

케라스를 이용한 딥러닝

목차

- 0. 소스공유/참고자료
- 1. 딥러닝이란 무엇인가?
- 2. 딥러닝을 위한 수학
- 3. 신경망 시작하기
- 4. 다층 신경망 이해
- 5. 주요 케라스 문법
- 6. 합성곱 신경망 이해
- 7. 순환 신경망 이해

github 접속 후 다운로드

<https://github.com/imguru-mooc/keras>

서적:



Do it! 딥러닝 입문
[박해선](#) 저 | 이지스퍼블리싱 | 2019년 09월 20일



케라스 창시자에게 배우는 딥러닝
프랑소와 솔레 저/박해선 역 | 길벗 | 2018년 10월 22일 |

온라인 강의 자료:

텐서 플로우 공식 사이트 : <https://www.tensorflow.org/tutorials>

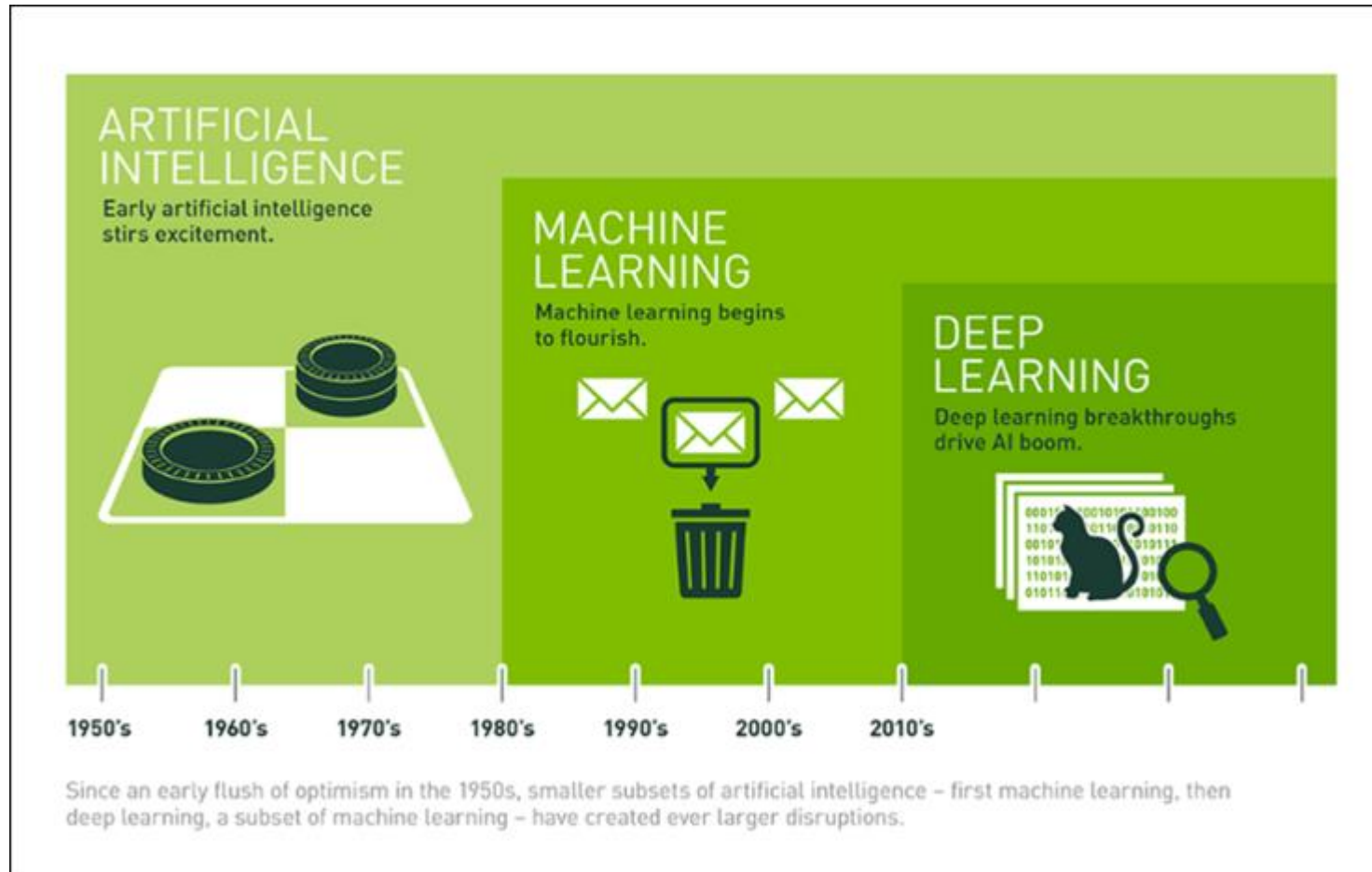
케라스 공식 사이트 : <https://keras.io>

1. 딥러닝이란 무엇인가?

1.1 인공지능과 머신러닝, 딥러닝

1.2 딥러닝 이전 : 머신 러닝의 간략한 역사

1.3 왜 딥러닝일까? 왜 지금일까?



*"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*

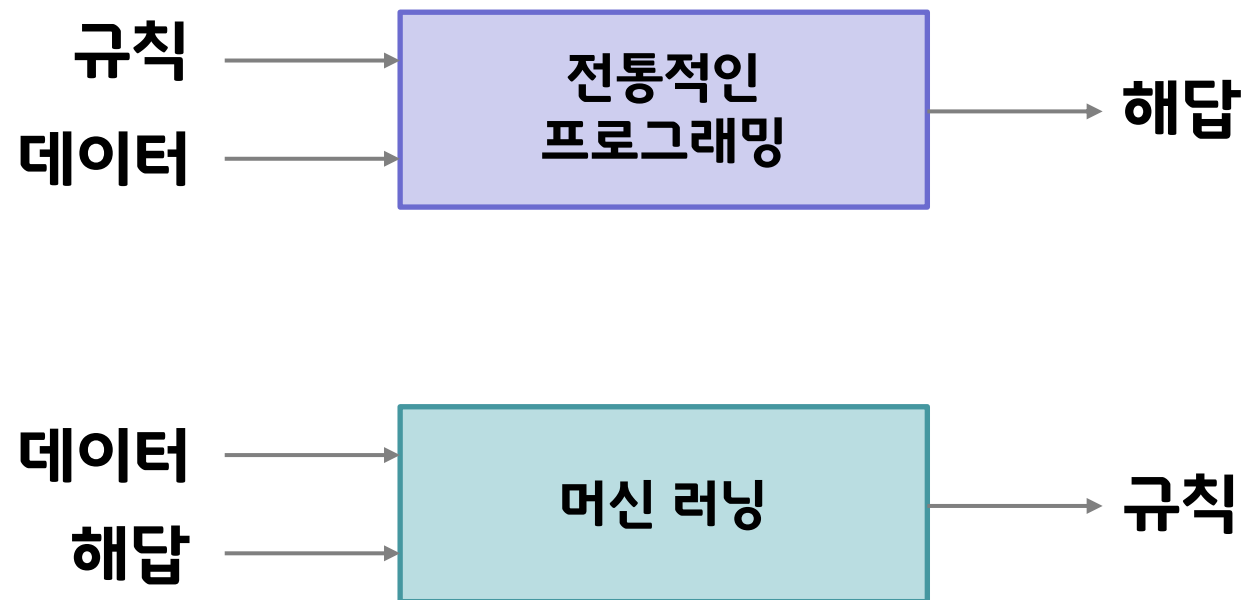
*명시적인 규칙을 충분하게 만들어 기계의 수준을 높이는 접근 방법을
symbolic AI 라고 하며, 1950년대부터 1980년대까지 AI의 지배적인
패러다임이었습니다.*

Basic concepts

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

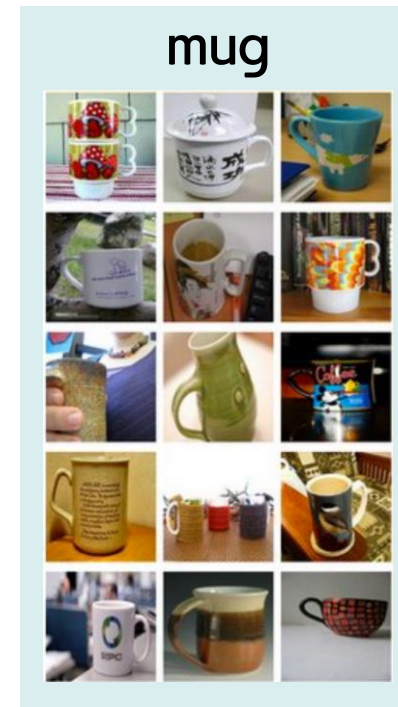
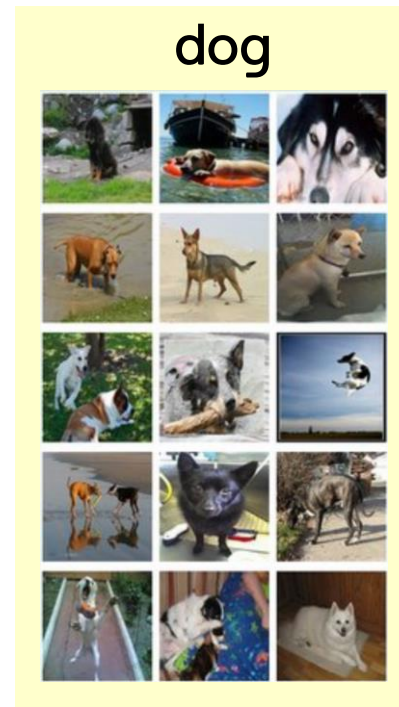
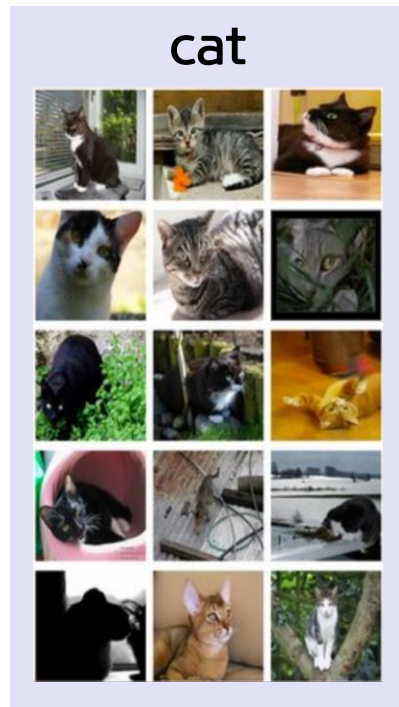
Machine Learning

- 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)



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

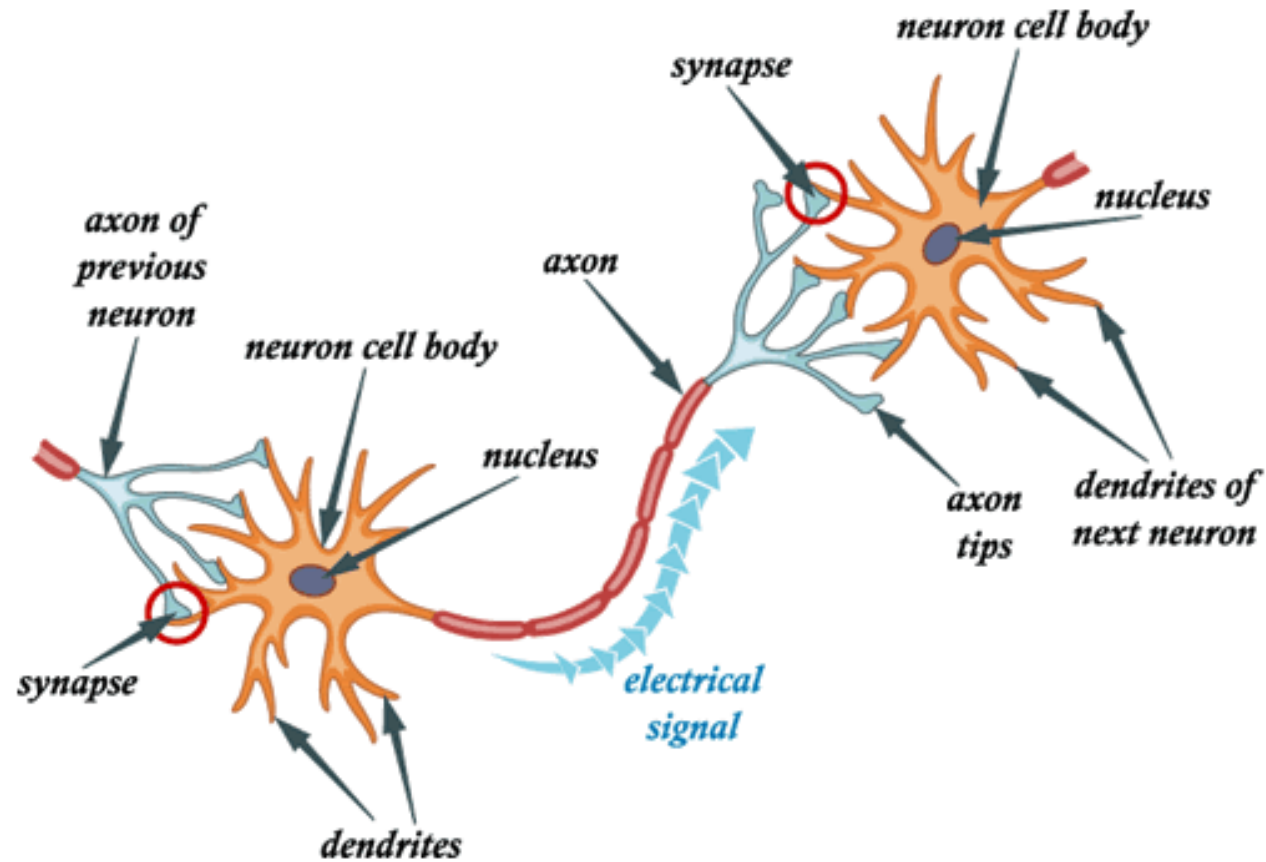
예) Google news grouping
페이스북 에서 특정 집단의 사람들을 그룹화
천체의 별 모양으로 분류

Supervised learning

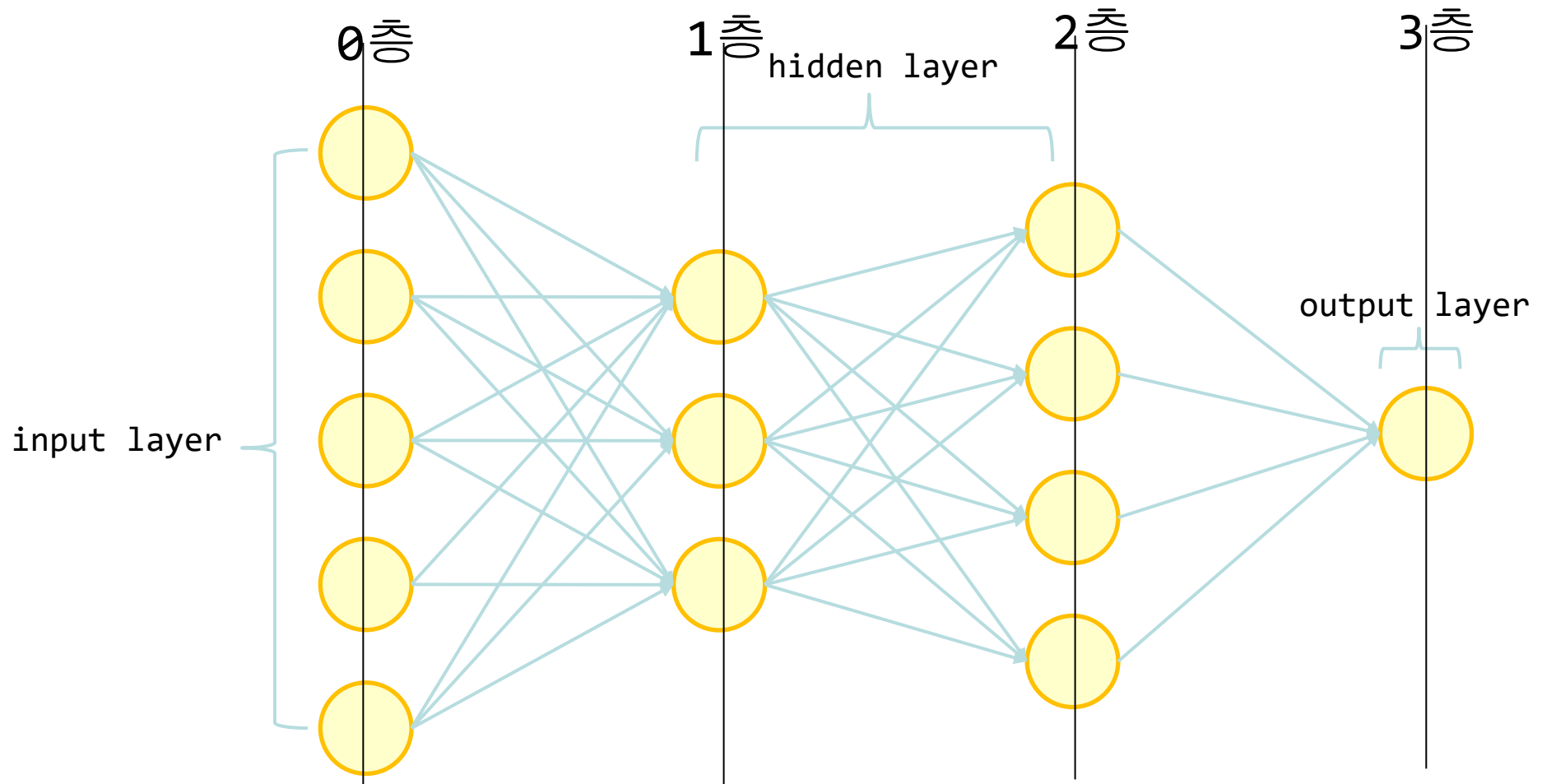
- ML에서 가장 일반적인 문제 유형
 - 시험 점수 예측 : 이전 시험에서 점수와 공부 시간
 - 이메일 스팸 필터 : 라벨이 있는 학습(스팸 또는 햄)
 - 이미지 라벨링 : 태그가 있는 이미지로 부터 학습

Types of supervised learning

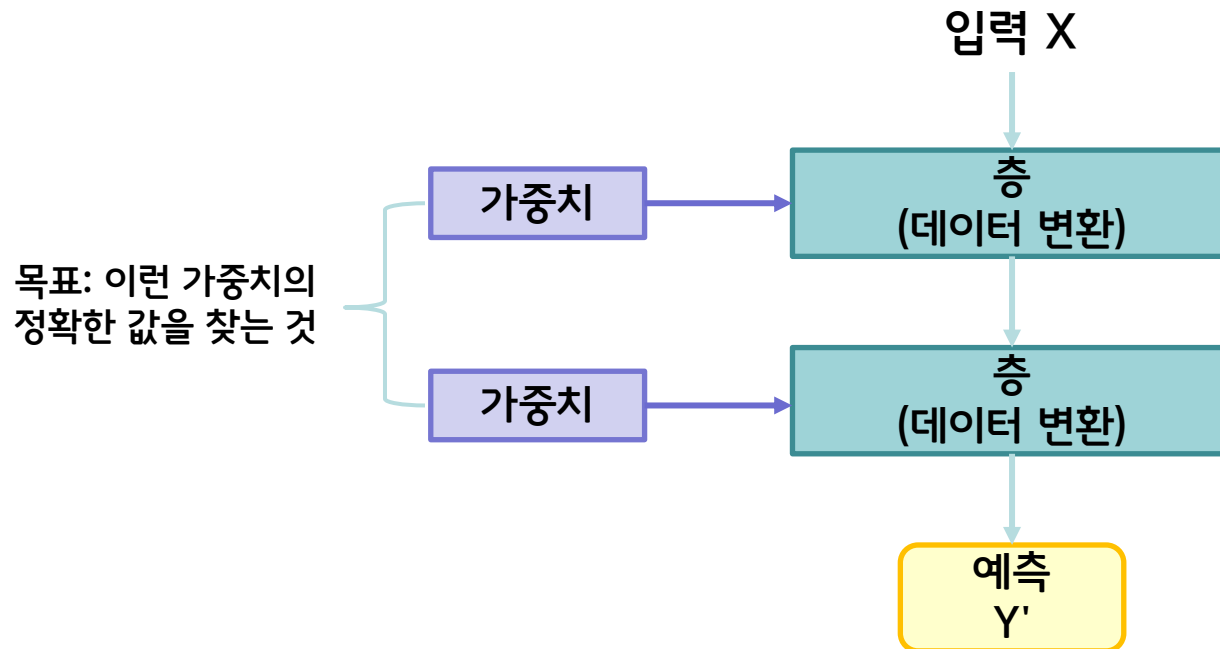
- 시험 점수 예측 : 이전 시험에서 점수와 공부 시간
 - regression
- 이메일 스팸 필터 : 라벨이 있는 학습(스팸 또는 햄)
 - binary classification
- 이미지 라벨링 : 태그가 있는 이미지로 부터 학습
 - multi-label classification



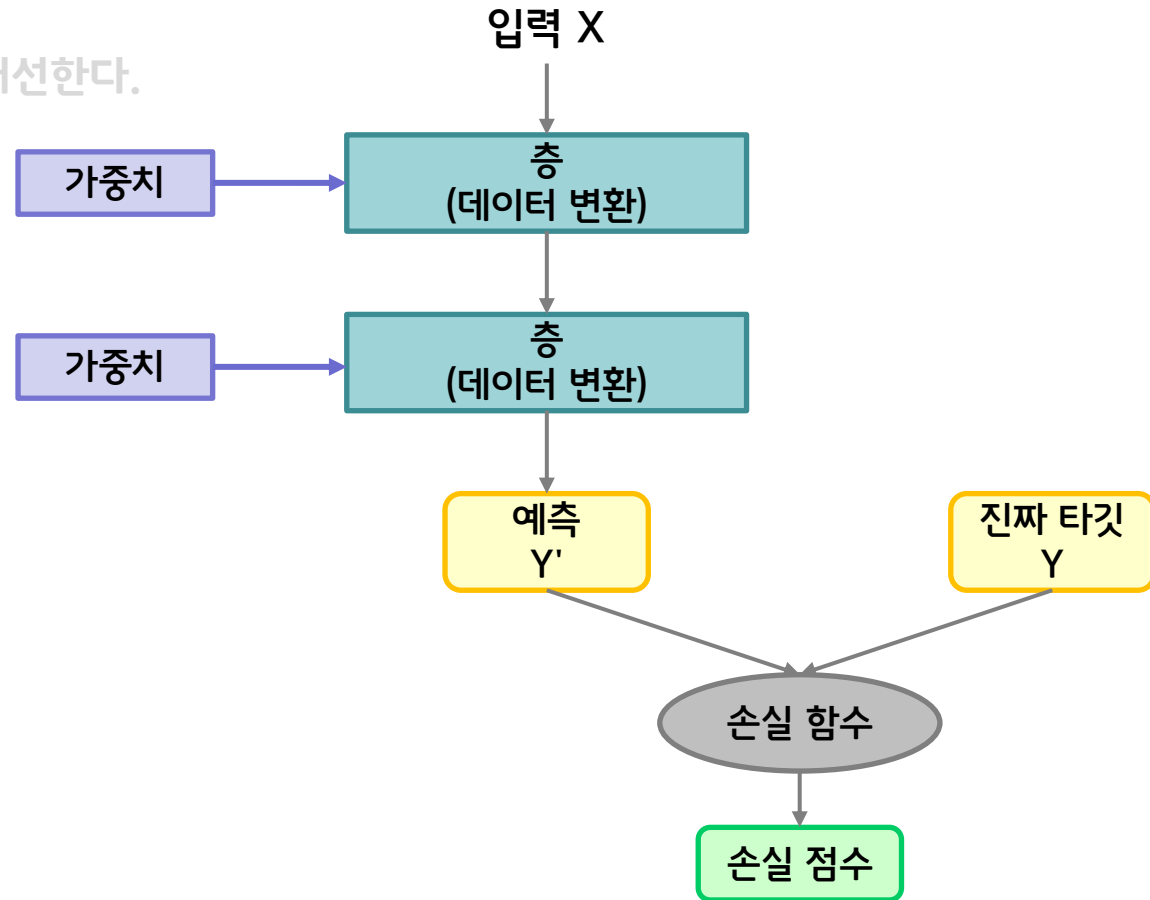
Multi-layer perceptron (MLP)는 Hidden layer라는 layer를 도입해 인풋을 한 차원 높은 단계의 특징, 즉 representation으로 나타낸다.



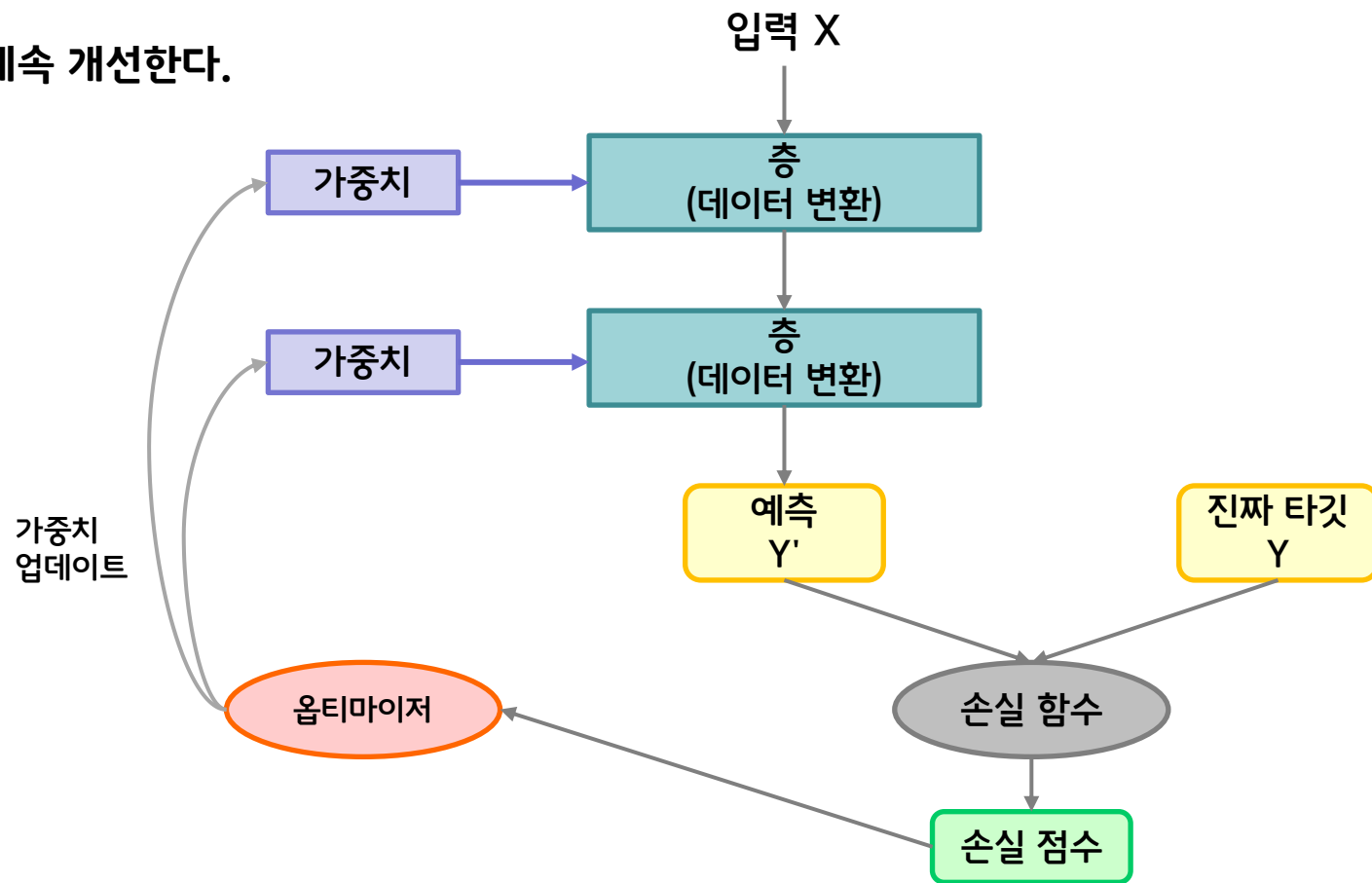
- 1.데이터를 입력한다.
- 2.여러 층을 통해 예상 결과값을 만든다. (매핑)
- 3.실제 값과 비교해서 그 차이를 구한다. (타겟과 손실함수)
- 4.차이를 줄이기 위한 방법으로 앞의 층들의 가중치를 수정해준다. (역전파)
- 5.이 방법의 반복으로 규칙을 계속 개선한다.



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



- ◆ 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 혹 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?

◆ 선형 회귀(Linear Regression)

- 예측 문제를 해결하는 사용
- 기존 데이터 셋을 이용하여 규칙을 찾는다.
- 머신 러닝에서는 규칙에 해당하는 W 와 b 를 스스로 학습한다.

◆ 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

◆ 1950년대

- 신경망의 핵심 아이디어 등장
- 본격적으로 시작되지 못함

◆ 1980년대

- 역전파 알고리즘 재발견
- 신경망에 역전파 알고리즘 적용 시작

◆ 1990년대

- 초창기 합성곱 신경망과 역전파를 연결
- 미국 우편 서비스에 이용

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 인터페이스 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.

2021에 발표한 TPU는 1 엑사플롭이다.(1000만개 노트북 PC용 프로세서와 동일)

‘데이터의 바다’라는 용어가 있듯이 현재는 데이터가 매우 많다.

저장 장치의 발전, 데이터 셋을 수집하고 배포할 수 있는 인터넷의 성장은 머신러닝에 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(2015)이후 많은 스타트업과 학생, 연구자들이 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 있다. 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)

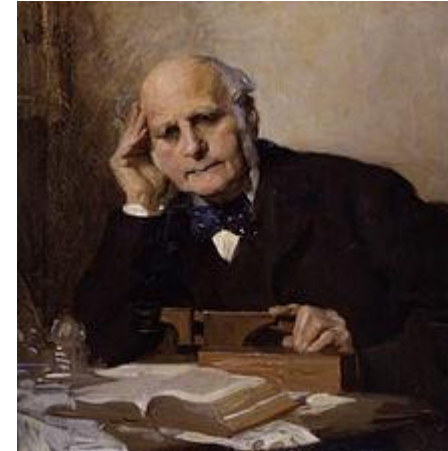
2.3 선형회귀 구현

2.4 시그모이드 함수

2.5 로지스틱 회귀 구현

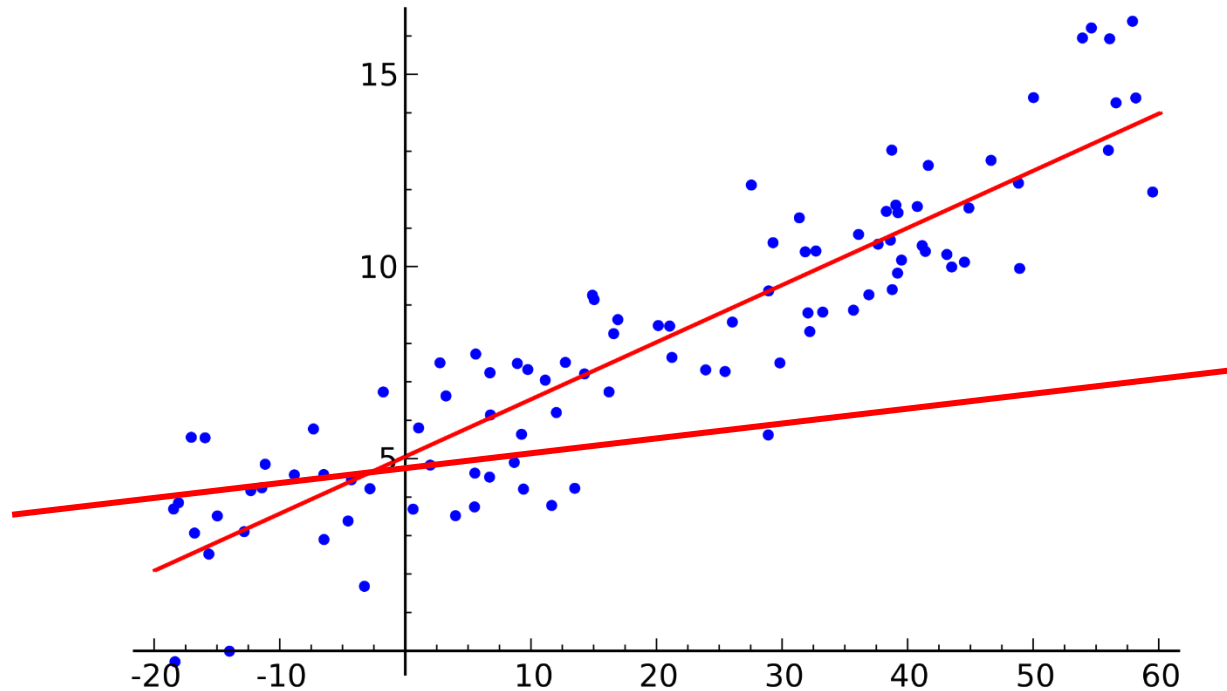
Regression - 회귀

"Regression toward the mean"



Sir Francis Galton
(1822 ~ 1911)

Linear Regression - 선형 회귀

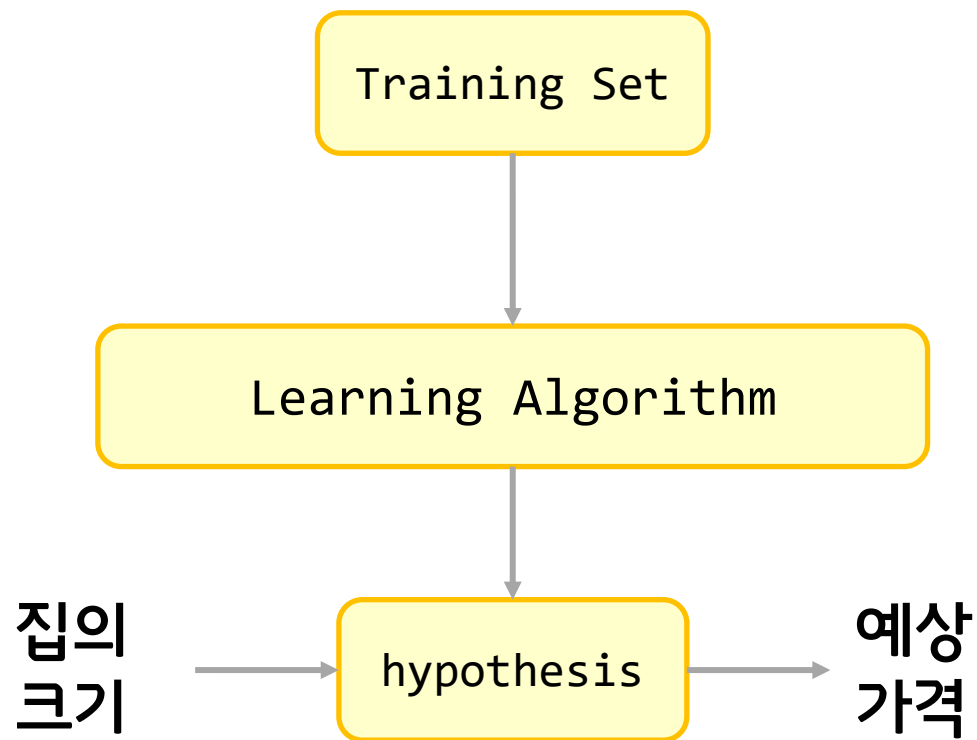


$$\frac{dy}{dx} = a$$

$$y = ax + b$$

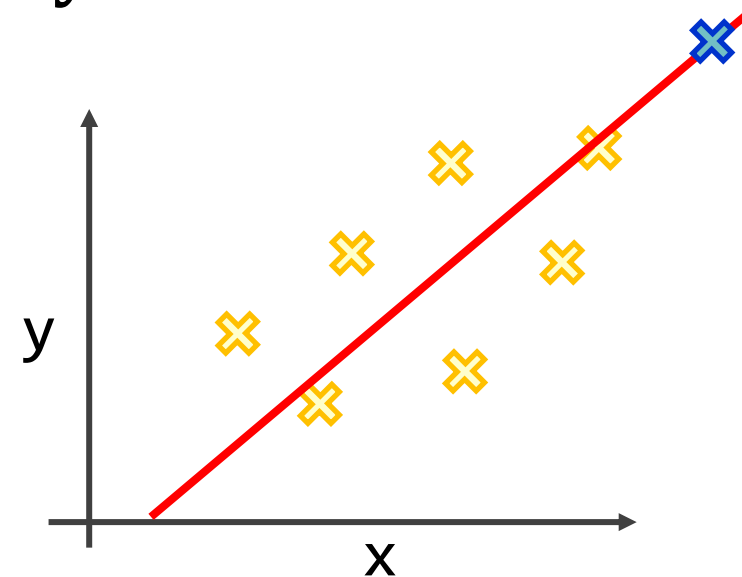
$$y = wx + b$$

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



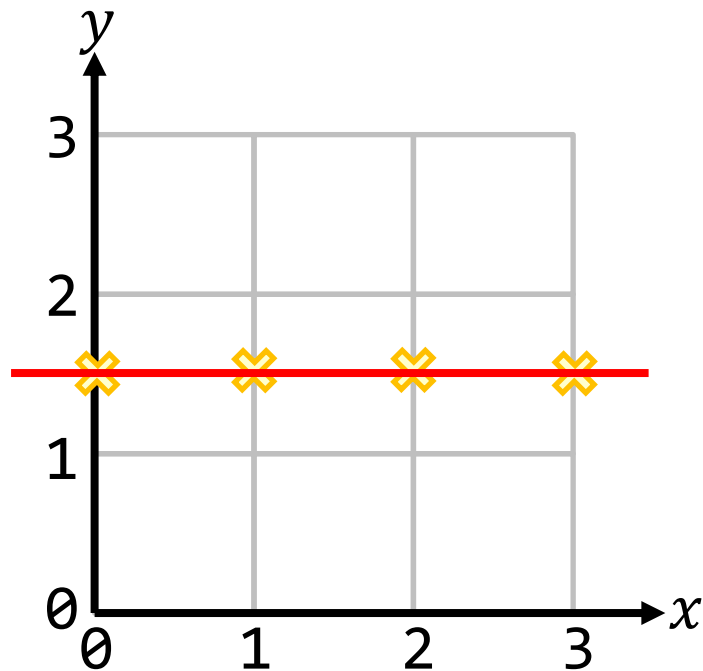
hypothesis ?

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



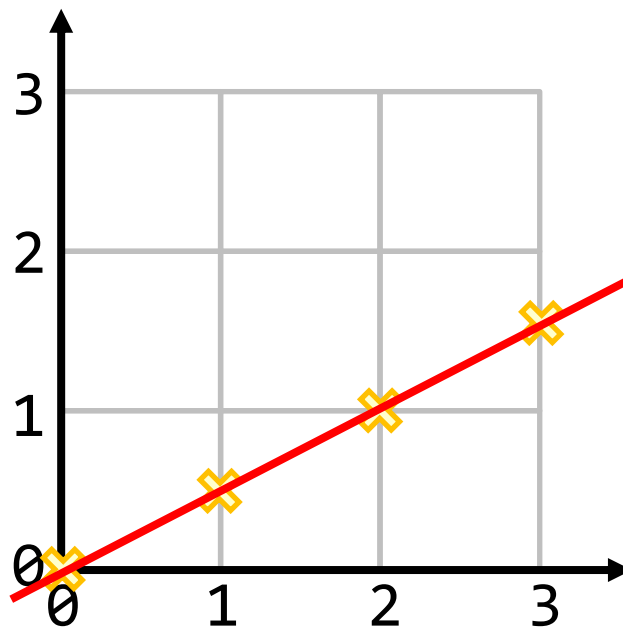
$$\hat{y} = wx + b$$

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



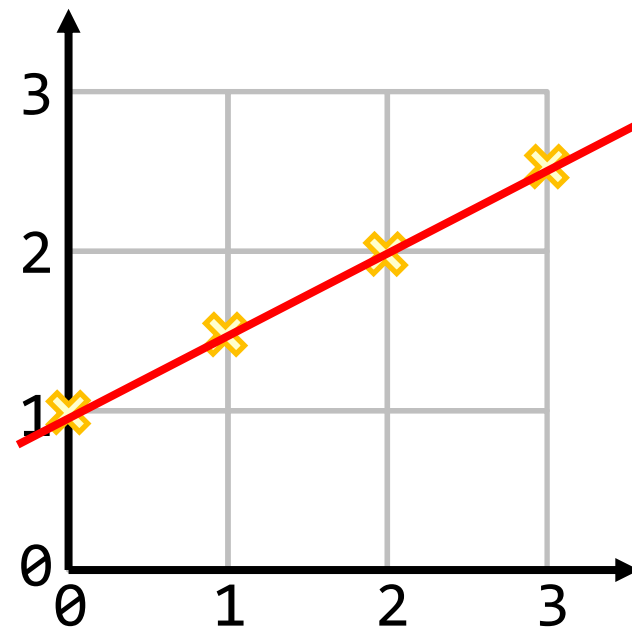
$$w = 0$$

$$b = 1.5$$



$$w = 0.5$$

$$b = 0$$



$$w = 0.5$$

$$b = 1$$

Hypothesis :

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

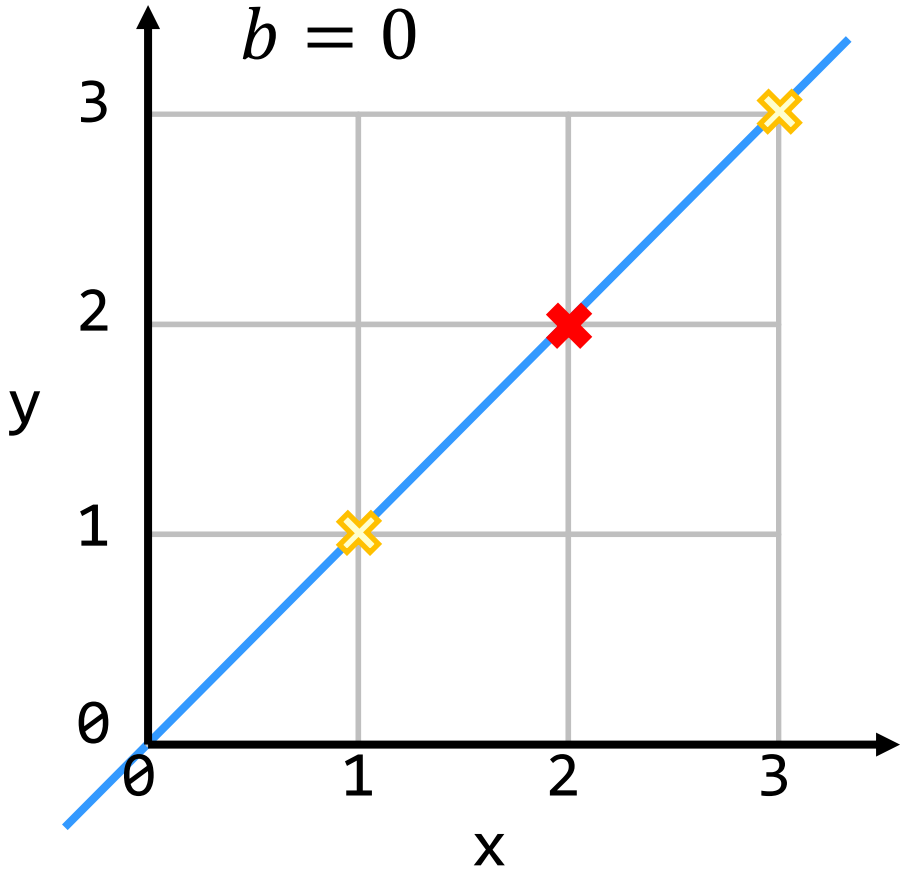
Parameters:

$$w, b$$

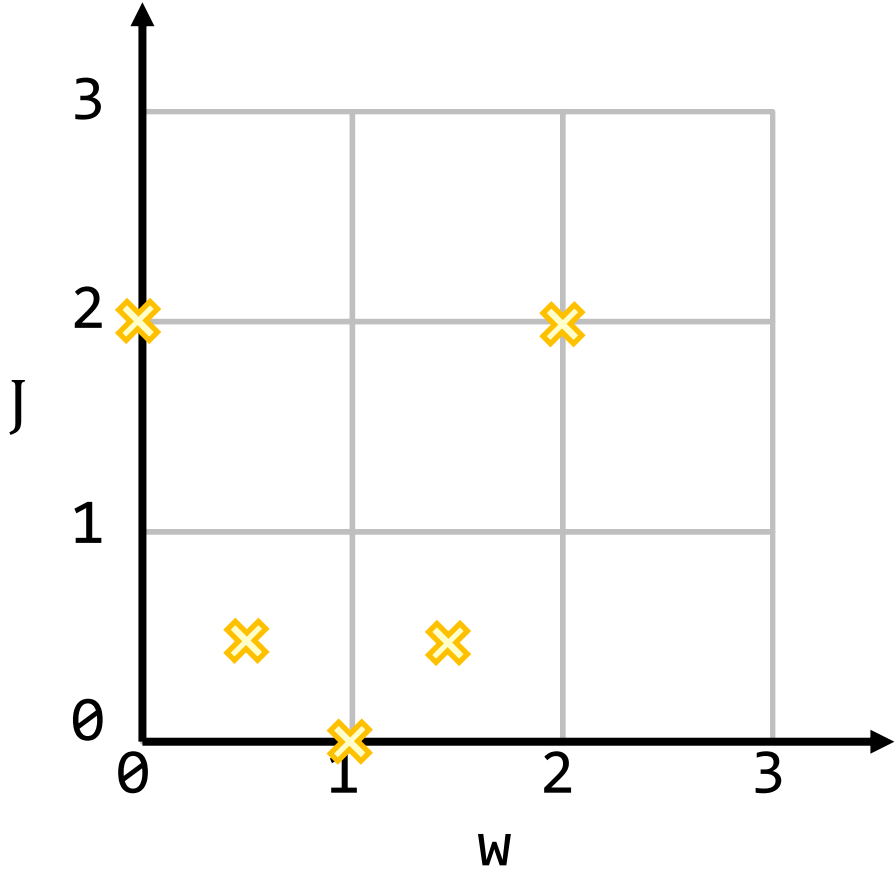
Cost(Loss) Function

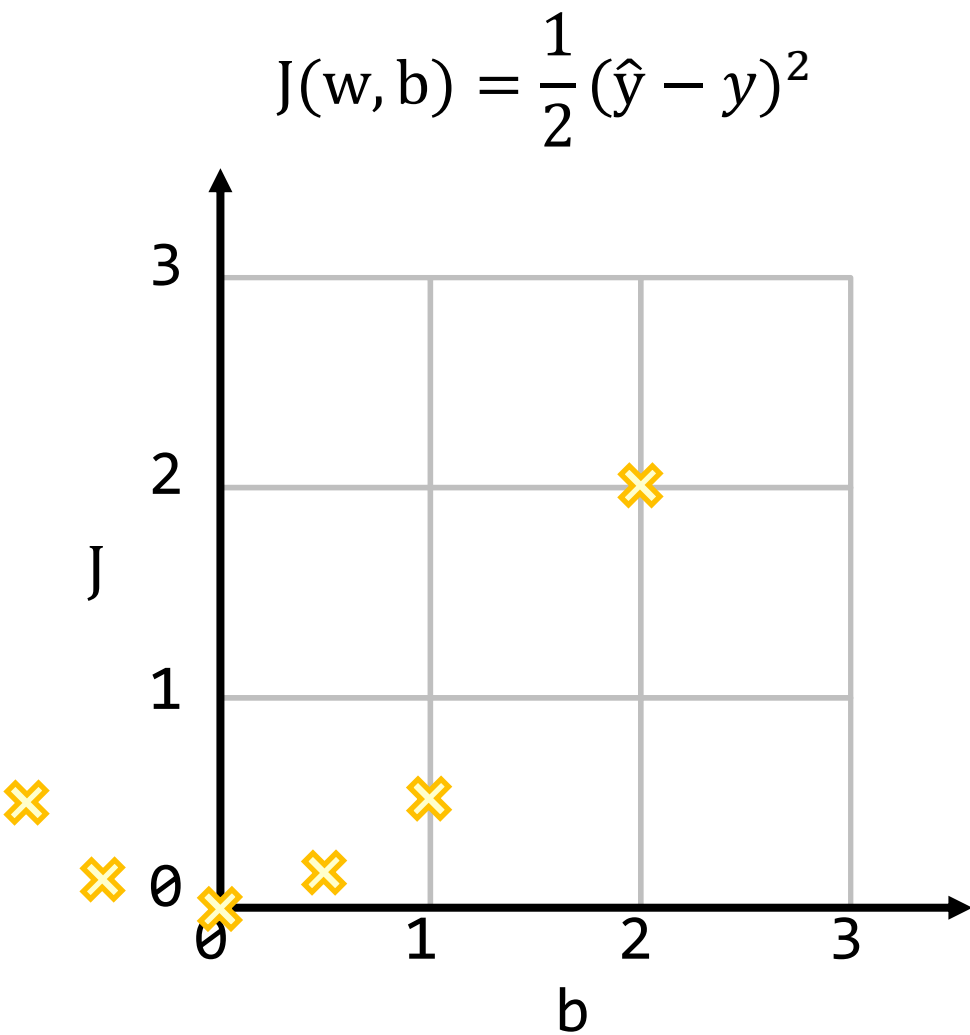
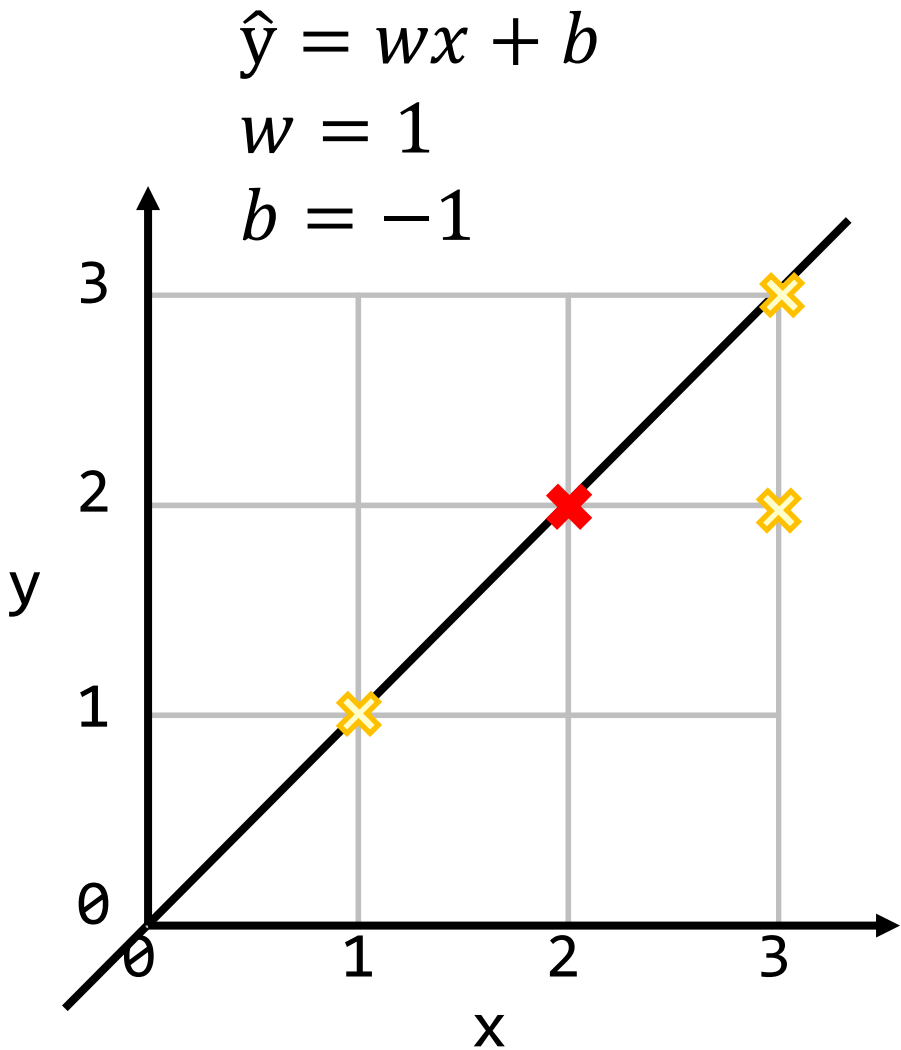
$$J(w, b) = \frac{1}{2} (\hat{y} - y)^2$$

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



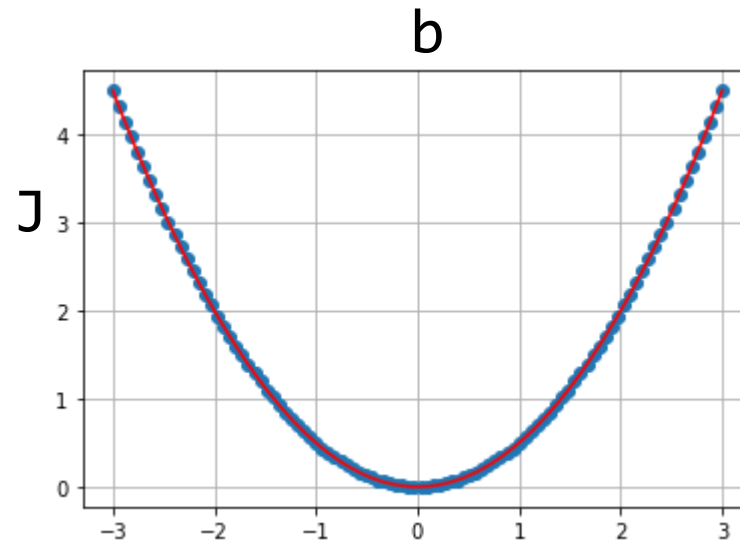
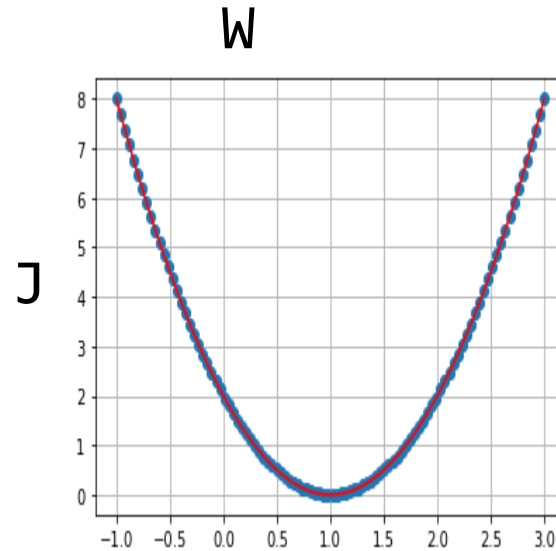
$J(w, b) = \frac{1}{2} (\hat{y} - y)^2$





$$y_{\text{hat}} = W * x + b$$

소스참조



2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 선형회귀 구현

2.4 시그모이드 함수

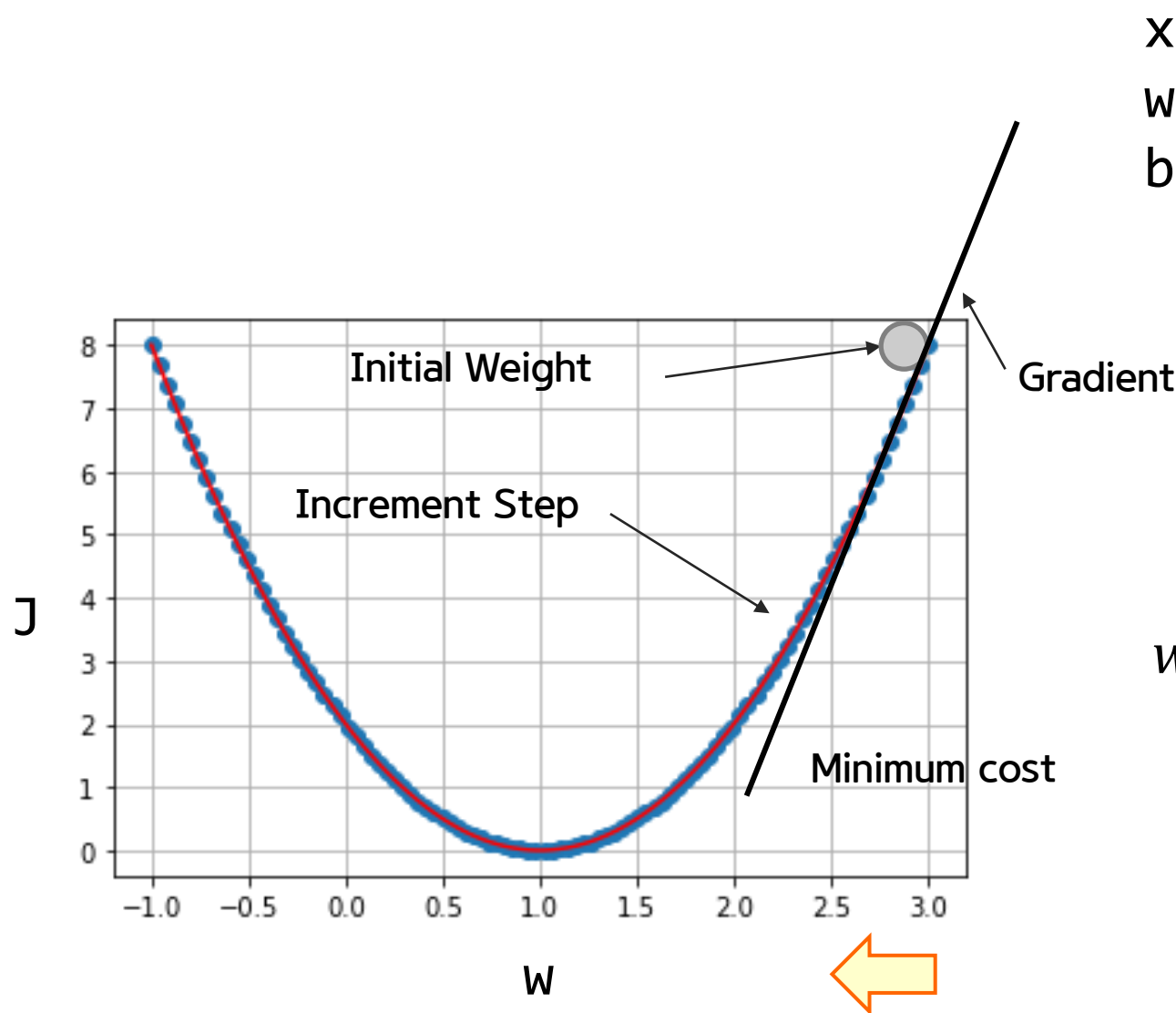
2.5 로지스틱 회귀 구현

Gradient descent algorithm

- 비용 함수 최소화
- 경사 하강은 많은 최소화 문제에 사용된다.
- 주어진 비용 함수, 비용 (W, b)에 대해 비용을 최소화하기 위해 W, b 를 찾는다.
- 일반적인 함수 : 비용 (w_1, w_2, \dots)에 적용 가능

작동 방식

- 초기 추측으로 시작
 - 0,0 (또는 다른 값)에서 시작
 - W 와 b 를 약간 변경하여 $\text{cost}(W, b)$ 의 비용을 줄이려고 노력
- 매개 변수를 변경할 때마다 가능한 가장 낮은 $\text{cost}(W, b)$ 을 감소시키는 기울기를 선택
- 반복
- 최소한의 지역으로 수렴할 때까지 수행



$$x=2, \quad y=2$$

$$w=3$$

$$b=0$$

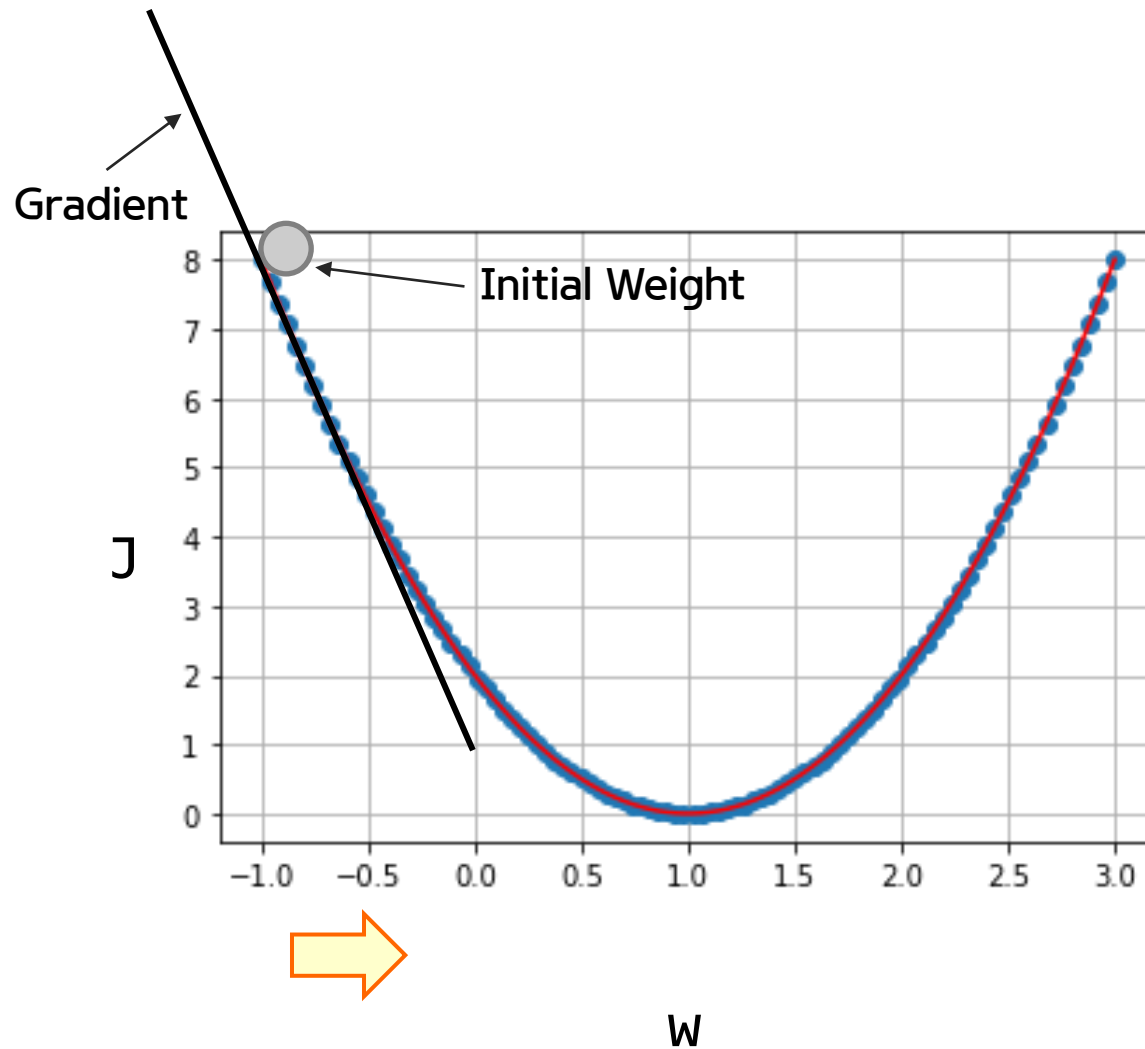
$$\hat{y} = wx + b$$

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

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

$$w = 3 - 0.08$$

$$w = 2.92$$



$$x=2, \quad y=2$$

$$w=-1$$

$$b=0$$

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

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

$$0.01$$

$$w = w - \alpha * \frac{dJ}{dw}$$

$$w = 1 + 0.00$$

$$w = 0.00$$

수렴까지 반복

{

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

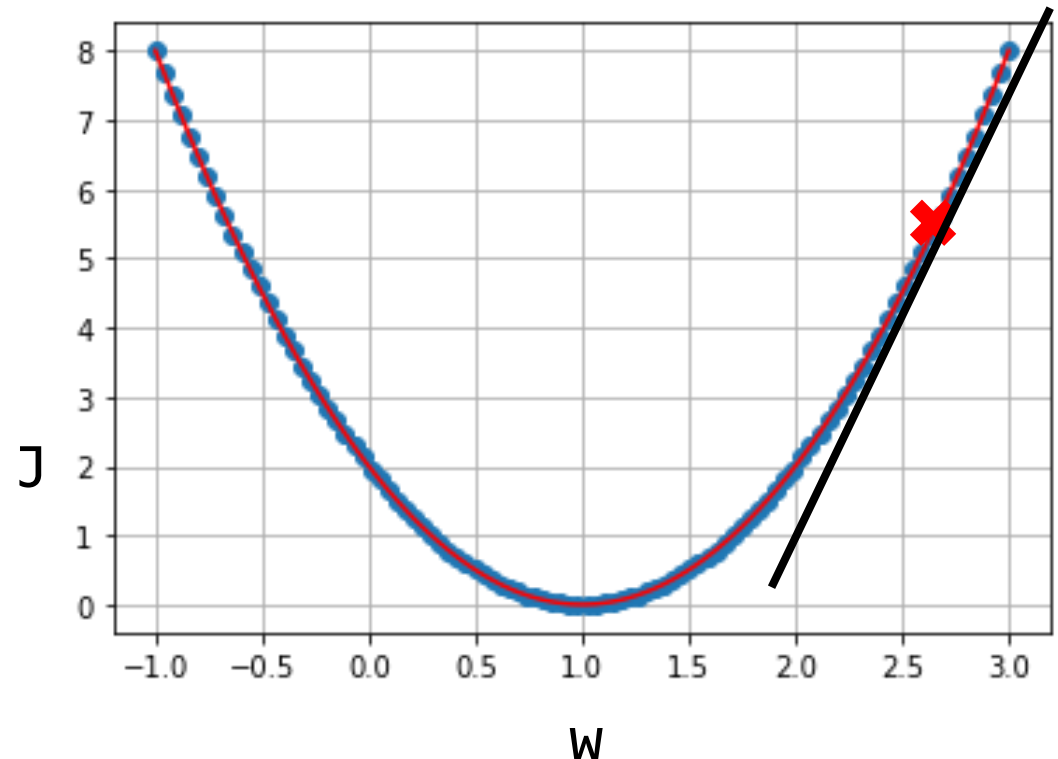
}

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 w 에 대해서 편미분 하면 된다.



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

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

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

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

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

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

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

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

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

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

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

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

$$= x$$

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

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

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

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

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

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

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

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

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

Chain Rule 사용

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

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

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

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

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

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

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

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

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

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

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

Chain Rule 사용

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

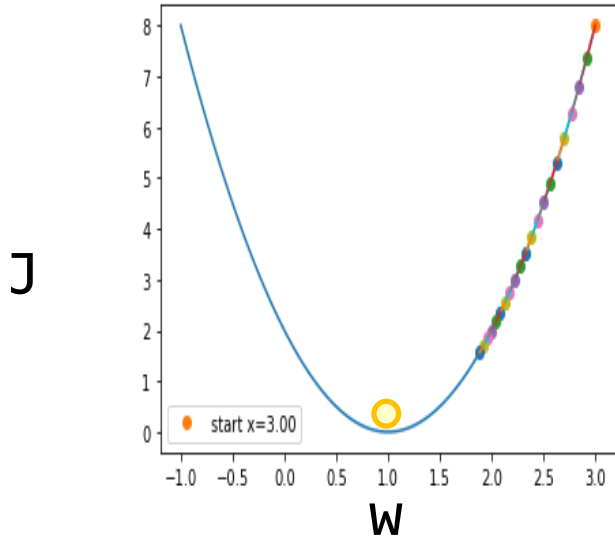
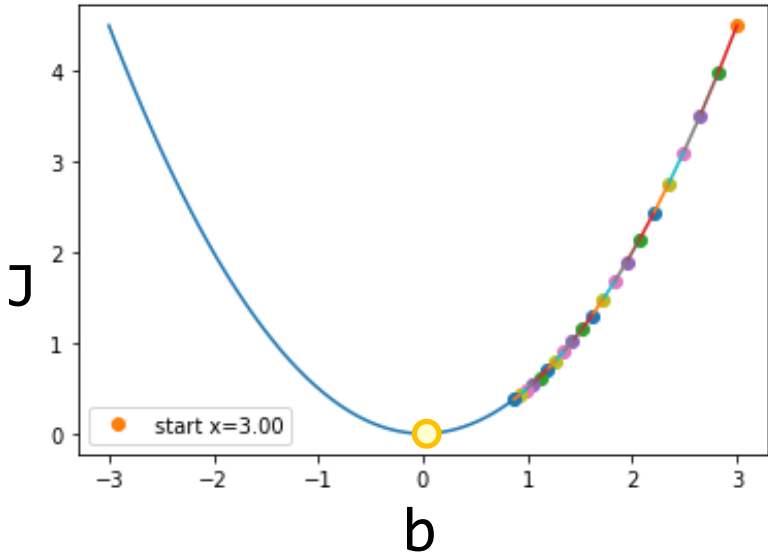
$$J(w, b) = \frac{1}{2} (\hat{y} - y)^2$$

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

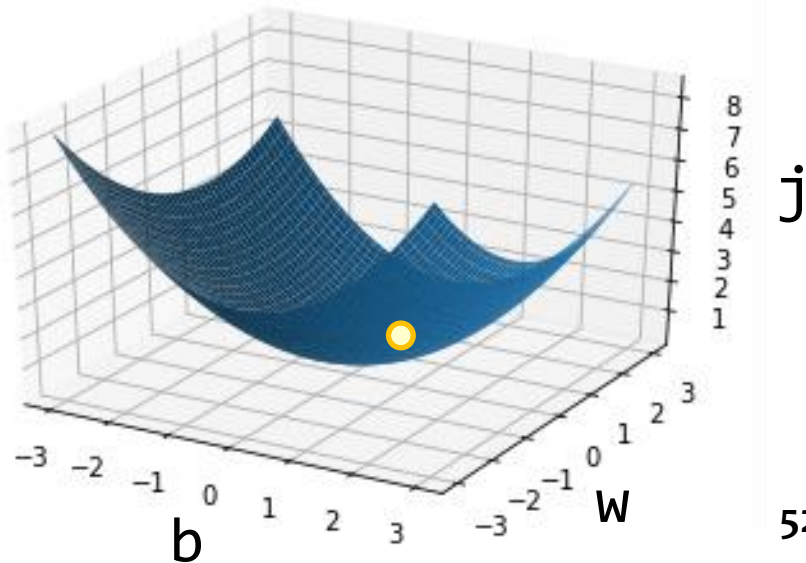
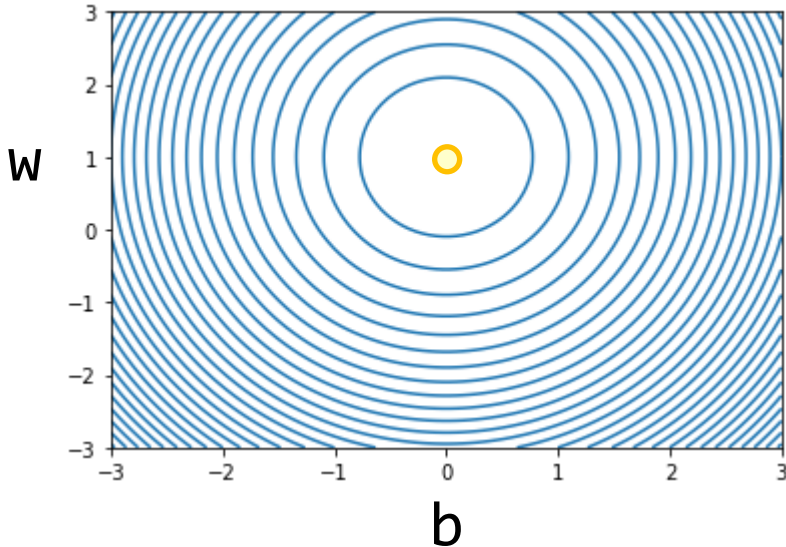
Gradient descent algorithm

2.2 SGD (Stochastic Gradient Descent)

소스 참조



$$J(w, b) = \text{Loss}$$



2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 선형회귀 구현

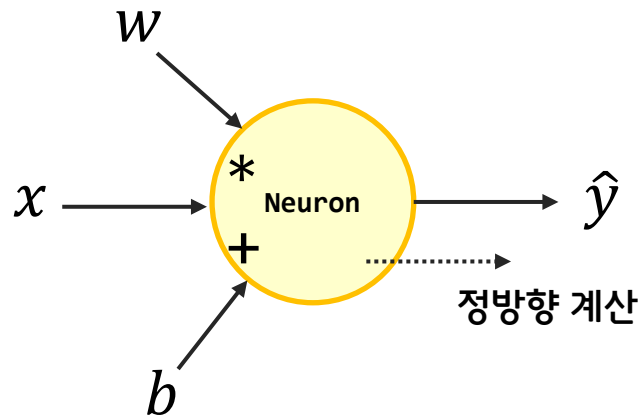
2.4 시그모이드 함수

2.5 로지스틱 회귀 구현

소스 참조

정방향 계산 만들기

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



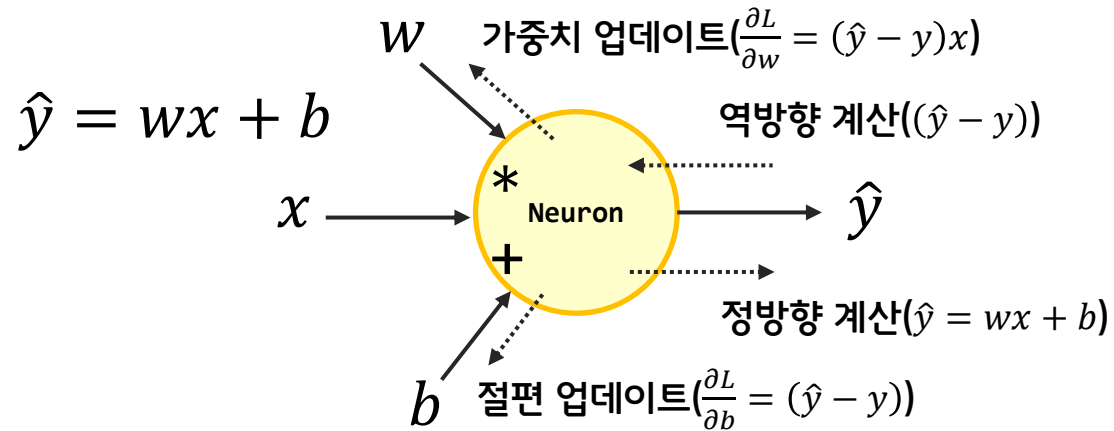
$$\hat{y} = wx + b \quad \# \text{ 선형연산}$$

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
```

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

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



소스 참조

2. 딥러닝을 위한 수학

2.1 MSE(Mean Squared Error)

2.2 SGD (Stochastic Gradient Descent)

2.3 선형회귀 구현

2.4 시그모이드 함수

2.5 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

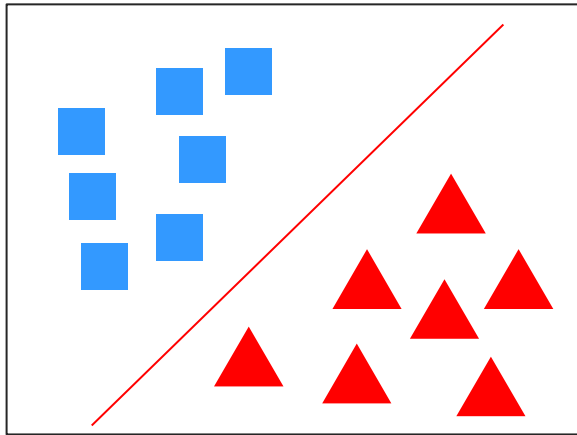
Binary Classification(이진 분류) 란?

: 값은 0 또는 1

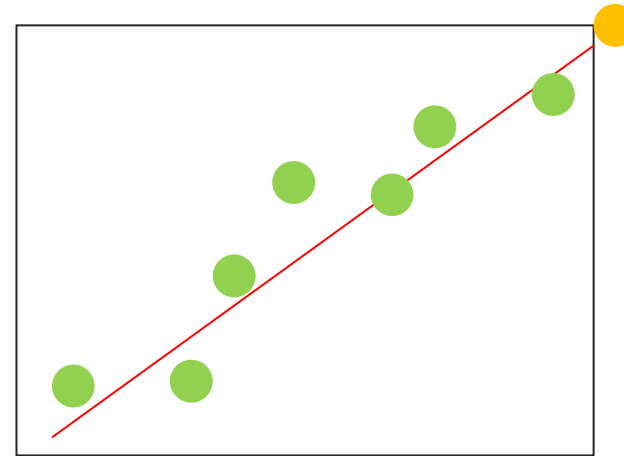
- Exam : Pass or **Fail**
- Spam : Not Spam or **Spam**
- Face : Real or **Fake**
- Tumor : Not Malignant or **Malignant**

Logistic vs Linear

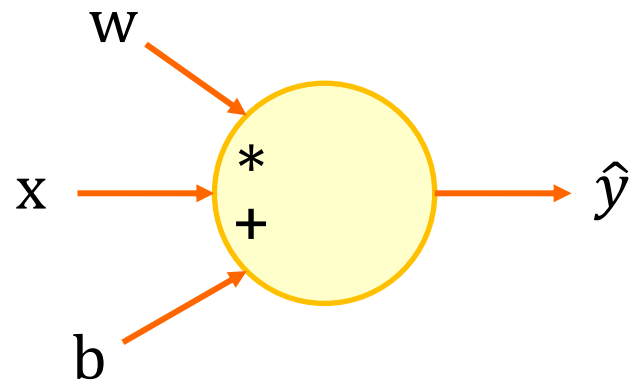
로지스틱 회귀와 선형 회귀의 차이점은?



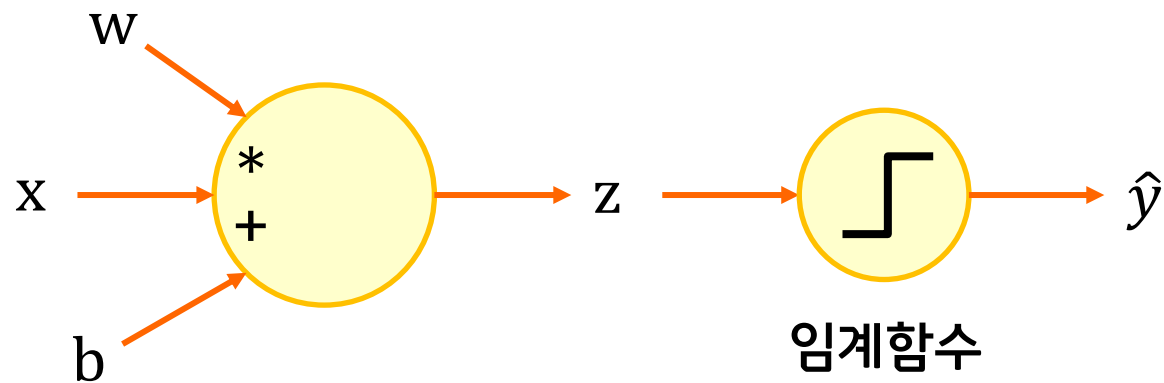
Discrete(분리) : 분류 목적
신발 사이즈 / 회사의 근로자수

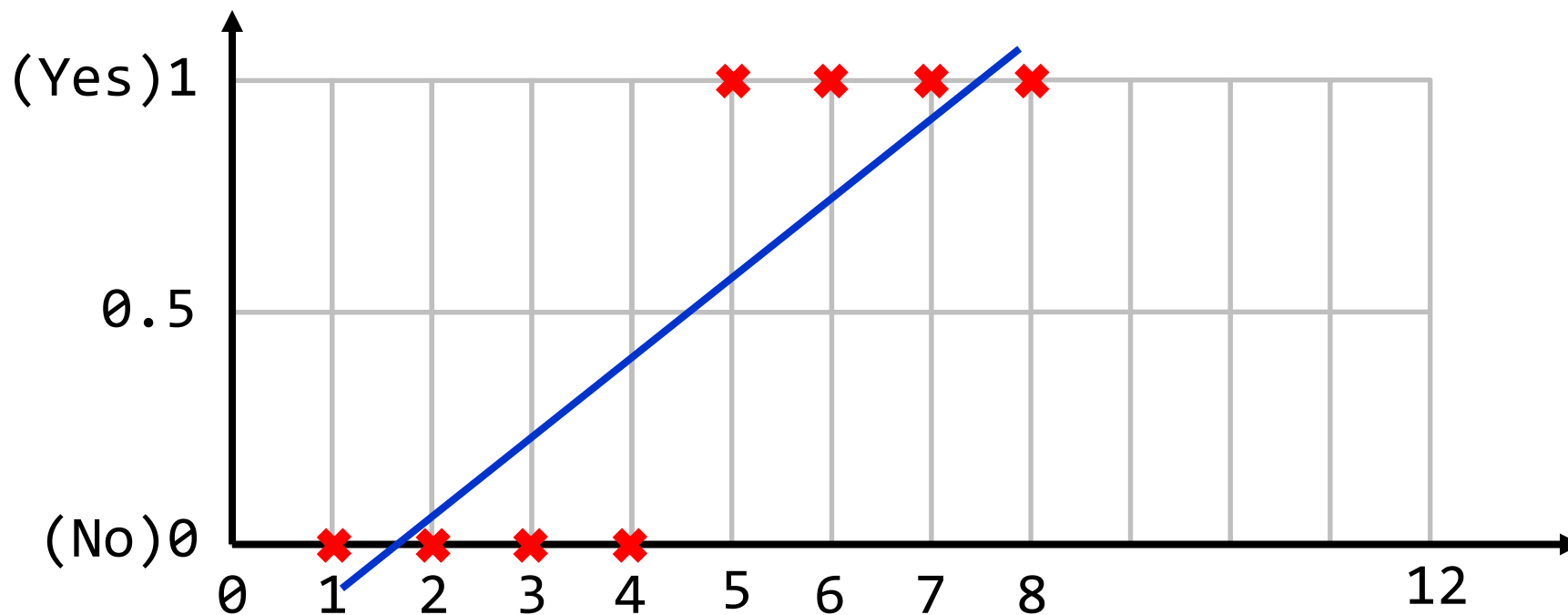


Continuous (지속적인) : 값의 예측
시간/ 무게 / 높이



퍼셉트론





Threshold classifier output at 0.5:

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

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

1 => 양성

2 => 양성

3 => 양성

4 => 양성

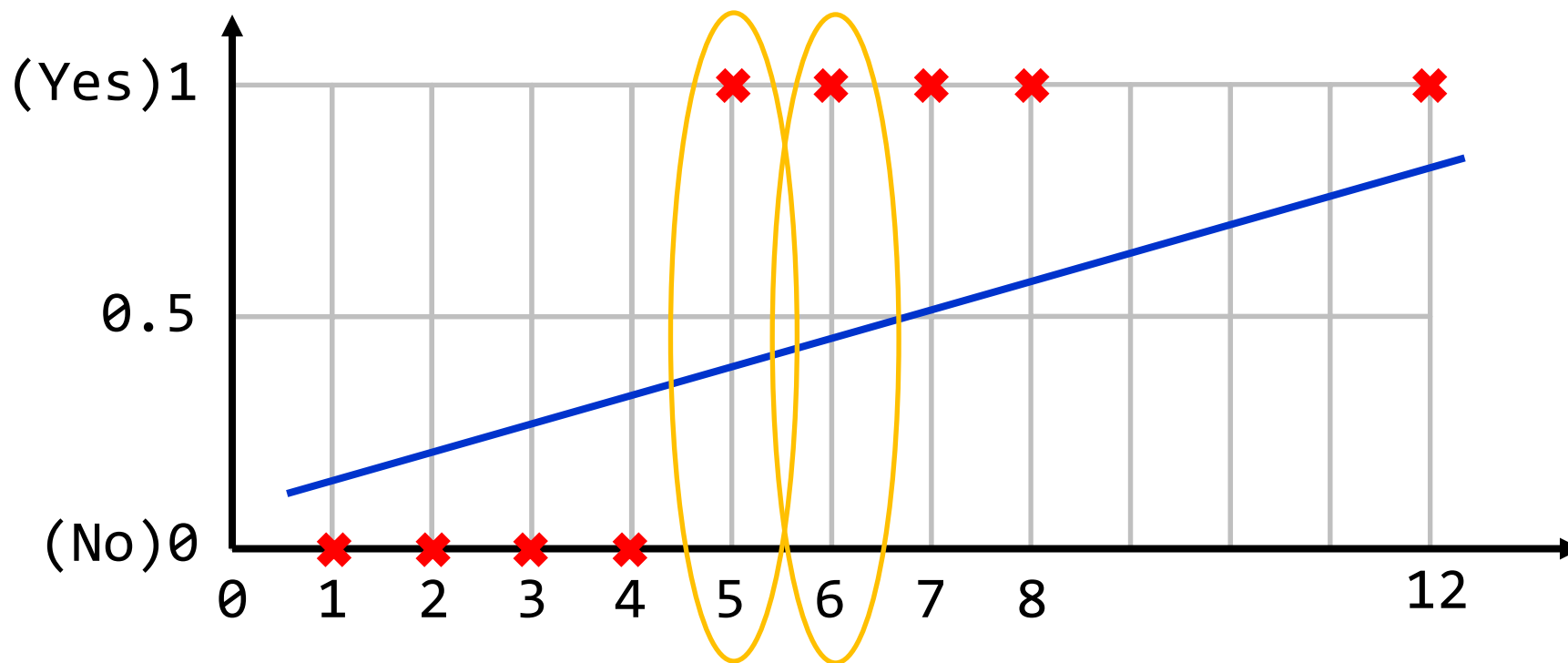
5 => 악성

6 => 악성

7 => 악성

8 => 악성

소스참조



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

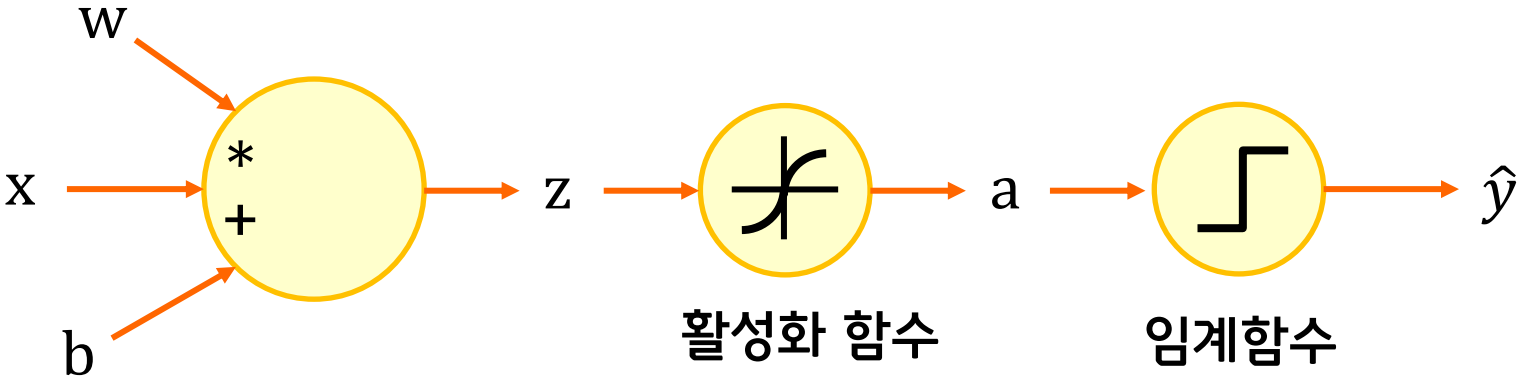
4 => 양성

5 => 양성

6 => 양성

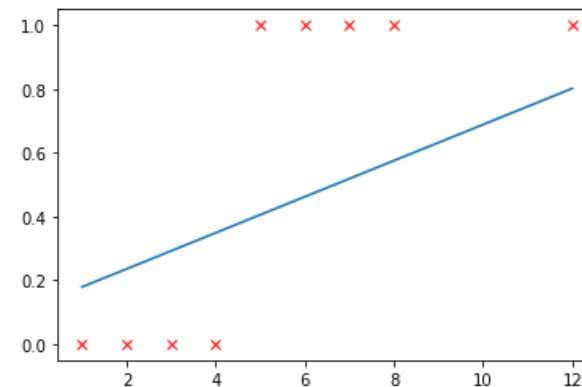
7 => 악성

8 => 악성



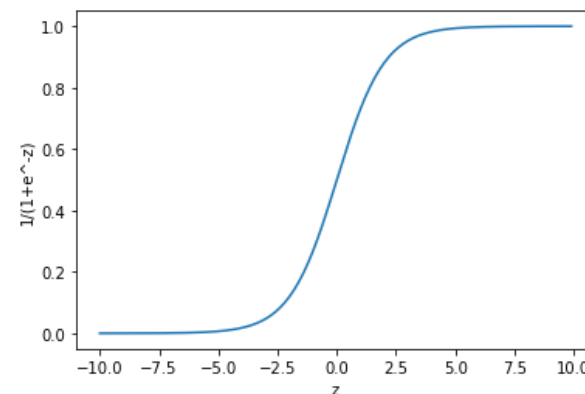
Classification: $y=0$ or 1

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



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

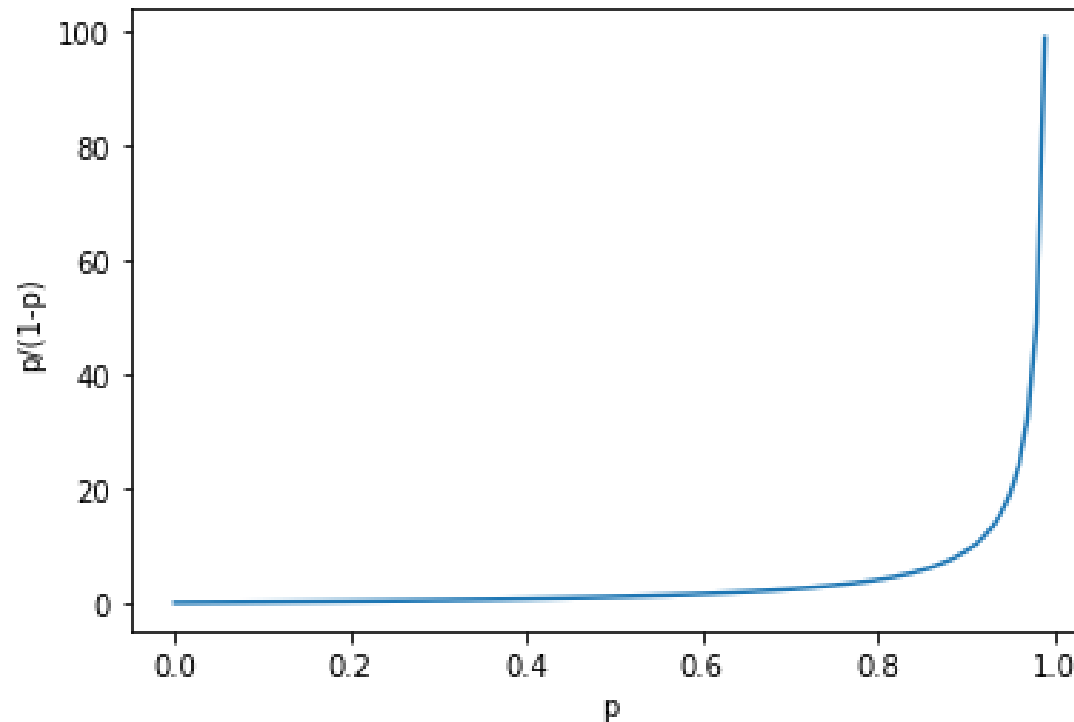
분류 문제 사용



소스참조

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

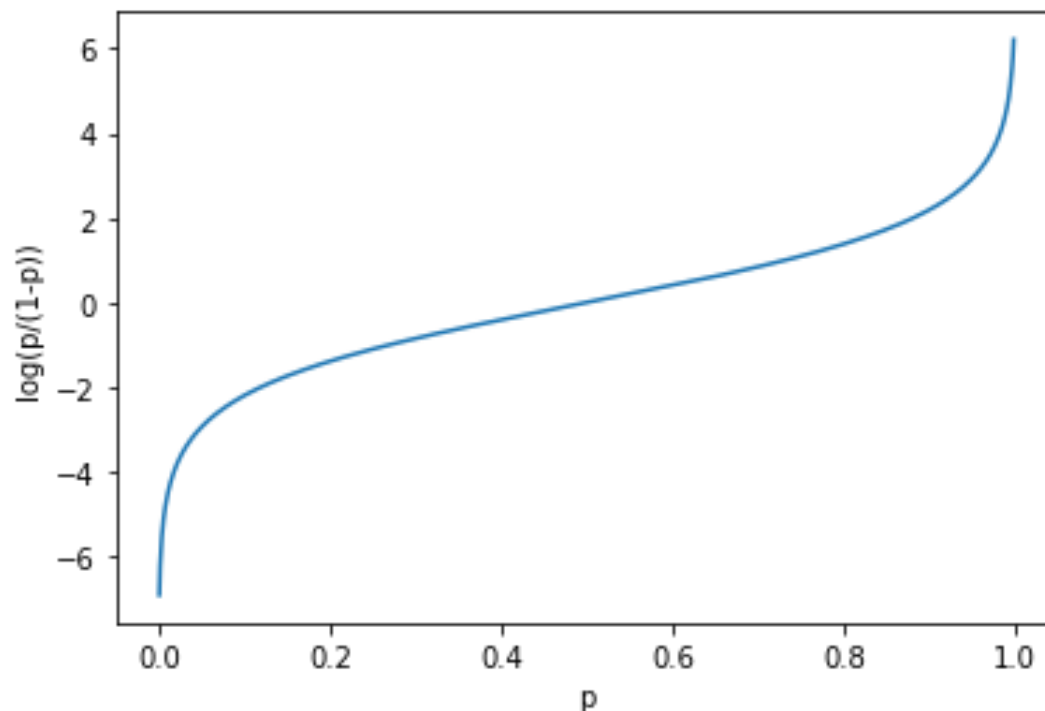
오즈 비를 그래프로 그리면 다음과 같다.
p가 0부터 1까지 증가할 때 오즈 비의 값은 처음에는
천천히 증가하지만 p가 1에 가까워지면 급격히 증가한다.



소스참조

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

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



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

소스참조

로짓 함수의 유도 : p에대해 정리

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

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

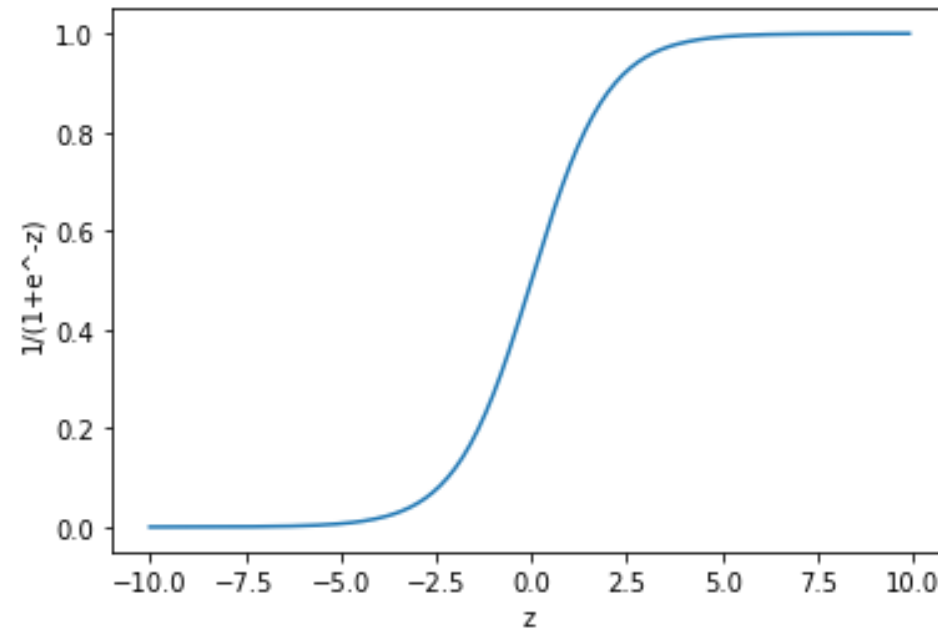
$$\frac{p}{1-p} = e^z$$

$$p = (1 - p) * e^z$$

$$p = e^z - p * e^z$$

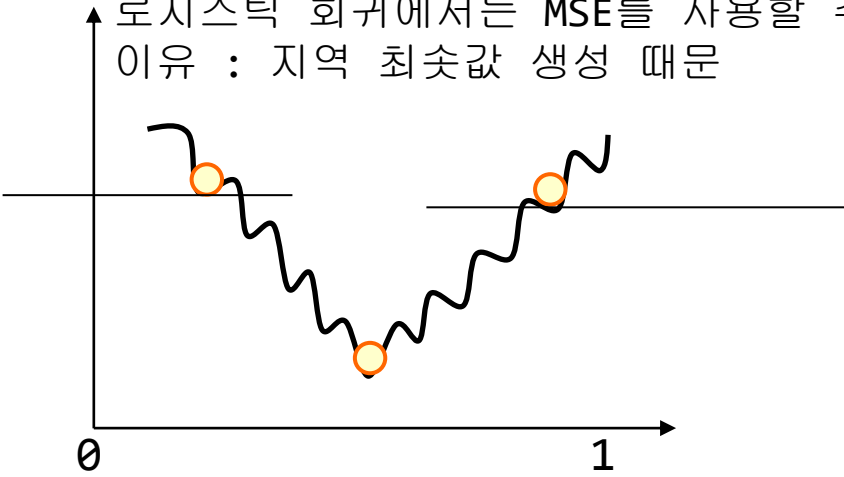
$$p + p * e^z = e^z$$

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



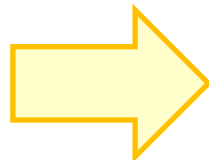
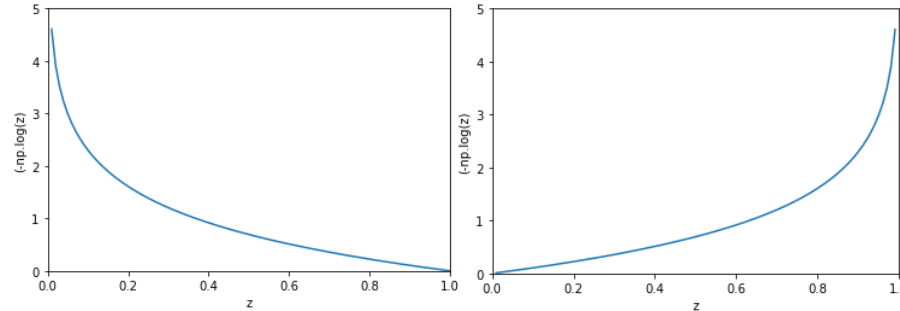
A convex logistic regression cost function

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



Binary Cross Entropy 함수

소스참조



$\int - \text{ / } = \text{ ~ }$

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

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

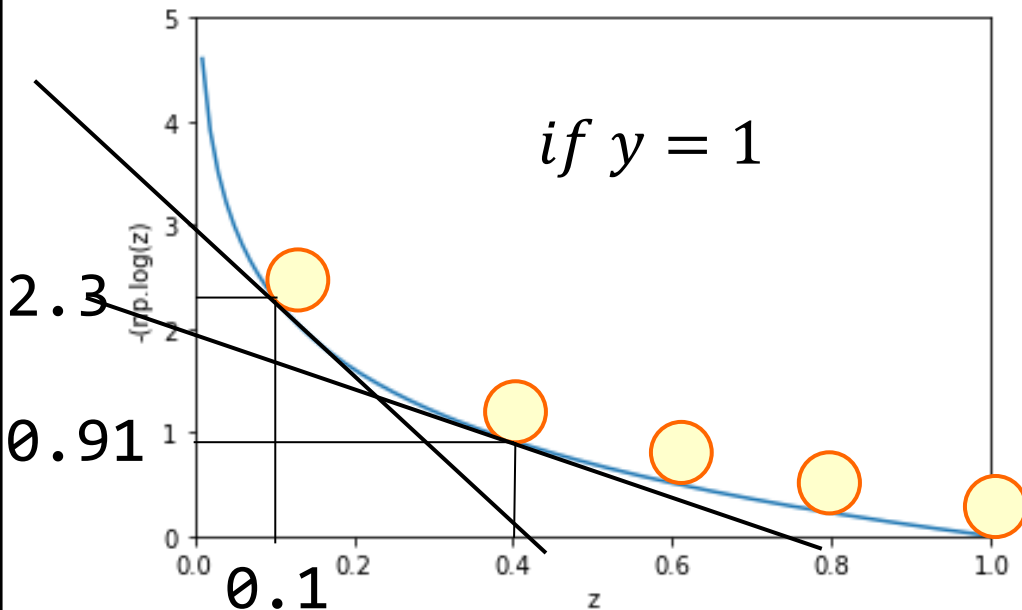
$$\text{Cost}(\hat{y}, y) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1 - \hat{y}) & \text{if } y = 0 \end{cases}$$

Logistic regression cost function

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

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

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



$\hat{y} \rightarrow 0.6$
Cost $\rightarrow 2.3 \rightarrow 0.91 \rightarrow 0.51$

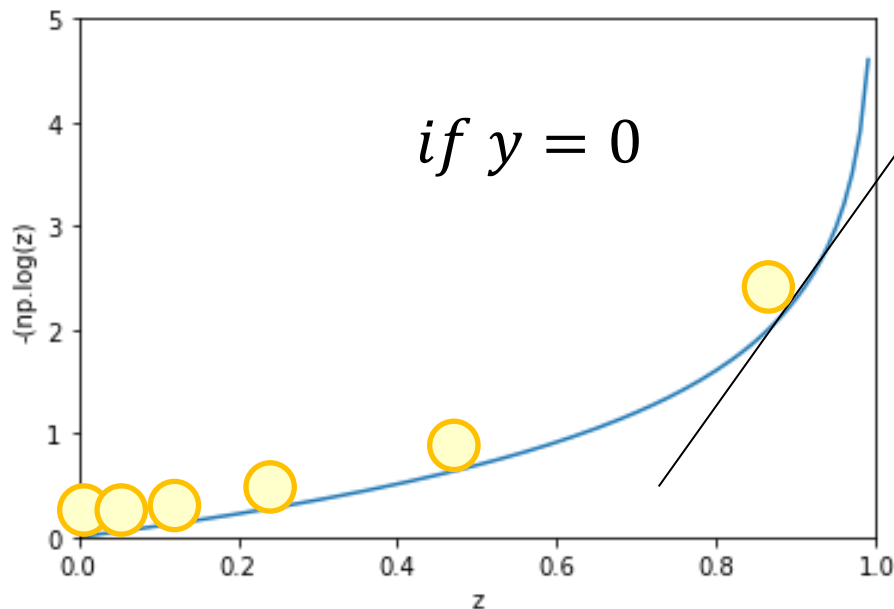
Logistic regression cost function

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

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

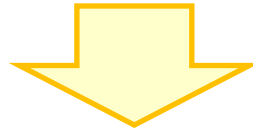
But as $\hat{y} \rightarrow 1$
Cost $\rightarrow \infty$

$\hat{y} \rightarrow 0.9 \rightarrow 0.5$
Cost $\rightarrow 2.3 \rightarrow 0.69$



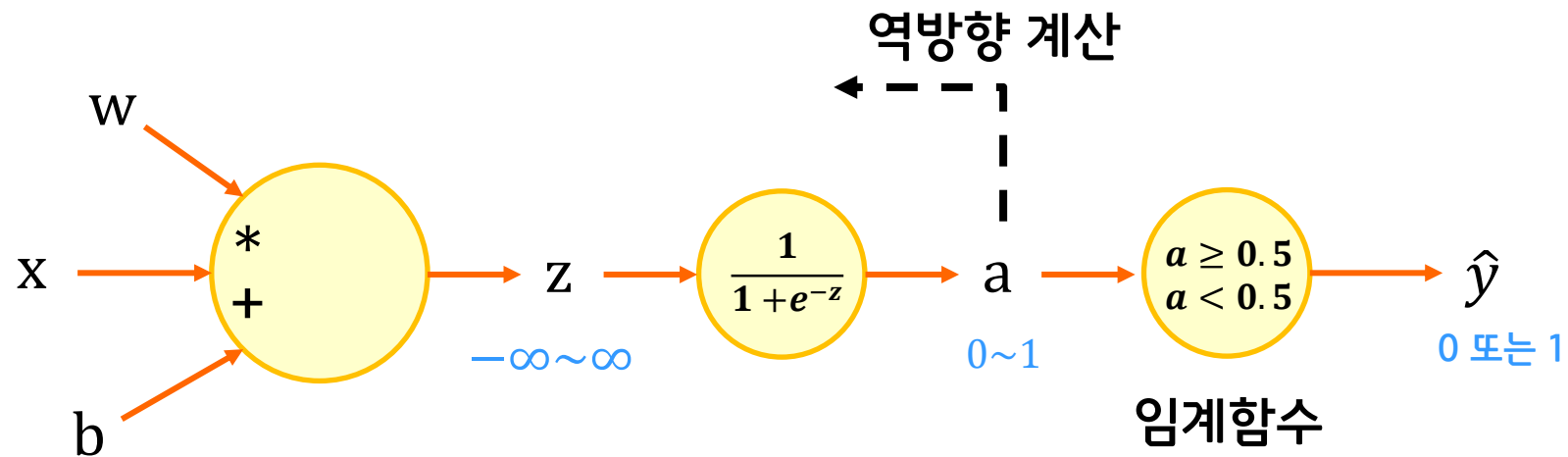
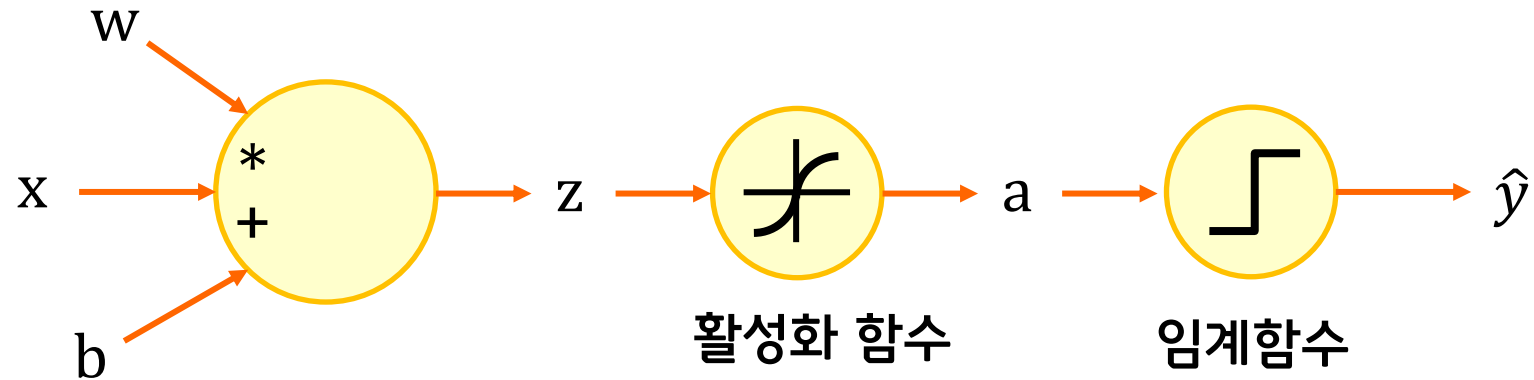
Logistic regression cost function

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



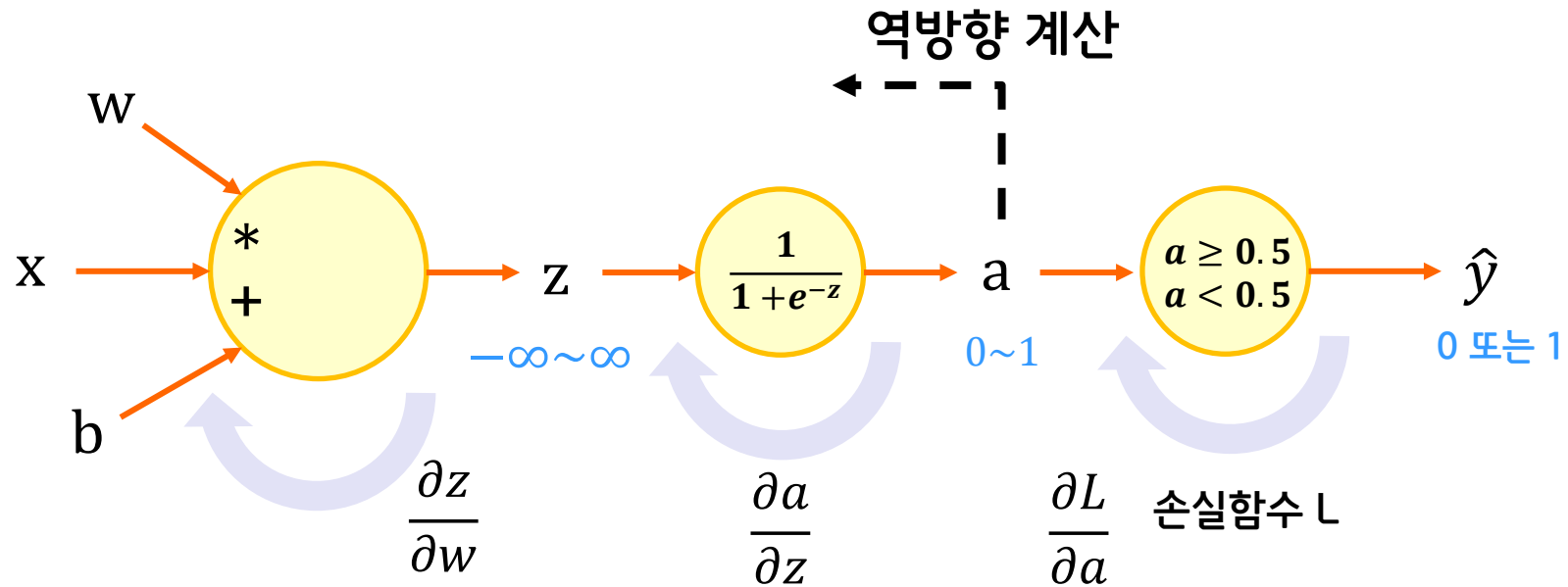
$$L = -(y * \log(a) + (1 - y) \log(1 - a))$$

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



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

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

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

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

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

$$= -\left(y \frac{1}{a} - (1 - y) \frac{1}{1 - a}\right) a(1 - a)$$

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

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

$$= (a - y)$$

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 한개인 경우 => x 1개

$$y = wx + b$$

Chain Rule 사용

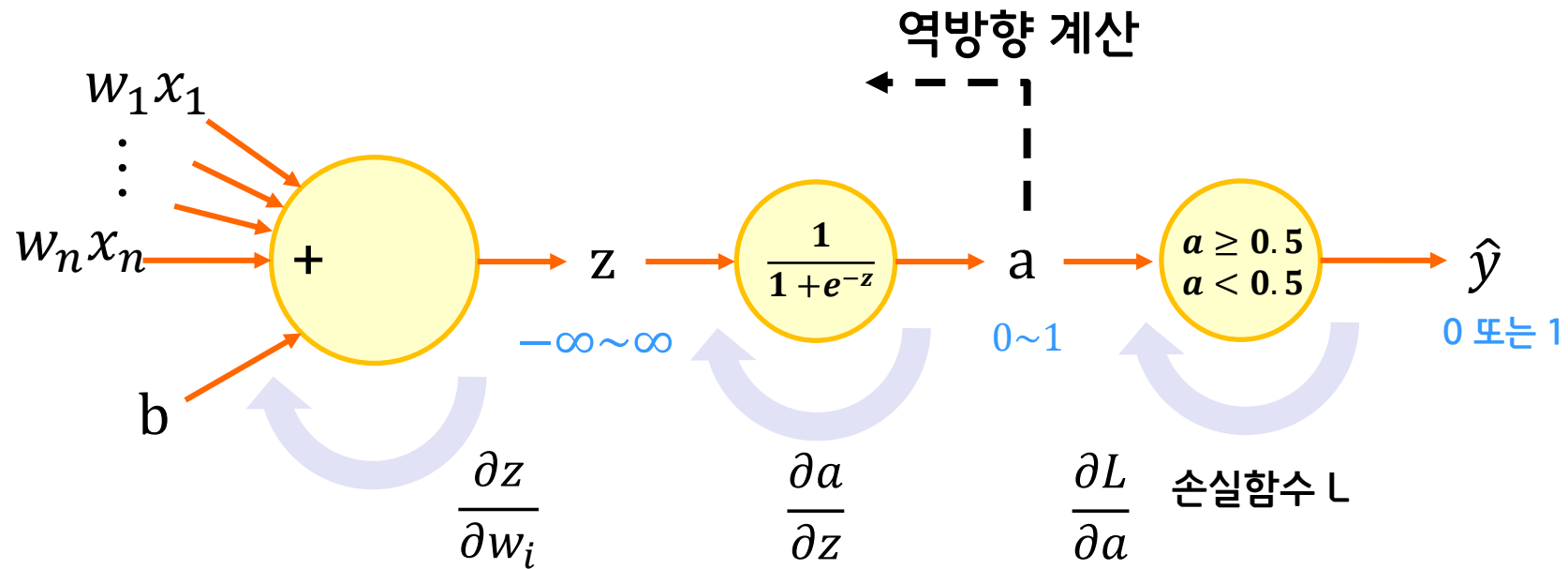
$$\hat{y} = w_1x_1 + w_2x_2 + b$$

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

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

특성이 여러개인 경우 => x n개

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



chain rule을 이용하여 각 단계의 미분 결과를 곱한다.

w에 대하여 미분

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

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

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

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

b에 대하여 미분

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

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

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

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

제곱 오차의 미분과 로지스틱 손실 함수의 미분은 \hat{y} 이 a 로 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 선형회귀 구현

2.4 시그모이드 함수

2.5 로지스틱 회귀 구현

소스참조

3. 신경망 시작하기

3.1 손실그래프와 스케일링

3.2 과대적합과 과소적합

3.3 규제방법 구현

3.4 교차 검증 구현

소스참조

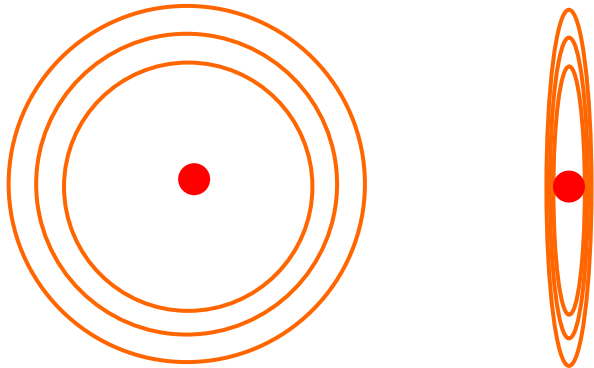
소스참조

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.



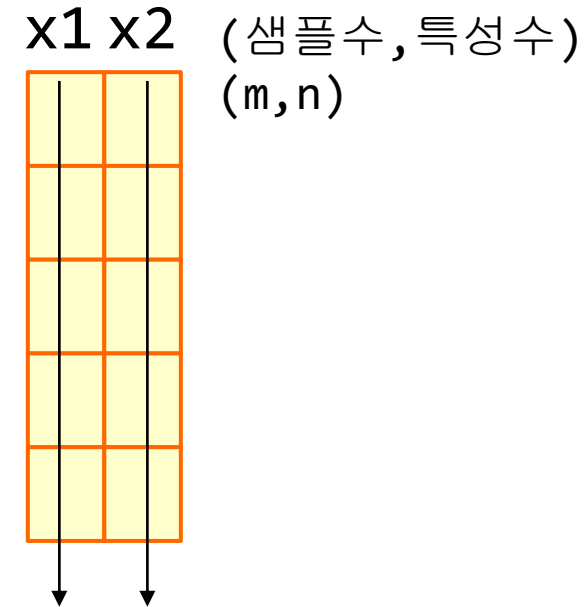
스케일 조정 : 평균 0, 표준편차 1 값이 된다.

$$Z = \frac{X - \mu}{s}$$

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



소스참조

3. 신경망 시작하기

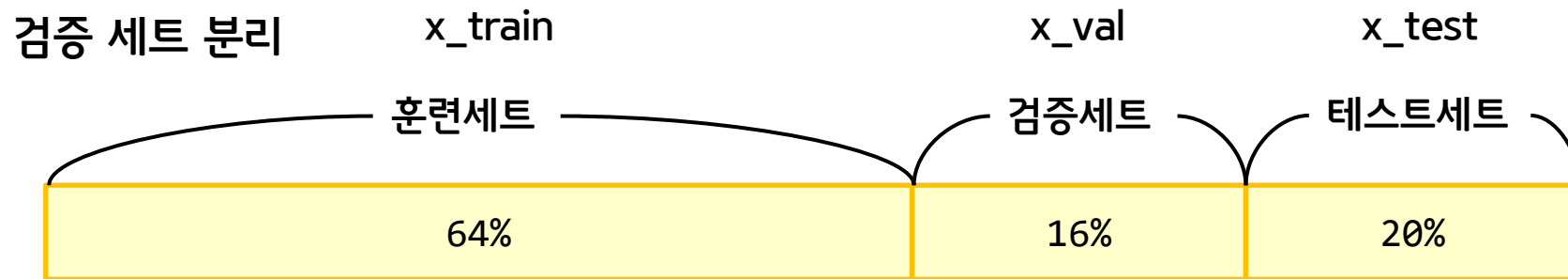
3.1 손실 그래프와 스케일링

3.2 과대적합과 과소적합

3.3 규제방법 구현

3.4 교차 검증 구현

"테스트 세트로 모델을 튜닝하면 실전에서 좋은 성능을 기대하기 어렵다"



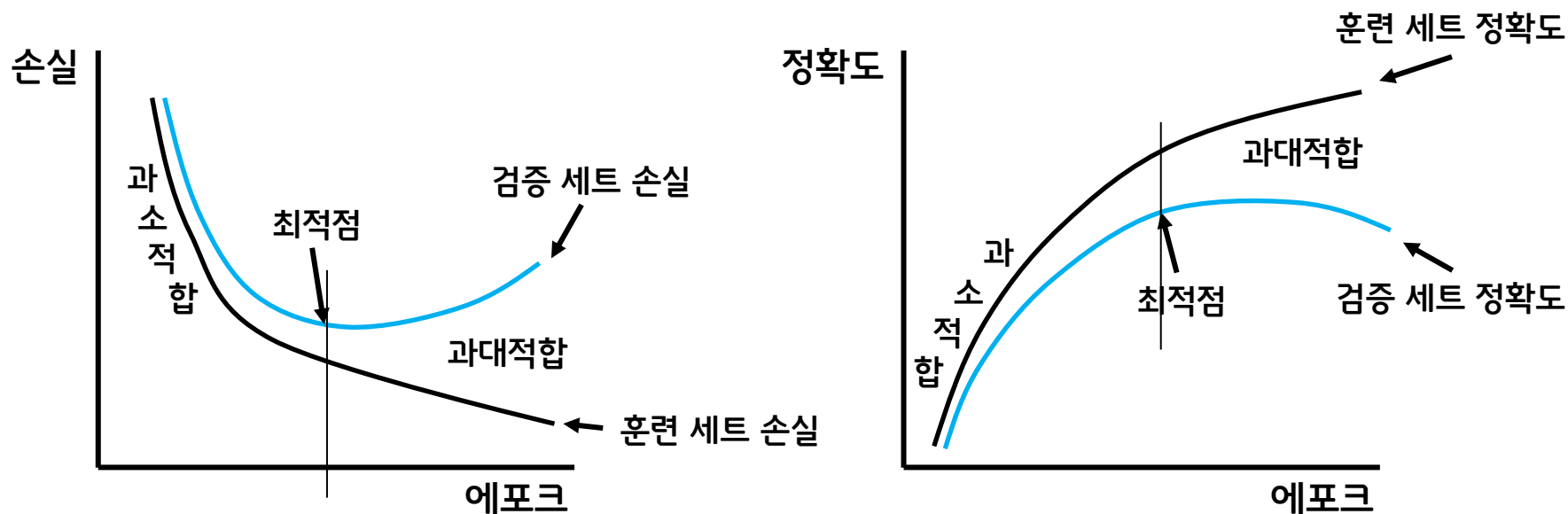
소스참조

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



소스참조

3. 신경망 시작하기

3.1 손실그래프와 스케일링

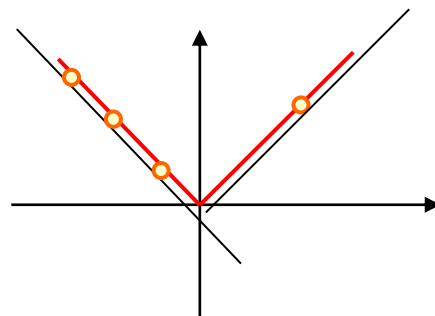
3.2 과대적합과 과소적합

3.3 규제방법 구현

3.4 교차 검증 구현

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

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



$$w_1x_1 + w_2x_2 + \dots + b$$

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

절대값을 미분하면 부호가 남는다.

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

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

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

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

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

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

파이썬으로 작성된 L1 규제 적용된 오차 역전파 구현

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

회귀 모델에 L1 규제를 추가한 것을 라쏘 모델이라 한다.

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

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

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

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

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

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

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.

소스참조

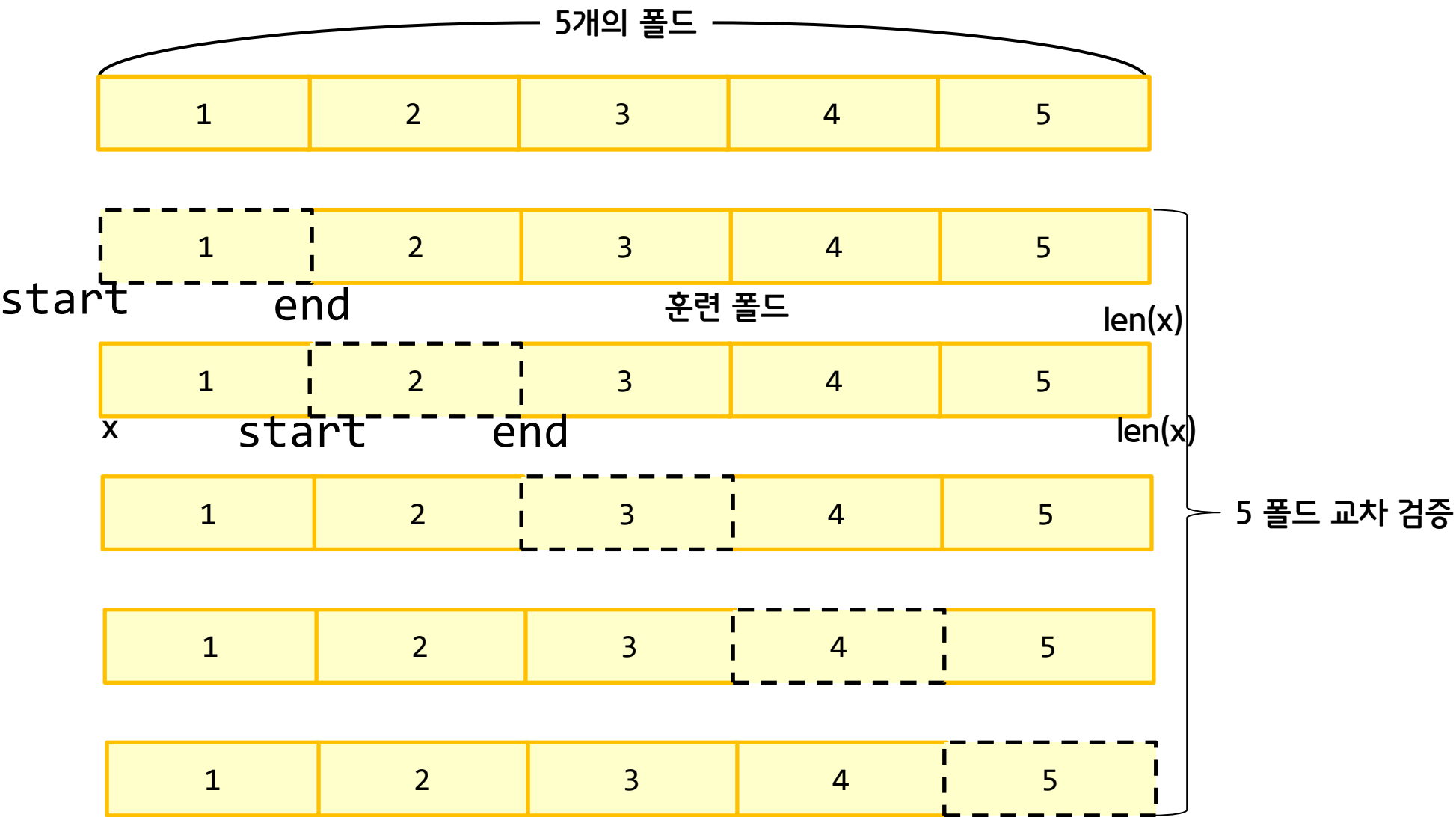
3. 신경망 시작하기

3.1 손실그래프와 스케일링

3.2 과대적합과 과소적합

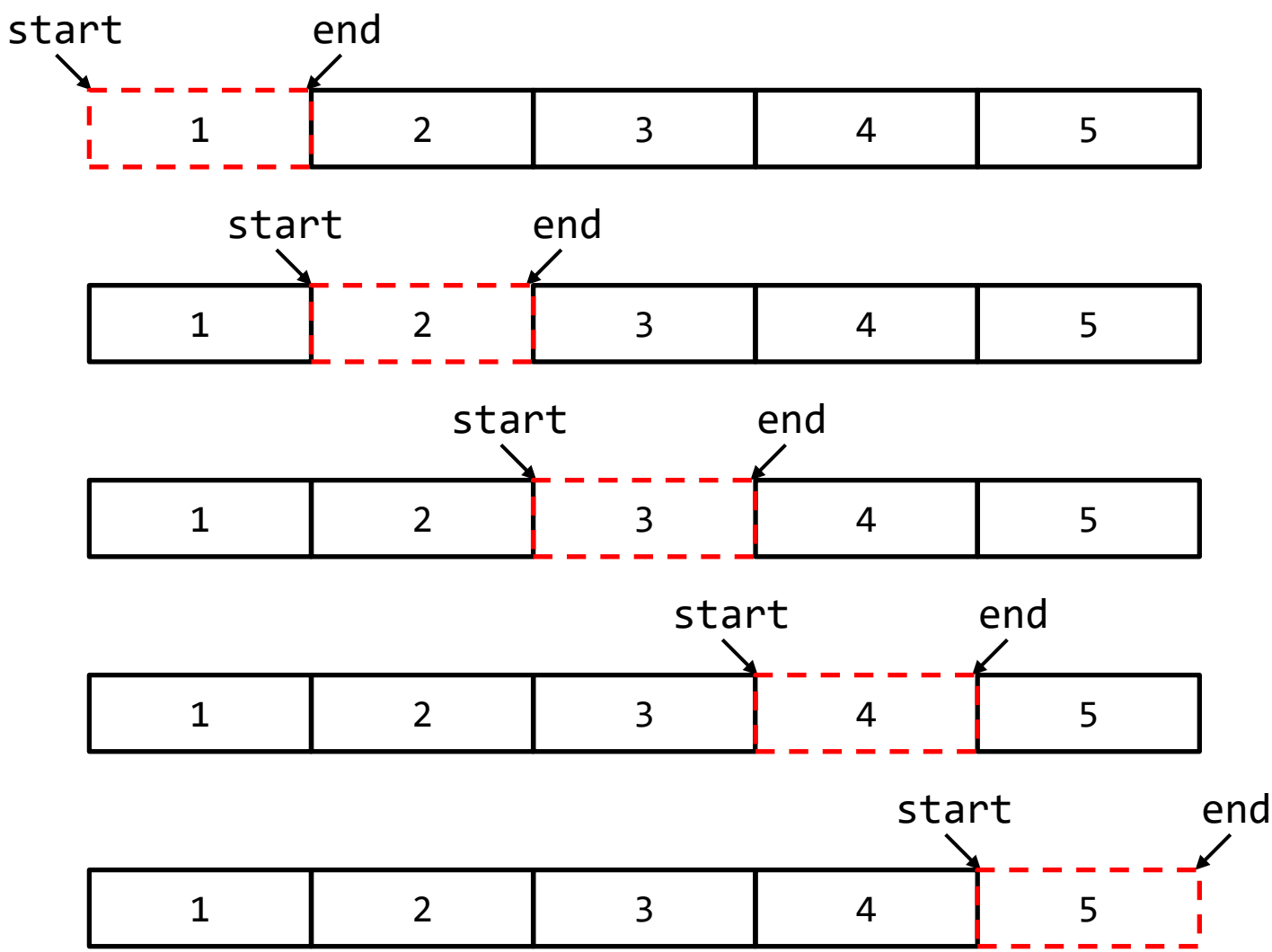
3.3 규제방법 구현

3.4 교차 검증 구현



교차 검증 과정

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



소스참조

4. 다층 신경망 이해

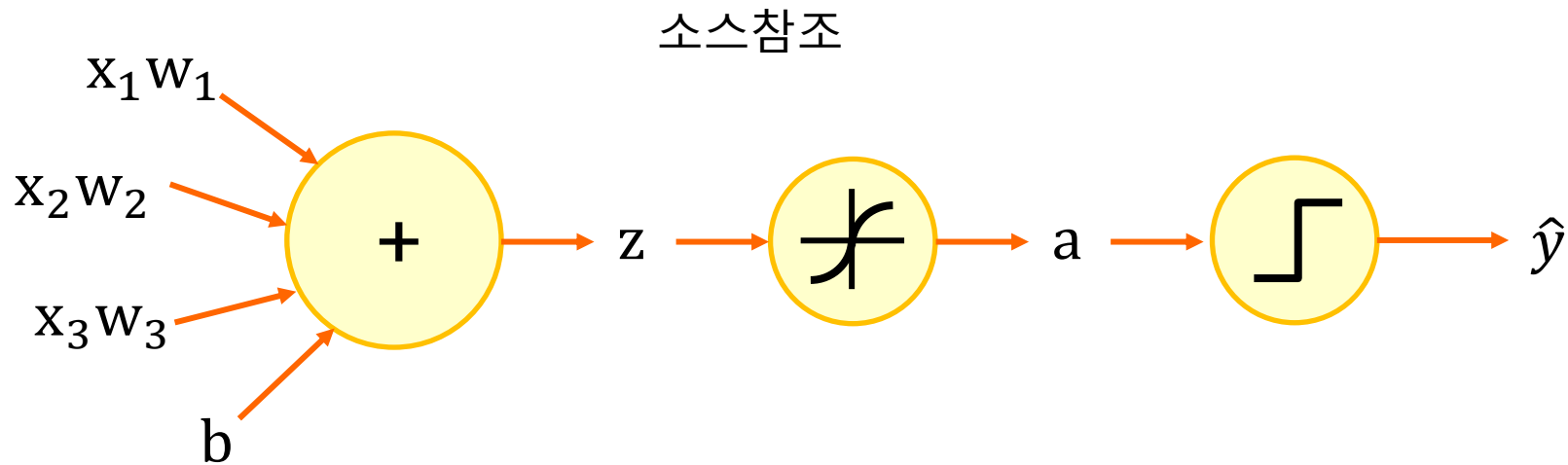
4.1 행렬 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 미니배치 경사 하강법 구현

4.5 다중분류 다층 신경망을 이해한다.

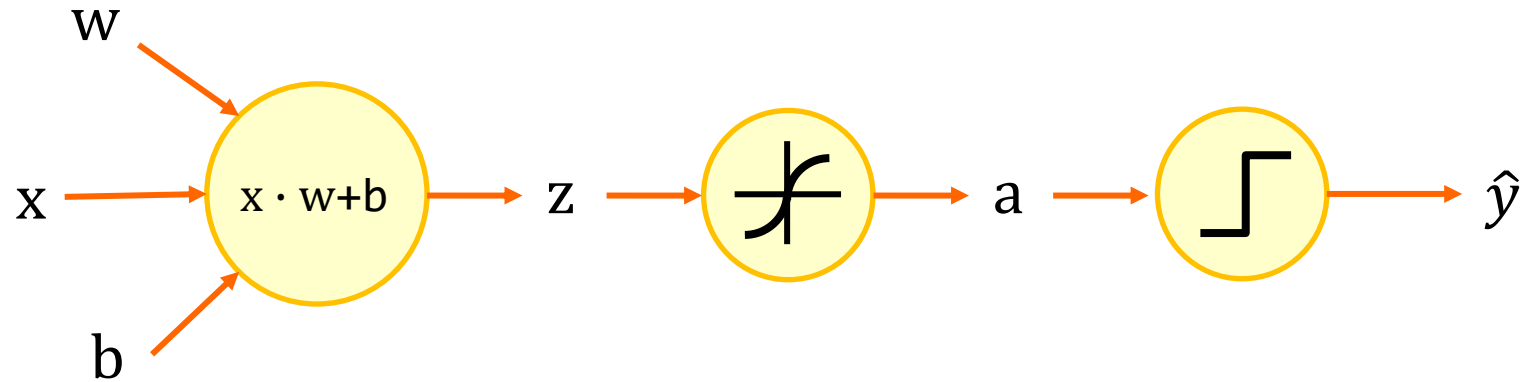


```
def forpass(self, x):
    z = np.sum(x * self.w) + self.b
    return z
```

넘파이의 원소별 곱셈

```

x = [x1, x1, ..., xn]
w = [w1, w, ..., wn]
x * w = [x1 * w1, x2 * w2, ..., xn * w1]
  
```



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)
```

a

1	2	3
---	---	---

b

4	5	6
---	---	---

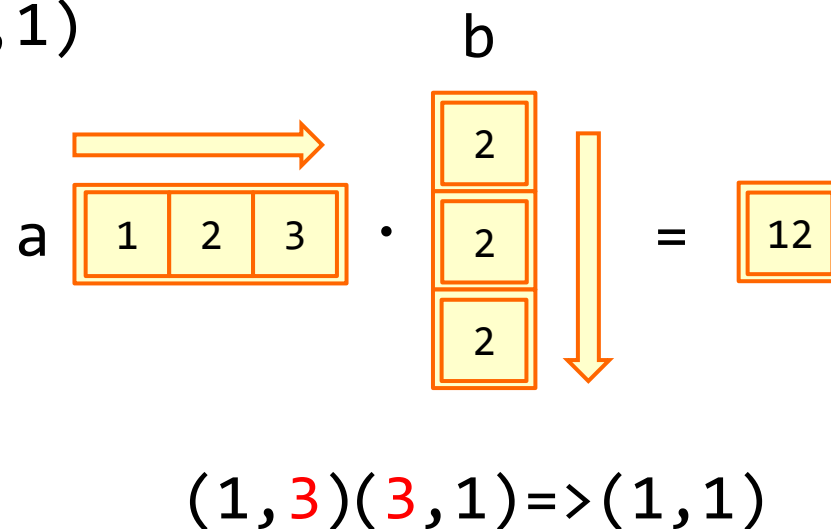
4	10	18
---	----	----

32

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

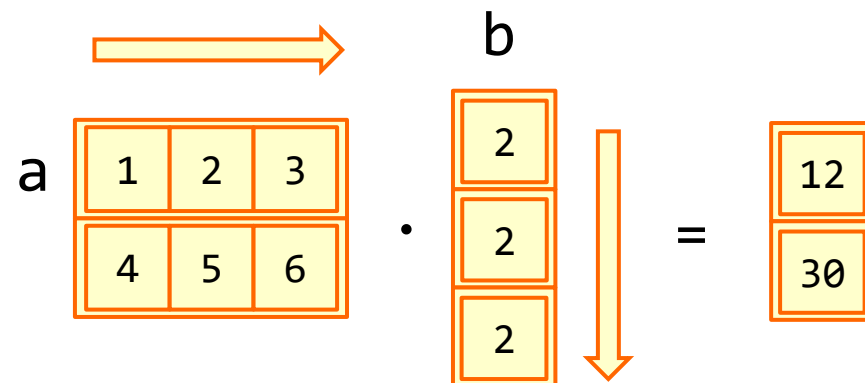
```

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]]
    
```



```
import numpy as np

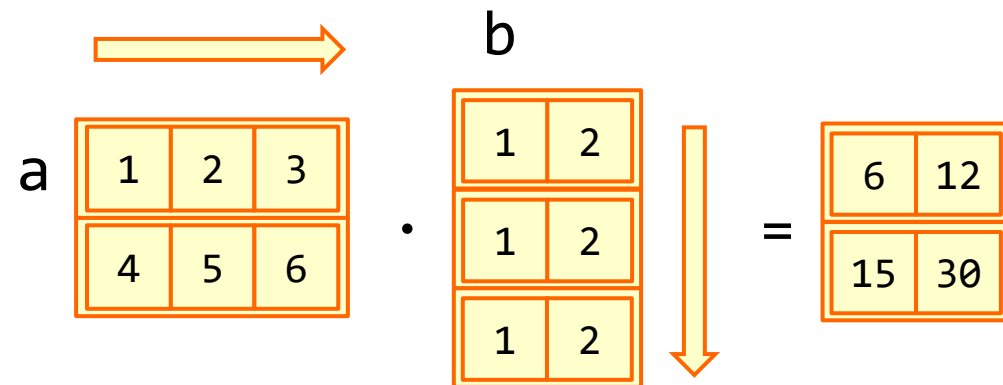
a = np.array([[1,2,3],
              [4,5,6]]) # (2,3)
print(a.shape)
b = np.array([[2],[2],[2]]) # (3,1)
print(b.shape)
c = np.dot(a,b) # (2,3)(3,1)
print(c.shape)
print(c.ndim)
print(c)
```



$$(2, 3)(3, 1) \Rightarrow (2, 1)$$

```
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)
```



$$(2, 3)(3, 2) \Rightarrow (2, 2)$$

1	2	3
---	---	---

(3,)

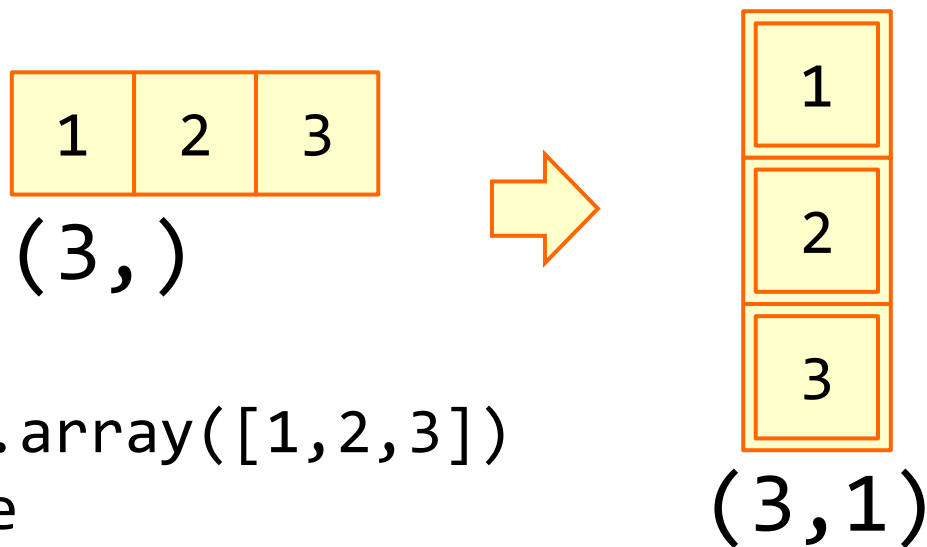
```
a = np.array([1,2,3])
```

```
a.shape
```

```
a.T
```

1	2	3
---	---	---

(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
```



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

```
a.shape
```

```
a = a.T
```

(2, 3)

(3, 2)

1	2	3
1	2	3

(2, 3)



1	1
2	2
3	3

(3, 2)

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

```
print(b)
```

1	2
1	2

(2, 2)



1	1
2	2

(2, 2)

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

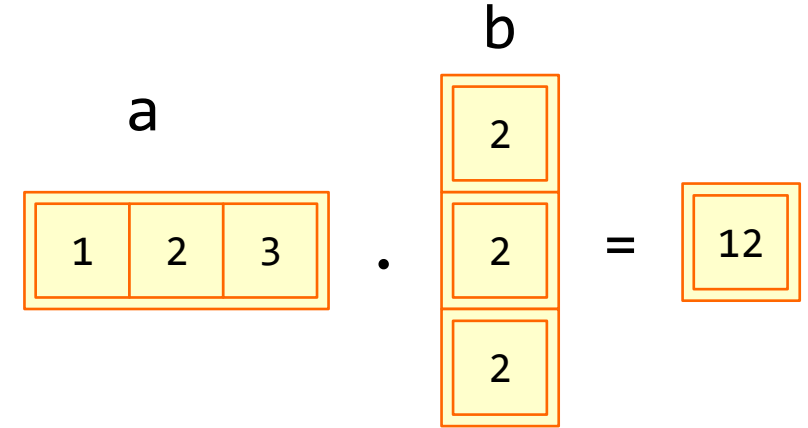
(2, 2)
~~(2, 2)~~

```
import numpy as np
```

$3 \times 4 = 4 \times 3$

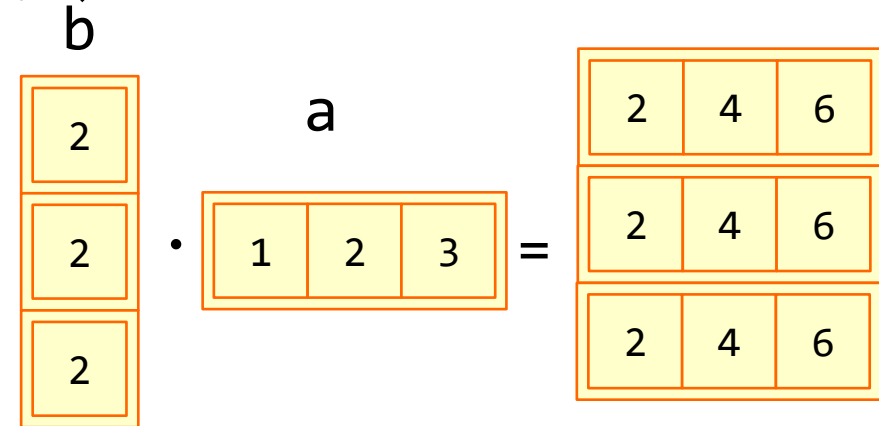
```
a = np.array([[1,2,3]]) # (1,3)
b = np.array([[2],[2],[2]]) # (3,1)
```

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



$(1, \textcolor{red}{3})(\textcolor{red}{3}, 1) \Rightarrow (1, 1)$

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



$(3, \textcolor{red}{1})(\textcolor{red}{1}, 3) \Rightarrow (3, 3)$

```
import numpy as np
```

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

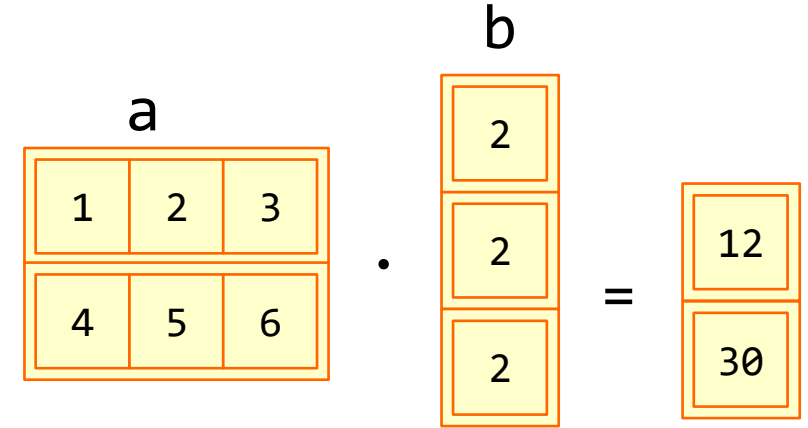
```
b = np.array([[2],[2],[2]]) # (3,1)
```

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

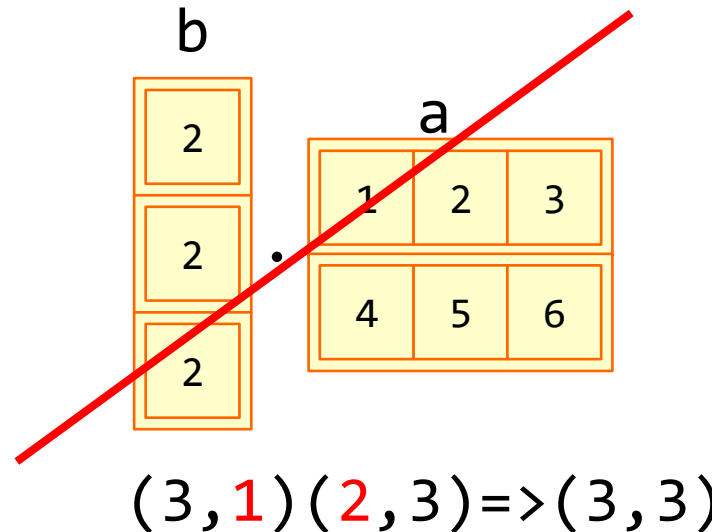
```
print(c.shape)
```

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

```
print(c.shape)
```



$(2, \textcolor{red}{3})(\textcolor{red}{3}, 1) \Rightarrow (2, 1)$



4. 다층 신경망 이해

4.1 행렬 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 미니배치 경사 하강법 구현

4.5 다중분류 다층 신경망을 이해한다.

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

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

$$(364, \textcolor{red}{30})(\textcolor{red}{30}, 1) \Rightarrow (364, 1)$$

행렬곱 가능과 곱 결과 크기

$$(m, \textcolor{red}{n}) \cdot (\textcolor{red}{n}, k) = (m, k)$$

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 곱셈으로 표현

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

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

$$Z = (364, 1)$$

$$A = \text{sigmoid}(Z)$$

$$L = \text{Loss}(A)/364$$

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

그레디언트 계산

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

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)

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

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

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

$$= X$$

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

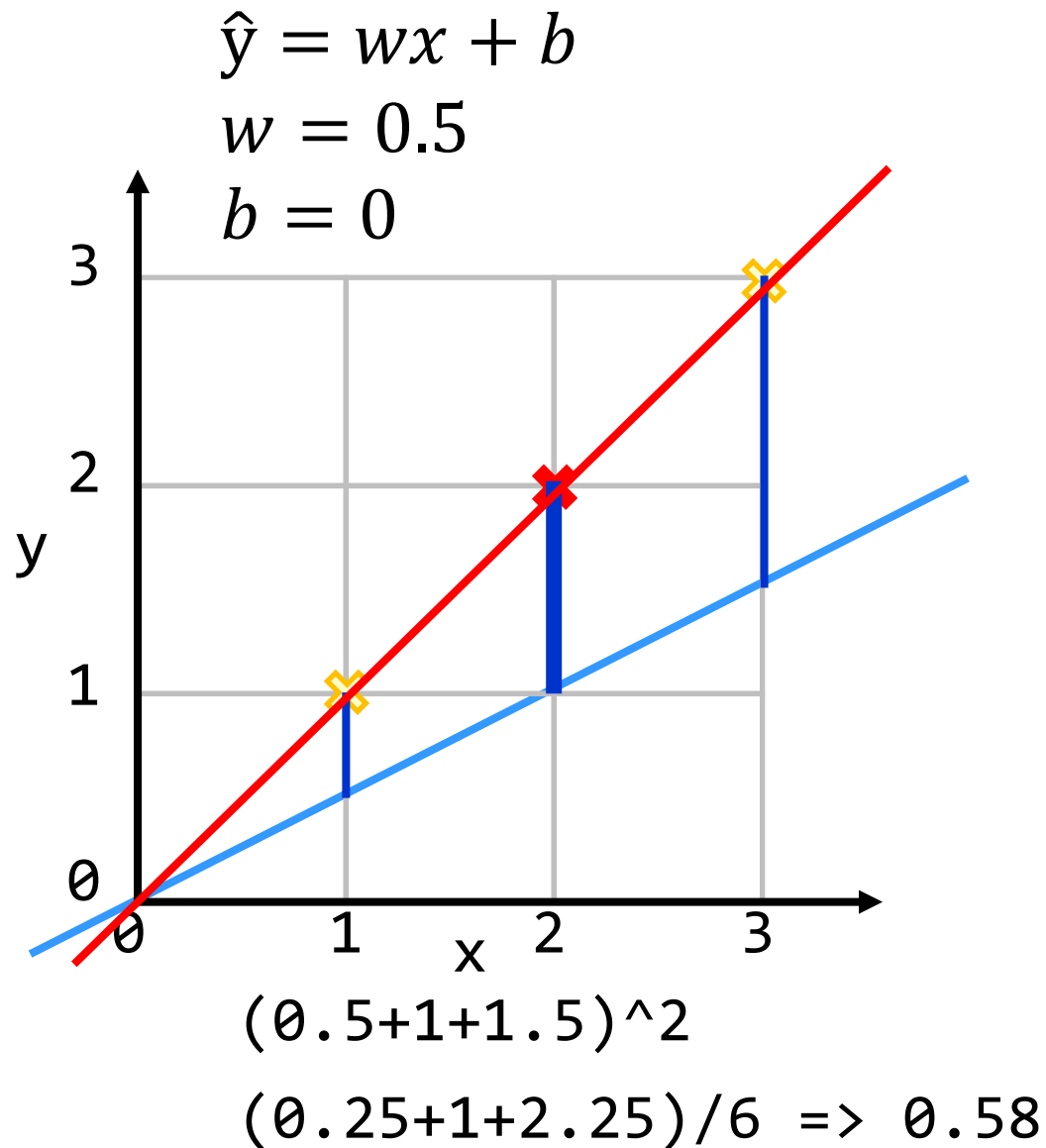
Chain Rule 사용

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

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

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

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



se

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

0.5

확률적 경사하강

mse

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

0.58

배치 경사하강

소스참조

4. 다층 신경망 이해

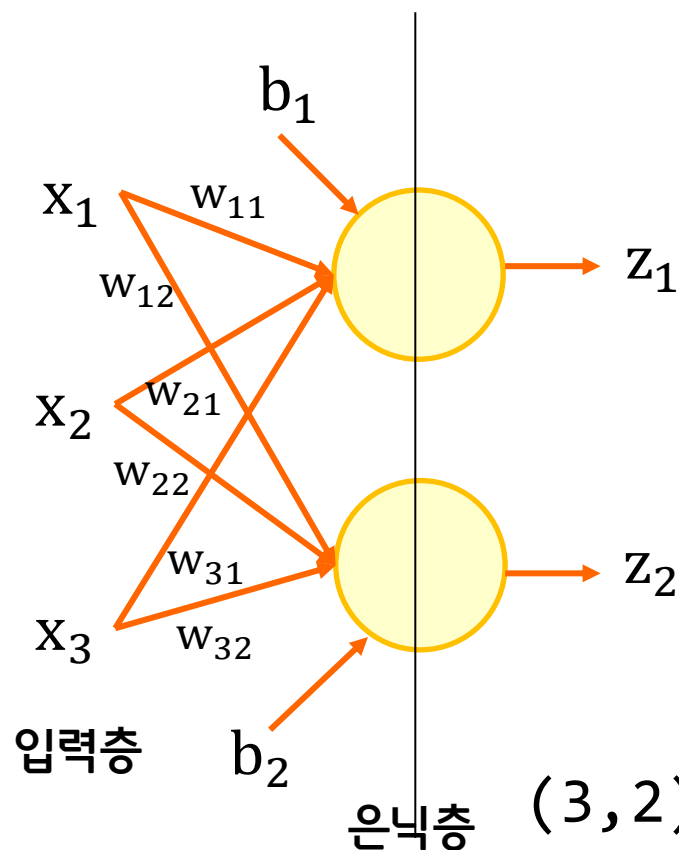
4.1 행렬 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 미니배치 경사 하강법 구현

4.5 다중분류 다층 신경망을 이해한다.



playground.tensorflow.org

$$XW_1 + b_1 = z_1$$

$$XW_2 + b_2 = z_2$$

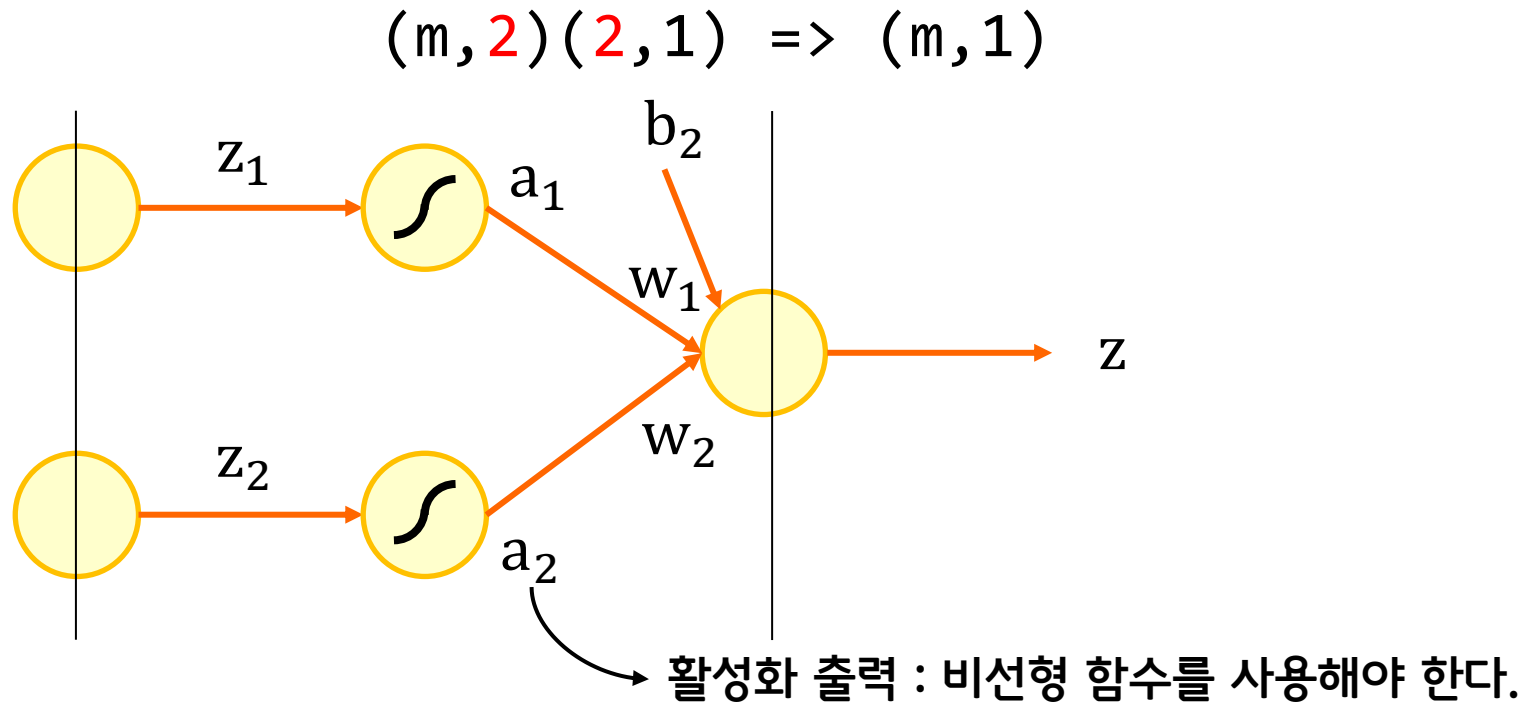
$$XW + b = Z$$

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

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

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

$$(m, n)(n, 2) \Rightarrow (m, 2)$$

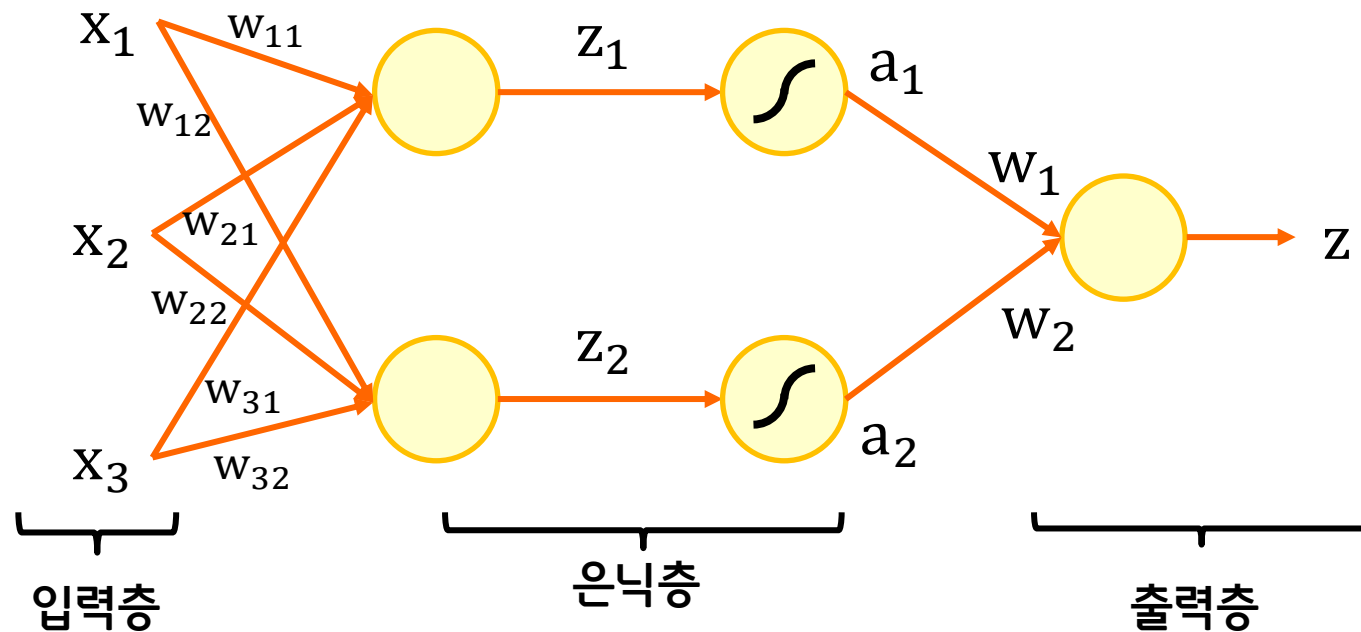


$$a_1 w_1 + a_2 w_2 + b_2 = z$$

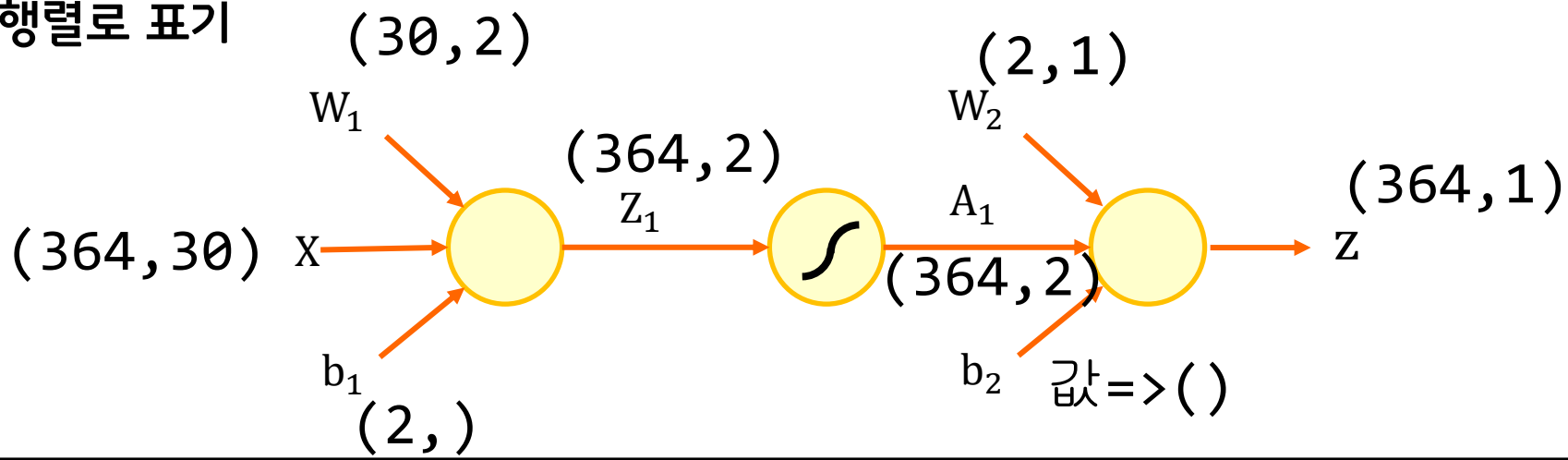
$$\begin{bmatrix} a_1 & a_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + b_2 = z$$

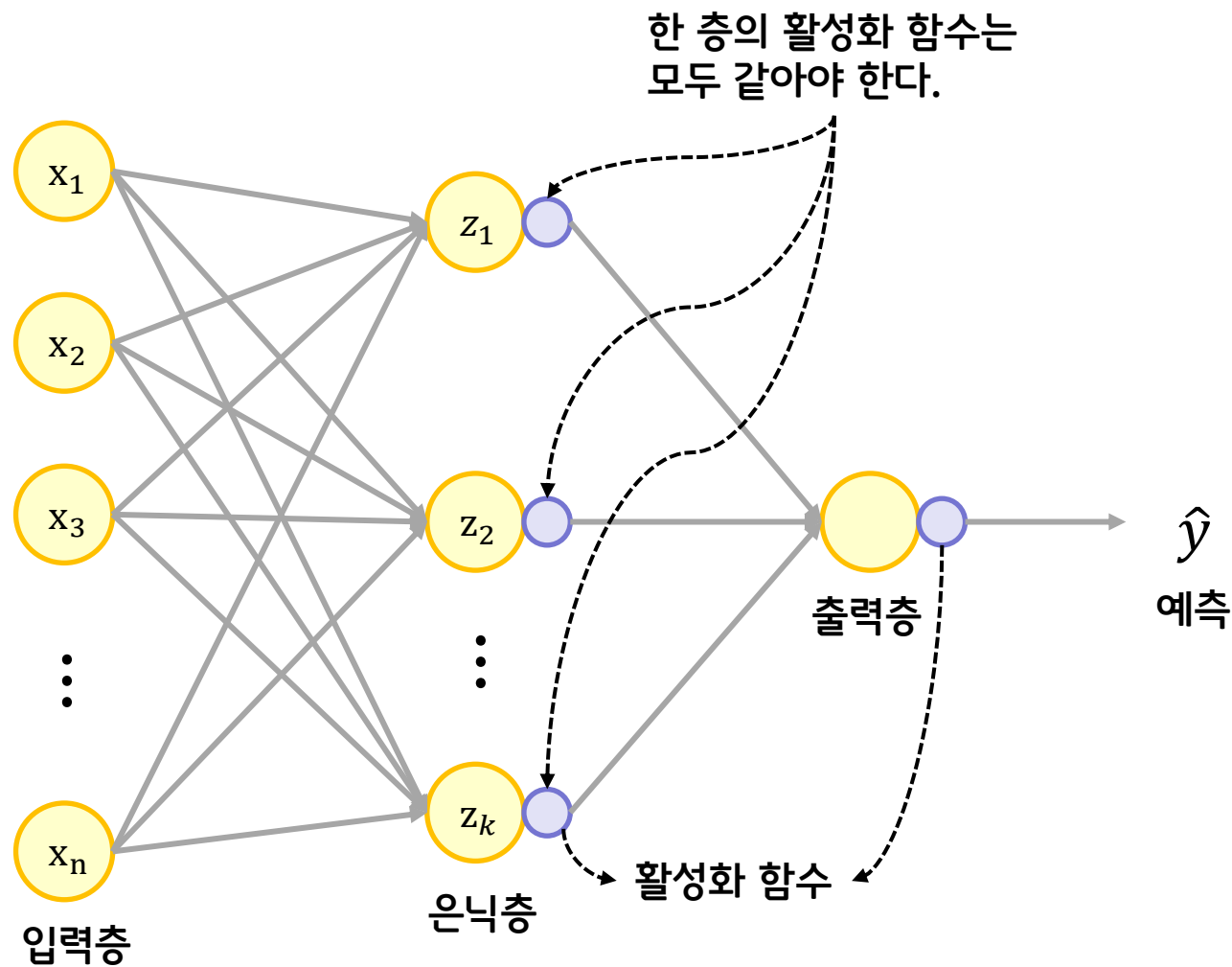
$$(m, 2)(2, 1) \Rightarrow (m, 1)$$

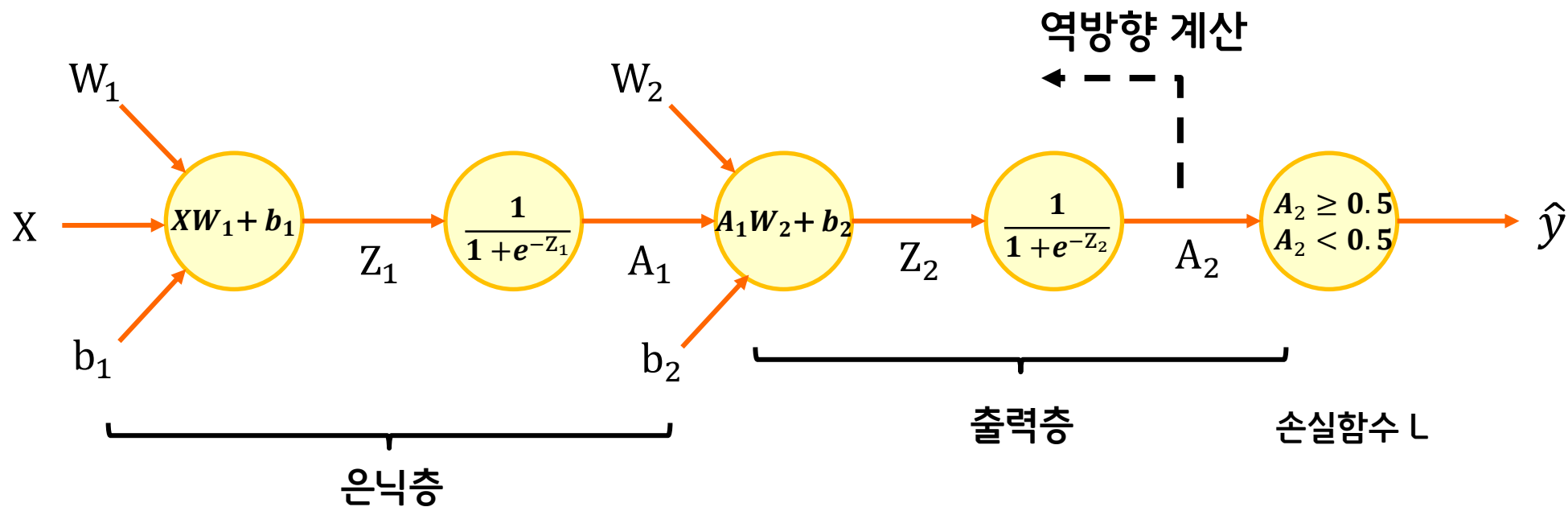
$$A_1 W_2 + b_2 = z_2$$



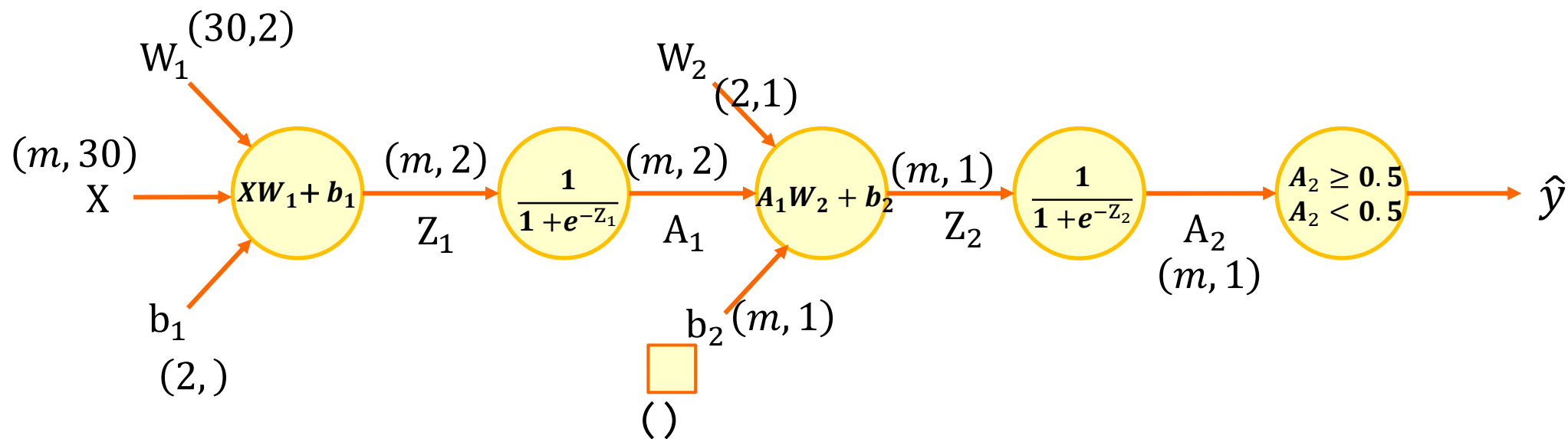
행렬로 표기





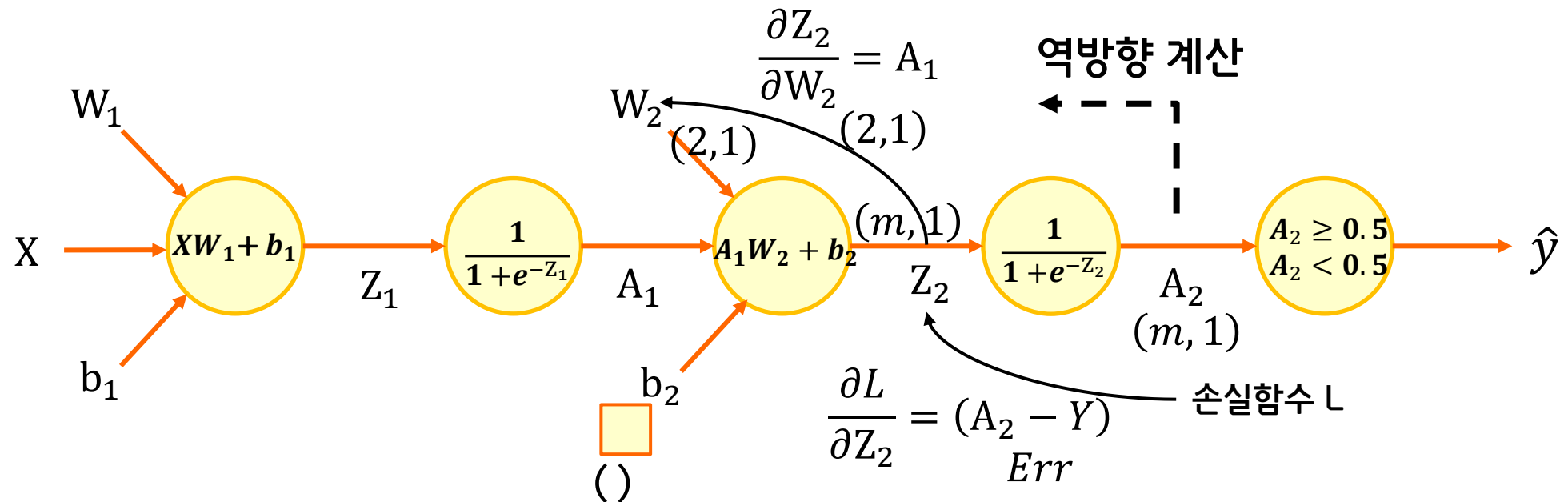


정방향 연산



가중치 W_2 대하여 손실 함수를 미분한다. $\frac{\partial L}{\partial W_2} = (A_1^T \cdot Err)/m$
 $(2, m)(m, 1) \Rightarrow (2, 1)$

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



편향 b_2 대하여 손실 함수를 미분한다.

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

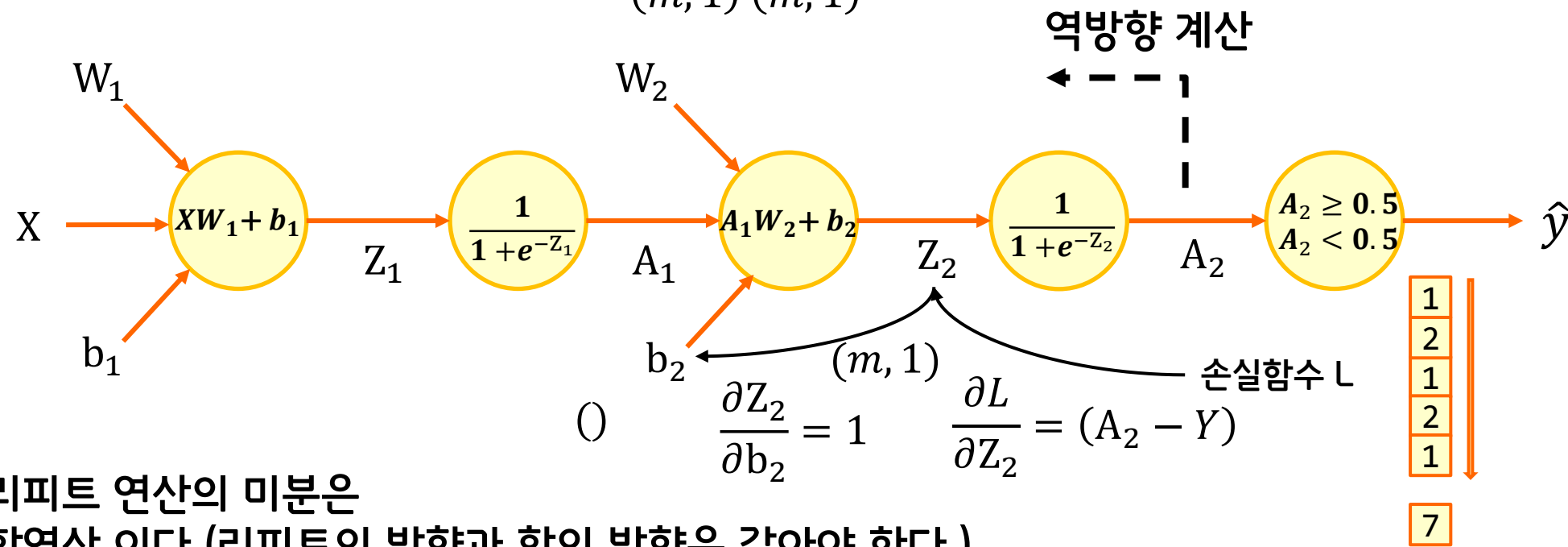
1
2
3
4
5

3
3
3
3
3

4
5
6
7
8

소스참조

$(m, 1) \quad (m, 1)$



가중치 W_2 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial Z_2} = (A_2 - Y) = \left[\begin{array}{c} 0.7 \\ 0.3 \\ \vdots \\ 0.6 \end{array} \right] \quad m\text{개}$$

$$\frac{\partial Z_2}{\partial W_2} = A_1 = \left[\begin{array}{cc} -1.37 & 0.96 \\ & \vdots \\ 2.10 & -0.17 \end{array} \right] \quad m\text{개}$$

첫 번째 뉴런의 활성화 출력 \swarrow
 \nwarrow 첫 번째 뉴런의 활성화 출력

가중치 W_2 대하여 손실 함수를 미분한다.

행렬의 구성을 보면 A_1 의 크기는 $(m,2)$ 이고 $(A_2 - Y)$ 의 크기는 $(m,1)$ 이므로 A_1 을 전치하여 $(A_2 - Y)$ 와 곱해야 한다.

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

$(m, k)(m, 1)$
 $(k, m)(m, 1) \Rightarrow (k, 1)$

m 개
↑
타겟과 예측의 차이
↓
그레디언트의 총 합

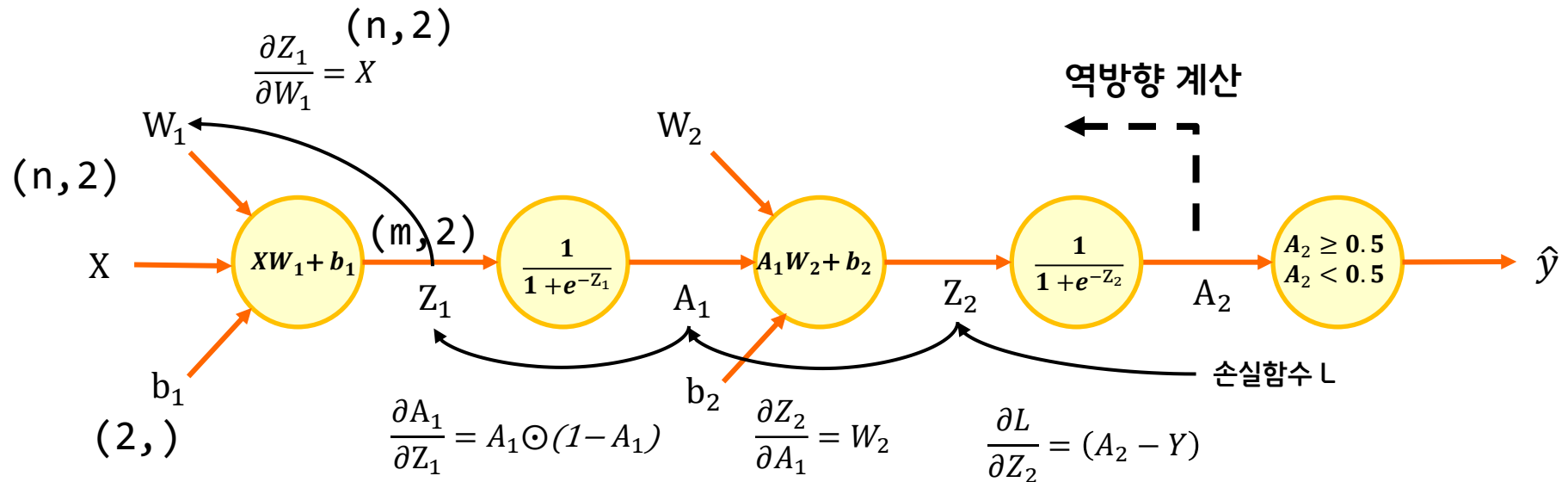
가중치 W_1 대하여 손실 함수를 미분한다.

$$Err \cdot W_2^T$$

err_to_hidden

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

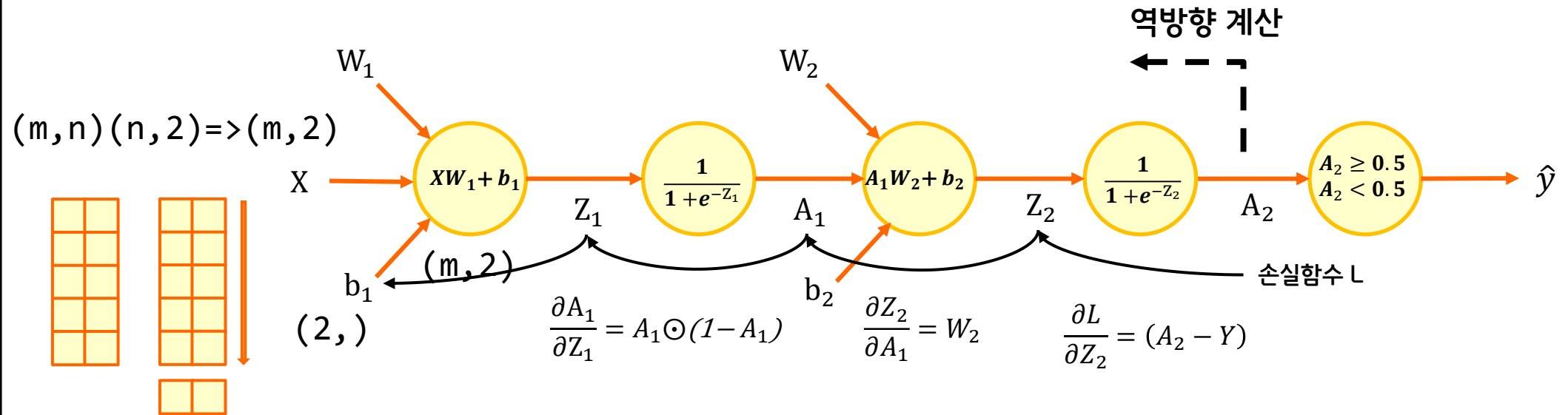
$(m, 1) (1, 2) \Rightarrow (m, 2)$
 $(n, m) (m, 2) \Rightarrow (n, 2)$



편향 b_1 대하여 손실 함수를 미분한다.

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_2} \frac{\partial Z_2}{\partial A_1} \frac{\partial A_1}{\partial Z_1} \frac{\partial Z_1}{\partial b_1} = \text{np.sum(err_to_hidden, axis=0)/m;}$$

$$(m, 2) \Rightarrow (2,)$$



4. 다층 신경망 이해

4.1 행렬 연산

4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 미니배치 경사 하강법 구현

4.5 다중분류 다층 신경망을 이해한다.

미니배치 경사 하강법이란?

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_size`를 인자로 받아서 멤버 함수에 저장한다.

소스참조

4. 다층 신경망 이해

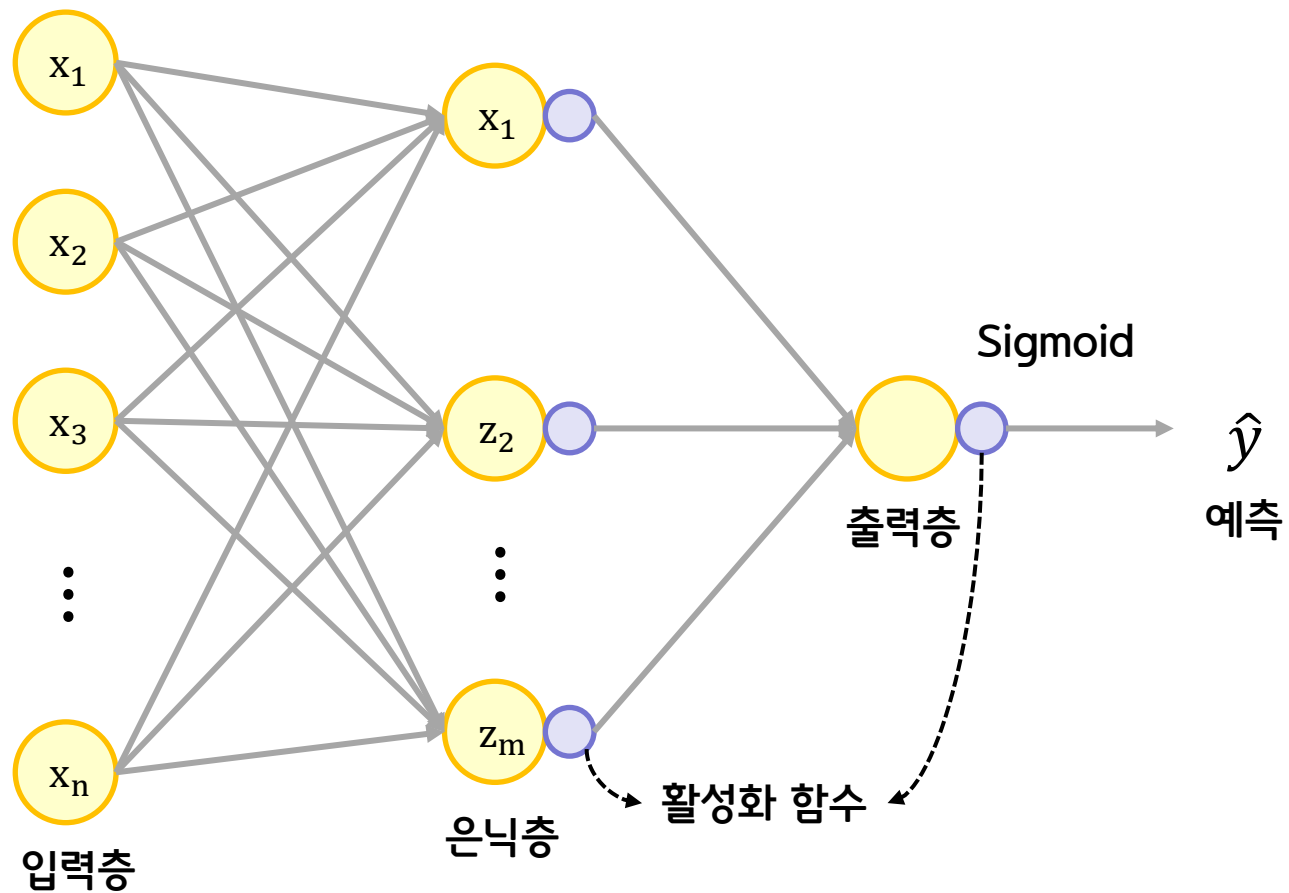
4.1 행렬 연산

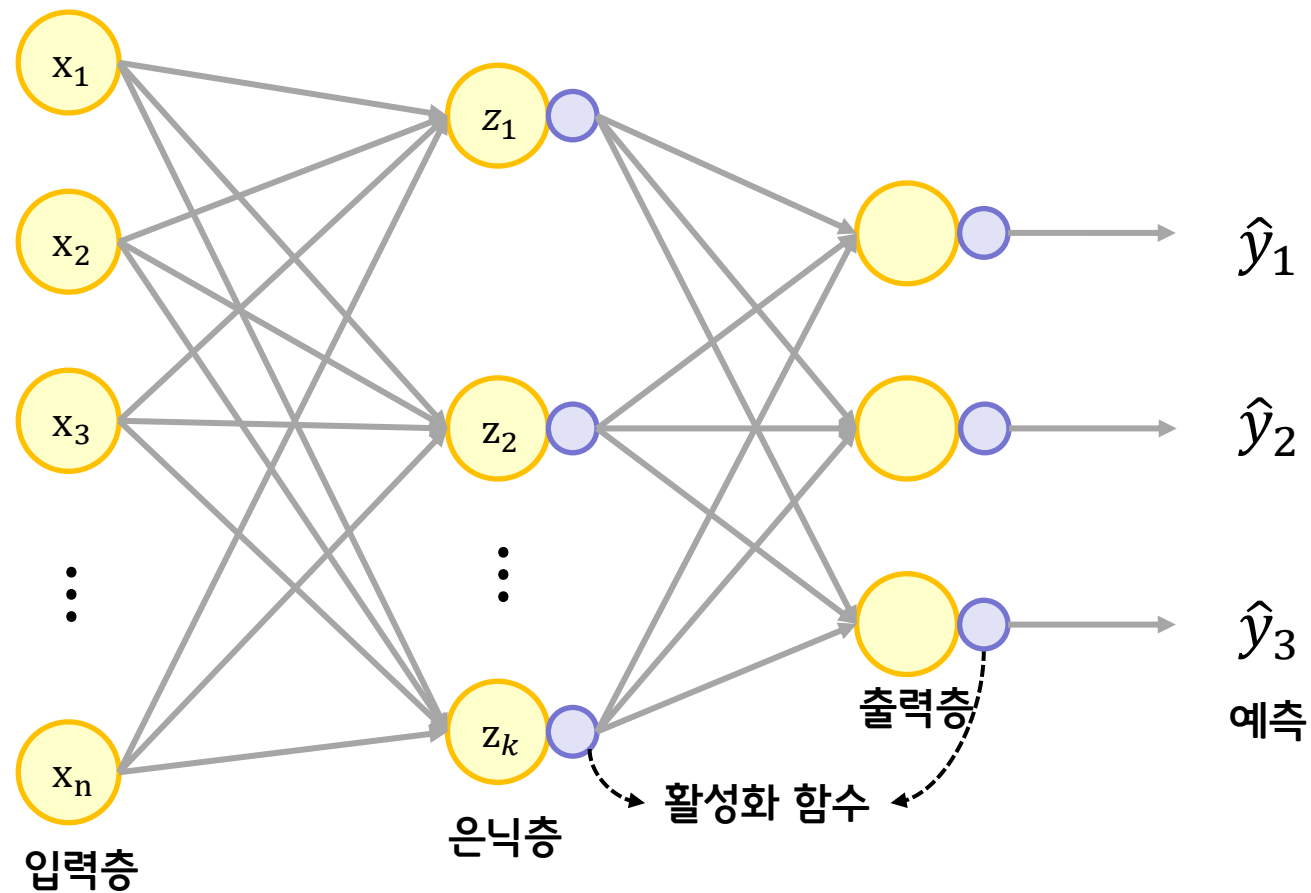
4.2 배치경사 하강법 구현

4.3 2개의 층을 가진 신경망 구현

4.4 미니배치 경사 하강법 구현

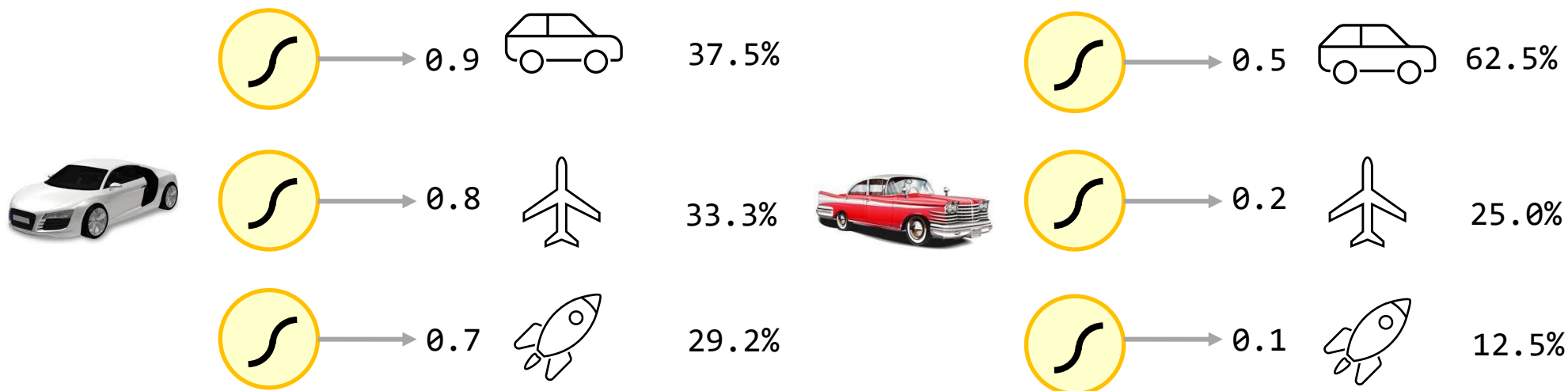
4.5 다중분류 다층 신경망을 이해한다.





활성화 출력의 합이 1이 아니면 비교하기 어렵다.

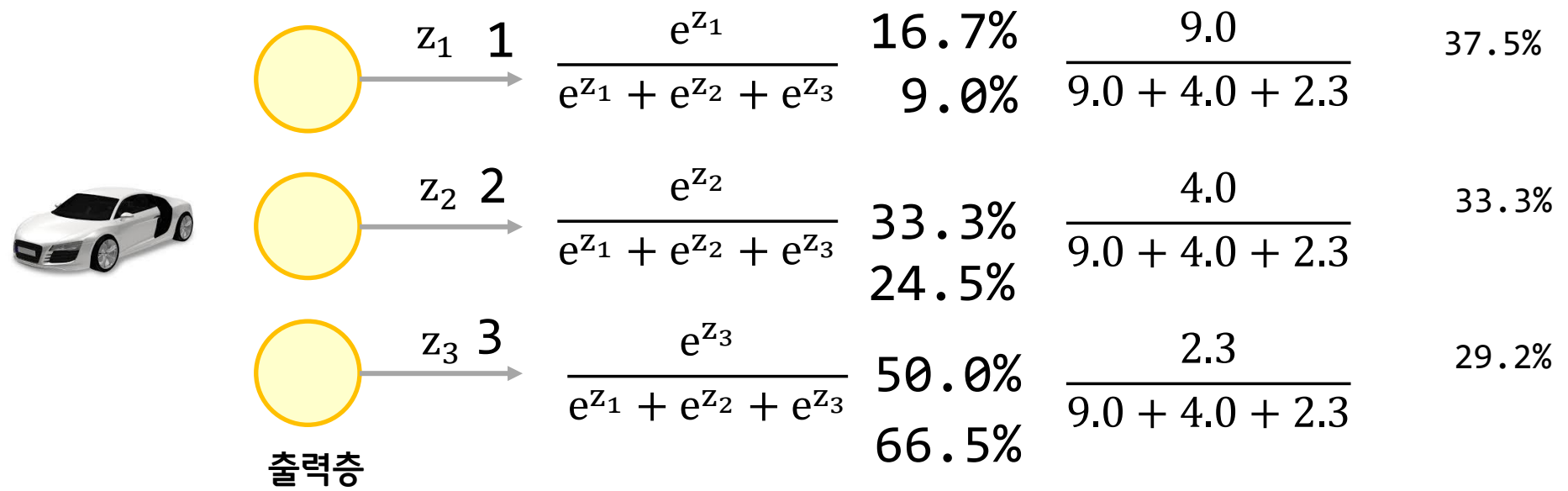
소프트맥스 함수를 적용해 출력 강도를 정규화 한다.



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

$$\frac{e^{z_i}}{e^{z_1} + e^{z_2} + e^{z_3}}$$
$$2/(1+2+3)$$
$$\frac{e^1}{e^1 + e^2 + e^3}$$
$$\frac{e^1}{30.19}$$

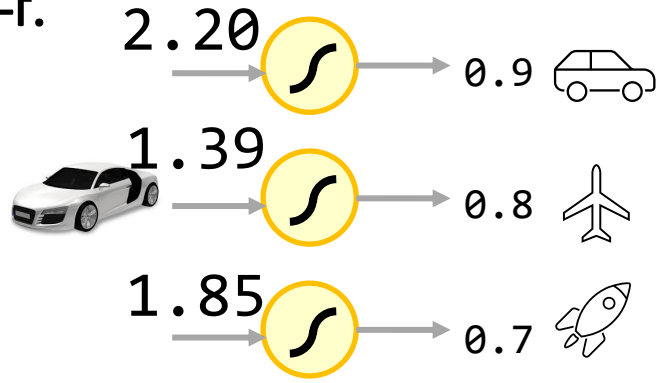
$$2.718/(2.718+7.389+20.085)$$



소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z에 대해 정리하면

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



$$z_1 = -\log\left(\frac{1}{0.9} - 1\right) = 2.20$$

$$z_2 = -\log\left(\frac{1}{0.8} - 1\right) = 1.39$$

$$z_2 = -\log\left(\frac{1}{0.7} - 1\right) = 0.85$$

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

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

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

자동차 59%

비행기 26%

로켓 15%

37.5%

15.380

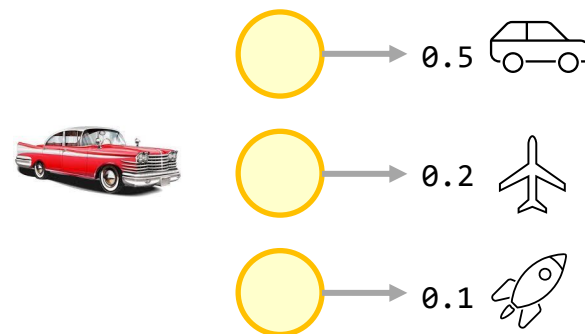
33.3%

29.2%

소프트맥스 함수를 적용해 출력 강도를 정규화 한다.

시그 모이드 함수를 z 에 대해 정리하면

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



$$z_1 = -\log\left(\frac{1}{0.5} - 1\right) = 0.00$$

$$z_2 = -\log\left(\frac{1}{0.2} - 1\right) = -1.39$$

$$z_3 = -\log\left(\frac{1}{0.1} - 1\right) = -2.20$$

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

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

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

자동차 74%

비행기 18%

로켓 8%

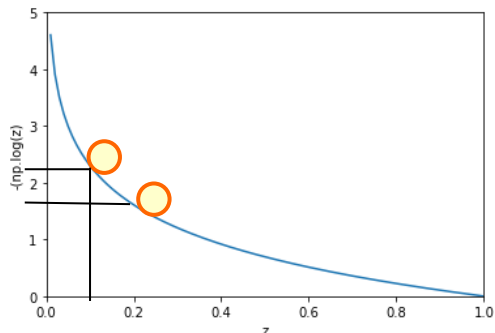
62.5%

25.0%

12.5%

소스참조


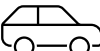


바이너리 크로스 엔트로피 손실 함수
카테고리컬 크로스 엔트로피 손실 함수
크로스 엔트로피 손실 함수



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

$-\log(0.2)$

$y=1$

	a	y	
	0.7	0	
	0.1	0	
	0.2	1	

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

크로스 엔트로피 손실 함수 미분

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

$$\frac{\partial a_1}{\partial z_1} = a_1(1-a_1) \quad \frac{\partial a_2}{\partial z_1} = -a_2 a_1 \quad \frac{\partial a_3}{\partial z_1} = -a_3 a_1$$

$$\begin{aligned} \frac{\partial L}{\partial z_1} &= \left(-\frac{y_1}{a_1}\right) a_1(1-a_1) + \left(-\frac{y_2}{a_2}\right) (-a_2 a_1) + \left(-\frac{y_3}{a_3}\right) (-a_3 a_1) \\ &= (a_1 - y_1) \end{aligned}$$

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

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

5. 주요 케라스 문법

5.1 케라스에 대하여

5.2 순차 모델

5.3 함수형 API

5.4 훈련 및 평가

5.5 사용자 정의 레이어 및 모델

5.6 저장 및 직렬화

5.7 전처리 레이어

소스참조

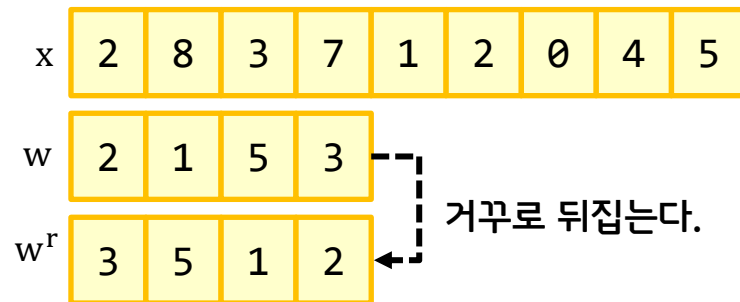
6. 합성곱 신경망 (CNN) 이해

6.1 합성곱 연산

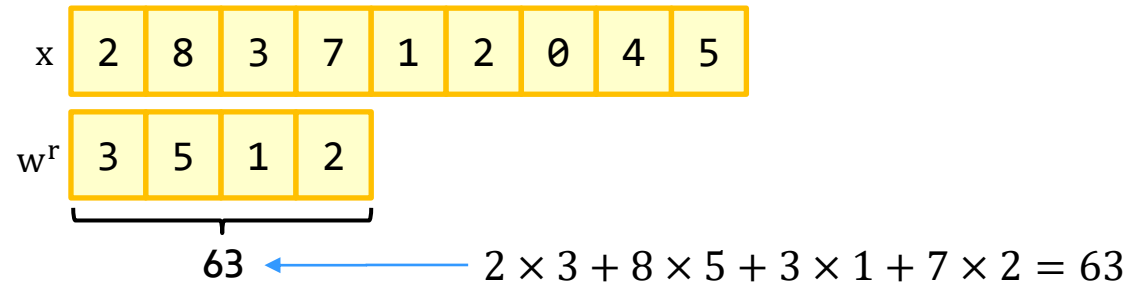
6.2 합성곱 신경망 구현

6.3 케라스로 합성곱 신경망 구현

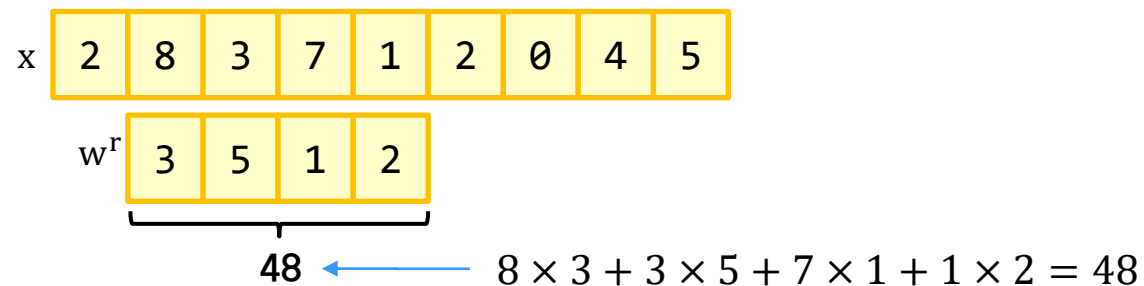
배열 하나 선택해 뒤집기



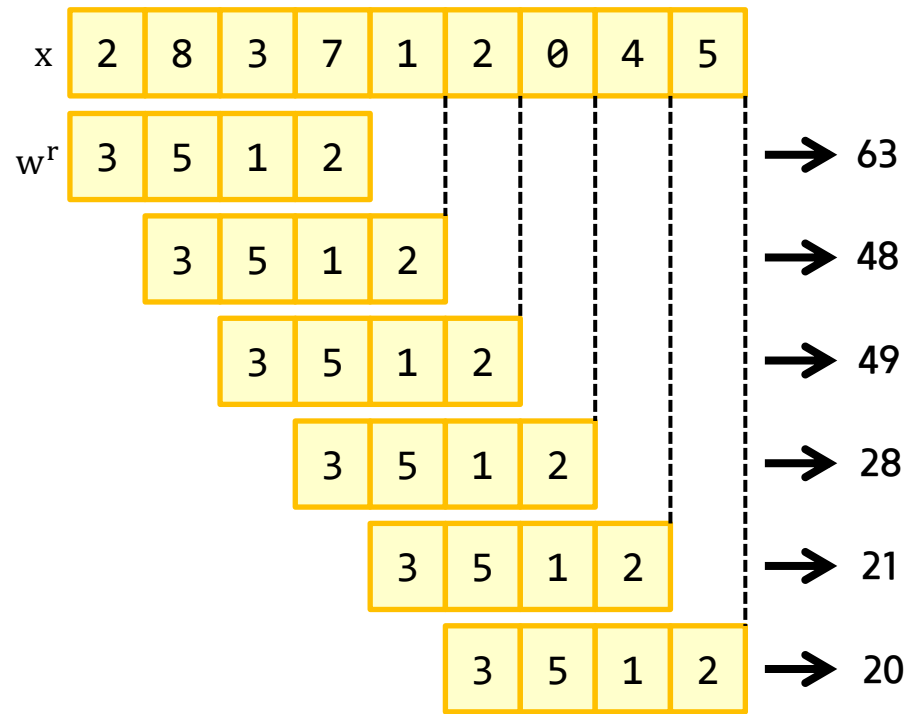
첫 번째 합성곱



두 번째 합성곱



전체 합성곱



$N-F+1 = \text{Out}$

$9-4+1 = 6$

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]

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

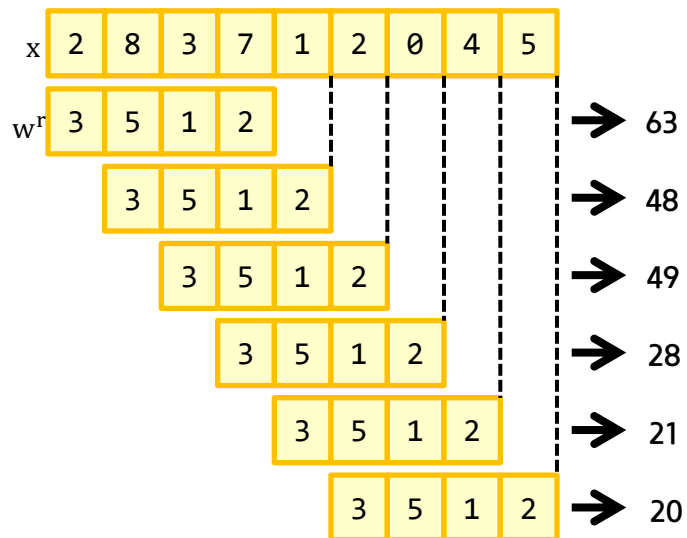
싸이파이로 합성곱 수행

```
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.

합성곱(convolve)



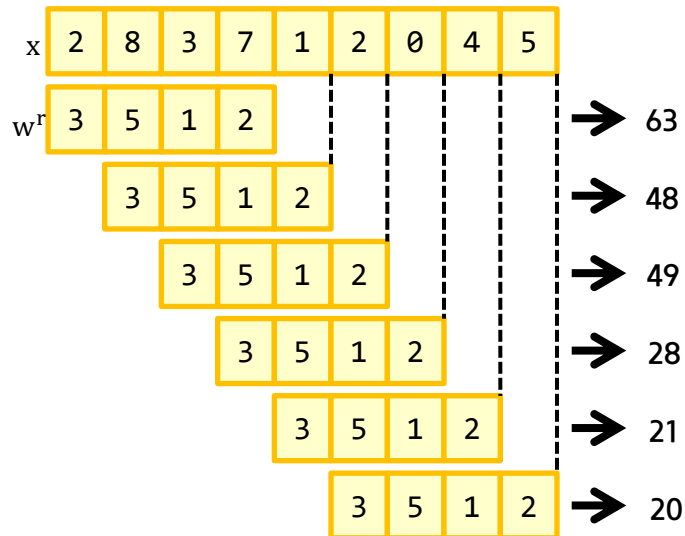
싸이파이로 교차상관 수행

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

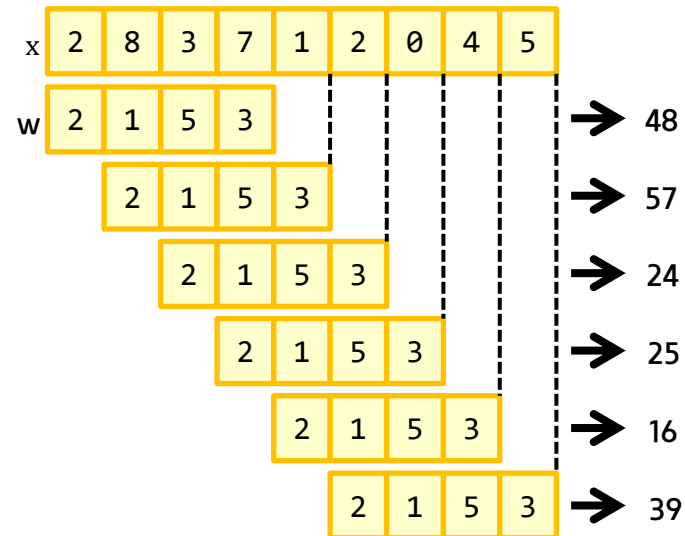
```
array([48, 57, 24, 25, 16, 39])
```

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

합성곱(convolve)

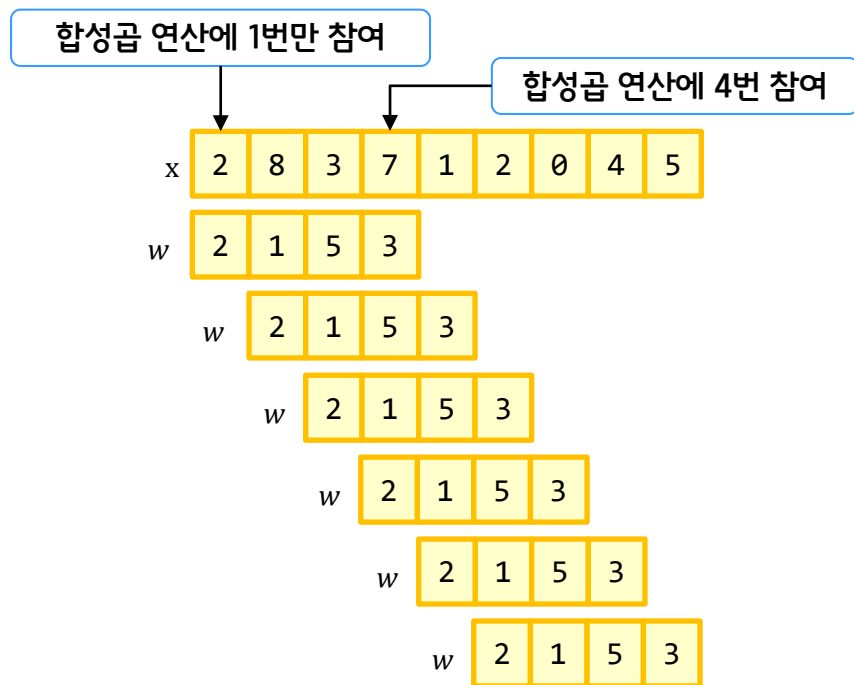


교차상관(correlate)



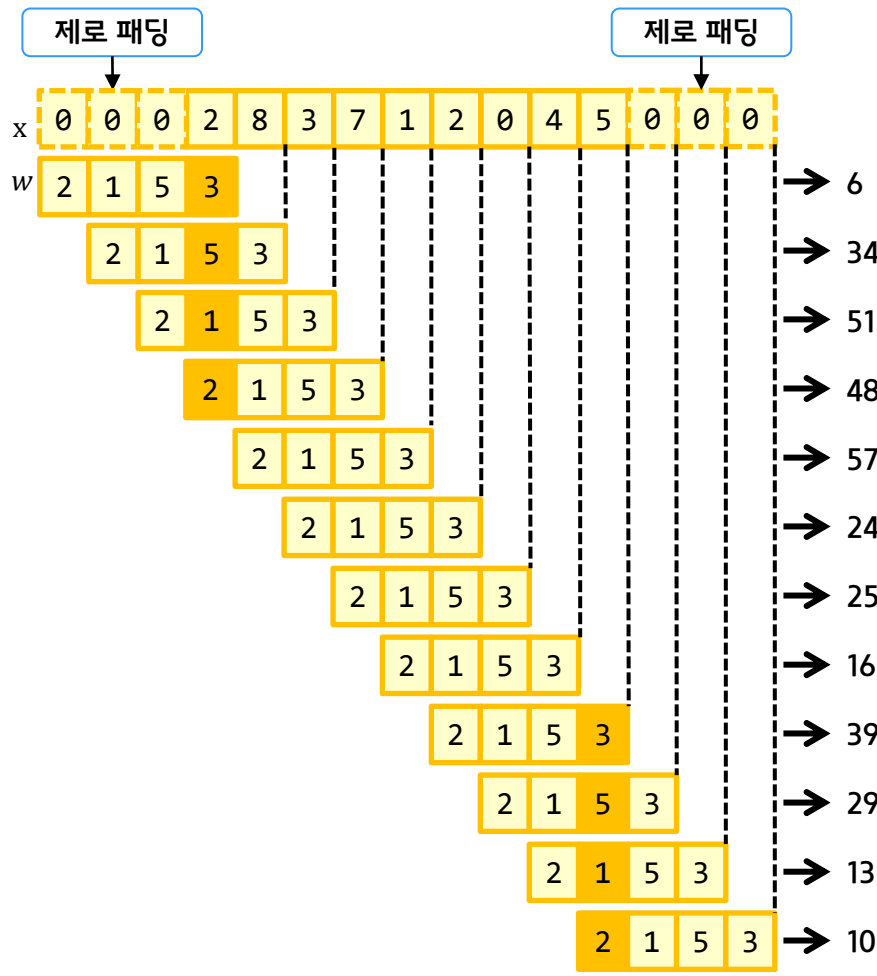
패딩과 스트라이드 이해

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')
```

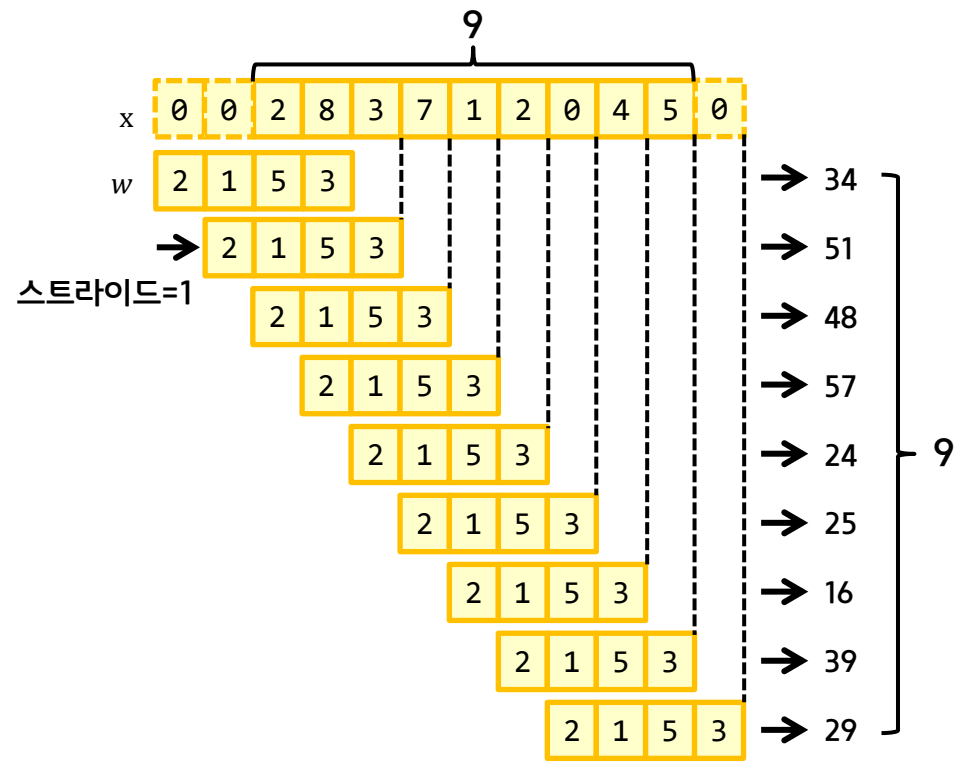
array([6, 34, 51, 48, 57, 24, 25, 16, 39, 29, 13, 10])

패딩과 스트라이드 이해

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

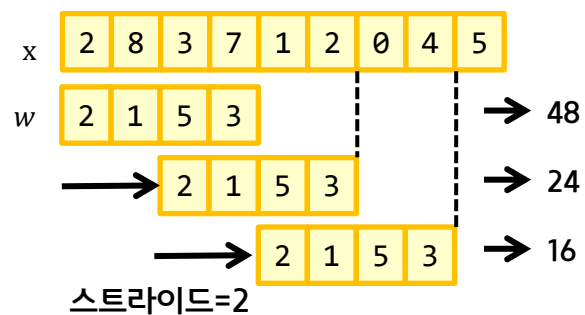
```
correlate(x, w, mode='same')
```

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

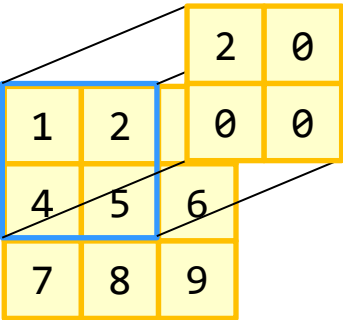
1	2	3
4	5	6
7	8	9

w

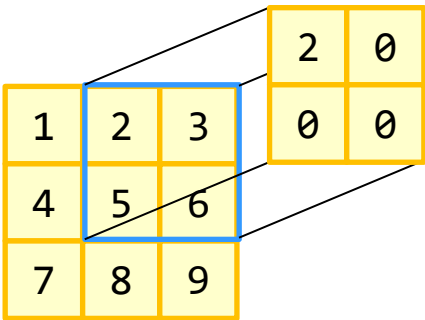
2	0
0	0

```
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')
```

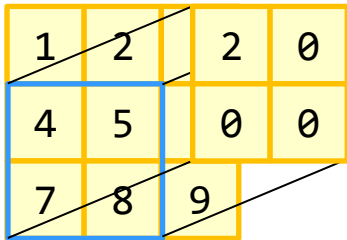
array([[2, 4],
 [8, 10]])



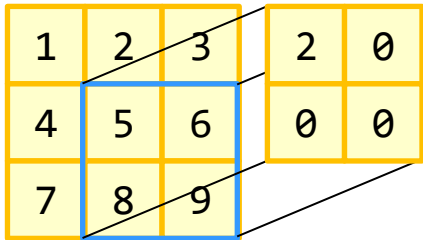
=> 2



=> 4



=> 8



=> 10

2	4
8	10

2차원 배열에서 same padding

$$N+P-F+1 = 0$$

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

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

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

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

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

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

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

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

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

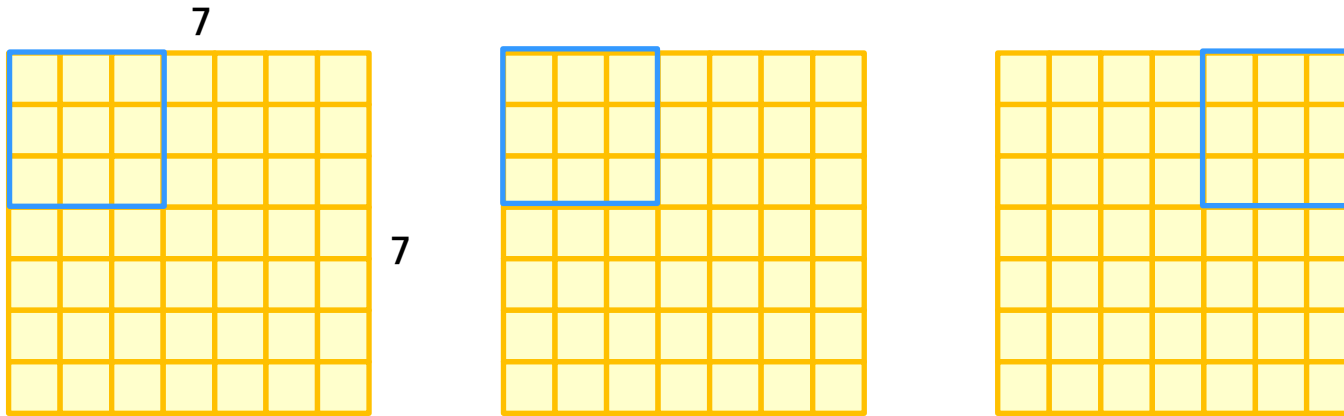
1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

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

```
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

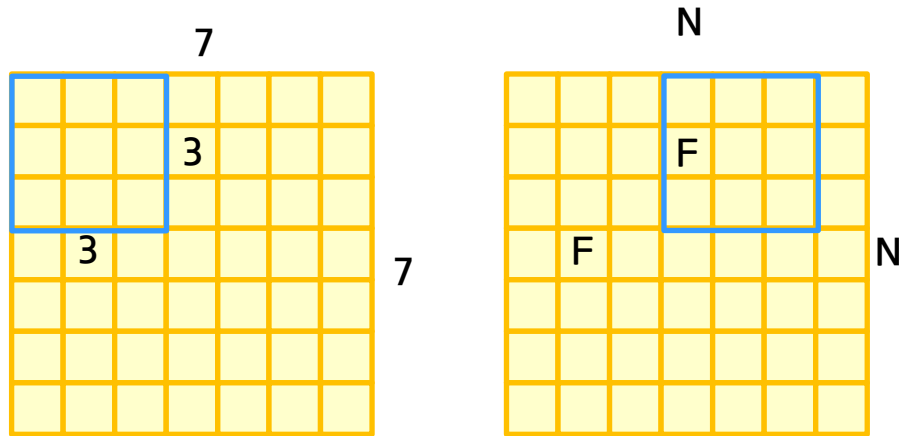
7x7 input
3x3 filter
mode = 'valid'
stride = 2

=> 3x3 output

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

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

2차원 배열에서 스트라이드 이해



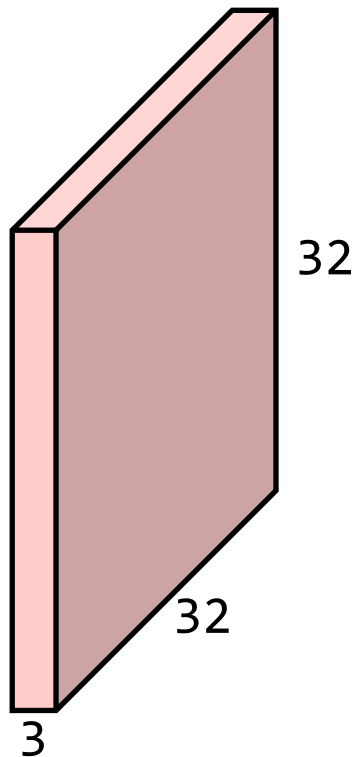
Output size :
 $(N - F) // \text{stride} + 1$

예) $N = 7, F = 3$
 stride 1 $\Rightarrow (7-3)//1+1 = 5$
 stride 2 $\Rightarrow (7-3)//2+1 = 3$
 stride 3 $\Rightarrow (7-3)//3+1 = 2$

Convolution Layer

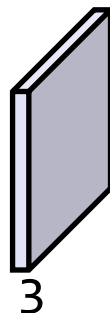
`image.shape => (32,32,3)`

32x32x3 image



필터는 항상 입력 볼륨의 전체 채널을 확장한다.

5x5x3 weight



277

183



ch=3 (R, G, B)

padding = valid

Convolution Layer

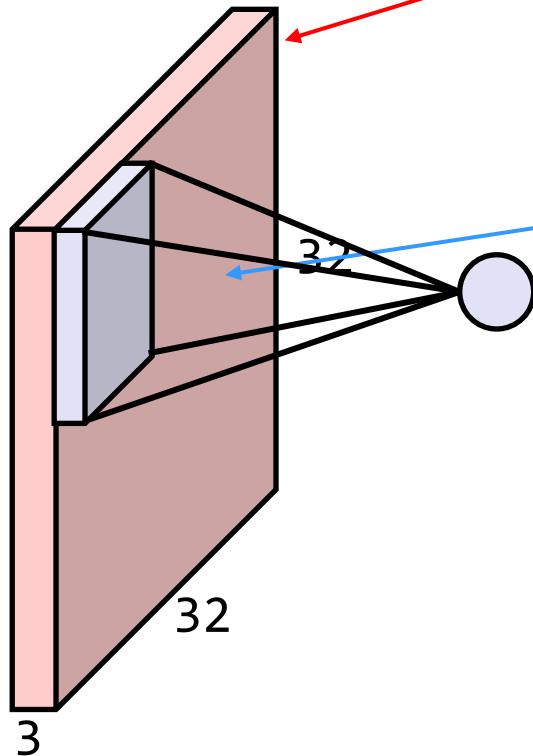
$(32, 32, 3)$

32x32x3 image

output.shape
 $(28, 28, 1)$

$(5, 5, 3)$

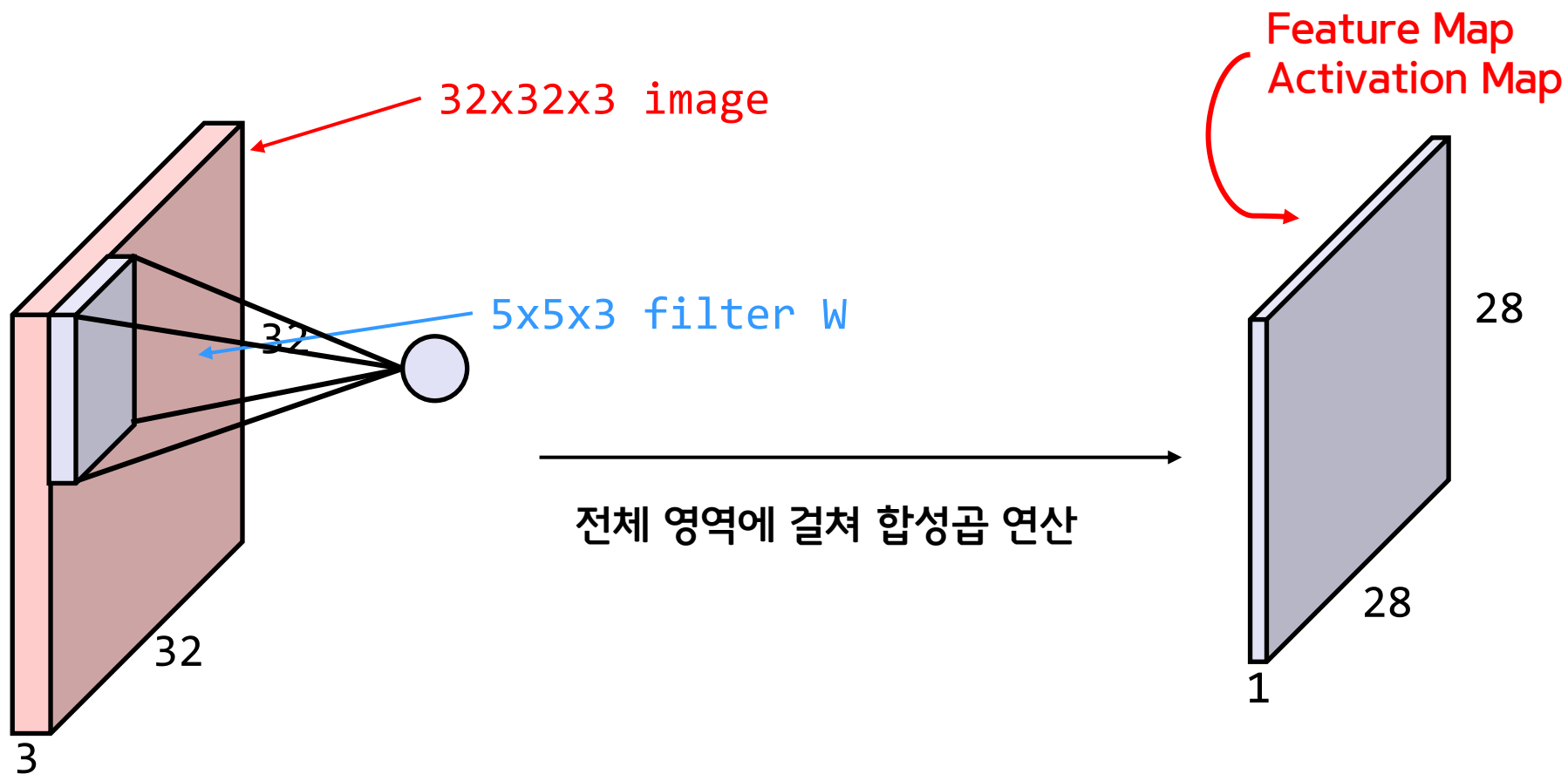
5x5x3 filter W



1 number :
필터와 이미지의 5x5x3영역 사이에서
내적을 취한 결과

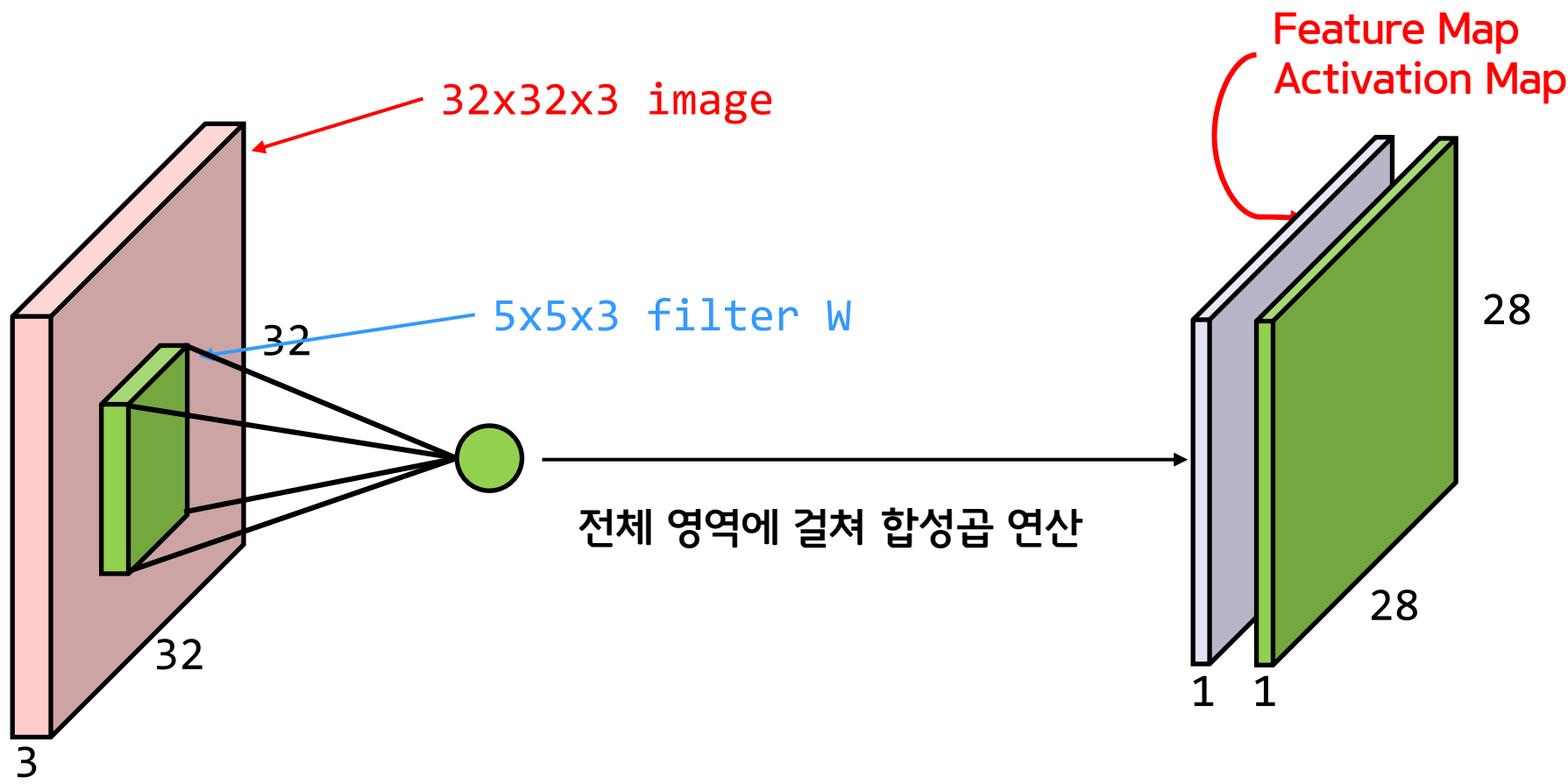
$XW + b$

Convolution Layer



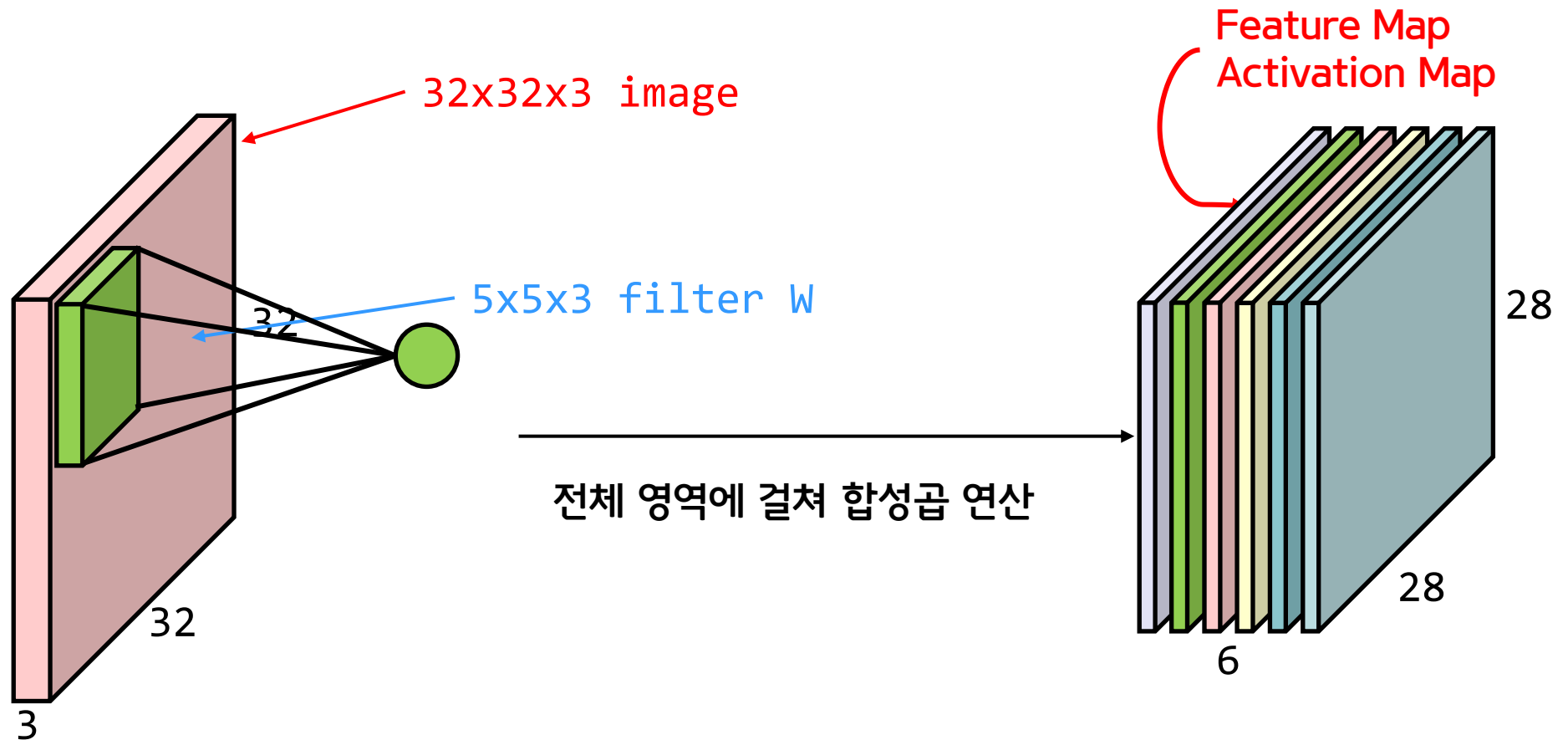
Convolution Layer

두번째 필터 동작



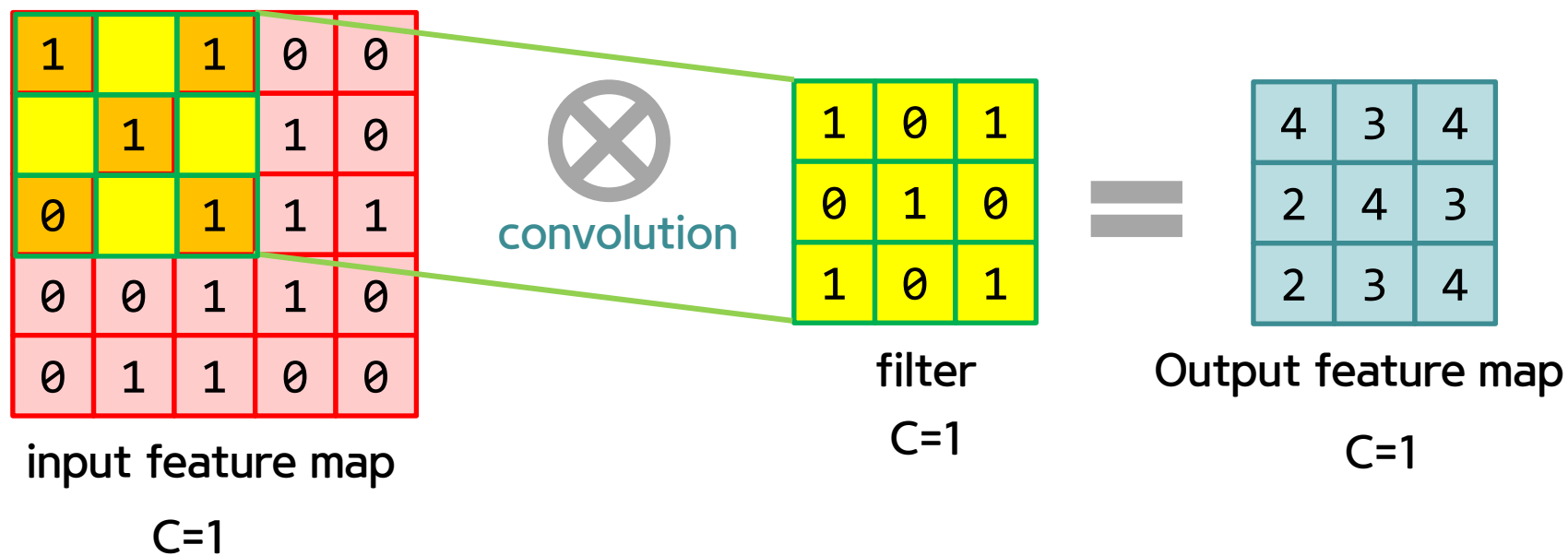
Convolution Layer

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

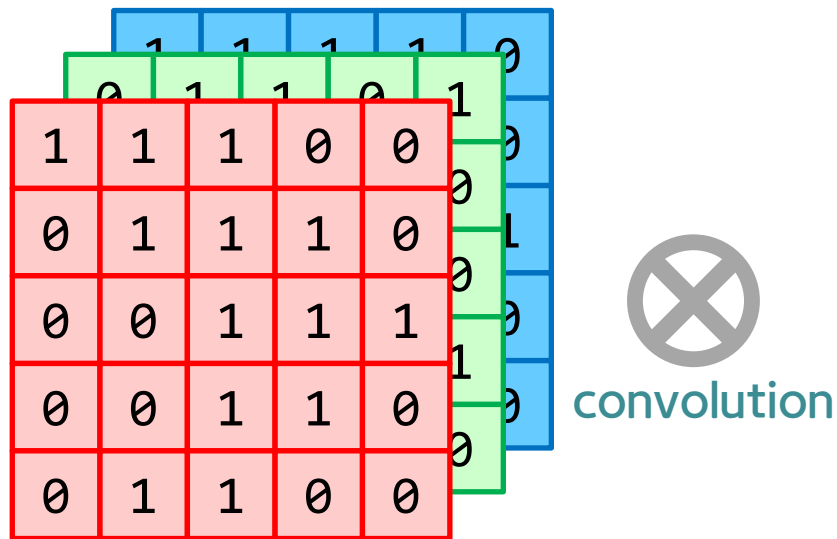


Convolution Layer - 계산

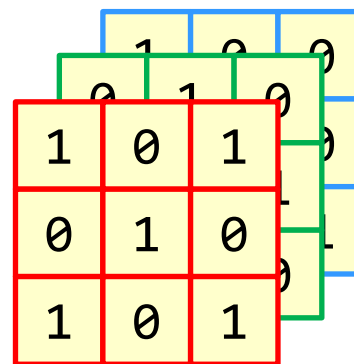
$$1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$



Convolution Layer - 계산



input channel = 3



filter channel = 3

of filters : 1

$$\begin{bmatrix} 3. & -1. & 3. \\ -2. & 0. & 2. \\ 1. & 3. & 4. \end{bmatrix}$$

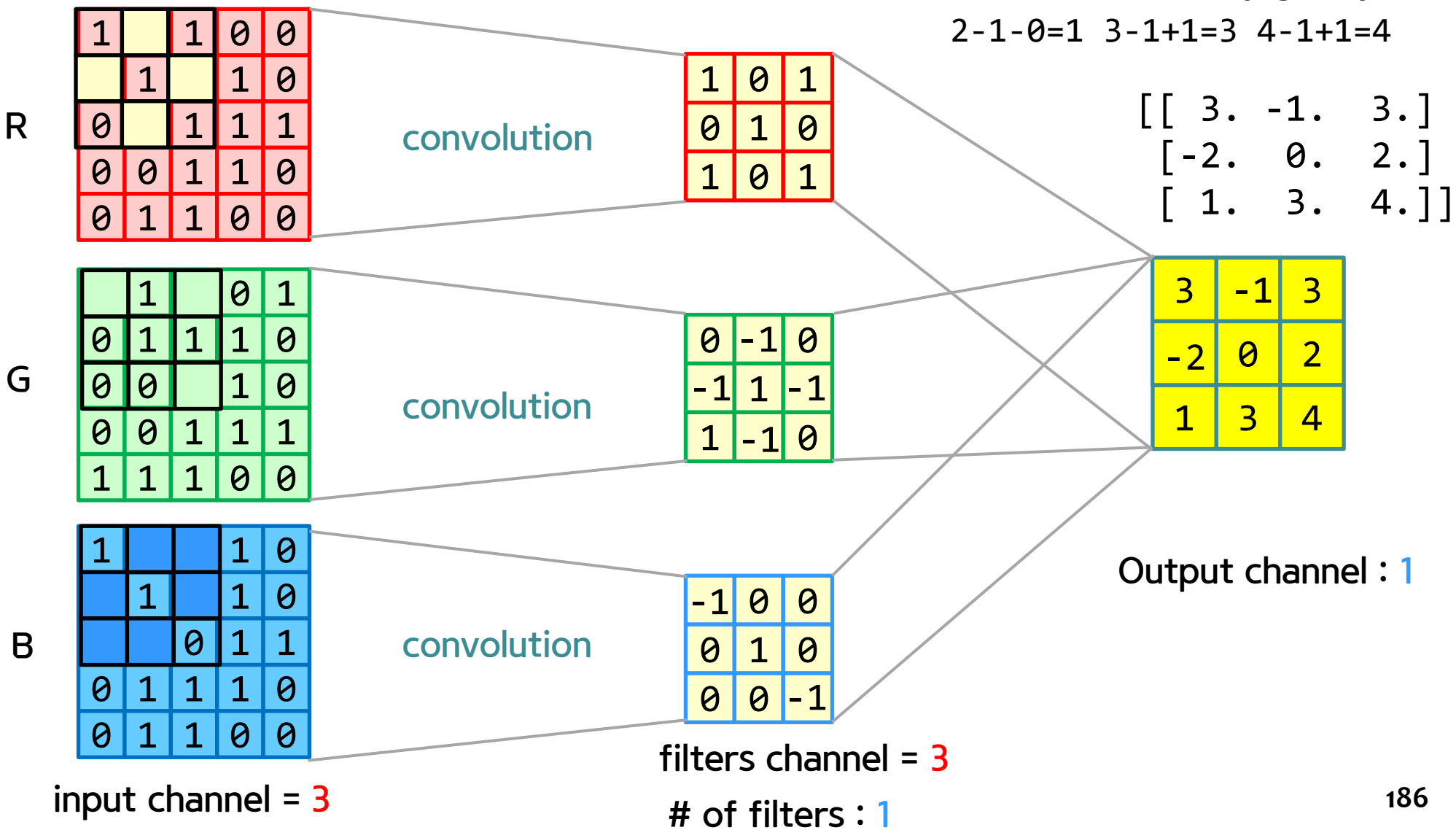
=

3	-1	3
-2	0	2
1	3	4

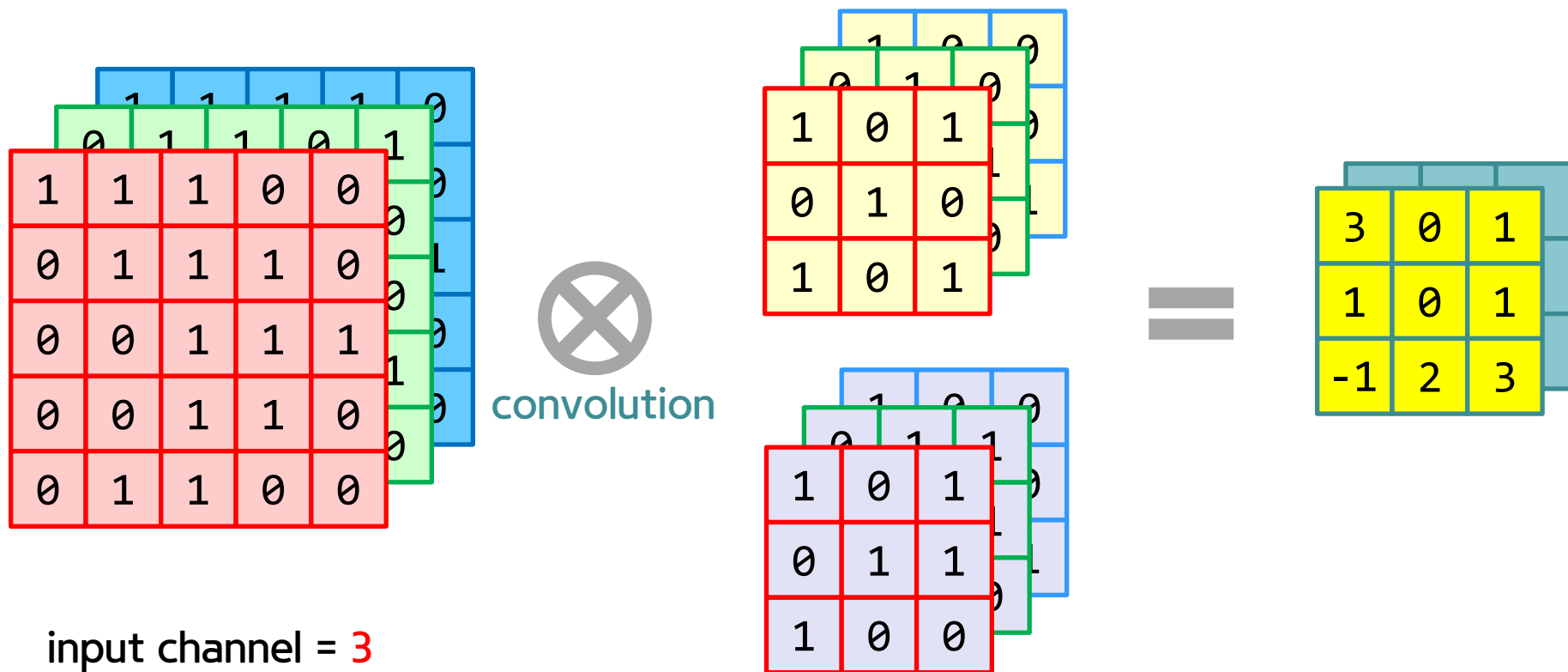
Output channel : 1

Convolution Layer - Multi Channel, Many Filters

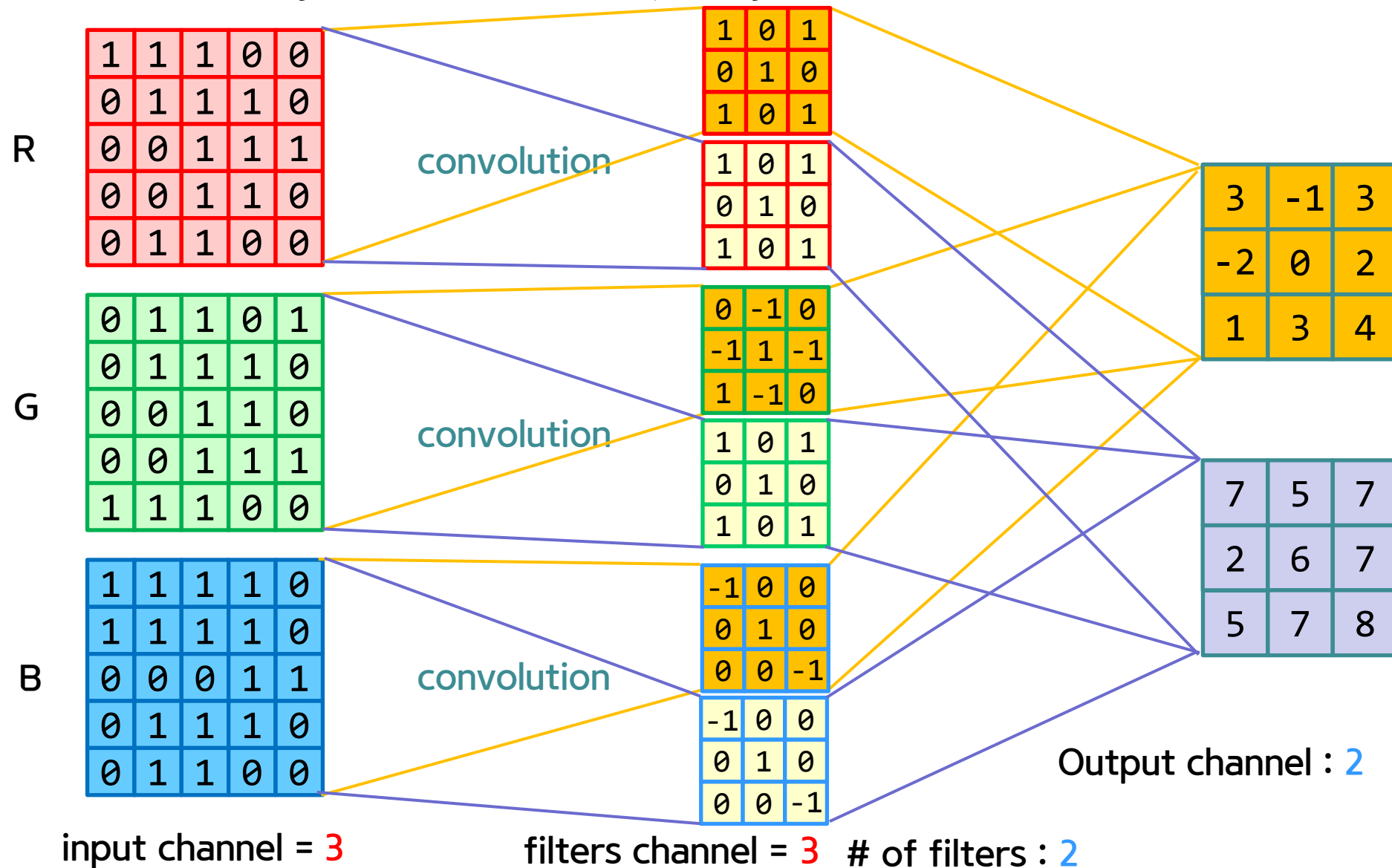
$4 - 1 + 0 = 3$ $3 - 3 - 1 = -1$ $4 + 0 - 1 = 3$
 $2 - 2 - 2 = -2$ $4 - 2 - 2 = 0$ $3 - 1 + 0 = 2$
 $2 - 1 - 0 = 1$ $3 - 1 + 1 = 3$ $4 - 1 + 1 = 4$



Convolution Layer - 계산



Convolution Layer - Multi Channel, Many Filters



tf.keras.layers.Conv2D

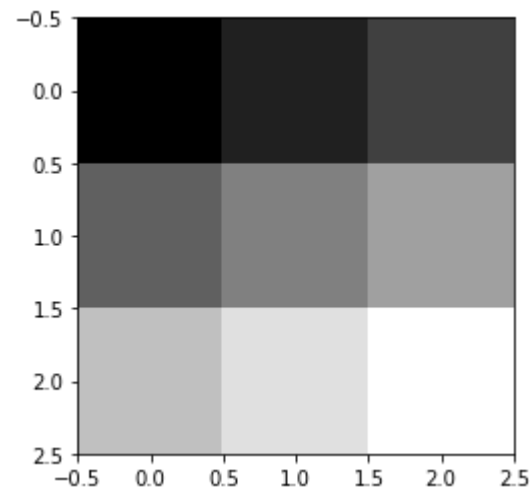
Arguments:

- **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 $\neq 1$ is incompatible with specifying any `dilation_rate` value $\neq 1$.
- **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".

```
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]]

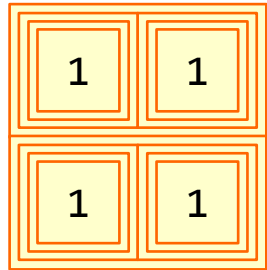
1	2	3
4	5	6
7	8	9

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

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

1	2	3
4	5	6
7	8	9

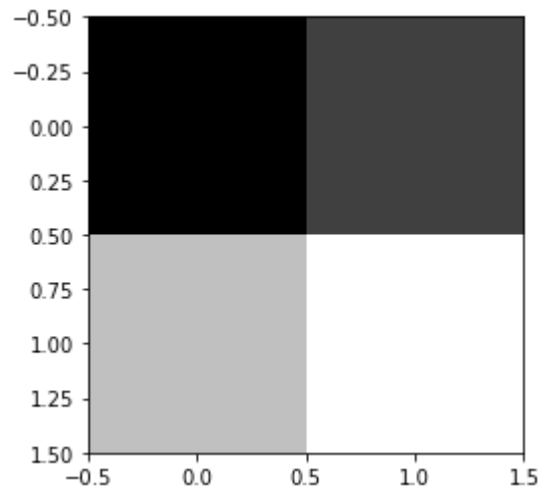

```
weight = np.array([[[[1.]], [[1.]]], [[1.]], [[1.]], [[1.]])
```



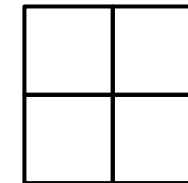
axis=0 axis=3
 ↓ ↓
(2,2,1,1)=>
(height,width,channel,filters)

```
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()
```

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



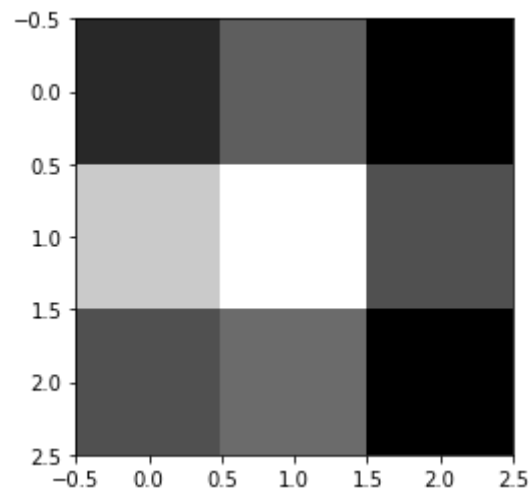
1	2	3
4	5	6
7	8	9



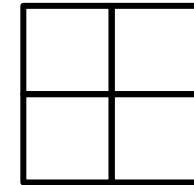
```
weight = np.array([ [ [ [ 1. ] ], [[1.]] ] , [[[1.]], [[1.]]] ])
print("weight.shape=", weight.shape)
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=1, kernel_size=2, padding='same',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape)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

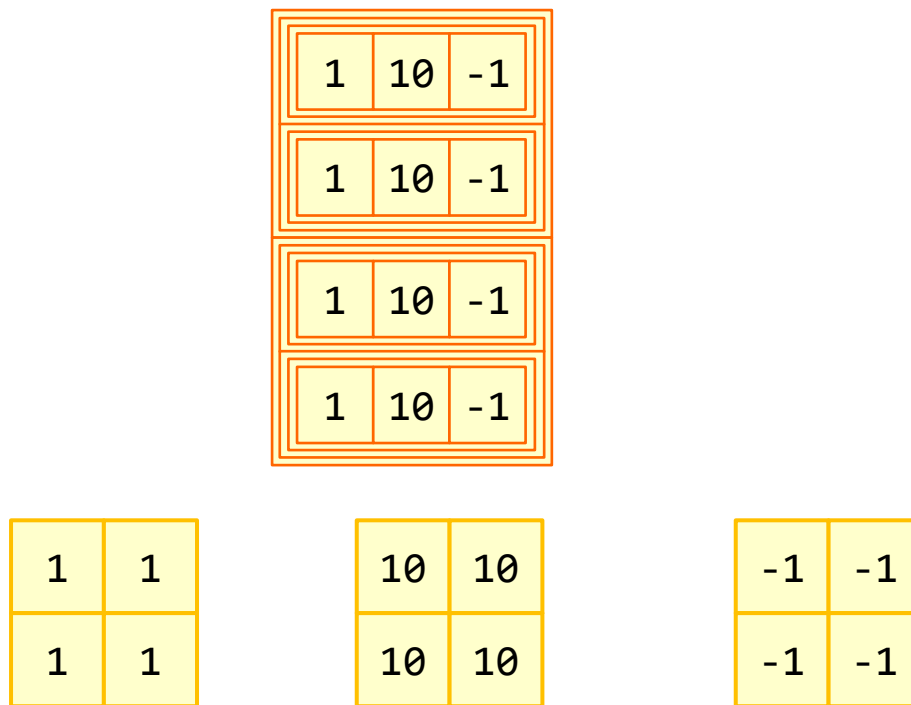


1	2	3	0
4	5	6	0
7	8	9	0
0	0	0	0

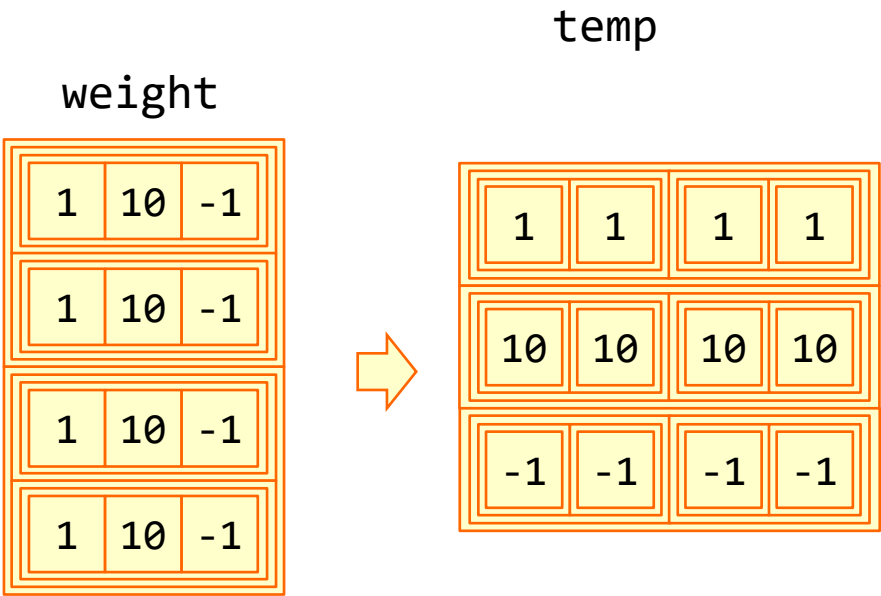


```
weight = np.array([[[[1.,10.,-1.]],[[1.,10.,-1.]],[[1.,10.,-1.]],[[1.,10.,-1.]])])  
print(weight.shape)
```

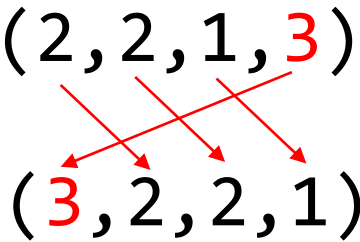
weight
(height,width,channel,nums)
(2,2,1,1) => (2,2,1,3)



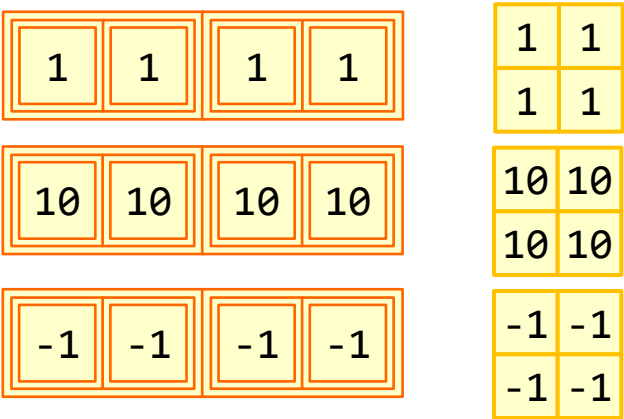
```
temp=np.transpose(weight,(3,0,1,2))
```



weight
(height,width,channel,nums)



```
temp=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

12	16
24	28

10	10
10	10

120	160
240	280

(1, 2, 2, 3)

-1	-1
-1	-1

-12	-16
-24	-28

conv2d (1,2,2,**3**)

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

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

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

12	120	-12
16	160	-16
24	240	-24
28	280	-28



12	16	24	28
120	160	240	280
-12	-16	-24	-28



12	16	24	28
120	160	240	280
-12	-16	-24	-28



12	16
24	28

120	160
240	280

-12	-16
-24	-28

```
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.]]
```

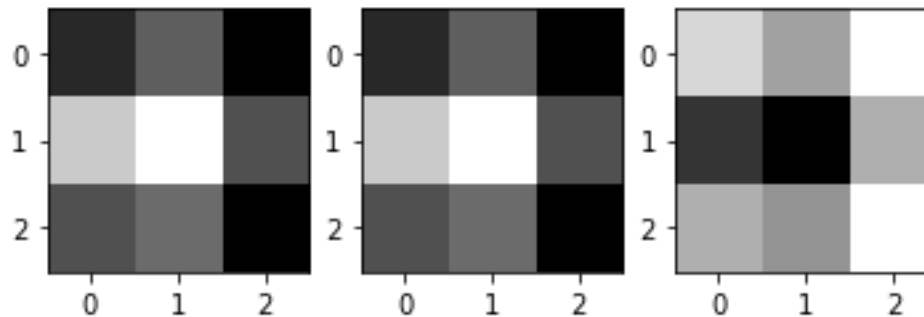
```
[[ -12. -16.  -9.]
 [-24. -28. -15.]
 [-15. -17.  -9.]]
```

(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)

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

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) (1,5,5,3)
for i, map in enumerate(maps):
    print(map.reshape(5,5))
```

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)

(**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))
```

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



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

$(3, 3, \textcolor{red}{3}, 1)$
 $(\textcolor{red}{3}, 3, 3, 1)$

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

```
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.]]
```

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

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

(3, 3, 2)

(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

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)

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

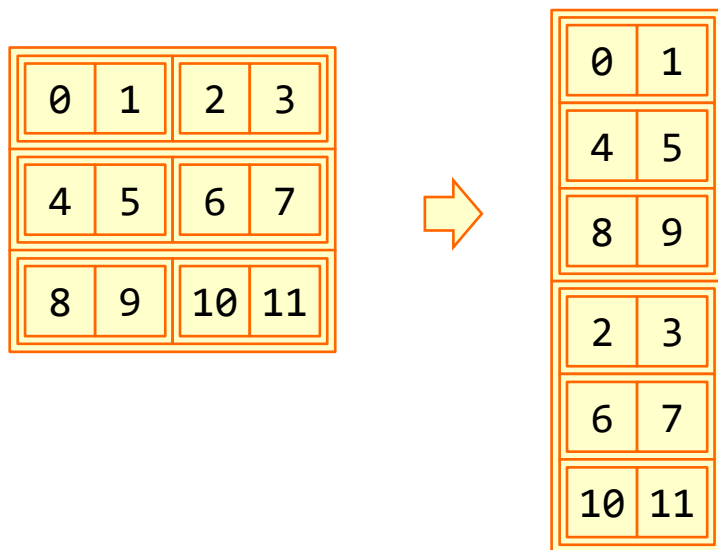
```
weight_init = tf.constant_initializer(weight)
conv2d = tf.keras.layers.Conv2D(filters=2, kernel_size=3, padding='valid',
kernel_initializer=weight_init)(image)
print("conv2d.shape", conv2d.shape) # ( 1,3,3,2)
feature_maps = np.swapaxes(conv2d, 0, 3)
for feature_map in feature_maps:
    print(feature_map.reshape(3,3))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

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

3	-1	3
-2	0	2
1	3	4

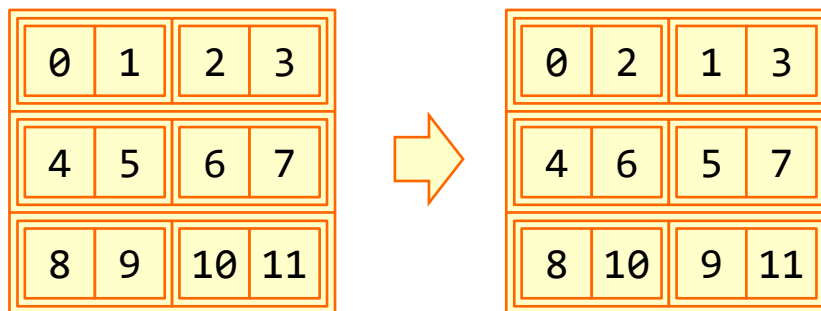
7	5	7
2	6	7
5	7	8

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



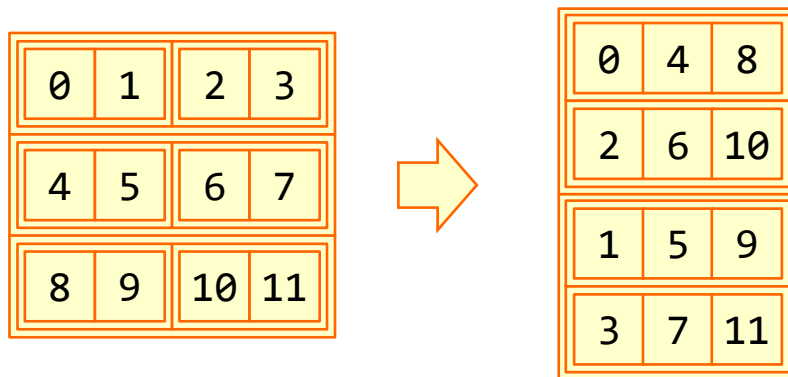
```
array([[[ 0,  1],
        [ 4,  5],
        [ 8,  9]],
       [[ 2,  3],
        [ 6,  7],
        [10, 11]]])
```

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



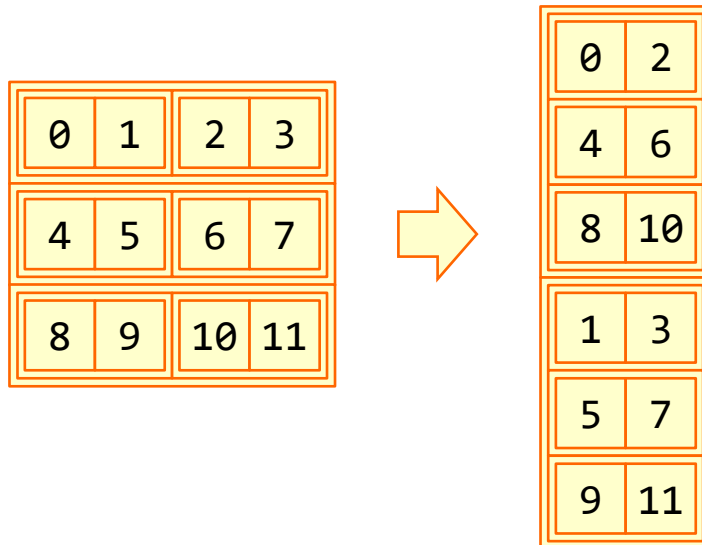
```
array([[[ 0,  2],
        [ 1,  3]],
       [[ 4,  6],
        [ 5,  7]],
       [[ 8, 10],
        [ 9, 11]]])
```

```
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))
```

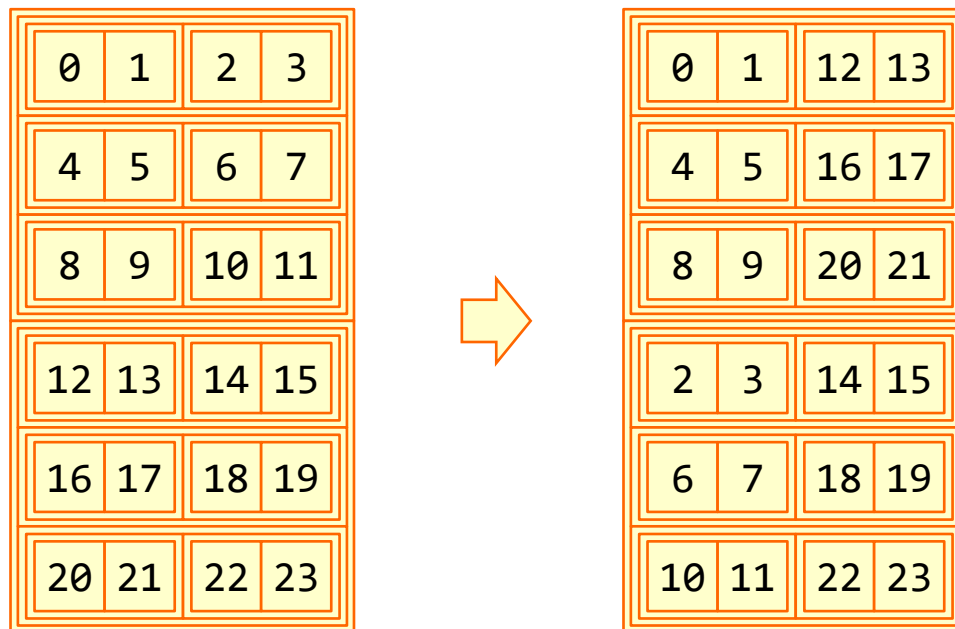


(3, 2, 2)
(2, 3, 2)

```
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)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23



0	4	8	12	16	20
1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23

```
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))
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23



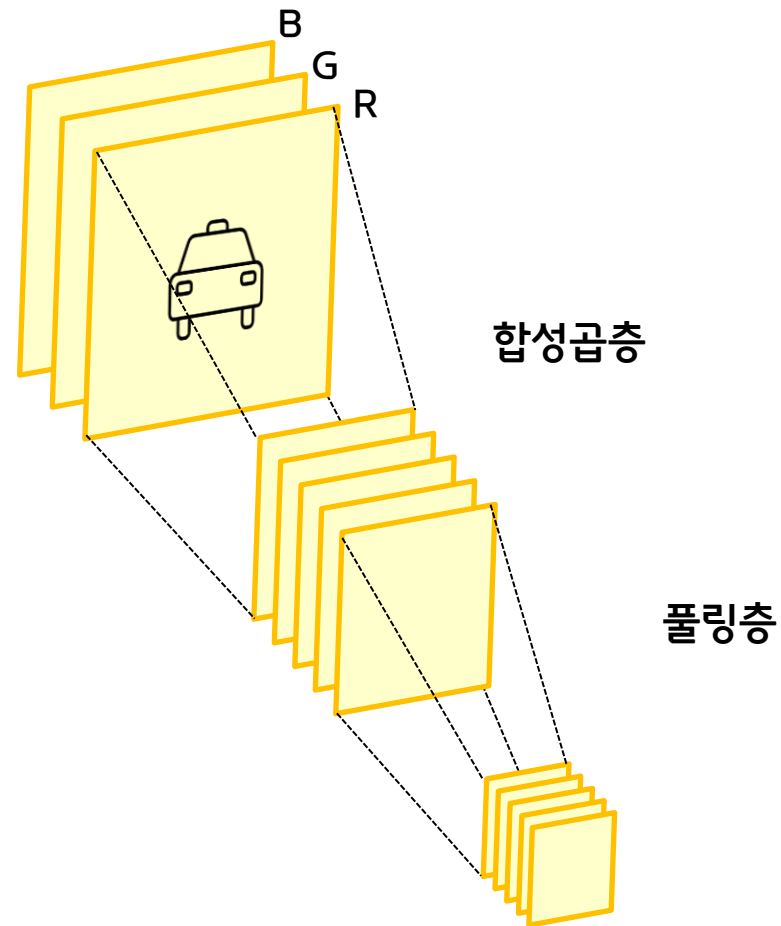
0	2	4	6	8	10
12	14	16	18	20	22
1	3	5	7	9	11
13	15	17	19	21	23

(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

풀링이란? 특성 맵을 스캔하며 최대값을 고르거나 평균값을 계산하는 것을 말함

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

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

최대 풀링

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



6

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



8

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



14

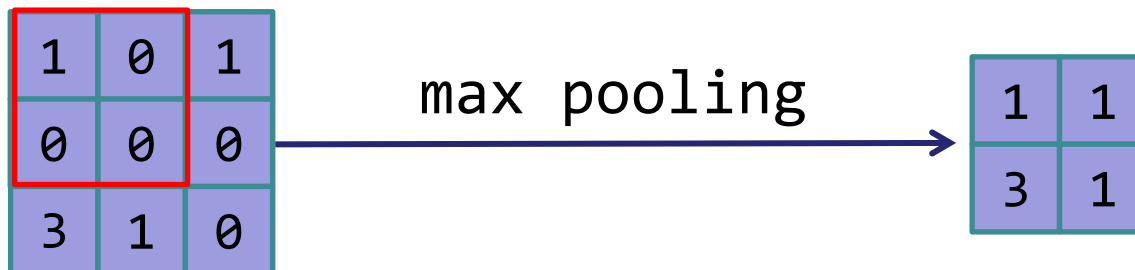
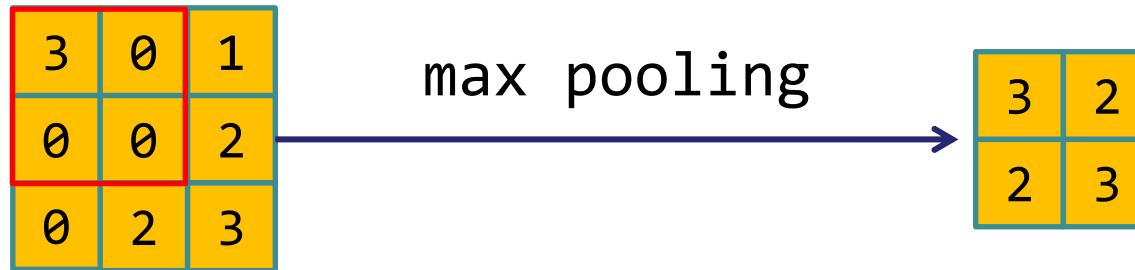
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



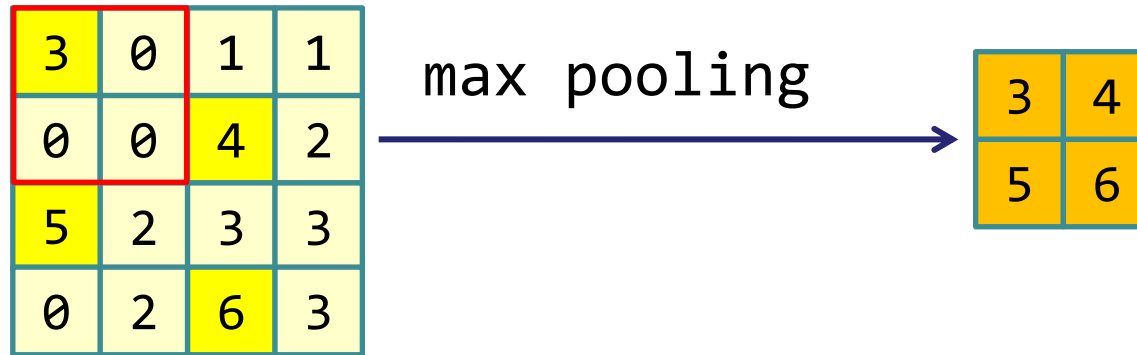
16

6	8
14	16

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



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



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

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



$$\frac{1 + 2 + 5 + 6}{4} = 3.5$$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



$$\frac{3 + 4 + 7 + 8}{4} = 5.5$$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



$$\frac{9 + 10 + 13 + 14}{4} = 11.5$$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16



$$\frac{11 + 12 + 15 + 16}{4} = 13.5$$

3.5	5.5
11.5	13.5

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".

```
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.]]]

4	3
2	1


```
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.]]]

4	3	0
2	1	0
0	0	0

4	3	0
2	1	0
0	0	0

4	3	0
2	1	0
0	0	0

4	3	0
2	1	0
0	0	0

```
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)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

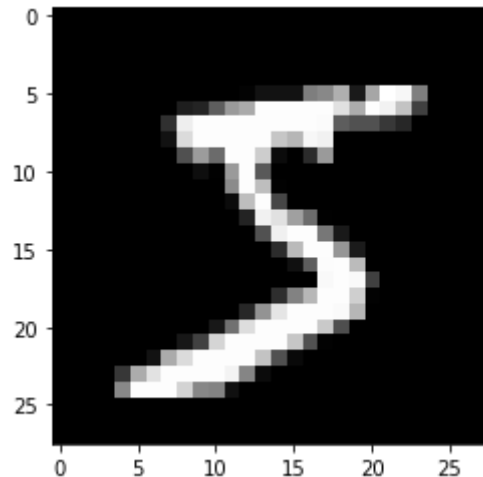
5	7
13	15

```
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.

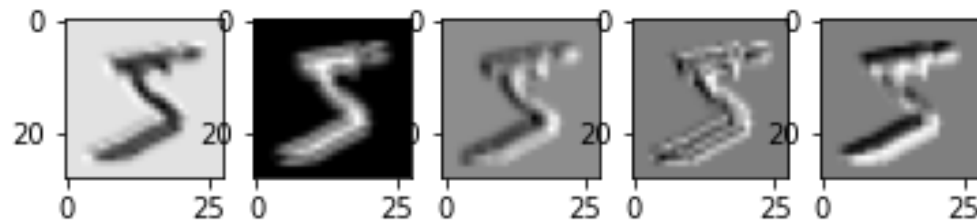
img = train_images[0]
plt.imshow( img, cmap='gray')
plt.show()
```



```
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()
```

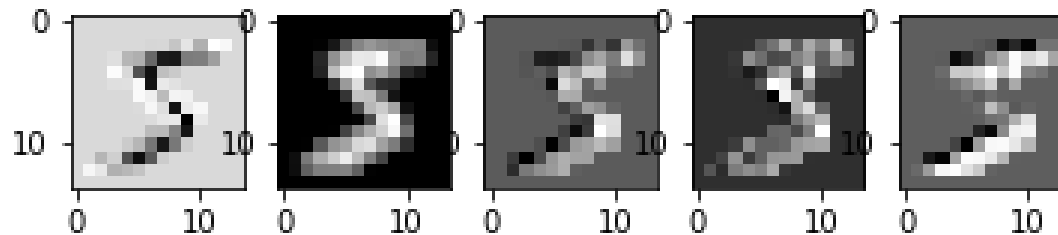
```
weight.shape (2, 2, 1, 5)
conv2d.shape (1, 28, 28, 5)
```



```
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)



활성함수는 네트워크에 비선형성(nonlinearity)을 추가하기 위해 사용됨

- 활성화 함수 없이 layer를 쌓은 네트워크는 1-layer 네트워크와 동일하기 때문에 활성화 함수는 비선형 함수로 불리기도 한다.
- 멀티레이어 퍼셉트론을 만들 때 활성화 함수를 사용하지 않으면 쓰나마이다.

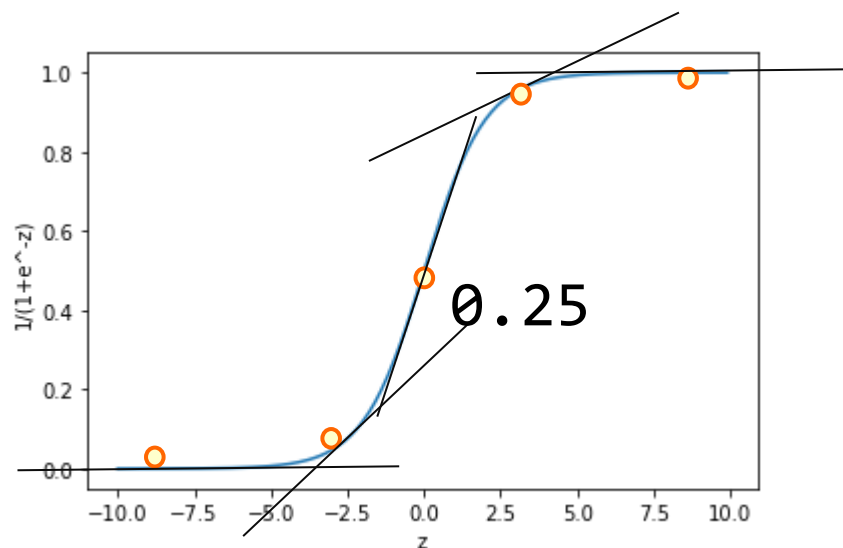
$$1000 * 0.25 = 250 * 0.25 =$$

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 지나가 그 다음 layer로 전파되지 않는다. 즉, 학습이 되지 않는다.

2) 활성화함수의 결과 값의 중심이 0이 아닌 0.5이다.

3) 계산이 복잡하다 (지수함수 계산)

!! Gradient Vanishing

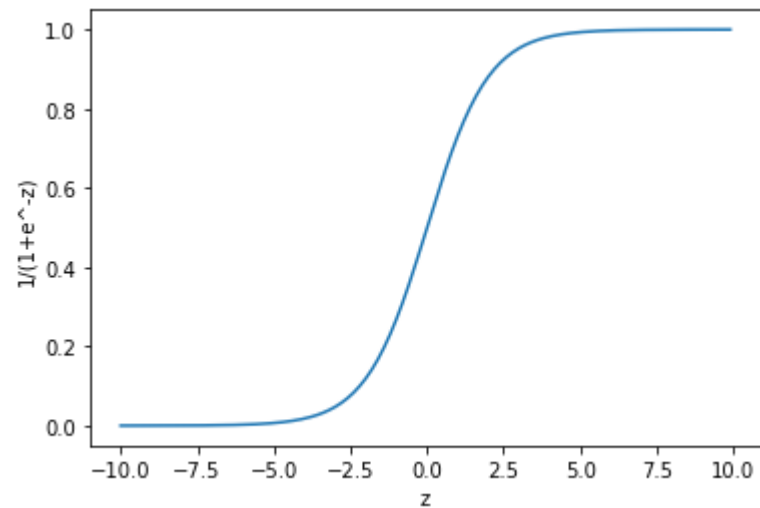
- 시그모이드와 같이 결과값이 포화(saturated)되는 함수는 gradient vanishing 현상을 야기

- 이전 레이어로 전파되는 그라디언트가 0에 가까워 지는 현상

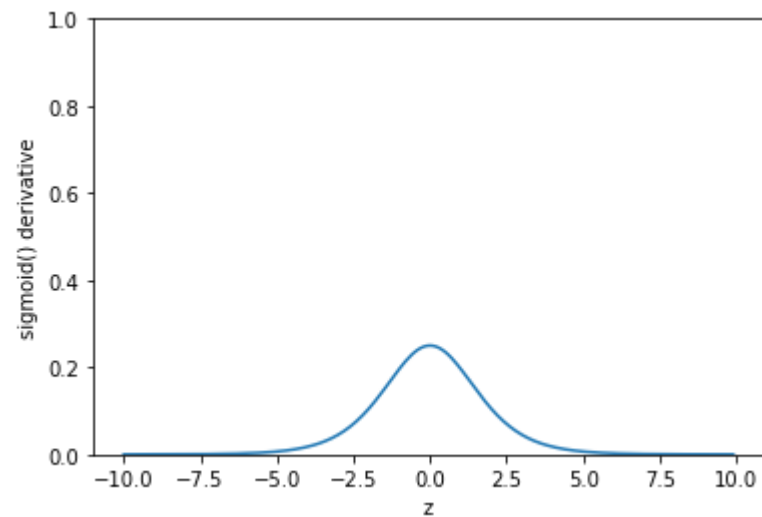
- 레이어를 깊게 쌓으면 파라미터의 업데이트가 제대로 이루어지지 않음

- 양 극단의 미분값이 0에 가깝기 때문에 발생하는 문제

시그모이드 함수



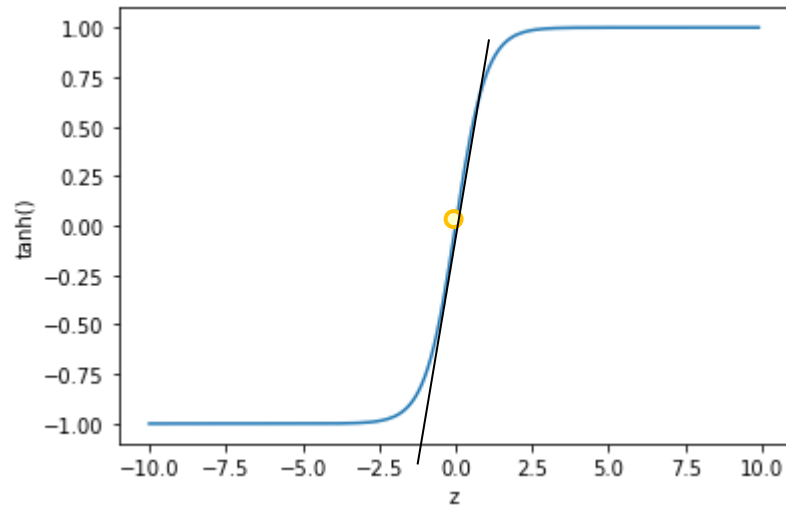
시그모이드 미분값



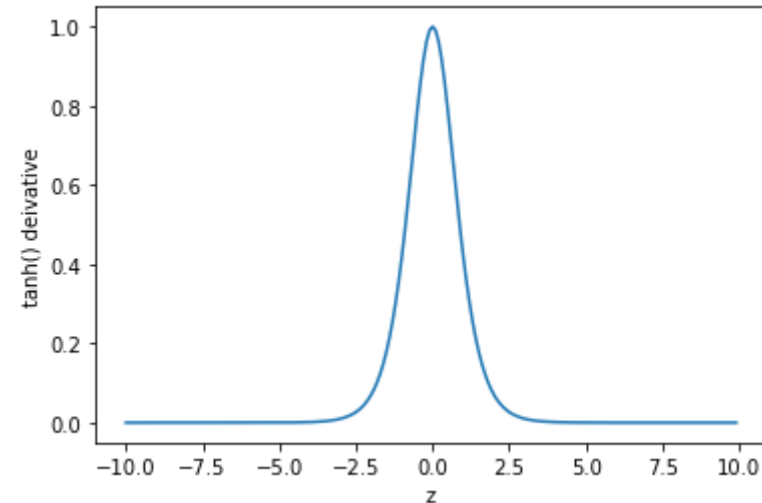
양쪽 꼬리가 0에 수렴하며 최대값이 0.25를 넘지 않는다.

2. 하이퍼 볼릭 탄젠트(tanh)

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



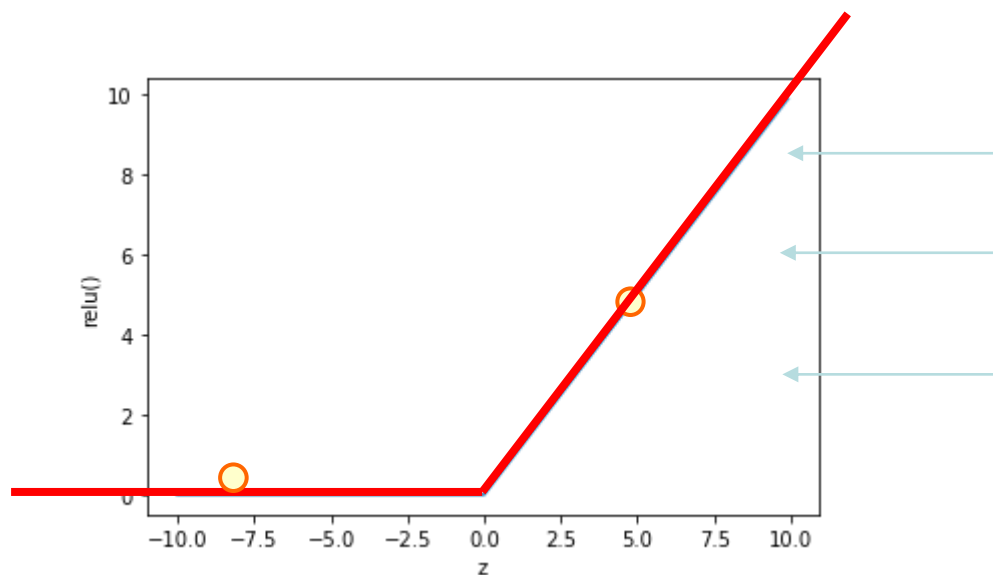
$$\tanh(x)' = (1 - \tanh(x))(1 + \tanh(x)) = 1 - H^2$$



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

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

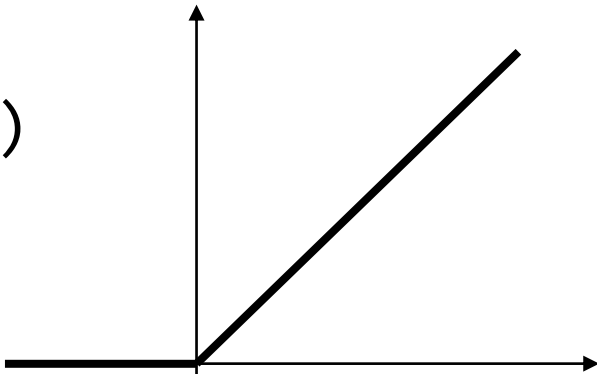
- (1) 양 극단값이 포화되지 않는다. (양수 지역은 선형적)
- (2) 계산이 매우 효율적이다 (최대값 연산 1개)
- (3) 수렴속도가 시그모이드류 함수대비 6배 정도 빠르다.

- 단점

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

Activation Function

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



3	0	1
-2	0	2
0	2	3

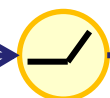
Relu



3	0	1
0	0	2
0	2	3

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

Relu



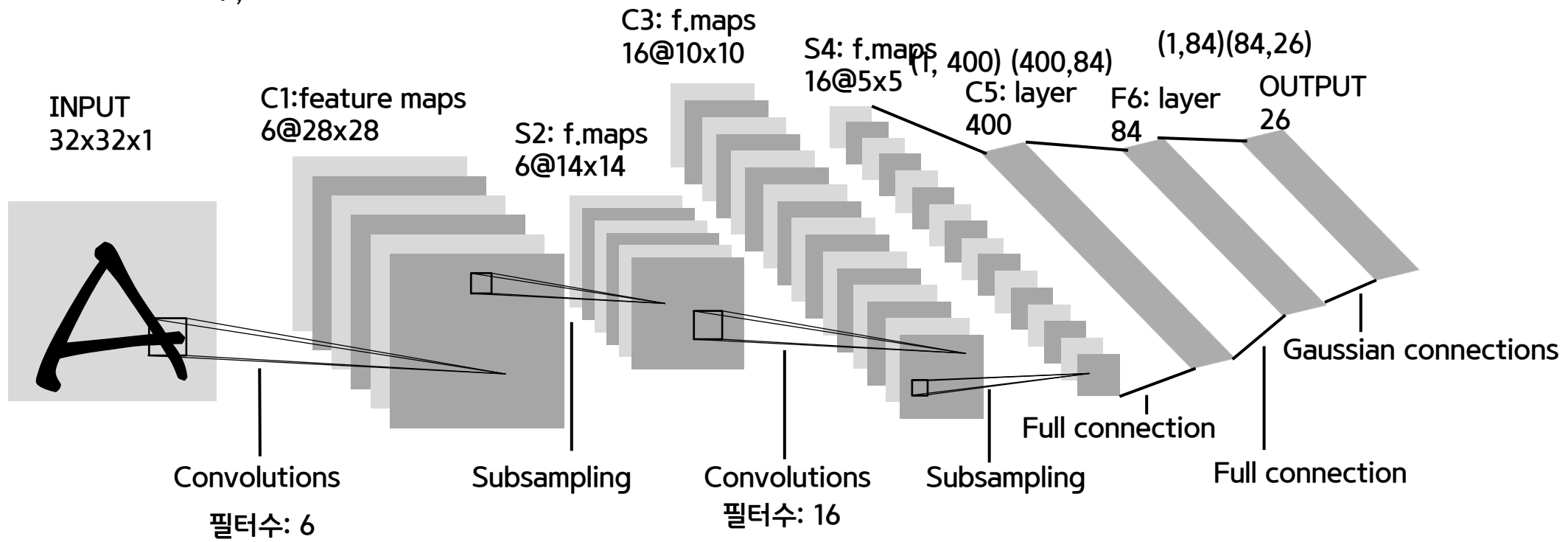
1	0	1
0	0	0
0	1	0

00000010 +2
11111110 -2 => 254

LeNet-5

$(1, 5, 5, 16) \Rightarrow (1, -1) \Rightarrow (1, 400) (400, 84) \Rightarrow (1, 84) (84, 26) \Rightarrow (1, 26)$

LeCun et al. , 1998



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

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

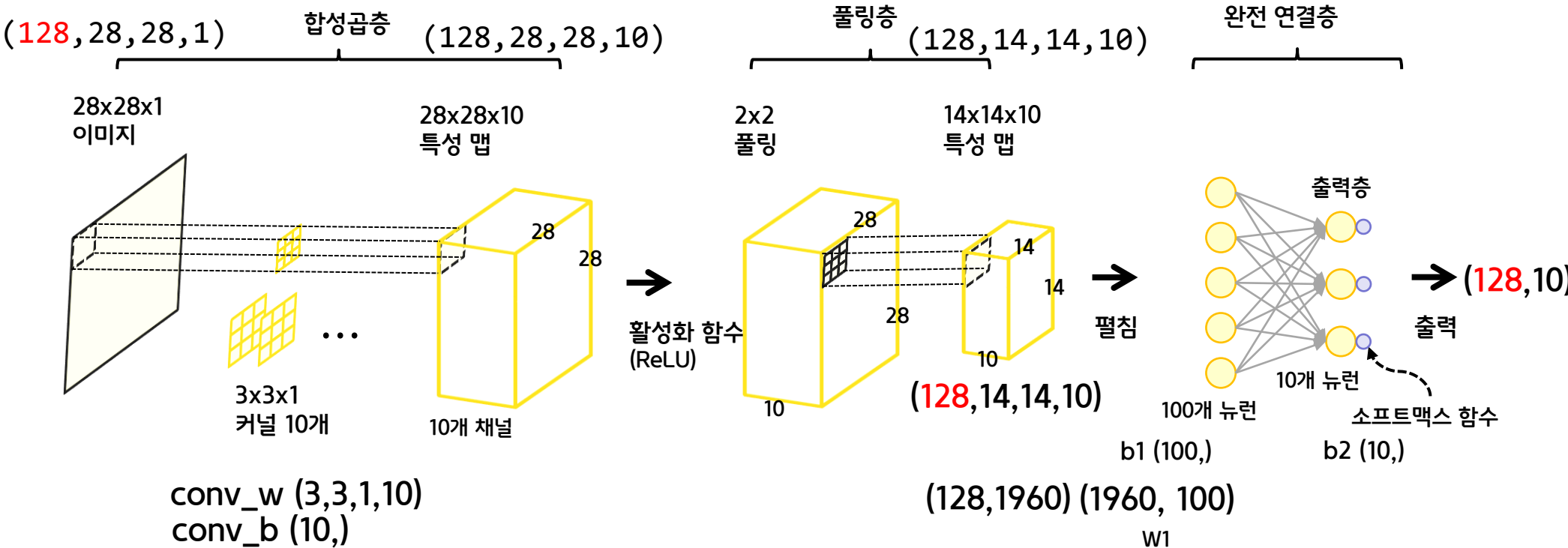
6. 합성곱 신경망 이해

6.1 합성곱 연산

6.2 합성곱 신경망 구현

6.3 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 타겟(`y`)을 기반으로 손실값을 계산한다.
- 이렇게 하면 크로스 엔트로피 손실과 그레디언트 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 그레이디언트를 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 `conv_w`와 `conv_b`를 포함하여 그레이디언트가 필요한 `weights_list`로 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

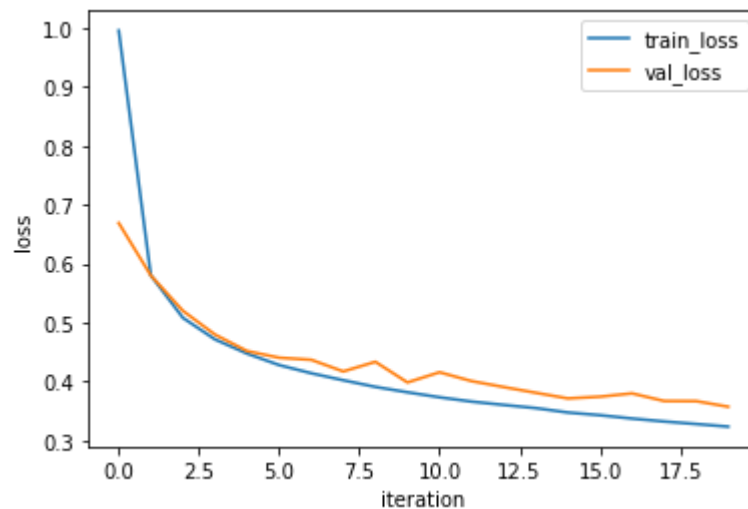
```
cn = ConvolutionNetwork(n_kernels=10, units=100, batch_size=128, learning_rate=0.01)
cn.fit(x_train, y_train_encoded,
      x_val=x_val, y_val=y_val_encoded, epochs=20)
```

```
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

6. 합성곱 신경망 이해

6.1 합성곱 연산

6.2 합성곱 신경망 구현

6.3 케라스로 합성곱 신경망 구현

소스참조

7. 순환 신경망(RNN)

7.1 RNN 이란

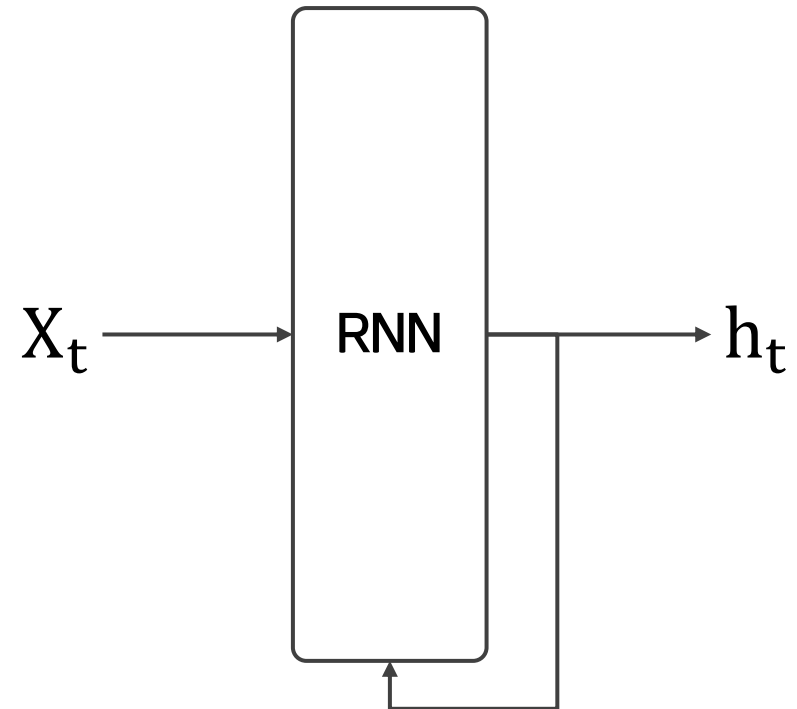
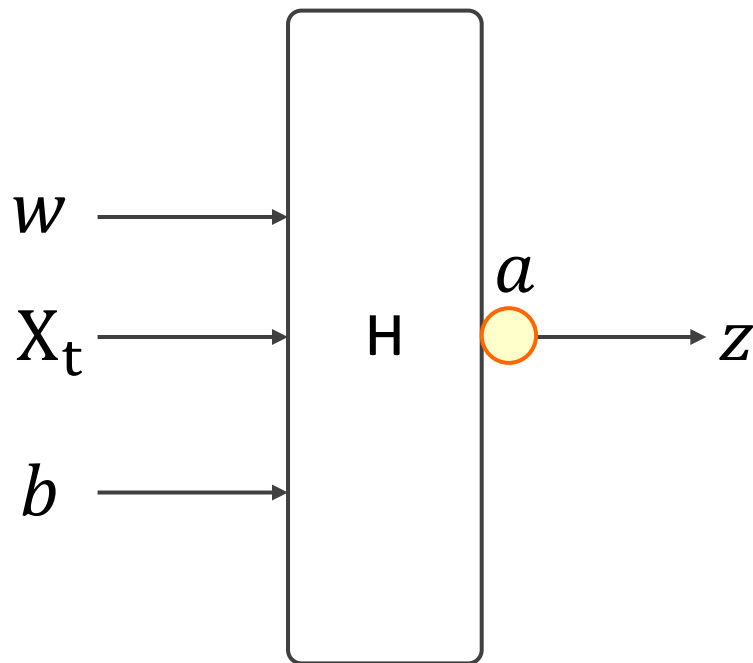
7.2 RNN 구현

7.3 다양한 자연어 처리 모델 구현

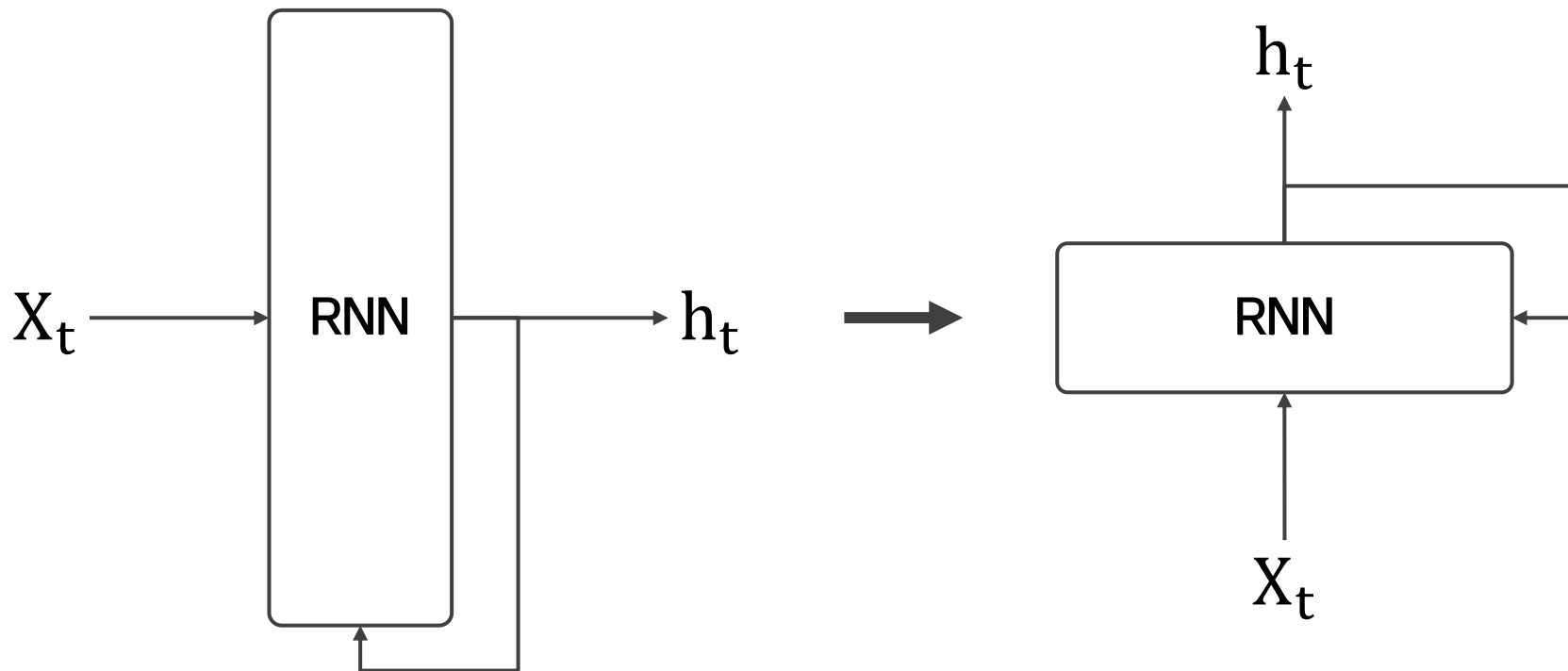
순환하기 위해서는 닫힌 경로가 필요하다.

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

순환 경로를 포함하는 RNN 계층



계층을 90도 회전시켜 그린다.



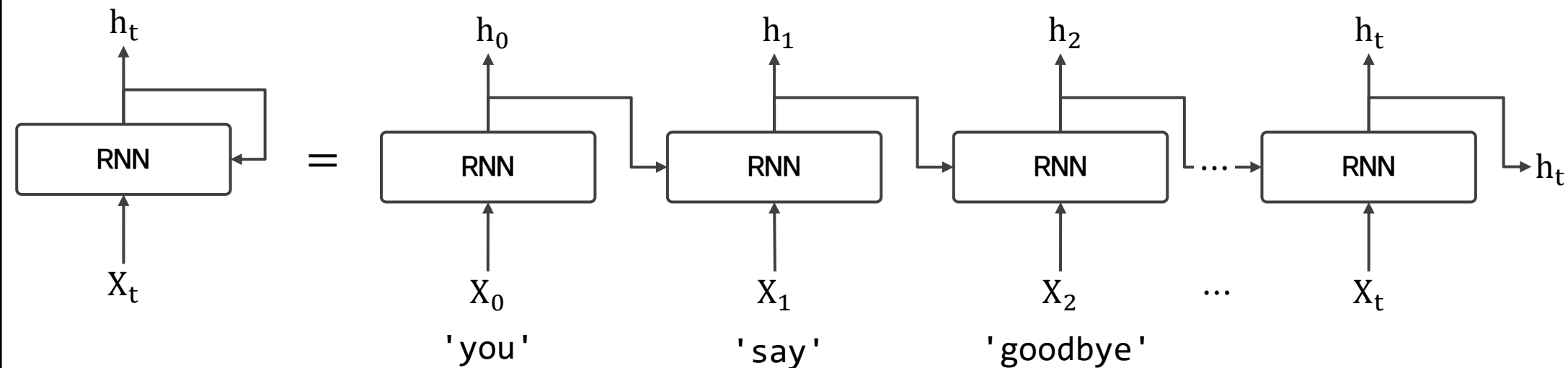
t: 시각

시계열 데이터($x_0, x_1, \dots, x_t, \dots$)가 RNN계층에 입력되고 이에 대응해 ($h_0, h_1, \dots, h_t, \dots$)가 출력된다.

각 시각에 입력되는 x_t 를 벡터라고 가정했을 때

문장(단어 순서)을 다루는 경우를 예로 든다면 각 단어의 분산 표현(단어 벡터)이 x_t 가 되며 이 분산 표현이 순서대로 하나씩 RNN계층에 입력된다.

RNN 계층의 순환 구조 펼치기



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

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

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

$$y = \text{sigmoid}(x) \Rightarrow y(1 - y)$$

$$h = \tanh(x) \Rightarrow 1 - h^2$$

W_x : 입력 x 를 출력 h 로 변환하기 위한 가중치

W_h : 1개의 RNN출력을 다음 시각의 출력으로 변환하기 위한 가중치

b : 편향

$h(t-1)$, x_t : 행벡터

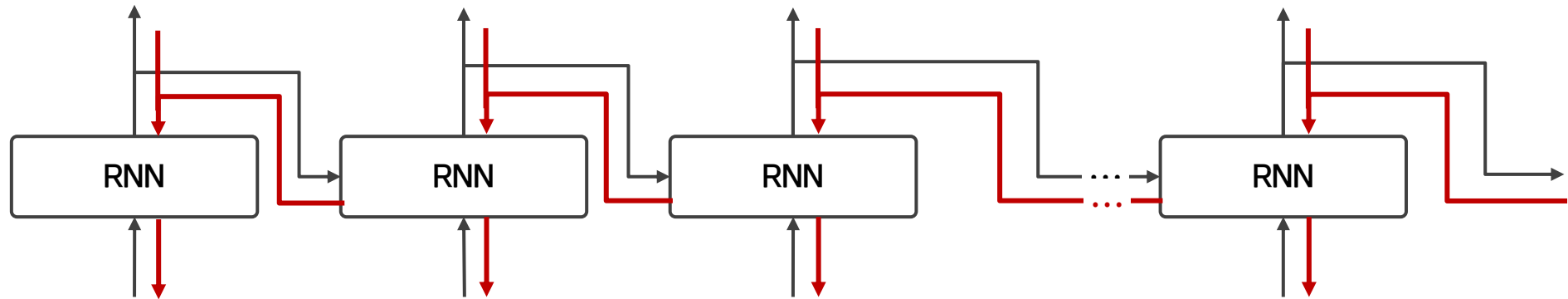
h_t 는 다른 계층을 향해 위쪽으로 출력되는 동시에 다음 시각의 RNN계층(자기 자신)을 향해 오른쪽으로도 출력된다.

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

RNN은 h 라는 '상태'를 가지고 있으며 위의 식의 형태로 갱신된다고 해석할 수 있다.

RNN계층을 '상태를 가지는 계층' 혹은 '메모리(기억력)이 있는 계층'이라고 한다.

순환 구조를 펼친 RNN 계층에서의 오차역전파



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

문제점

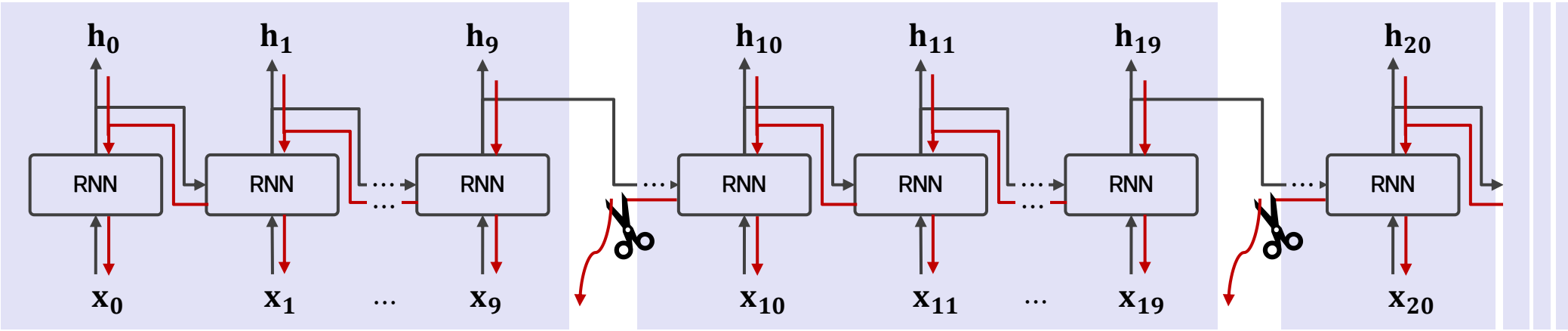
- 시계열 데이터의 시간 크기가 커지는 것에 비례하여 BPTT가 소비하는 컴퓨팅 자원도 증가
- 시간 크기가 커지면 역전파 시의 기울기가 불안정해짐

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

- 계층이 너무 길면 계산량과 메모리 사용량 등이 문제가 되고 계층이 길어짐에 따라 신경망을 하나 통과할 때마다 기울기 값이 조금씩 작아져서 이전 시각 t 까지 역전파되기 전에 0이 되어 소멸할 수도 있다.
- 순전파의 연결을 그대로 유지하면서(데이터를 순서대로 입력해야 한다) 역전파의 연결은 적당한 길이로 잘라내 잘라낸 신경망 단위로 학습을 수행한다.
- 역전파의 연결을 잘라버리면 그보다 미래의 데이터에 대해서는 생각할 필요가 없어지기 때문에 각각의 블록 단위로 미래의 블록과는 독립적으로 오차역전파법을 완결시킨다.
 - 블록: 역전파가 연결되는 일련의 RNN계층
- 순전파를 수행하고 그 다음 역전파를 수행하여 원하는 기울기를 구한다.
- 다음 역전파를 수행할 때 앞 블록의 마지막 은닉 상태인 h_t 가 필요하다.
 h_t 로 순전파가 계속 연결될 수 있다.

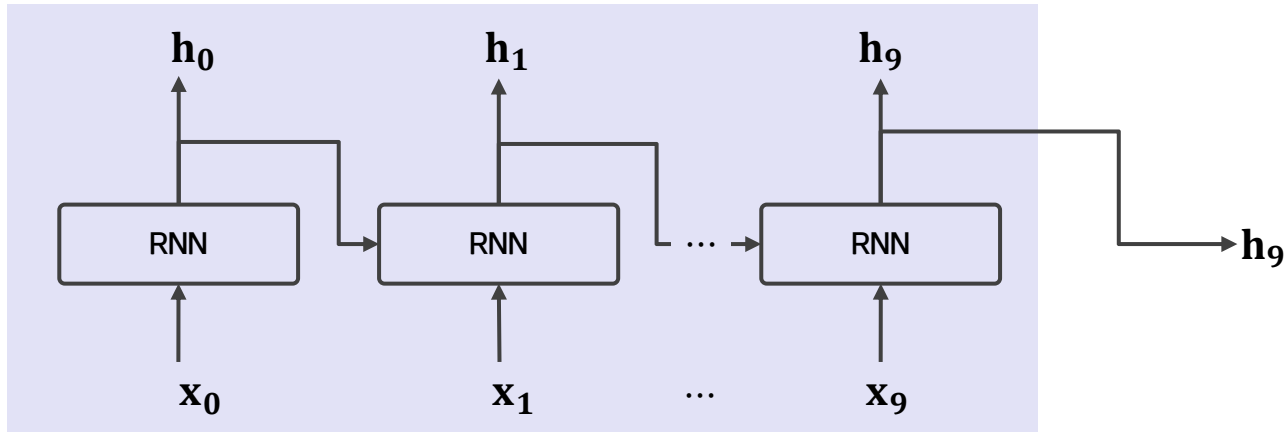
역전파의 연결을 적당한 지점에서 끊는다.

$T=0$

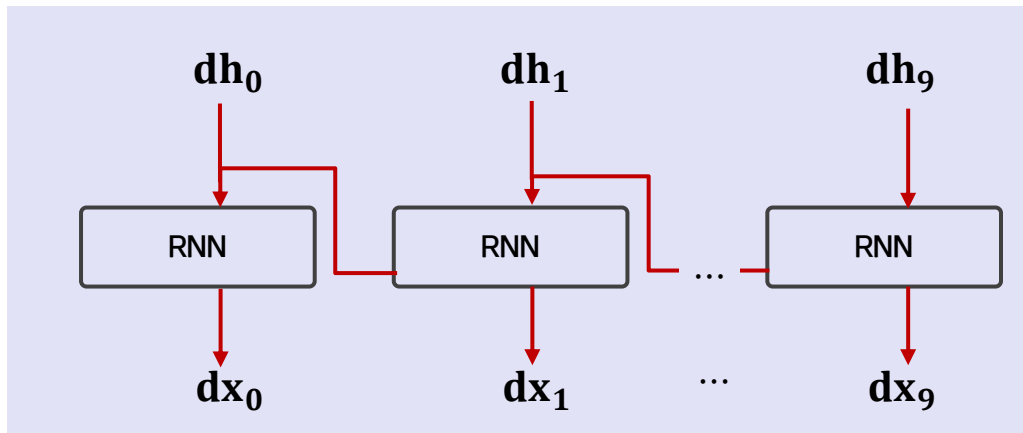


첫 번째 블록의 순전파와 역전파 : 이보다 앞선 시각으로부터의 기울기는 끊겼기 때문에 이 블록 내에서만 오차역전파법이 완결된다.

순전파

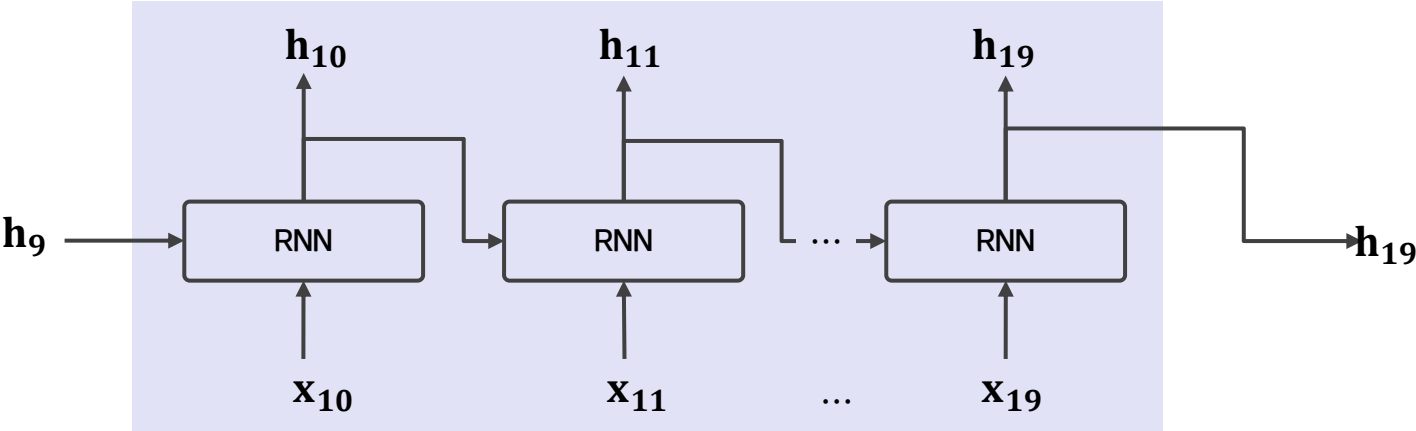


역전파

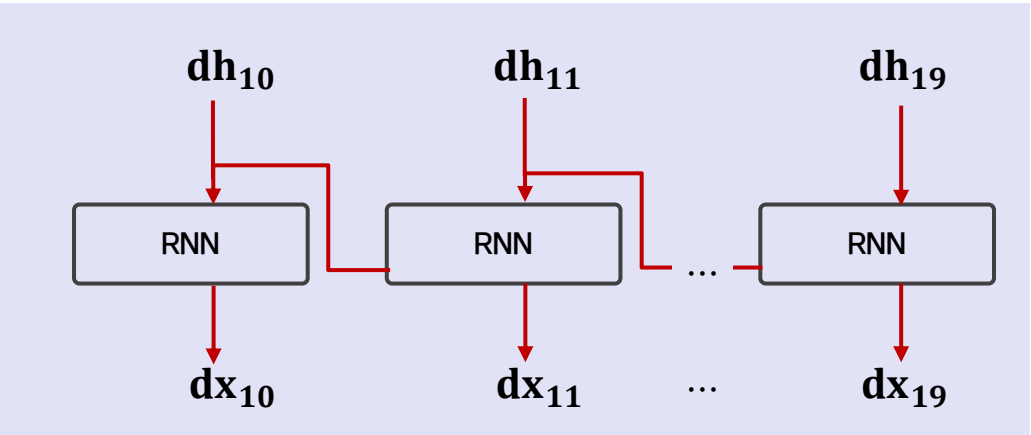


두 번째 블록의 순전파와 역전파

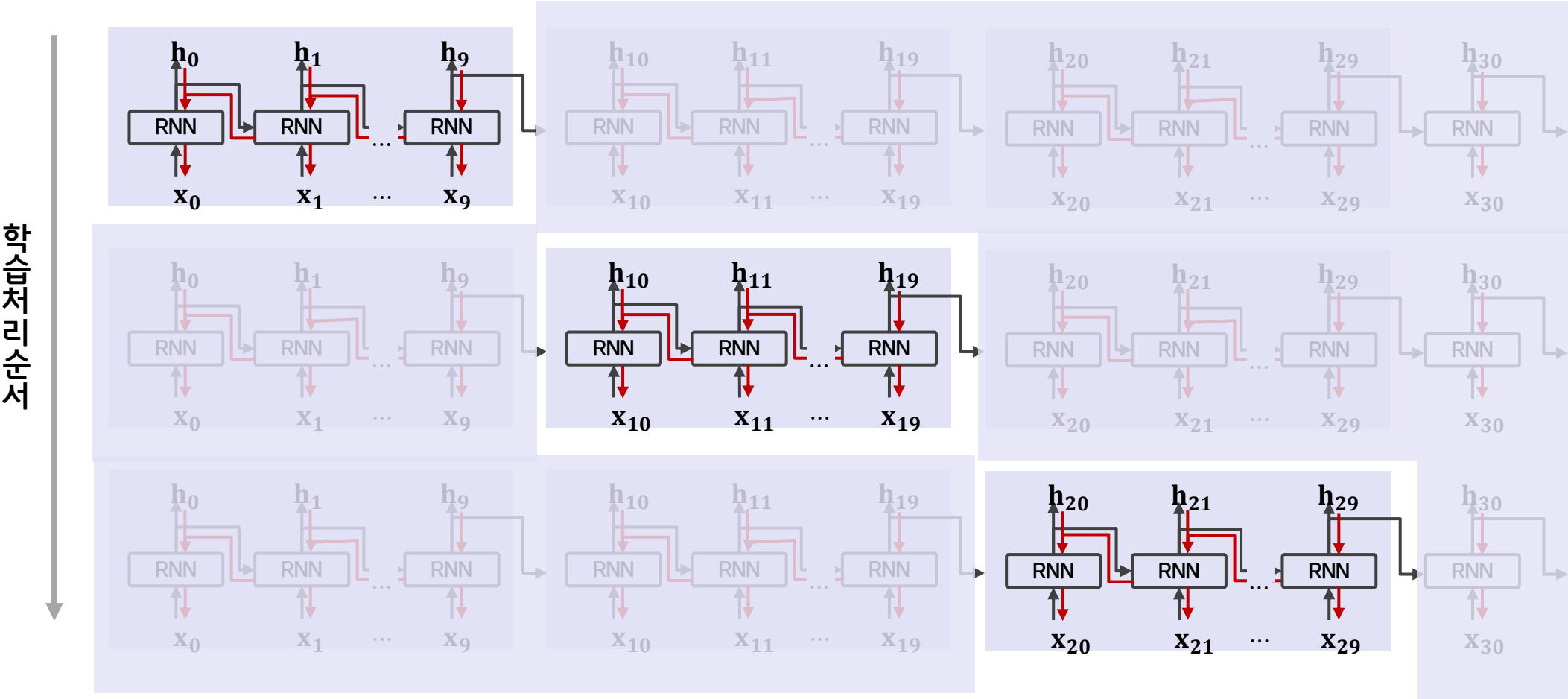
순전파



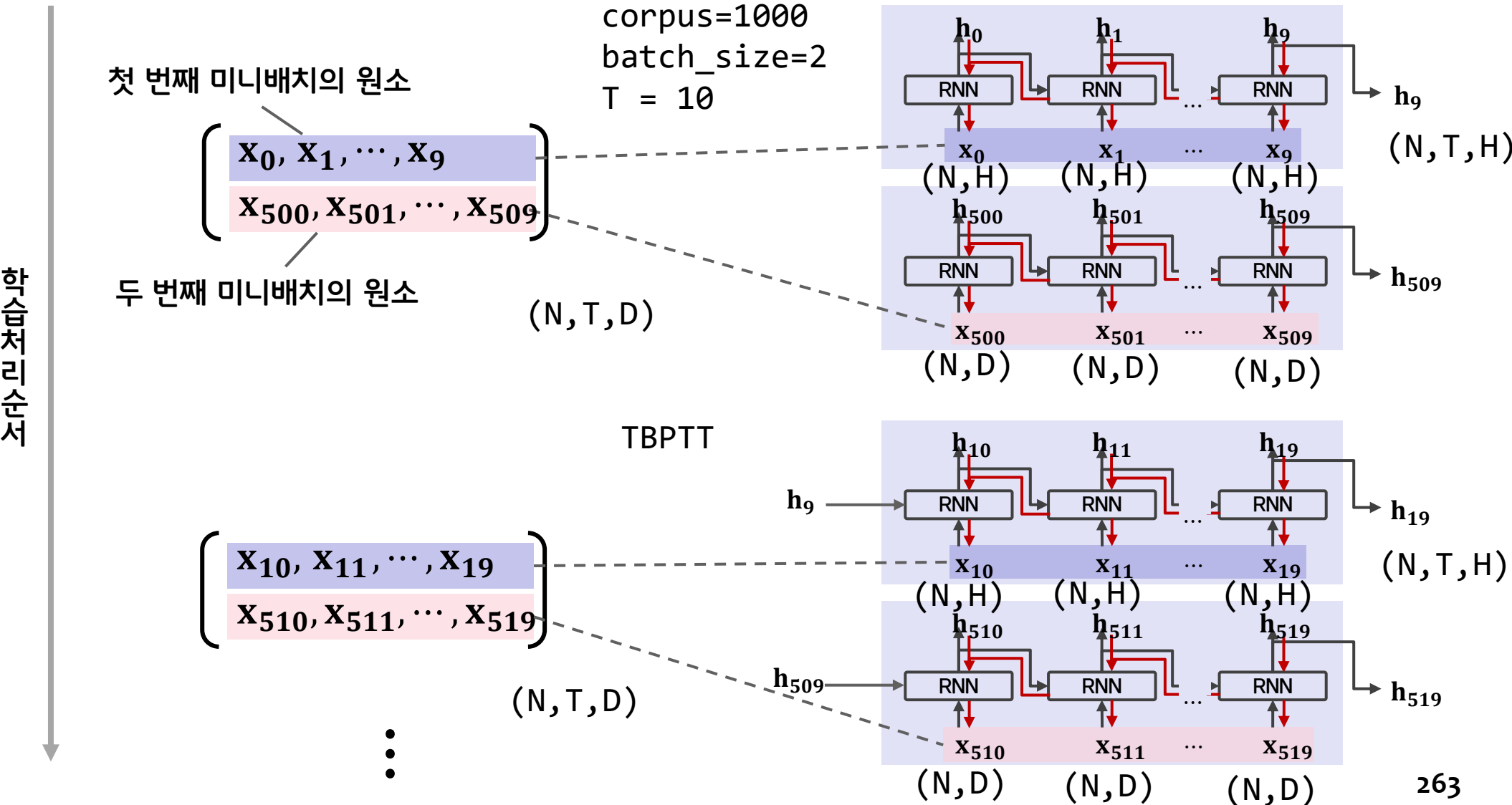
역전파



Truncated BPTT의 데이터 처리 순서



미니배치 학습 시 데이터를 제공하는 시작 위치를 각 미니배치로 옮긴다.



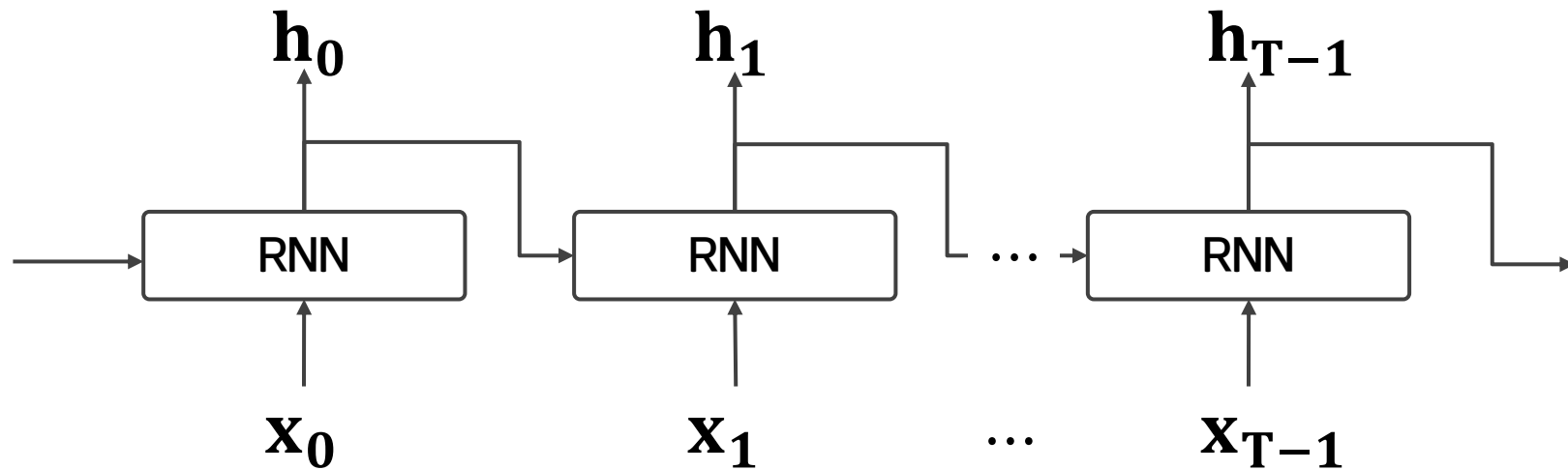
7. 순환 신경망(RNN)

7.1 RNN 이란

7.2 RNN 구현

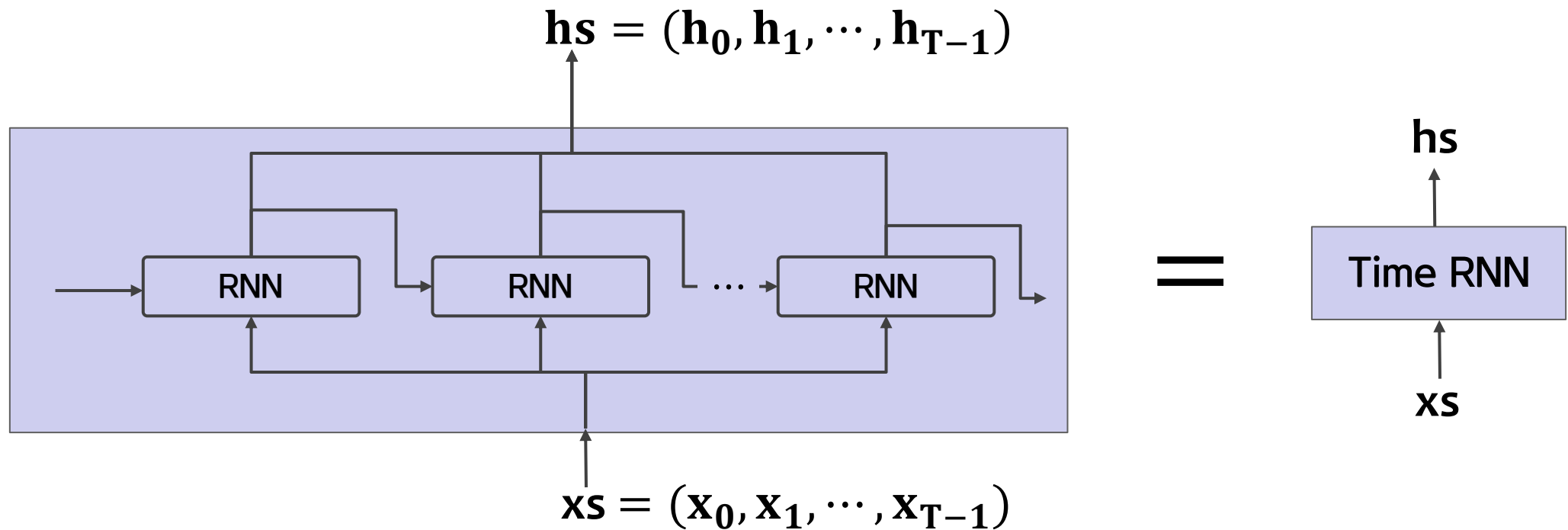
7.3 다양한 자연어 처리 모델 구현

RNN에서 다루는 신경망



길이가 T 인 시계열 데이터를 받는다.
각 시각의 은닉 상태를 T 개 출력한다.
모듈화를 생각해 위의 그림의 신경망을 '하나의 계층'으로 구현한다.

Time RNN 계층 : 순환 구조를 펼친 후의 계층들을 하나의 계층으로 간주한다.



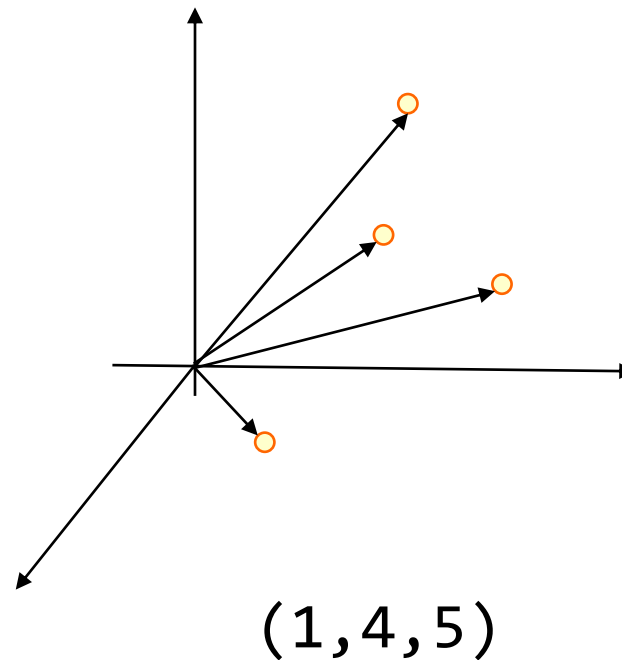
xs 를 입력하면 hs 를 출력하는 단일 계층

Time RNN계층 내에서 한 단계의 작업을 수행하는 계층을 'RNN계층'이라 하고
T개 단계분의 작업을 한꺼번에 처리하는 계층을 'Time RNN계층'이라 한다.

word = 'student'

입력 벡터의 차원수

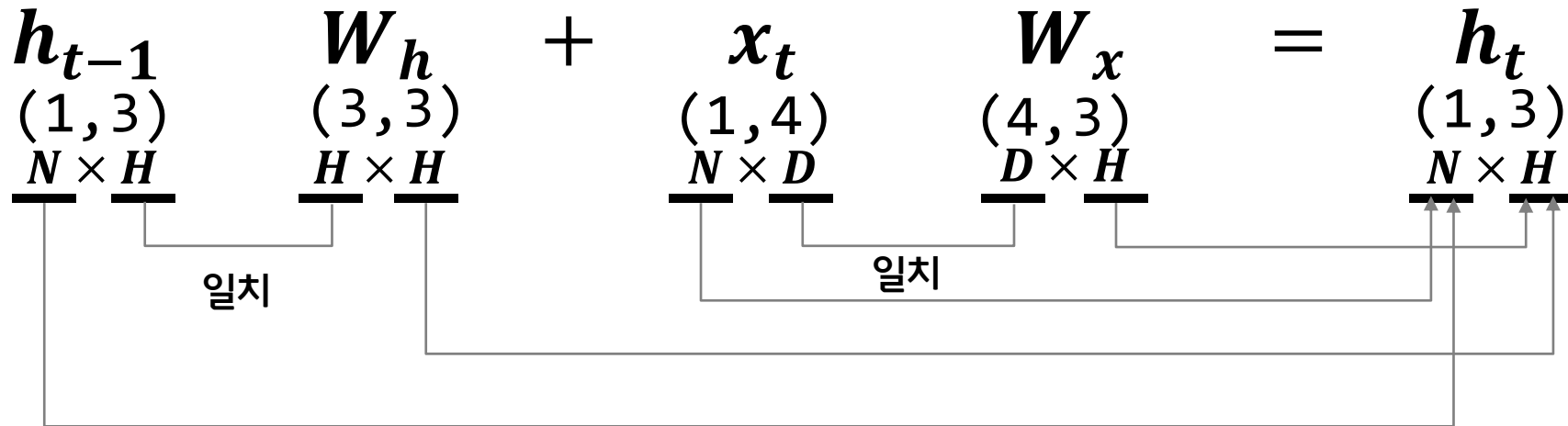
word_to_vector => embedding 층



512 차원

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

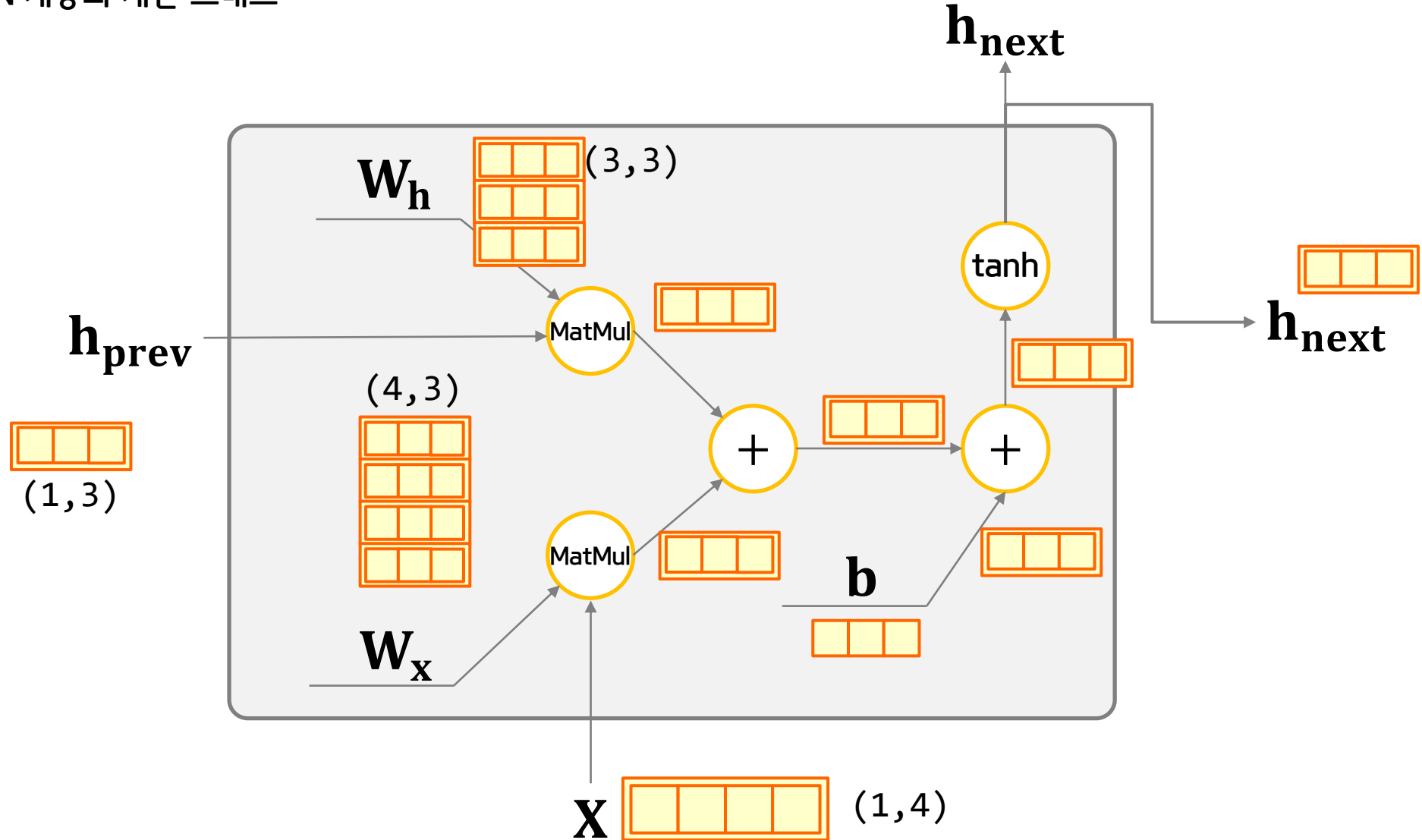
형상 확인 : 형렬 곱에서는 대응하는 차원의 원소 수를 일치시킨다.



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

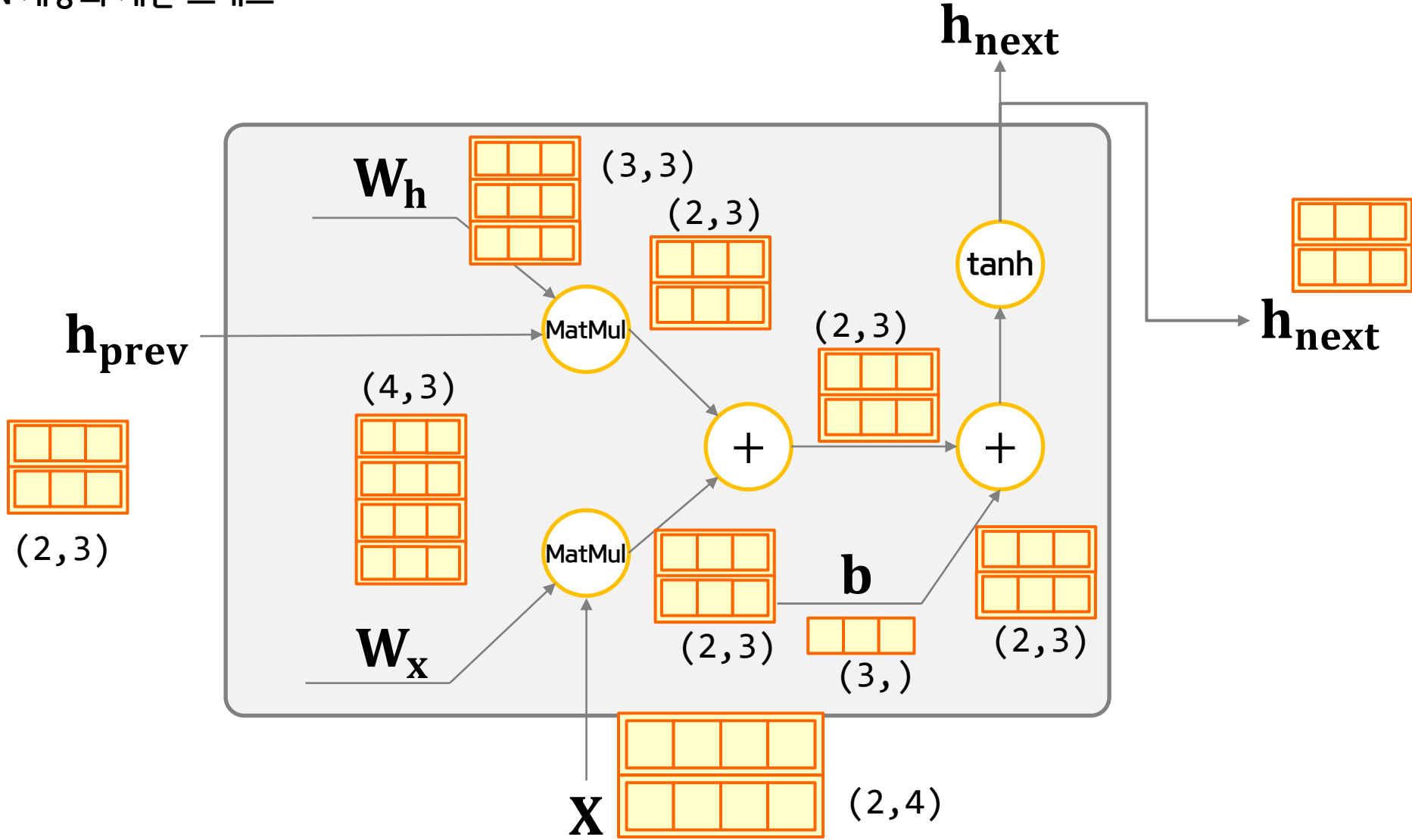
RNN 계층의 계산 그래프

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



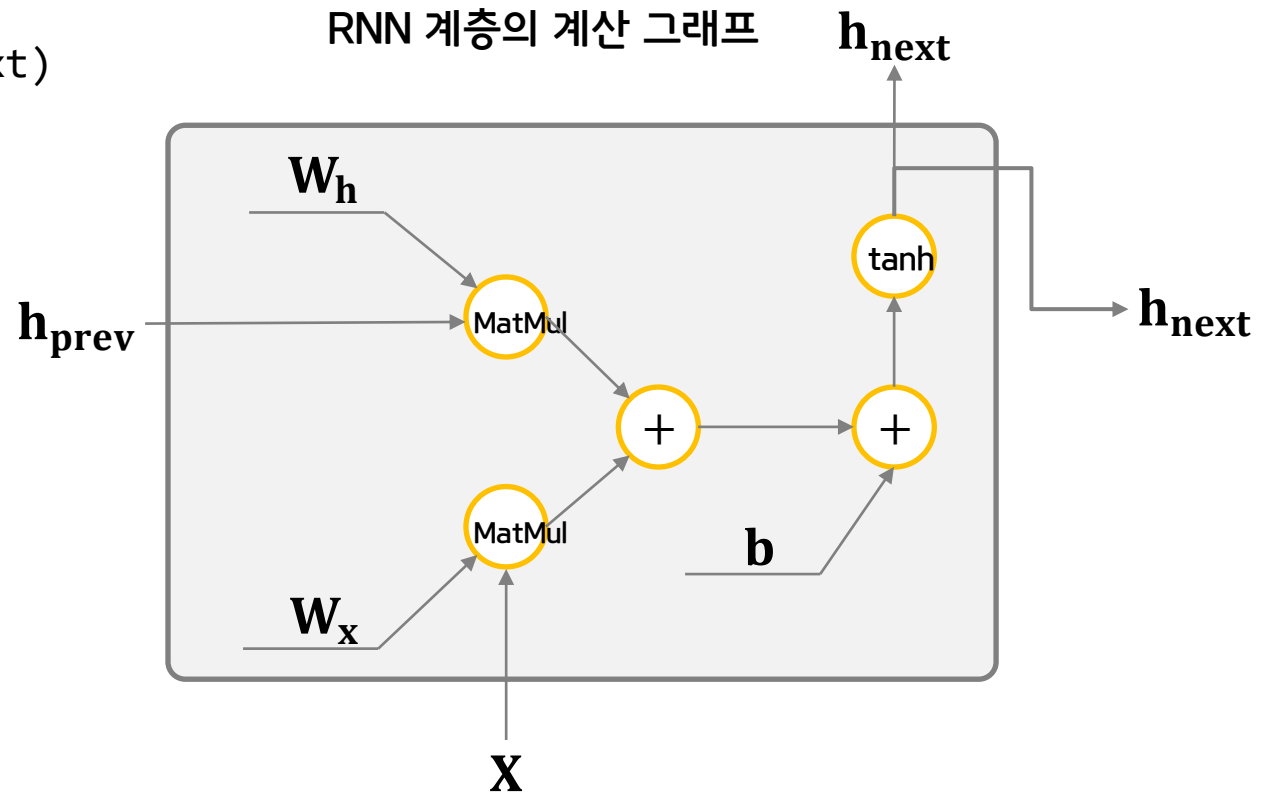
RNN 계층의 계산 그래프

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

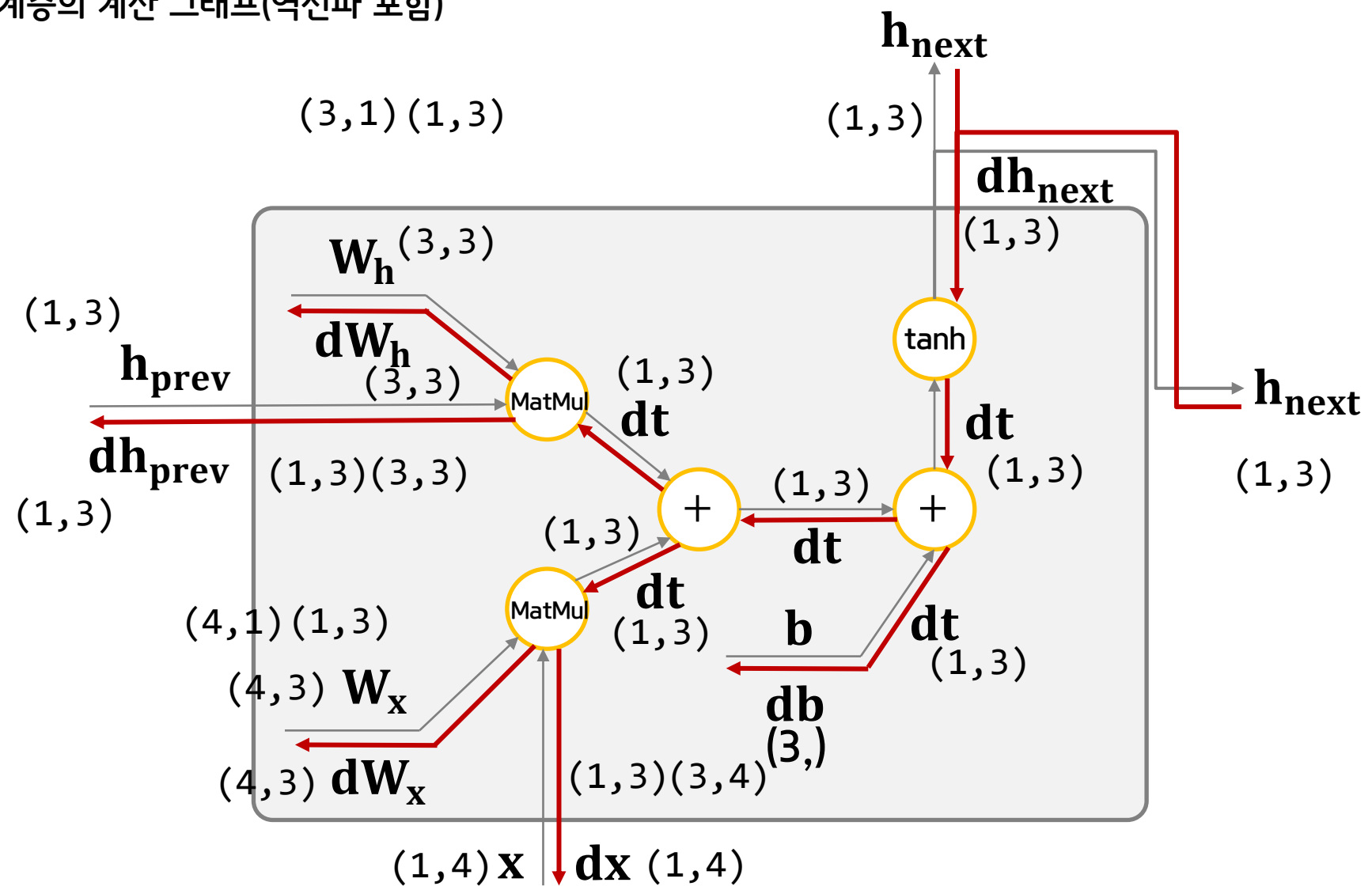


```
def forward(self, x, h_prev):  
    Wx, Wh, b = self.params  
    t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b  
    h_next = np.tanh(t)
```

```
self.cache = (x, h_prev, h_next)  
return h_next
```



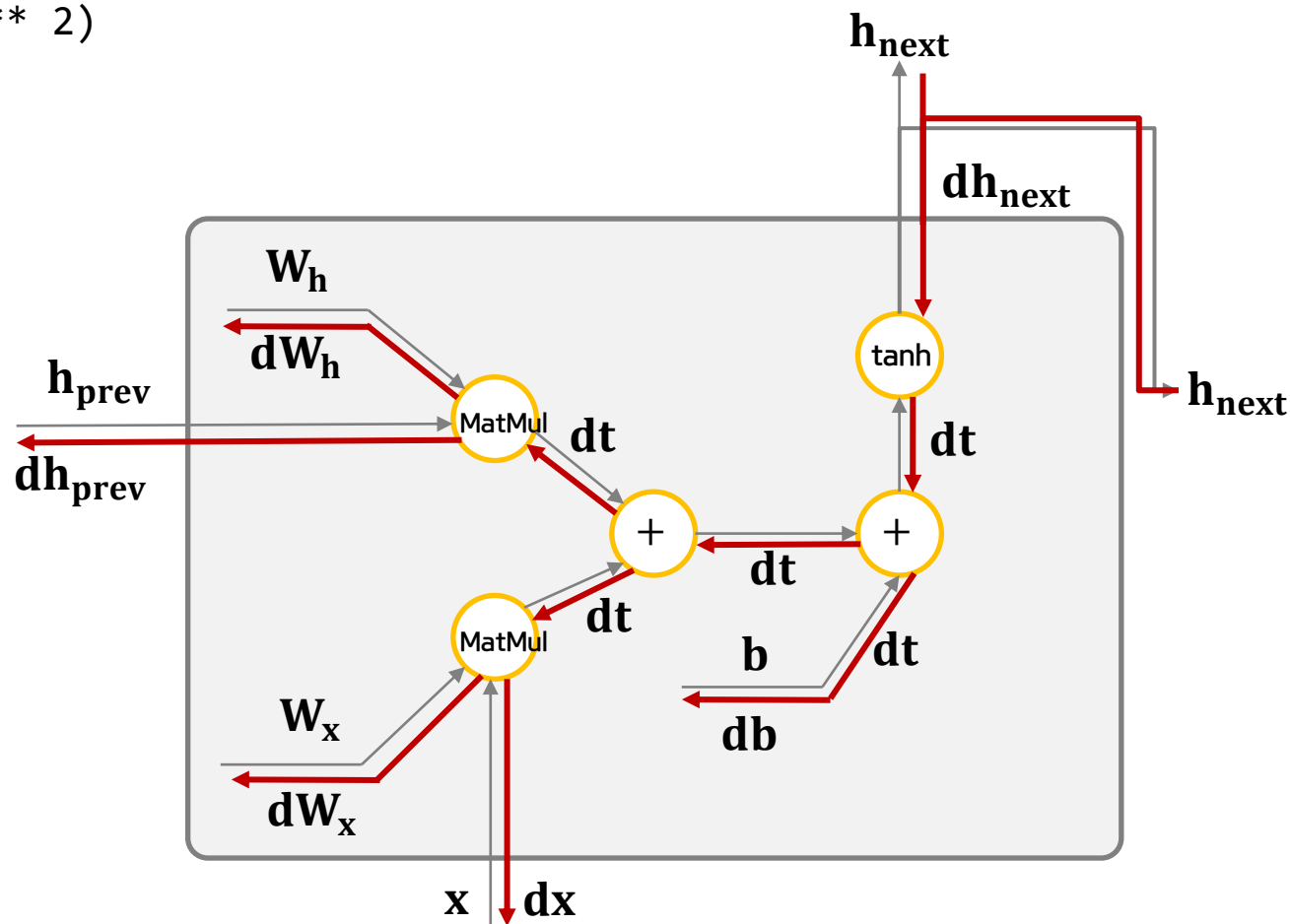
RNN 계층의 계산 그래프(역전파 포함)




```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.dot(h_prev.T, dt)
    dh_prev = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)
```

RNN 계층의 계산 그래프(역전파 포함)



소스참조

7. 순환 신경망(RNN)

7.1 RNN 이란

7.2 RNN 구현

7.3 자연어 처리 모델 구현

소스참조