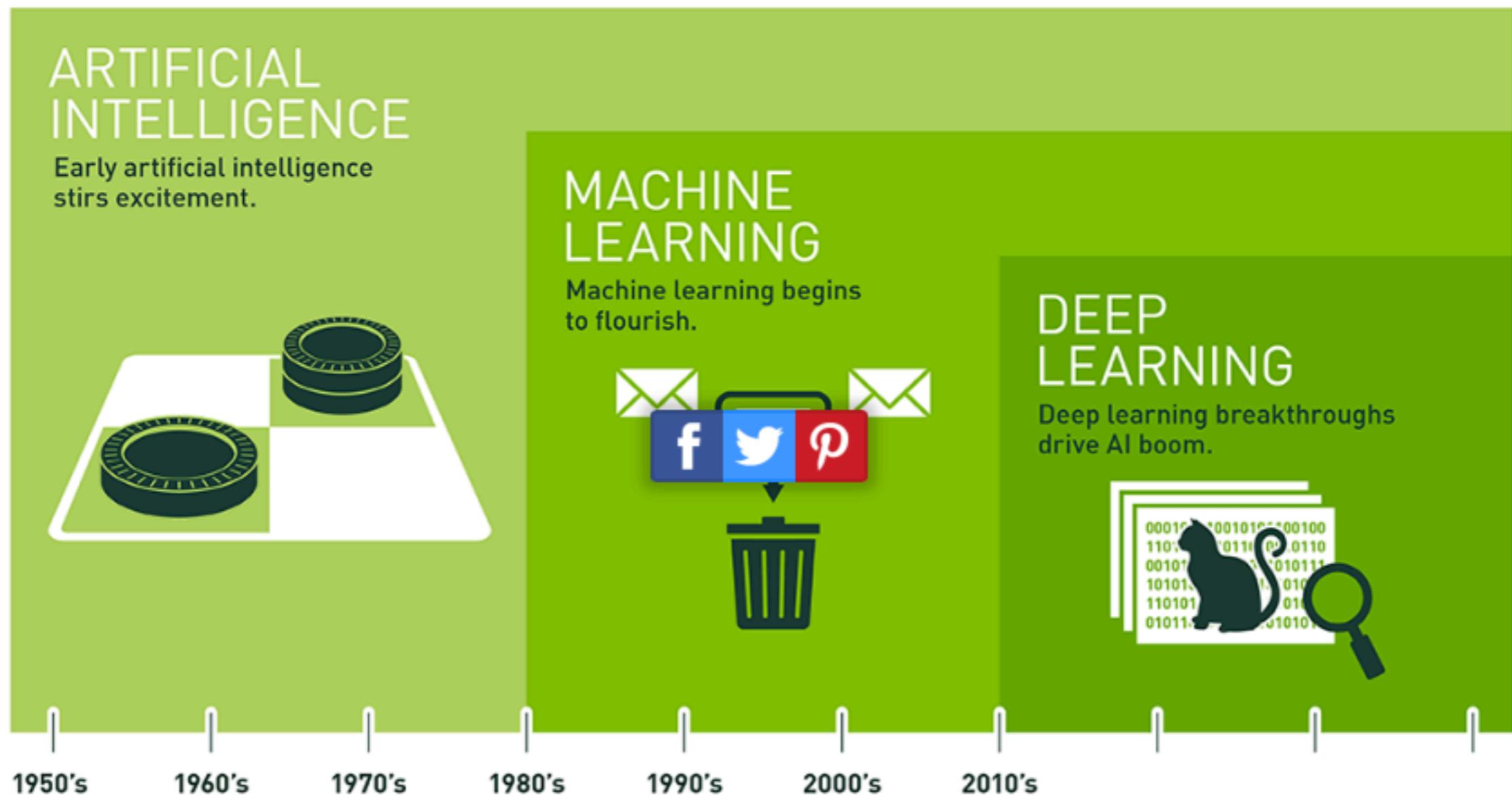


Neural Network

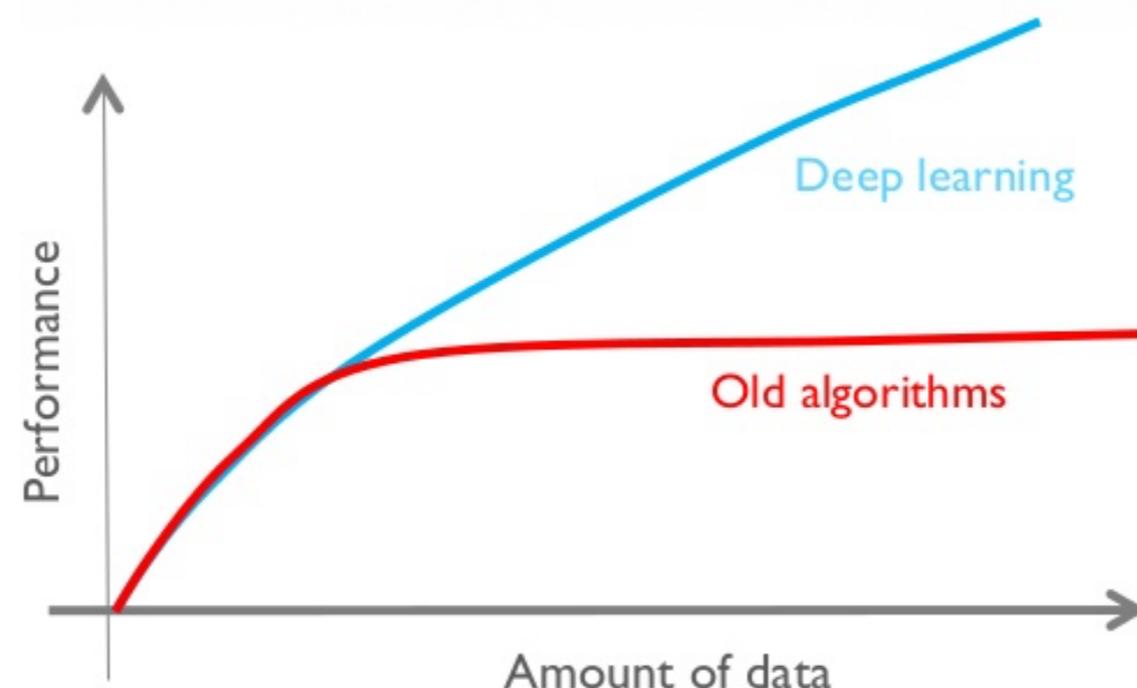
안남혁

머신러닝? 딥러닝?



딥러닝

- 머신러닝의 한 갈래로, 주로 **뉴럴 네트워크** 구조를 사용
- 왜 딥러닝이 인기있나요?
 - 사실 뉴럴 네트워크는 매우 오래된 구조
 - 최근 딥러닝의 발전은 아래 3가지 요소에 의해 되었다고 볼 수 있음
 - **방대한 데이터, 높은 컴퓨터 성능과 개선된 학습방법**

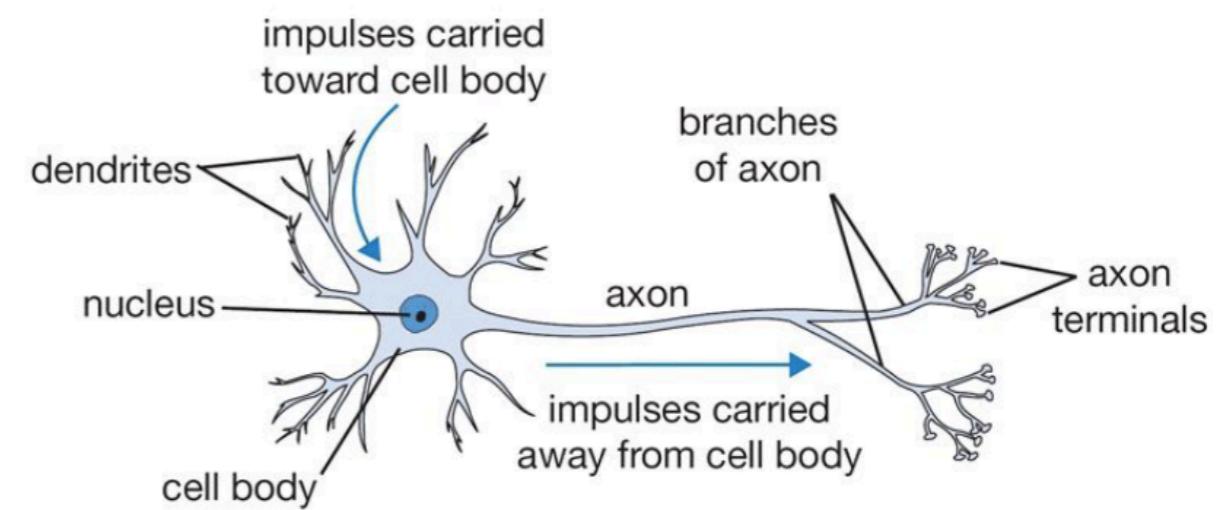
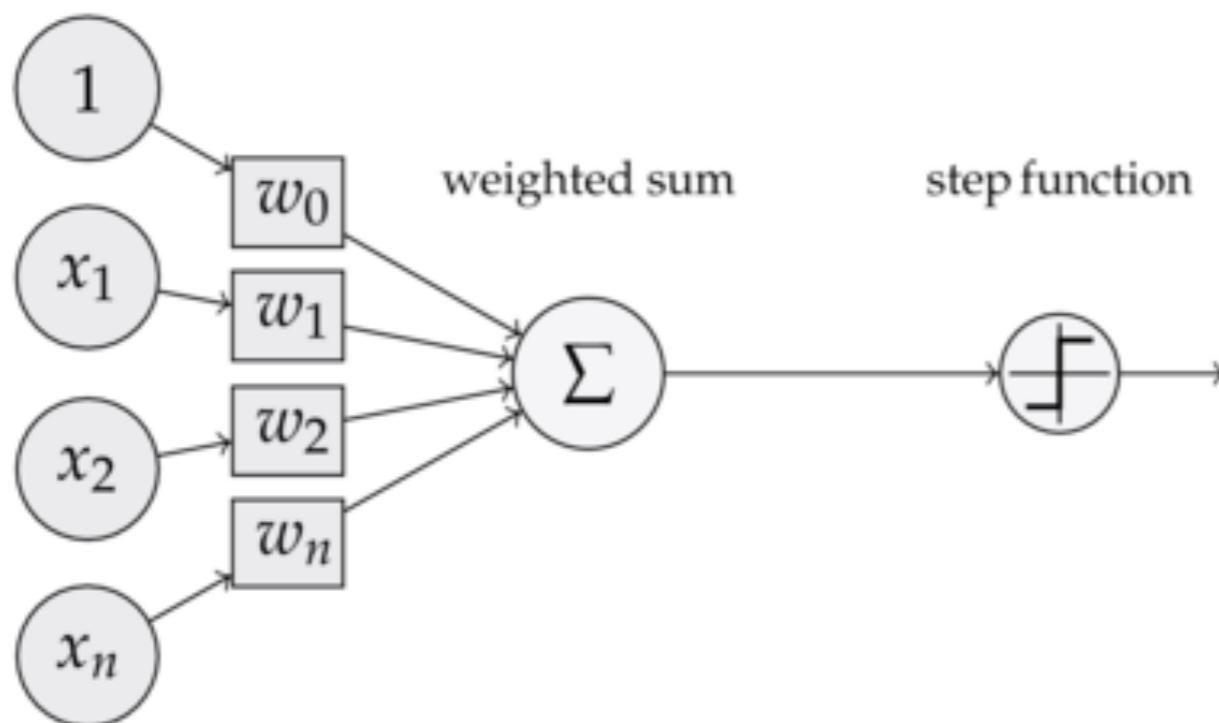


뉴럴 네트워크의 역사

- 퍼셉트론 (1958)
- XOR 문제 (1969)
- Back-propagation (1970~80s)
- LeNet (1990s)
- AlexNet (2012)

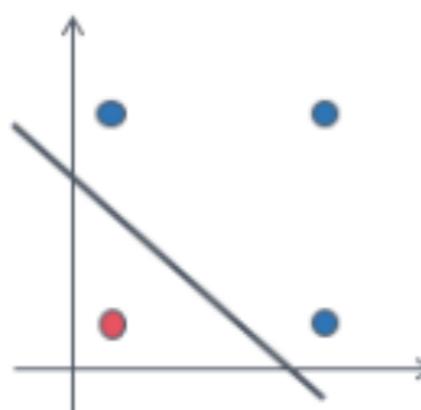
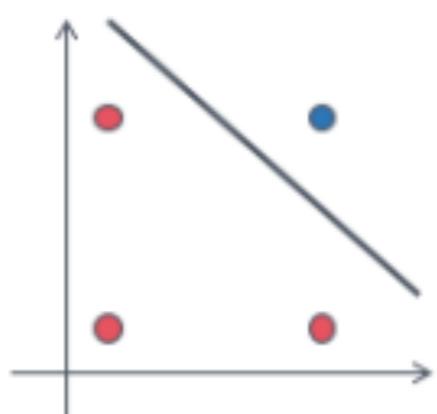
퍼셉트론

- 인간의 신경(뉴런)을 모방한 머신러닝 알고리즘
(그래서 인공신경망이라고 불림)
 - x_1, \dots, x_n 이 입력으로 들어오면, 이를 $w_1 \dots w_n$ 과 곱한 뒤, 덧셈
 - 덧셈 결과는 활성함수 (계단 함수)에 의해 0 혹은 1로 결과값이 나옴
 - 활성함수를 시그모이드 함수로 바꾸면 로지스틱 회귀와 유사



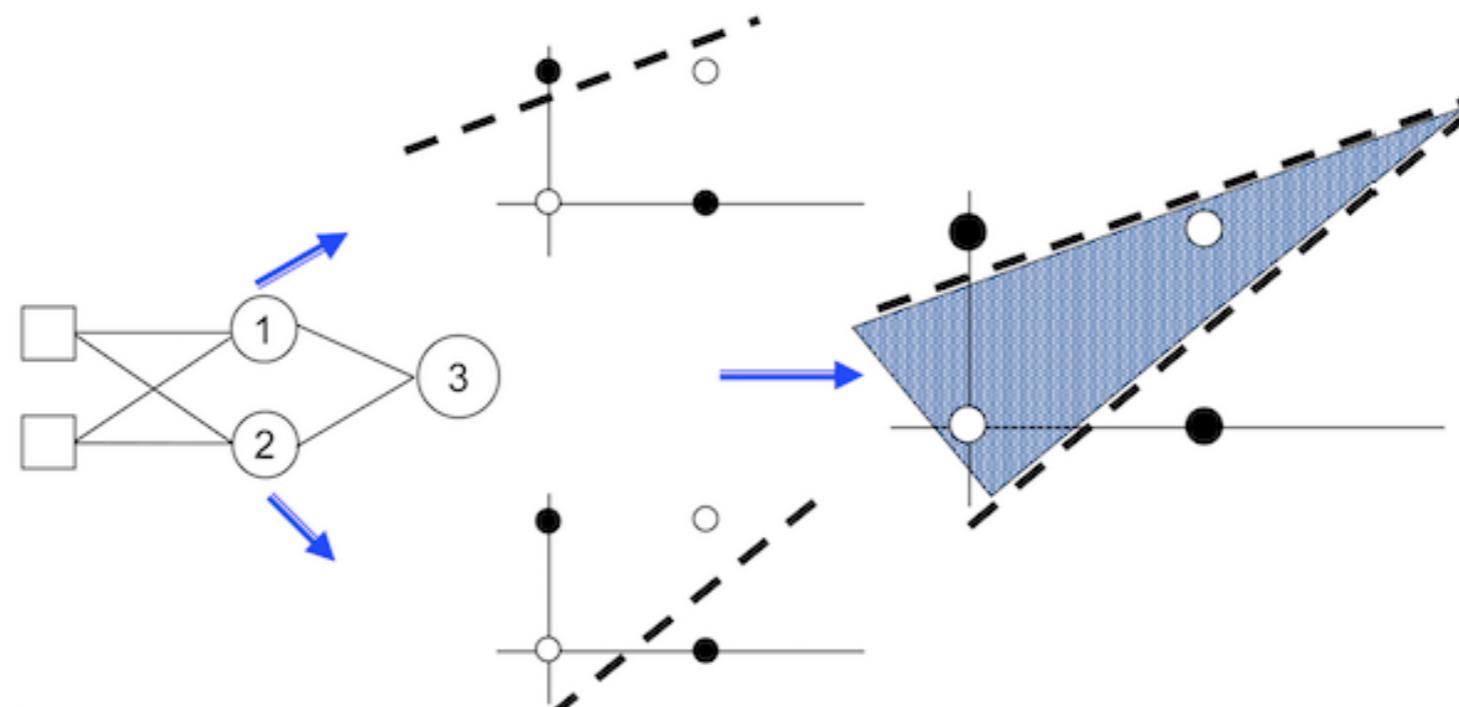
그러나..

- 퍼셉트론은 XOR 문제도 못푼다고 알려짐 (마빈 민스키, 1969)
 - 이는 퍼셉트론이 **선형분리** (linearly separable) 문제만 풀 수 있기 때문
 - 쉽게 말해, 선 하나로 데이터를 분류할 수 있으면 선형 분리
 - 더군다나 퍼셉트론을 학습할 방법도 없었음
- 뉴럴 네트워크 분야의 첫 **빙하기**



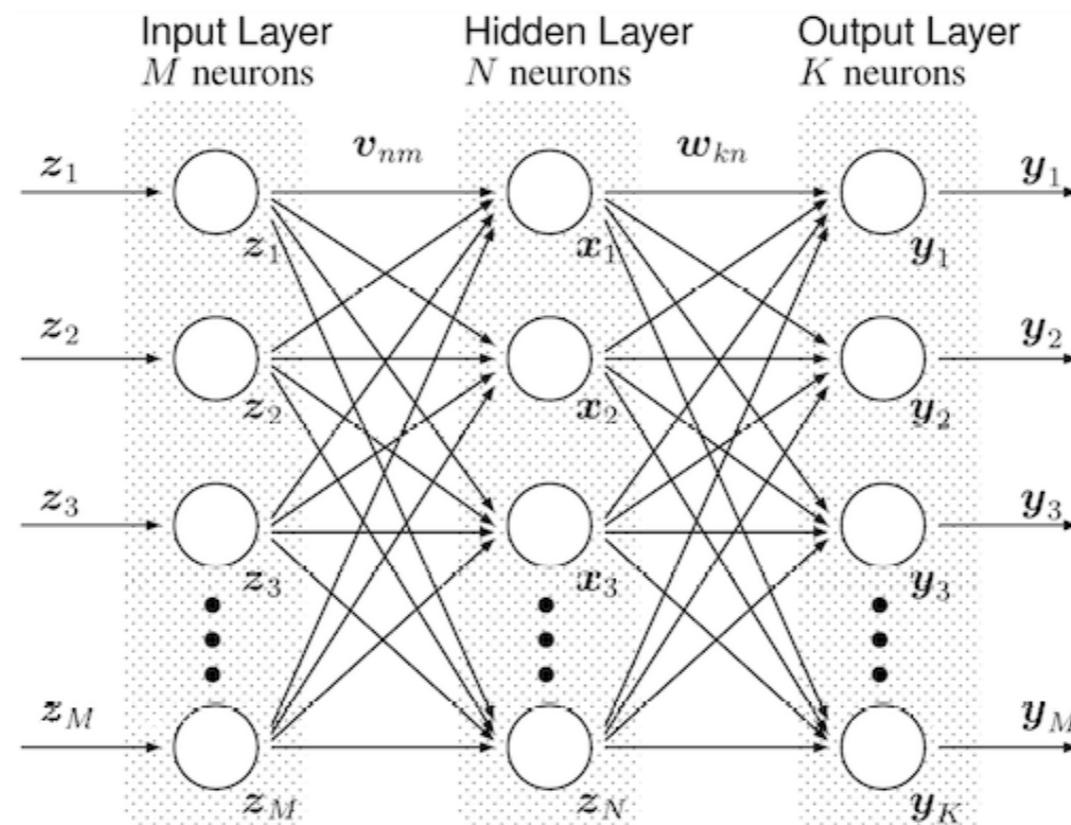
해결책

- XOR 문제
 - 퍼셉트론을 여러층 쌓으면 해결 → **멀티 레이어 퍼셉트론 (MLP)**
- 학습 문제
 - **Back-propagation** 알고리즘 등장 (1970~80s)



멀티 레이어 퍼셉트론

- 말 그대로 층이 여러개인 퍼셉트론 구조
 - **입력층, 은닉층과 출력층**으로 이루어짐
 - 아래 구조는 은닉층이 1개인 MLP
 - 퍼셉트론과 동일하게 활성함수가 레이어마다 존재

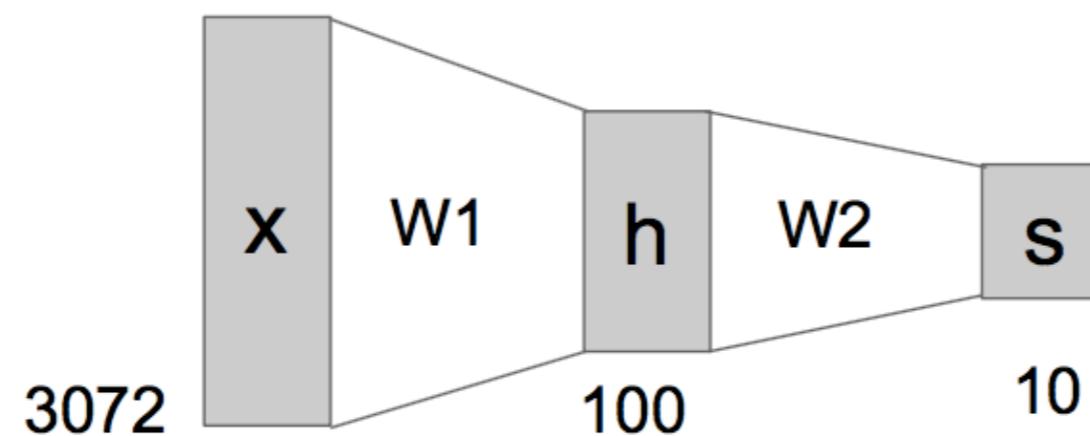


정리하면,

- 선형 함수: $f = Wx$
- 2층 MLP (뉴럴 네트워크): $f = W_2 \max(0, W_1x)$

- ...

- ...



참고: weight / bias

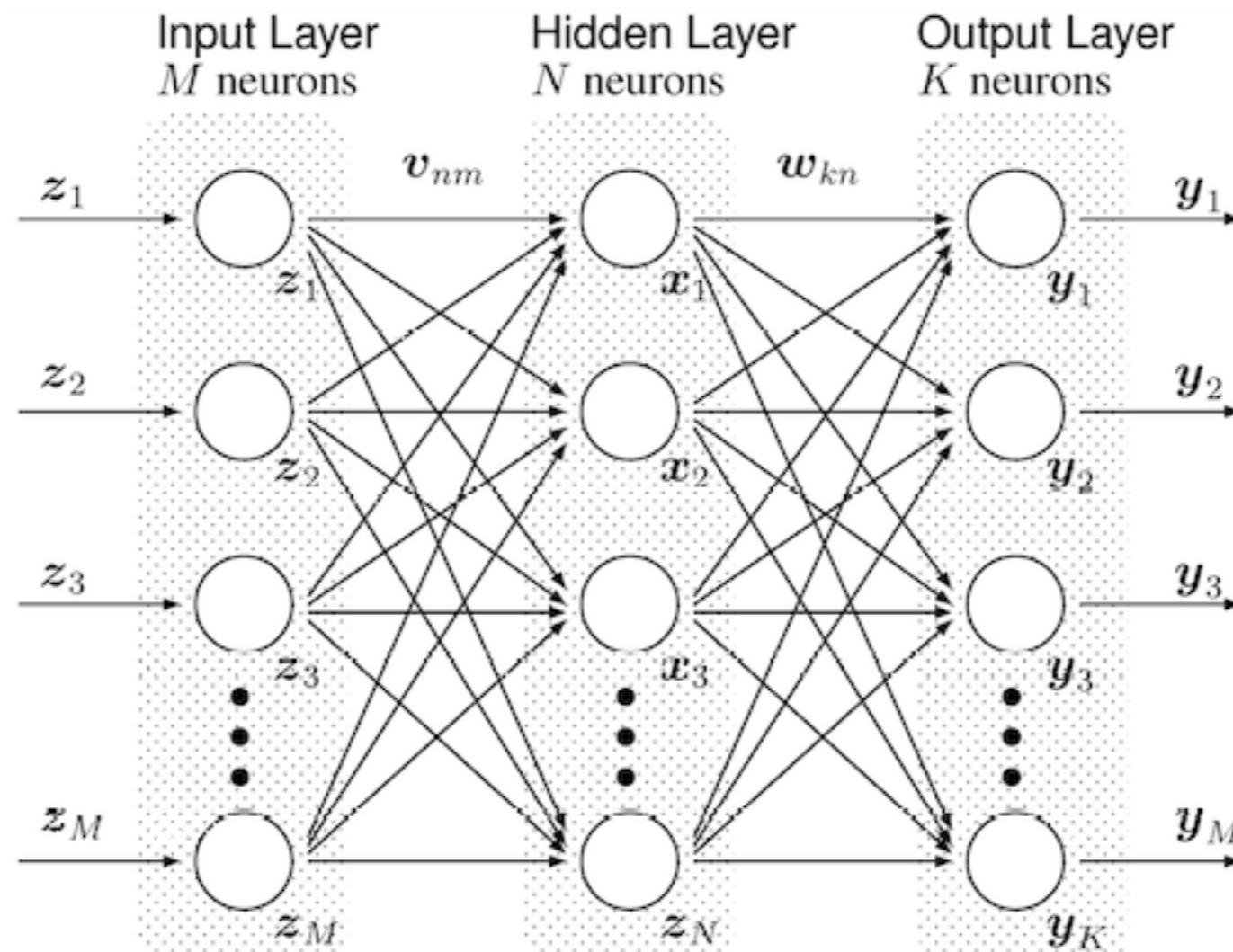
- $f = Wx + b$
- weight는 W 를, bias는 b 를 의미
- weight와 bias를 합쳐서 **파라미터**

뉴럴 네트워크의 학습 방법

- **Forward-propagation**
 - 입력값이 주어지면 NN을 통과시켜 생성되는 최종 **score**와 **loss** 출력
- **Backward-propagation**
 - 도출된 loss를 이용해 레이어를 **거꾸로** 순회하면서 NN 학습
- 네트워크의 학습은 **함수 최적화**의 방법에 따라 진행
 - Backward-propagation시 gradient descent 알고리즘을 사용하여 학습

뉴럴 네트워크의 학습 방법

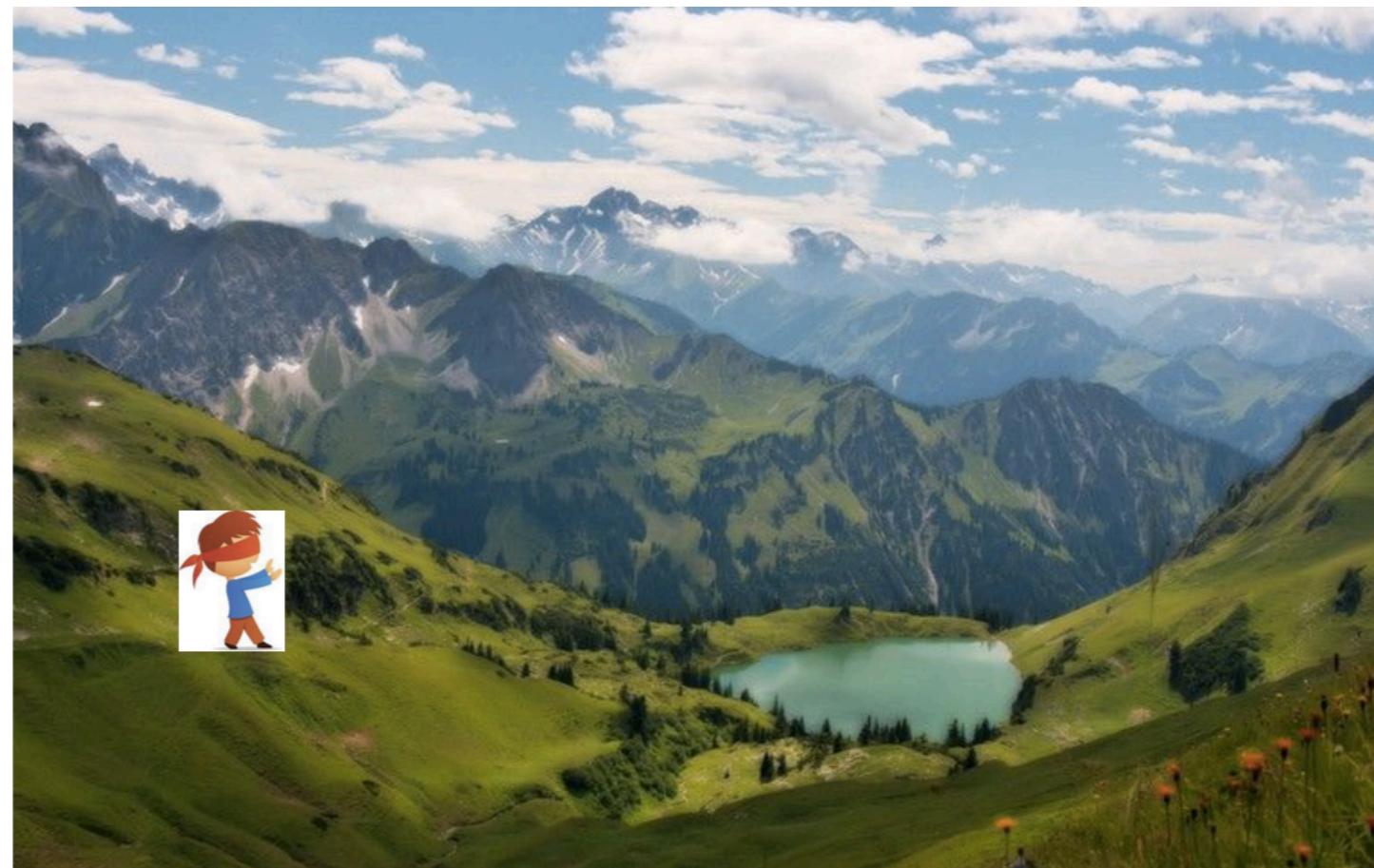
2. Backward-propagation



1. Forward-propagation

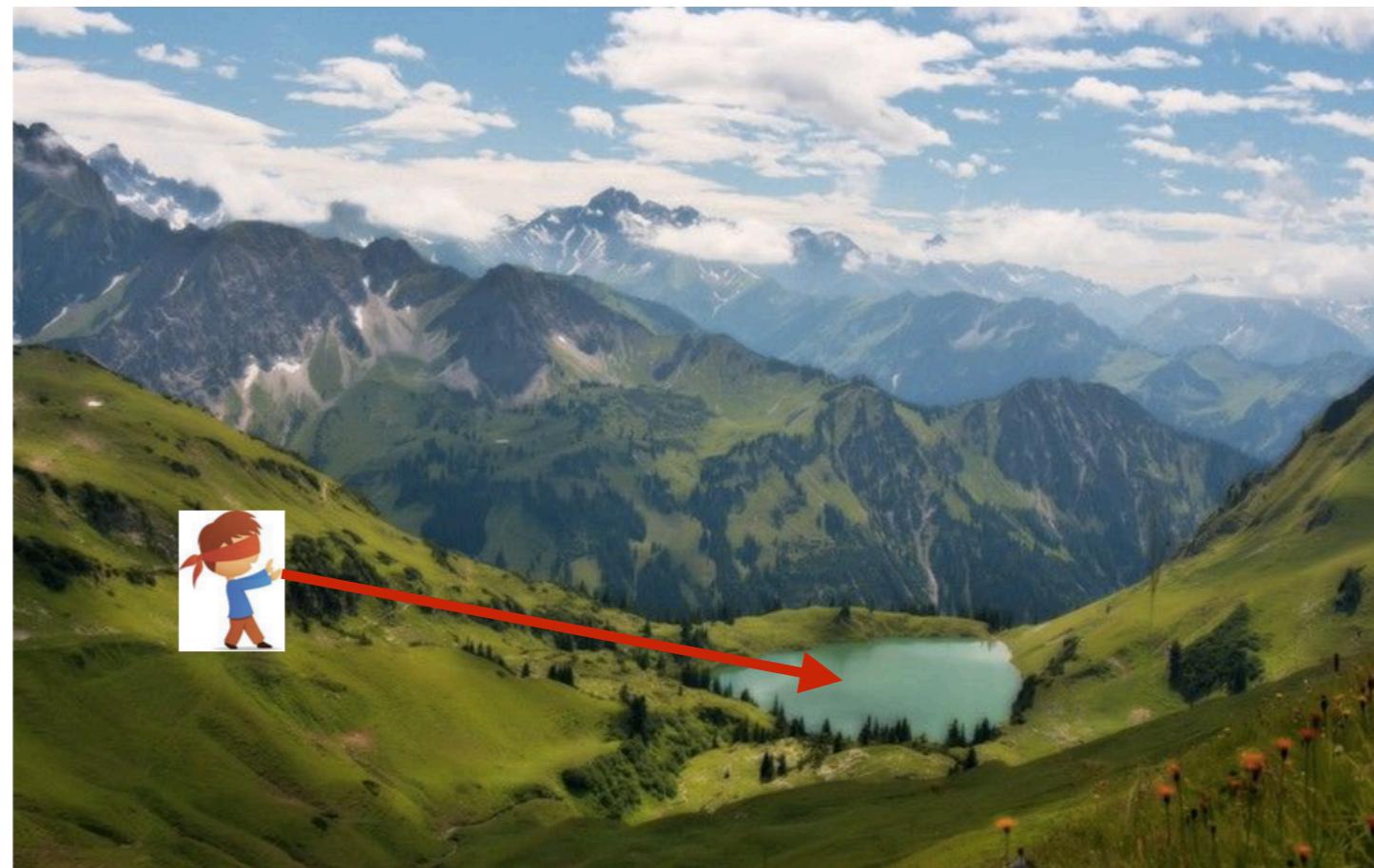
함수 최적화

- 뉴럴 네트워크의 학습은 **함수 최적화**의 문제
 - 직관적으로 “산에서 가장 낮은 곳을 찾아 내려가는” 문제로 생각
 - **그라디언트**를 통해 기울기가 가장 급격한 방향을 찾고, 그라디언트 방향으로 움직이기



함수 최적화

- 뉴럴 네트워크의 학습은 **함수 최적화**의 문제
 - 직관적으로 “산에서 가장 낮은 곳을 찾아 내려가는” 문제로 생각
 - **그라디언트**를 통해 기울기가 가장 급격한 방향을 찾고, 그라디언트 방향으로 움직이기



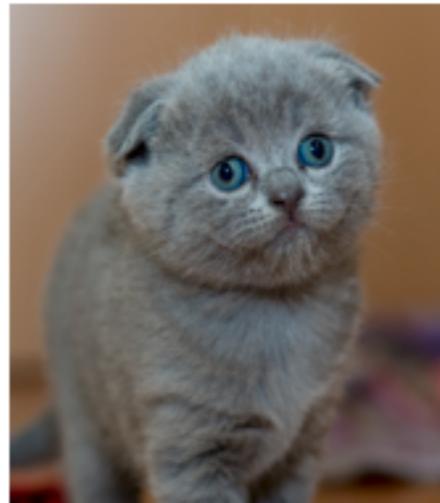
Gradient descent (경사 하강법)

- Gradient descent / ascent
 - 그라디언트를 구한 후 그라디언트 방향으로 움직이는 알고리즘
 - 최소값으로 향하면 descent, 최대값으로 향하면 ascent
 - 일반적으로 뉴럴 네트워크의 학습은 gradient descent 사용
- 그래서,
 - 최적화 하고 싶은 함수가 무엇인가? -> score와 **Loss 함수**
 - 그라디언트는 어떻게 구하나? -> **Backward propagation**

Score

- 뉴럴 네트워크를 통해 출력된 값

$$f = W_2 \max(0, W_1 x)$$



고양이

6.5

2.3

3.3

강아지

2.0

7.1

-1.5

사자

3.3

-3.5

4.3

LOSS

- 현재 학습된 모델이 얼마나 **좋은** 모델인지 알려주는 척도
- 뉴럴 네트워크를 통해 얻은 score를 변형하여 계산
 1. Softmax 적용
 2. Negative log-likelihood 계산
- 계산된 **loss를 최소화** 하는 방향으로 모델을 학습하게 됨

Softmax

- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화
- $$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$
- 이 때, $s_i = f(x_i; W)$ 는 score 함수

Softmax

- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화



$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

이 때, $s_i = f(x_i; W)$ 는 score 함수

고양이

6.5

강아지

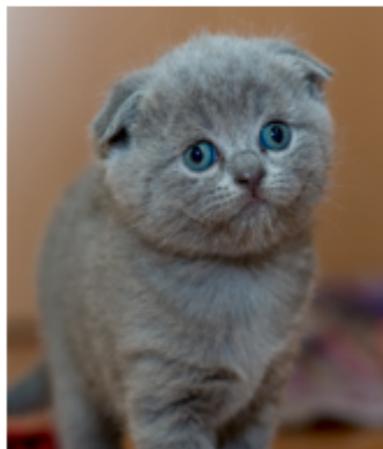
2.0

사자

3.3

Softmax

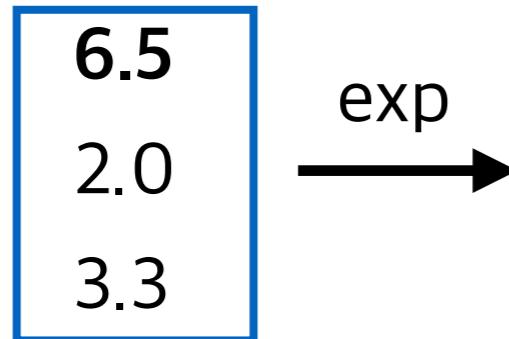
- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화



$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

이 때, $s_i = f(x_i; W)$ 는 score 함수

고양이
강아지
사자



Softmax

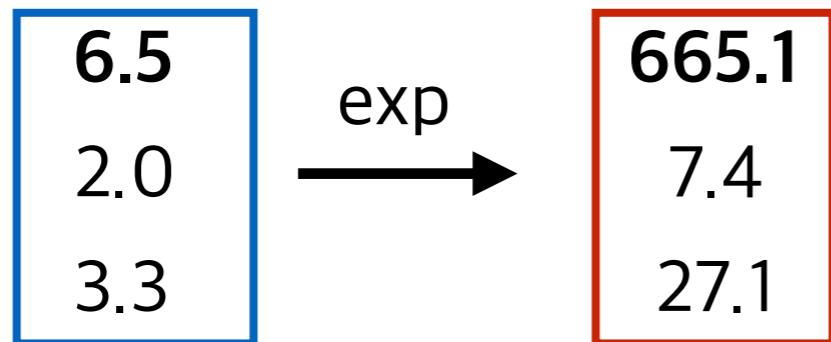
- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화



$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

이 때, $s_i = f(x_i; W)$ 는 score 함수

고양이
강아지
사자



Softmax

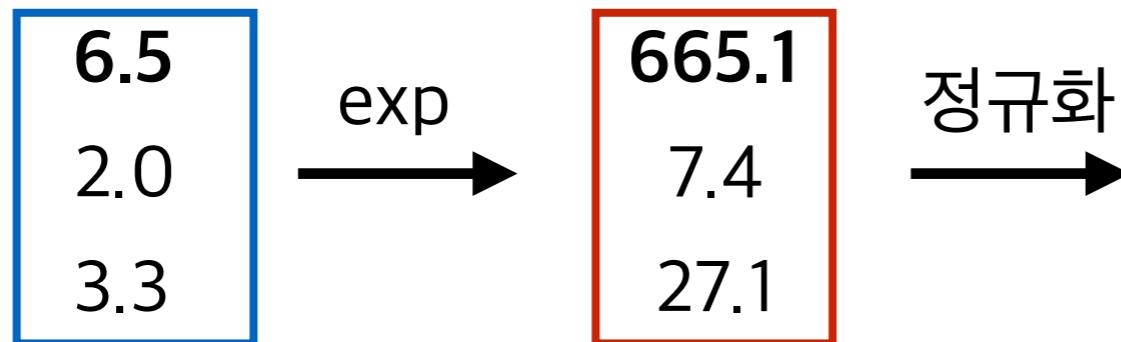
- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화



$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

이 때, $s_i = f(x_i; W)$ 는 score 함수

고양이
강아지
사자



Softmax

- 계산된 score 값을 확률값으로 변환하는 함수
 - 정답인 클래스에 해당하는 확률값이 1.0, 나머지는 0.0일 때 최상
 - $\exp \rightarrow$ 스코어가 가장 높은 클래스에 확률값을 몰아주자
(\max 연산은 미분 불가이므로 적용이 어려움)
 - 결과값의 합이 1이어야 함 \rightarrow 정규화



$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

이 때, $s_i = f(x_i; W)$ 는 score 함수

고양이
강아지
사자

6.5
2.0
3.3

\exp

665.1
7.4
27.1

정규화

0.95
0.01
0.04

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습

$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Softmax 

정답 클래스 

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

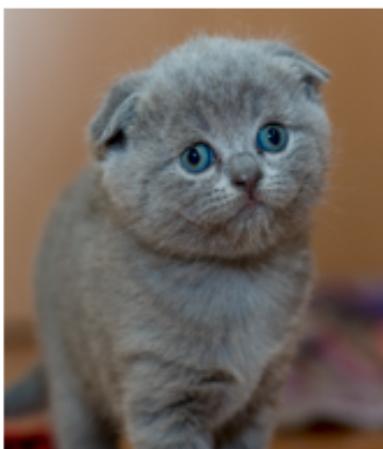
Softmax

정답 클래스

고양이	6.5
강아지	2.0
사자	3.3

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

정답 클래스
Softmax

고양이
강아지
사자

6.5
2.0
3.3

exp →

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

정답 클래스
Softmax

고양이
강아지
사자

6.5	exp	665.1
2.0		7.4
3.3		27.1

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습

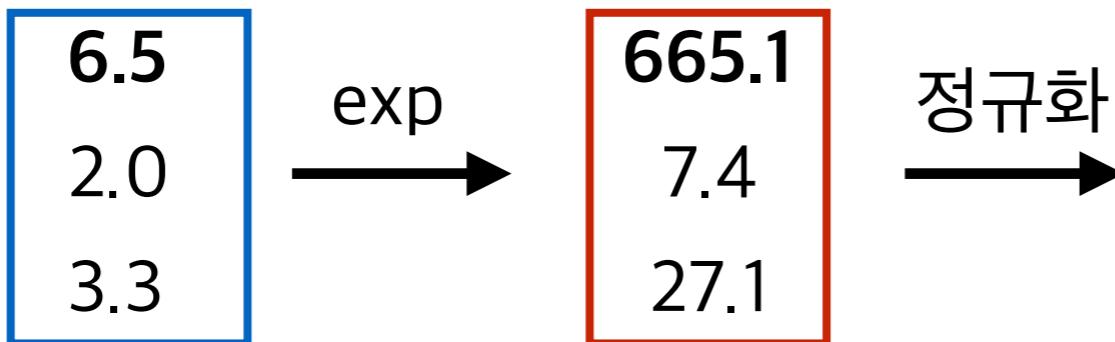


$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Softmax

정답 클래스

고양이
강아지
사자



Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Softmax

정답 클래스

고양이
강아지
사자

6.5
2.0
3.3

\exp

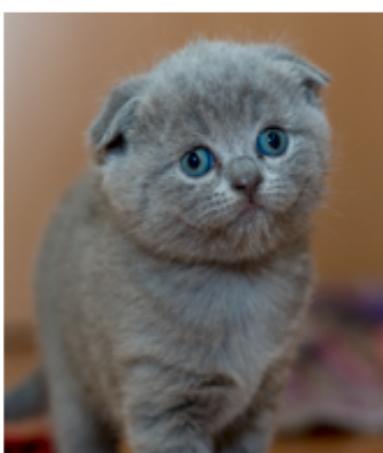
665.1
7.4
27.1

정규화

0.95
0.01
0.04

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Softmax

정답 클래스

고양이
강아지
사자

6.5
2.0
3.3

\exp

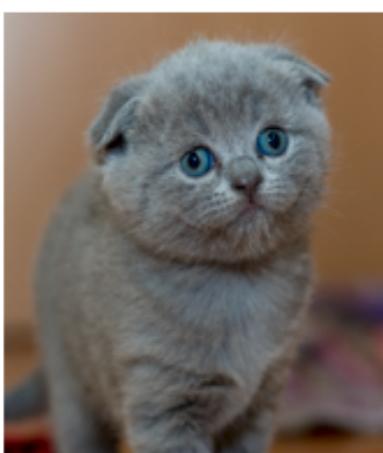
665.1
7.4
27.1

정규화

0.95
0.01
0.04

Negative log-likelihood (NLL)

- Softmax를 통해 얻은 확률값을 이용하여 loss를 계산
 - Log: 로그 함수는 단조 증가 함수이므로 취하더라도 결과는 동일, 곱셈이 덧셈으로 바뀌는 등의 연산이 편리하기 때문에 자주 사용되는 트릭
 - 계산된 NLL loss를 최소화 하는 방향으로 네트워크 학습



$$L_i = \log \left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right)$$

Softmax

정답 클래스

고양이
강아지
사자

6.5
2.0
3.3

\exp

665.1
7.4
27.1

정규화

0.95
0.01
0.04

$$\begin{aligned} -\log(0.95) \\ = 0.022 \end{aligned}$$

다른 예제



고양이
강아지
사자

$$\begin{bmatrix} 3.3 \\ -1.5 \\ 4.3 \end{bmatrix} \xrightarrow{\text{exp}} \begin{bmatrix} 27.1 \\ 0.2 \\ 73.7 \end{bmatrix}$$

정규화

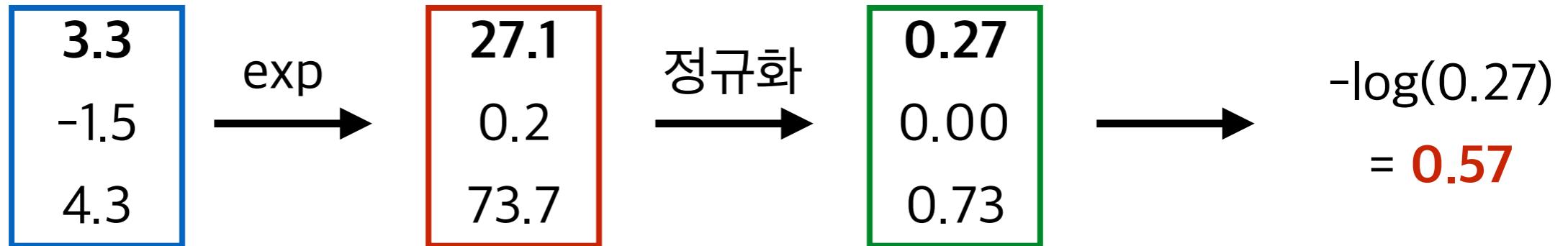
$$\begin{bmatrix} 0.27 \\ 0.00 \\ 0.73 \end{bmatrix}$$

$$\begin{aligned} -\log(0.27) \\ = 0.57 \end{aligned}$$

다른 예제



고양이
강아지
사자

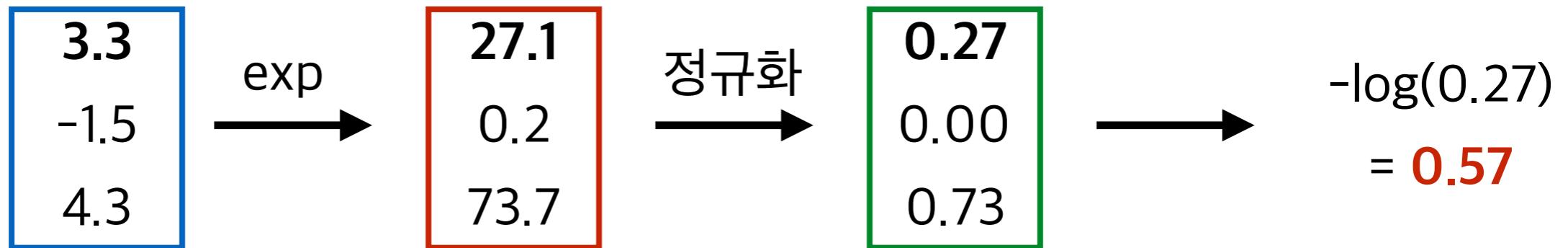


- 고양이 클래스에 해당하는 확률이 1.0 \rightarrow loss?

다른 예제



고양이
강아지
사자

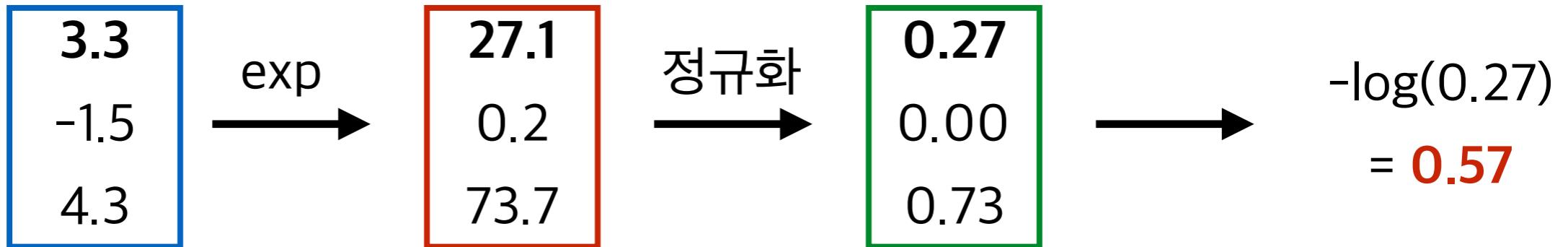


- 고양이 클래스에 해당하는 확률이 1.0 \rightarrow loss? **0.0**

다른 예제



고양이
강아지
사자

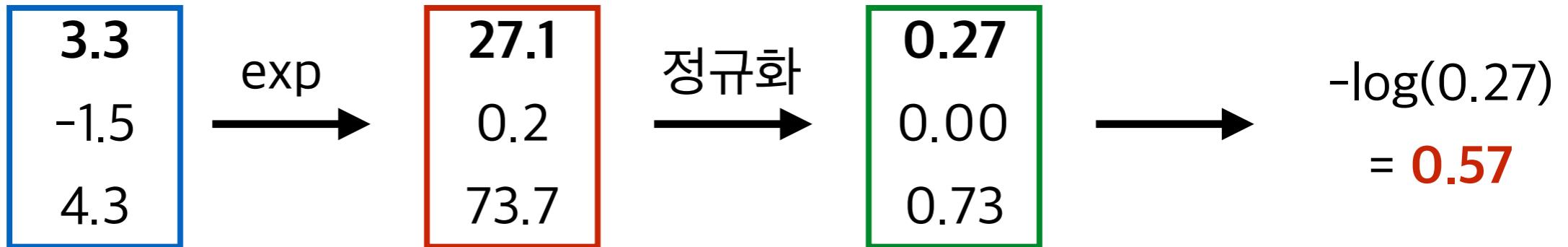


- 고양이 클래스에 해당하는 확률이 1.0 -> loss? **0.0**
- 고양이 클래스에 해당하는 확률이 0.0 -> loss?

다른 예제



고양이
강아지
사자

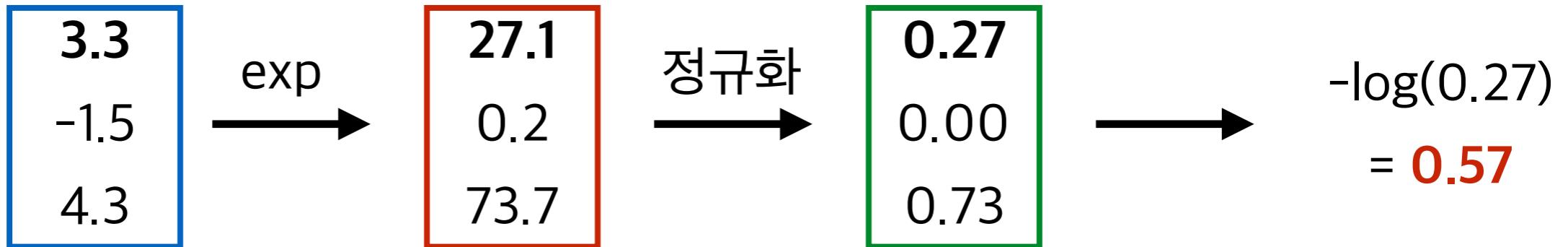


- 고양이 클래스에 해당하는 확률이 1.0 -> loss? **0.0**
- 고양이 클래스에 해당하는 확률이 0.0 -> loss? **정의되지 않음**

다른 예제



고양이
강아지
사자



- 고양이 클래스에 해당하는 확률이 1.0 -> loss? **0.0**
- 고양이 클래스에 해당하는 확률이 0.0 -> loss? **정의되지 않음**
 - TensorFlow에서는 NaN 에러를 냄 (Not a number)
 - 따라서 안정성을 위해 log를 취하기 전 매우 작은 값을 더해주는 것이 일반적

Cross-entropy loss

- Softmax와 NLL loss를 크로스 엔트로피 loss라고 부름
- 참고: 데이터 클래스 개수가 2개일 때 (이진 분류)
 - Softmax 대신 시그모이드 함수를 사용하여 확률 예측
 - 이진 크로스 엔트로피 (Binary cross-entropy)

Forward/backward propagation

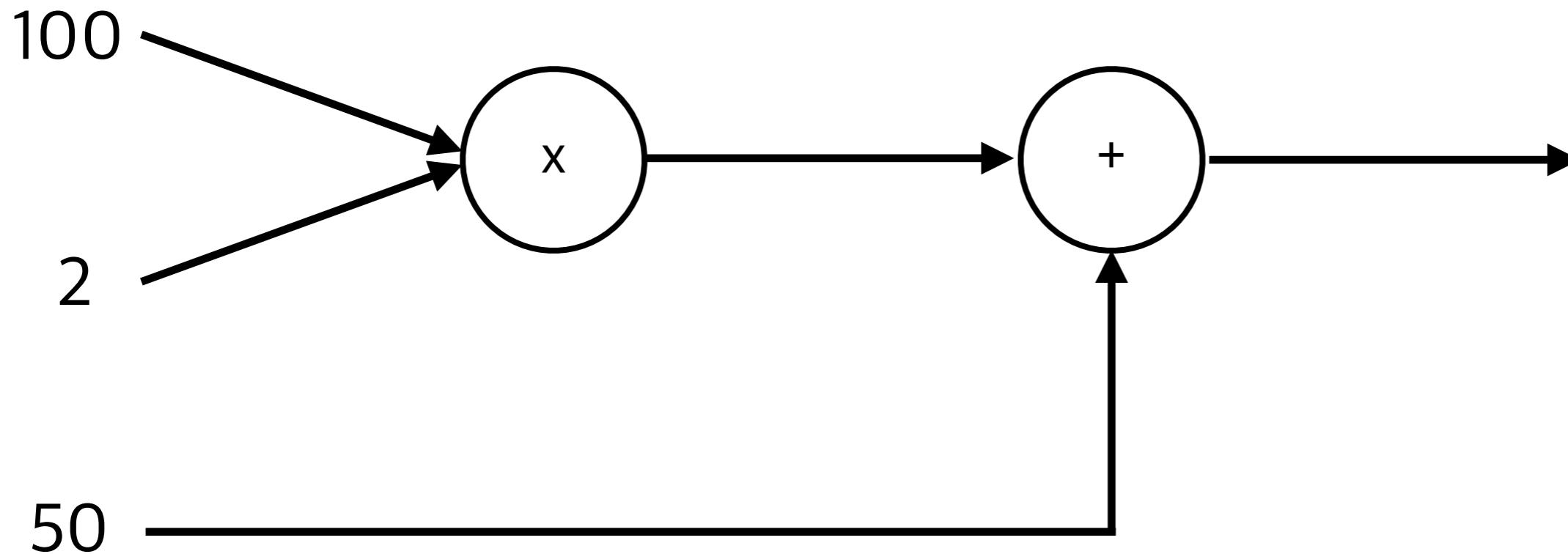
- **Forward propagation**
 - 입력을 받아 최종 loss값을 출력하는 단계
- **Backward propagation**
 - 레이어를 역으로 순회하면서 출력된 loss를 통해 각 파라미터의 **그라디언트를 계산**하는 단계
 - 그라디언트를 계산 한 뒤 gradient descent 방법으로 파라미터 업데이트

Forward-propagation

- 간단 예제: $f = (100^2) + 50$

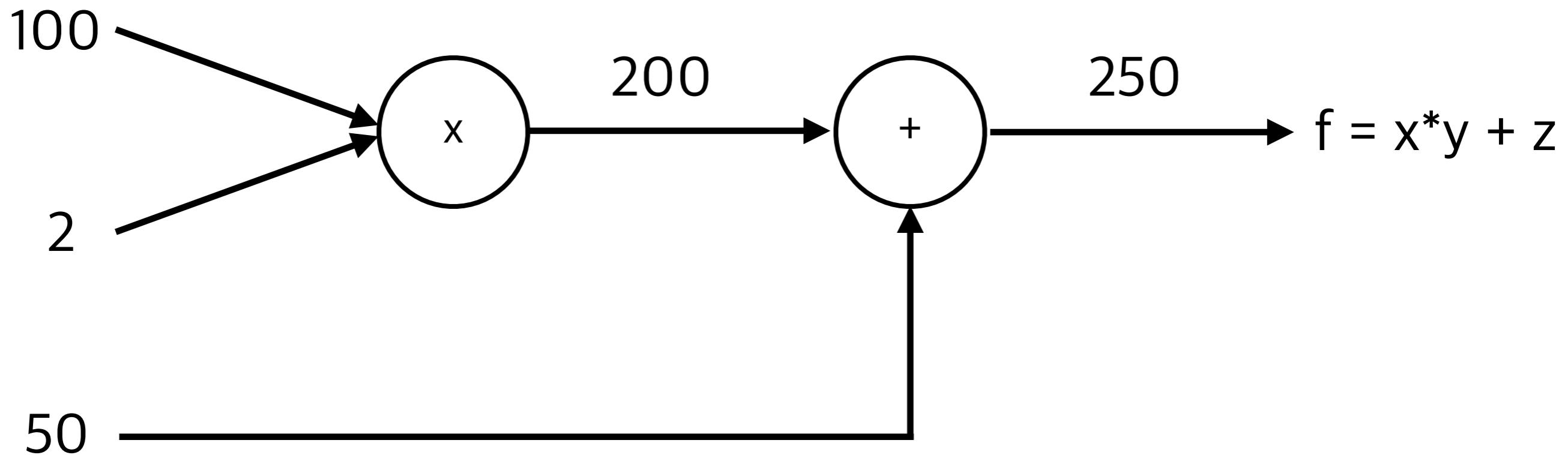
Forward-propagation

- 간단 예제: $f = (100*2) + 50$



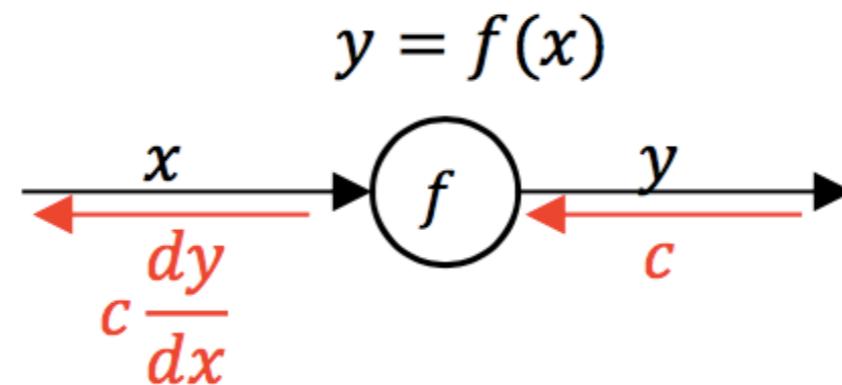
Forward-propagation

- 간단 예제: $f = (100*2) + 50$



Backward-propagation

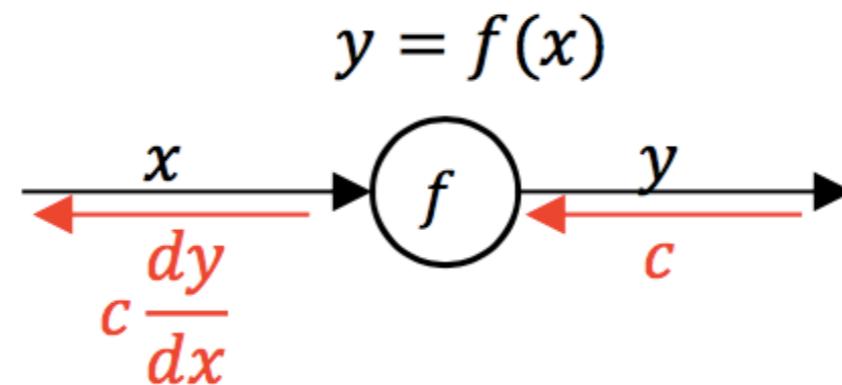
- Backward propagation은 각 연산의 **그라디언트**가 전파되는 방식으로 동작



e.g. $f(x) = x^2$ $y = f(x) = x^2$ $\frac{dy}{dx} = 2x$

Backward-propagation

- Backward propagation은 각 연산의 **그라디언트**가 전파되는 방식으로 동작

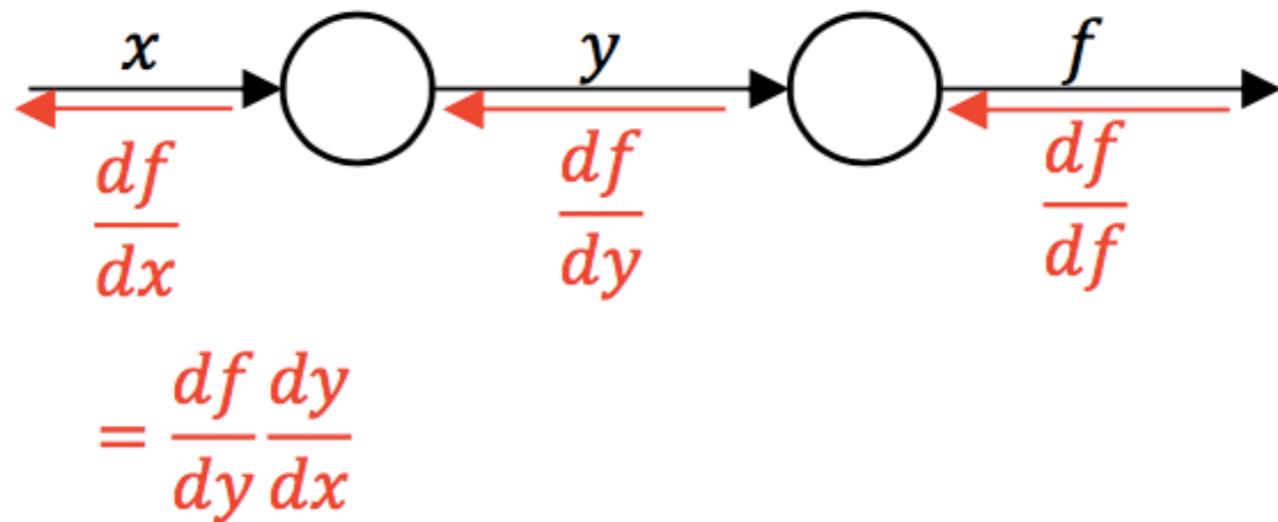


e.g. $f(x) = x^2$ $y = f(x) = x^2$ $\frac{dy}{dx} = 2x$

Backward propagation: $c \frac{dy}{dx} = c(2x)$

Backward-propagation

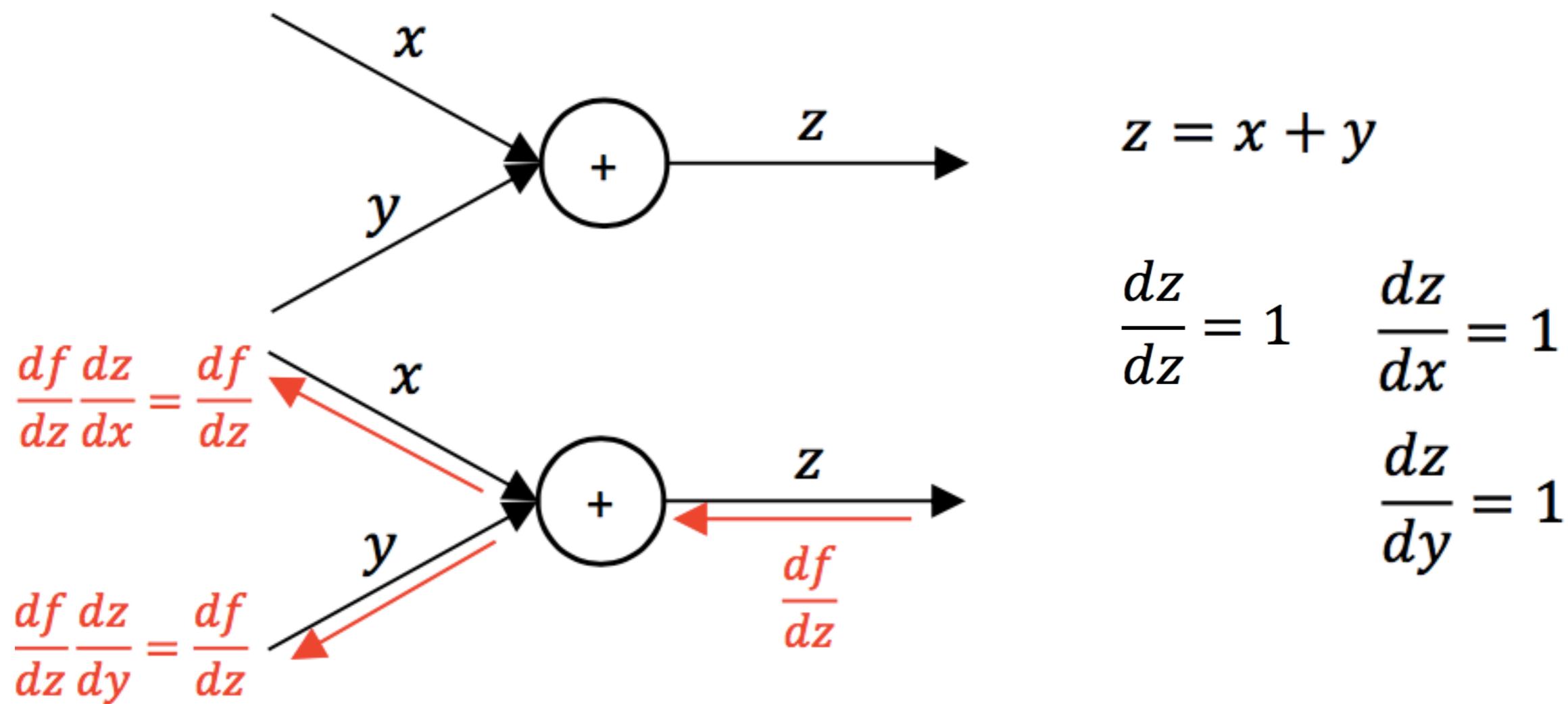
- 여러 함수(레이어)가 중첩되어 있을 경우 미분의 **체인룰**을 사용해서 그라디언트 전파



- 체인룰: $\frac{df}{dx} = \frac{df}{dy} \frac{dy}{dx}$

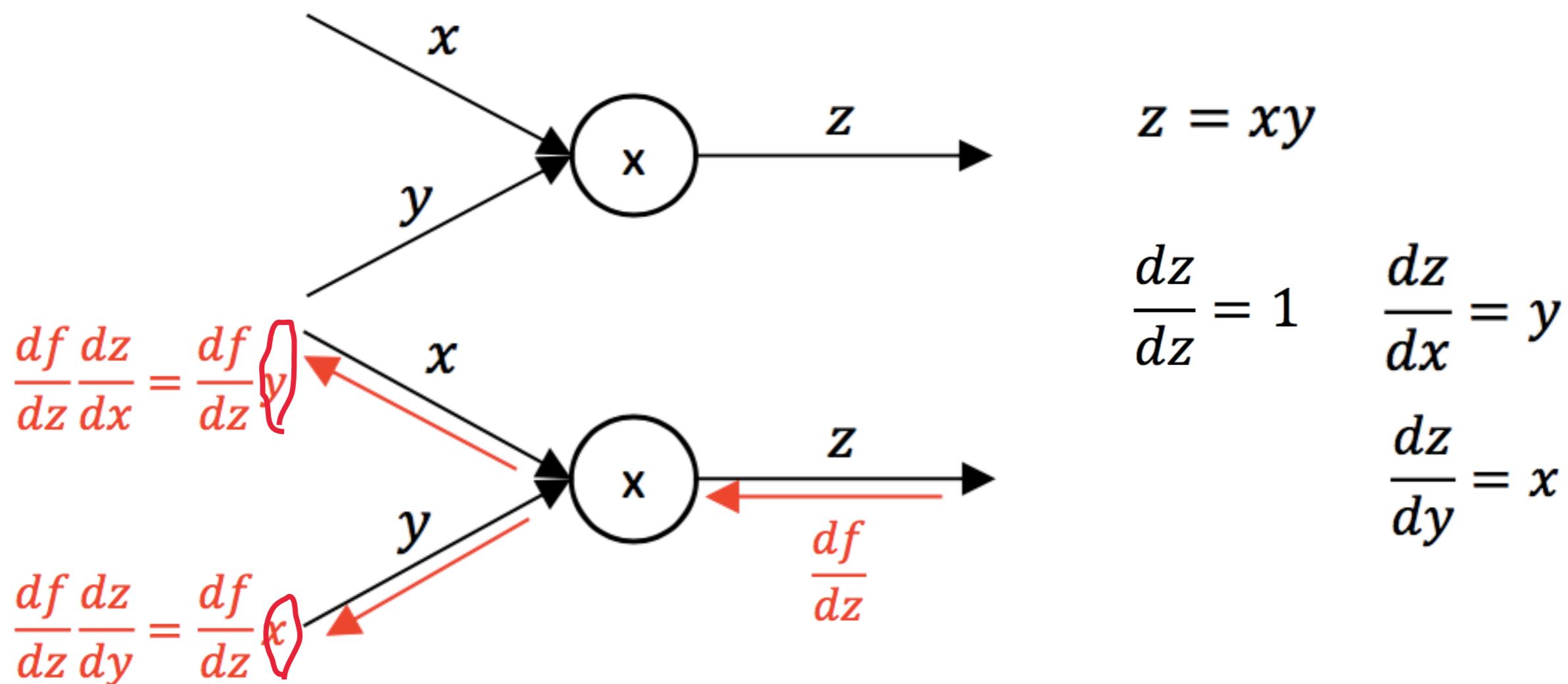
Backward-propagation

- 덧셈 연산의 그라디언트 및 backward-propagation



Backward-propagation

- 곱셈 연산의 그라디언트 및 backward-propagation

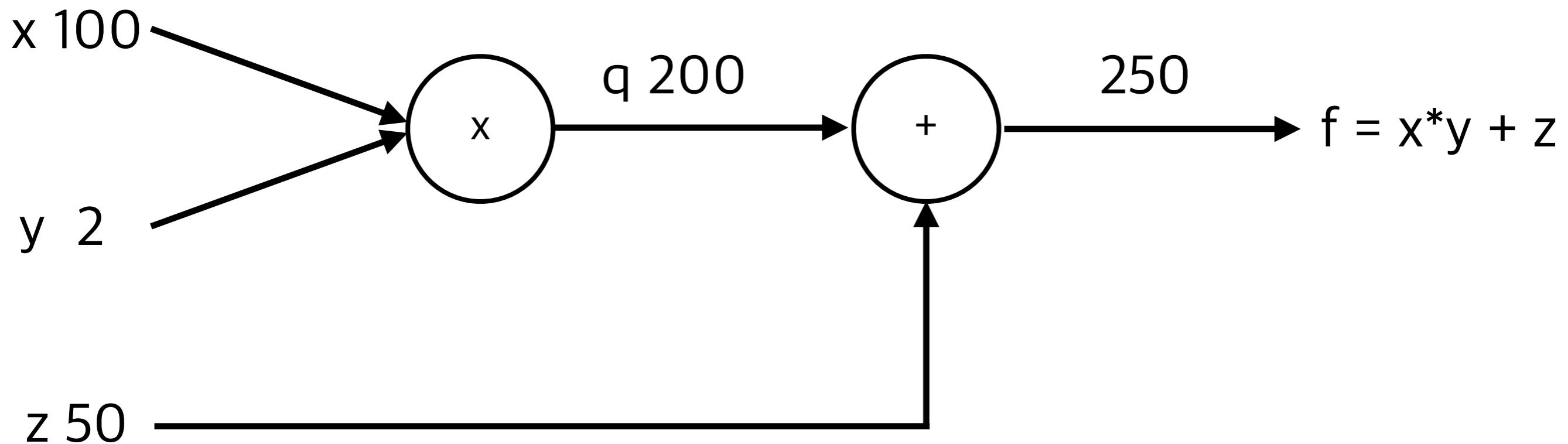


Backward-propagation

$$z = x + y \quad \frac{dz}{dx} = 1 \quad \frac{dz}{dy} = 1$$

- 간단 예제: $f = (100*2) + 50$

$$z = xy \quad \frac{dz}{dx} = y \quad \frac{dz}{dy} = x$$

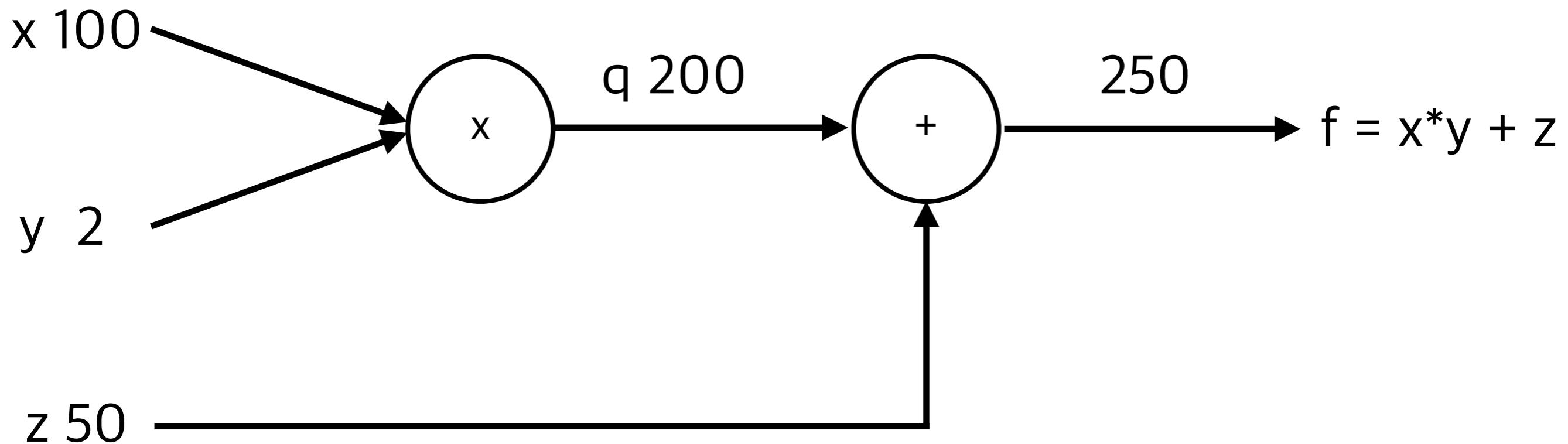


Backward-propagation

$$z = x + y \quad \frac{dz}{dx} = 1 \quad \frac{dz}{dy} = 1$$

- 간단 예제: $f = (100*2) + 50$

$$z = xy \quad \frac{dz}{dx} = y \quad \frac{dz}{dy} = x$$

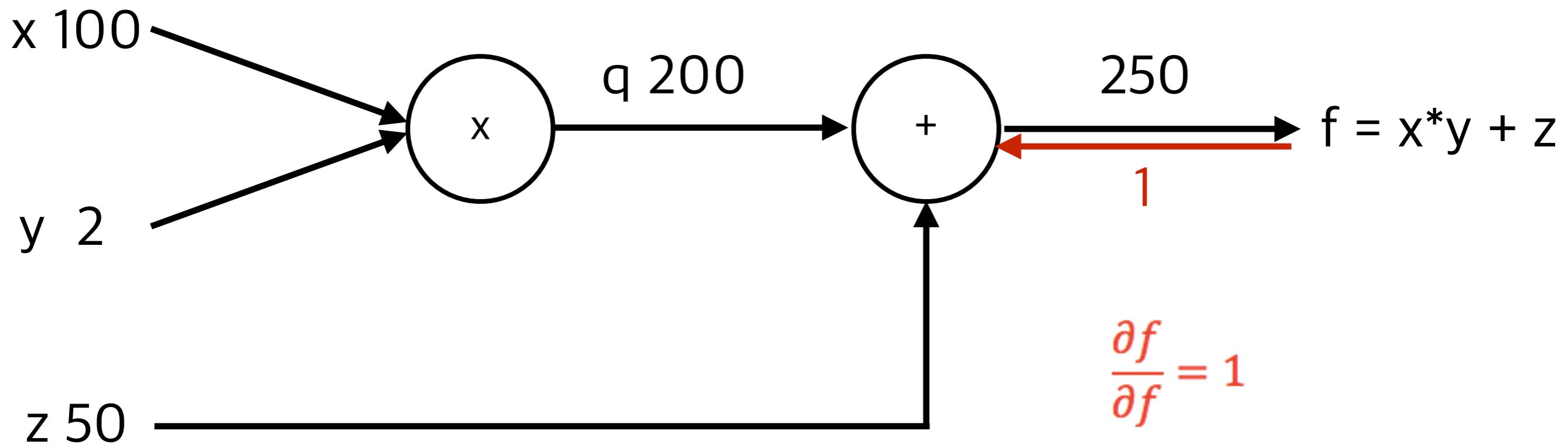


Backward-propagation

$$z = x + y \quad \frac{dz}{dx} = 1 \quad \frac{dz}{dy} = 1$$

- 간단 예제: $f = (100*2) + 50$

$$z = xy \quad \frac{dz}{dx} = y \quad \frac{dz}{dy} = x$$

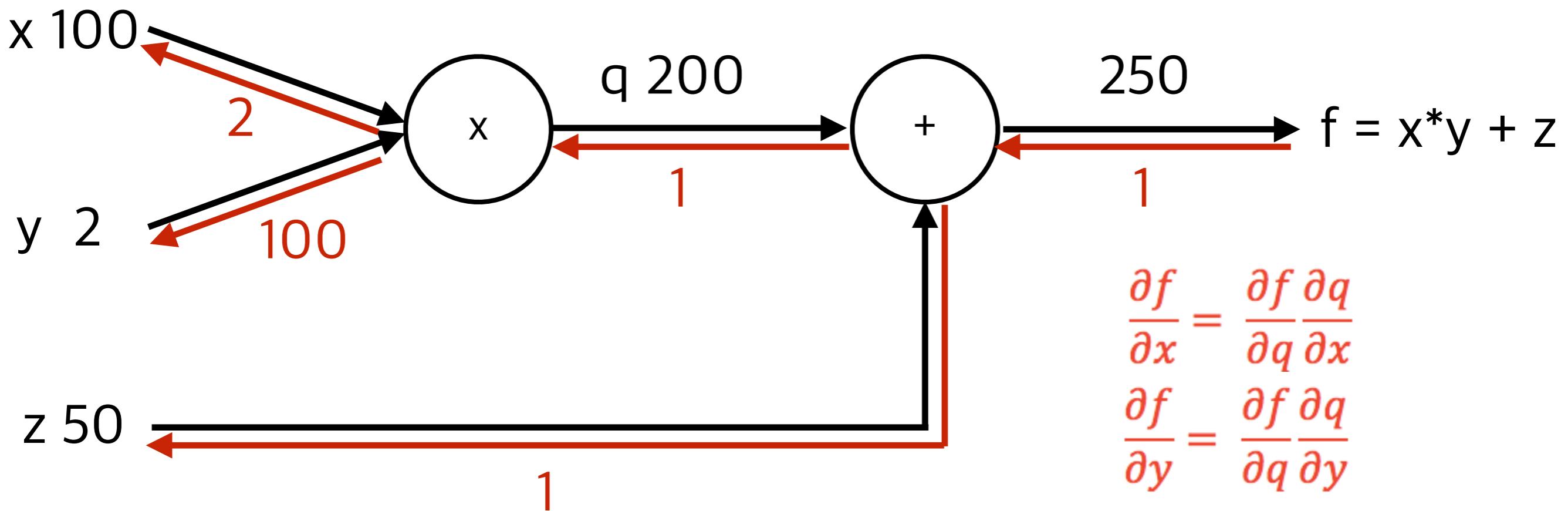


Backward-propagation

- 간단 예제: $f = (100*2) + 50$

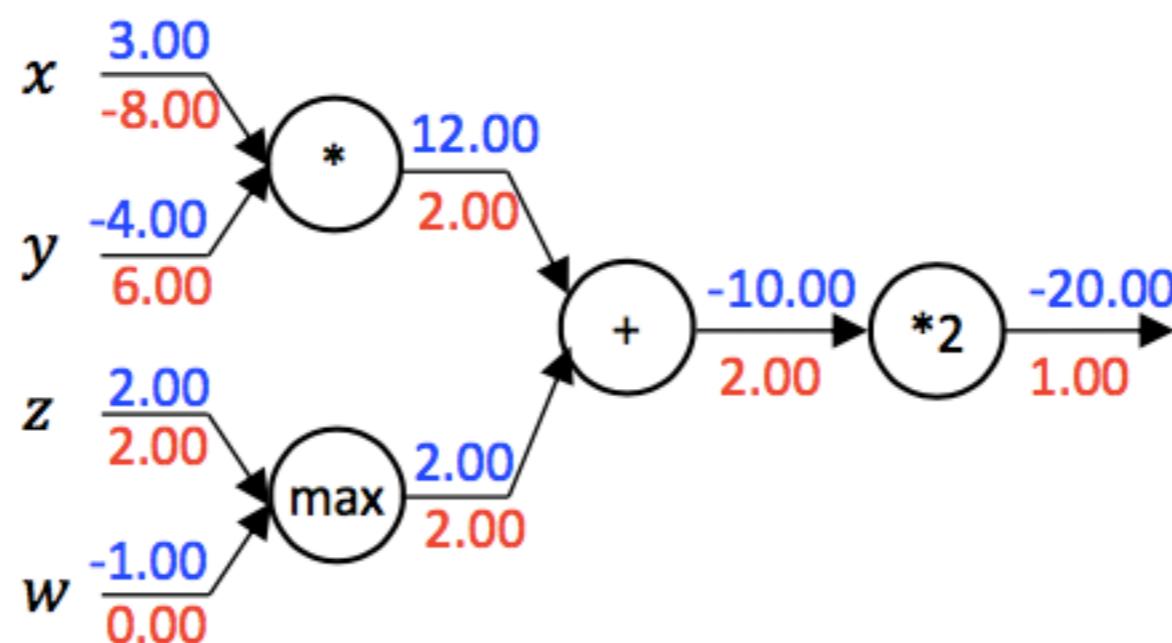
$$z = x + y \quad \frac{dz}{dx} = 1 \quad \frac{dz}{dy} = 1$$

$$z = xy \quad \frac{dz}{dx} = y \quad \frac{dz}{dy} = x$$



그라디언트 전파 패턴

- 덧셈: 그라디언트 분배 (그라디언트를 그대로 전파)
- 곱셈: 전파된 그라디언트 * 다른 노드의 값
- Max: 값이 큰 노드로 그라디언트 전파



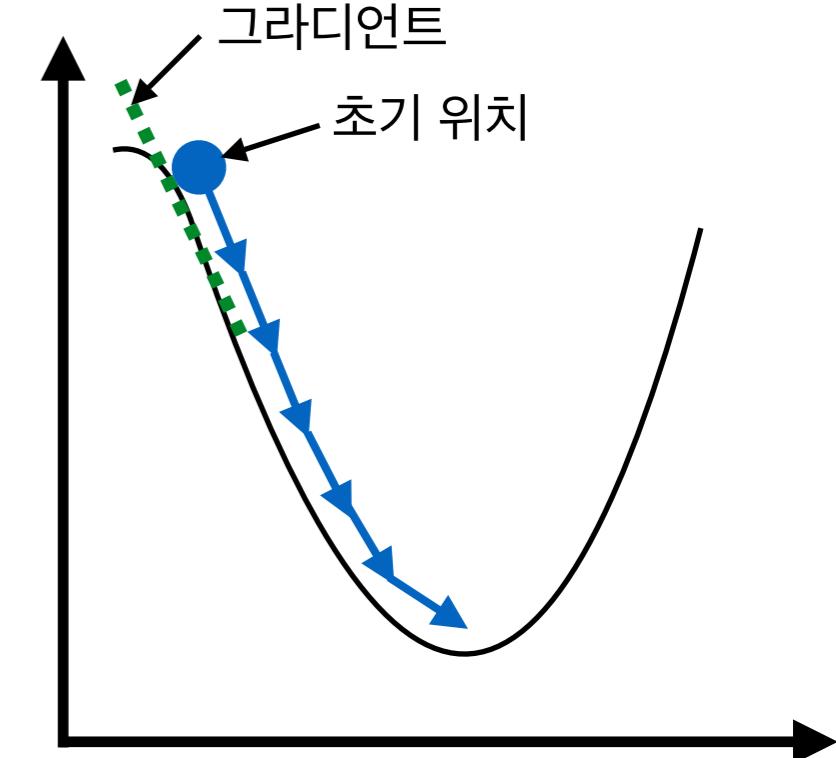
정리

- 지금까지 했던 것들
 - Loss를 정의: 크로스 엔트로피 loss
 - Forward propagation: 입력값을 모델에 넣고 loss를 계산
 - Back propagation: 계산된 loss를 이용 각 파라미터의 그라디언트 계산
- 남은 것
 - 그라디언트로 파라미터 업데이트 하기: gradient descent

Gradient descent

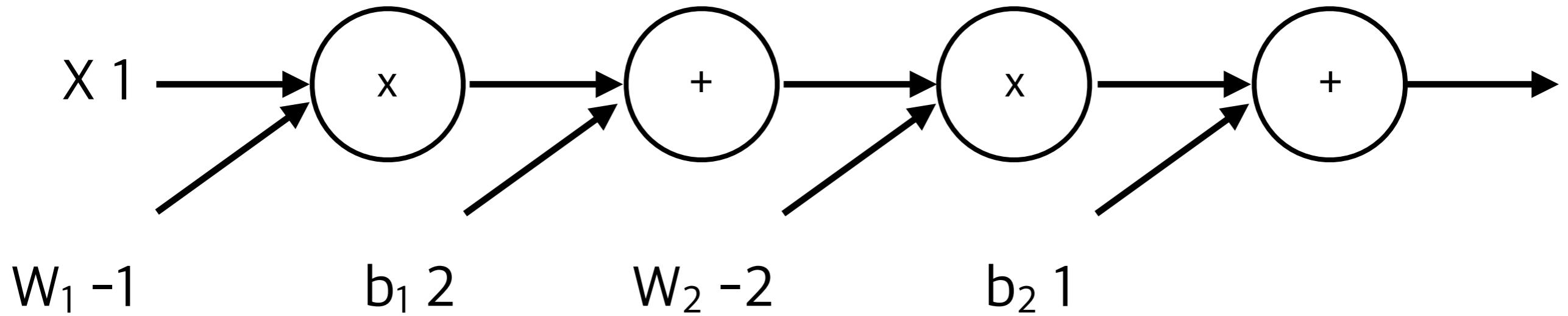
- 그라디언트를 이용하여 최저점을 점진적으로 찾아가는 방법
 - 수학적인 표기: $\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma_i \nabla f(\mathbf{x}_i)$
 - 코드:

```
while True:  
    grad = calculate_gradient(loss_func, parameters)  
    parameters -= learning_rate * grad
```
- Learning rate: 한 스텝동안 얼마나 움직일지 결정
하이퍼파라미터로 사용자가 정해주어야 함



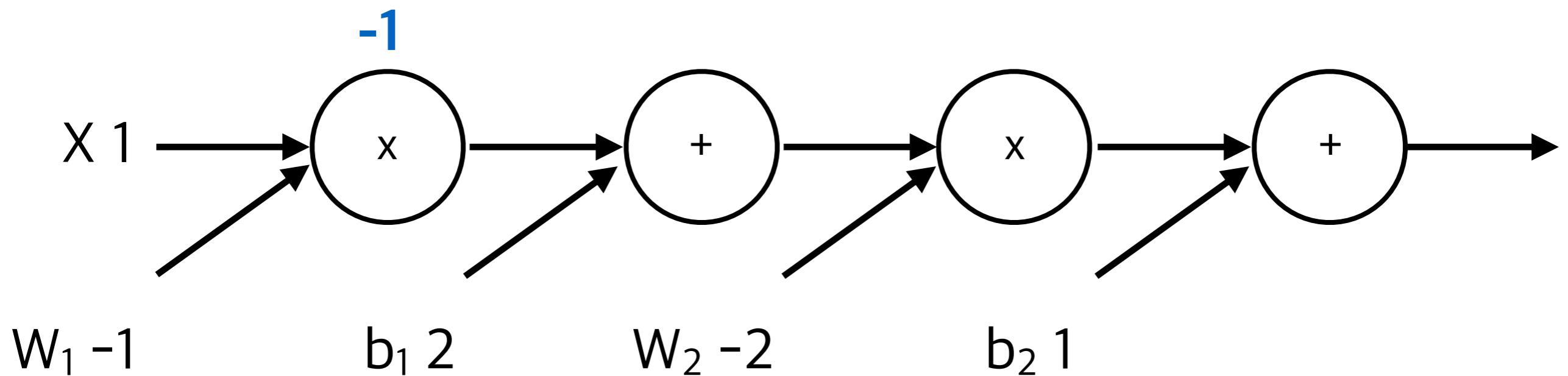
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



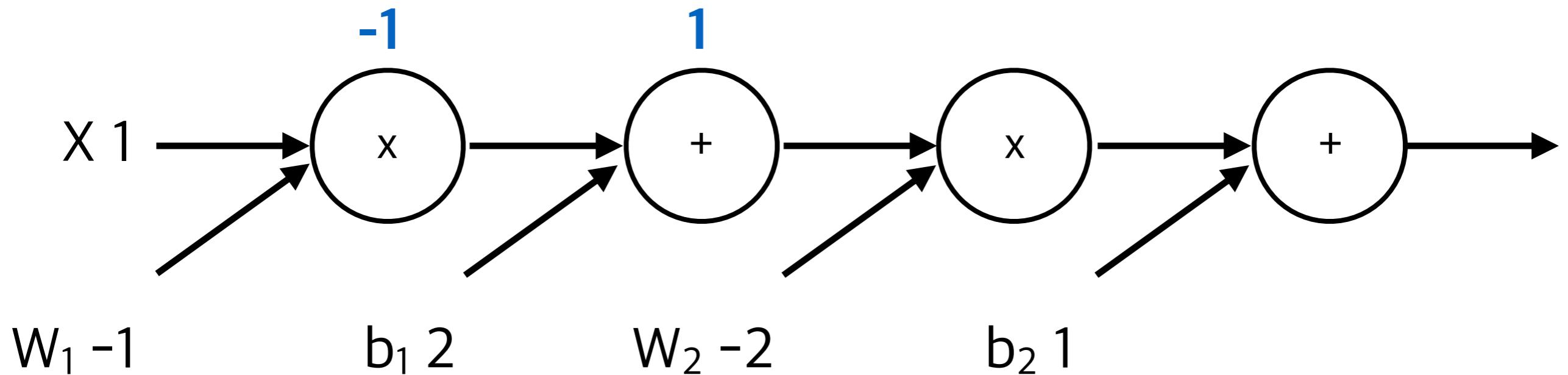
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



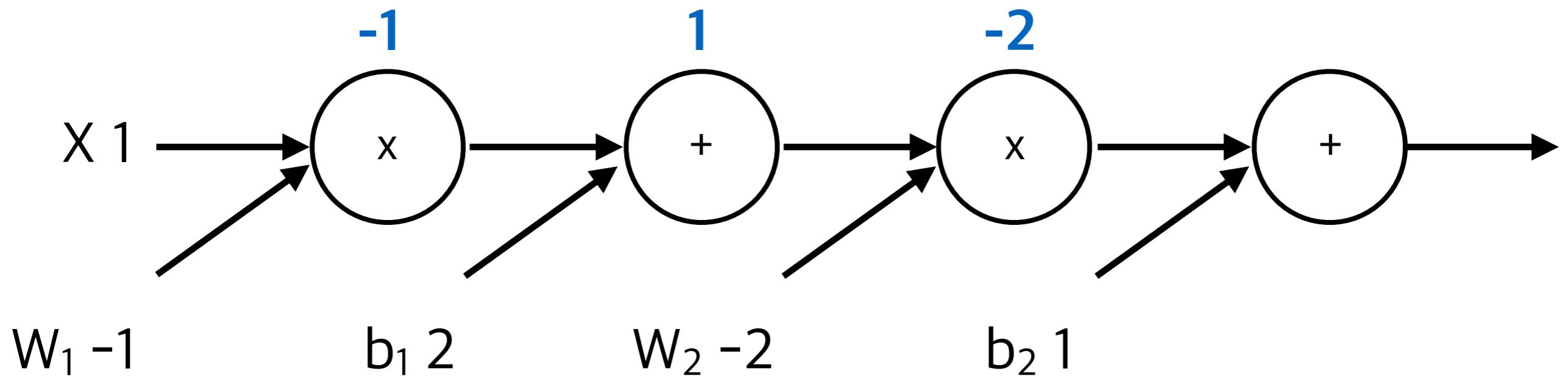
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



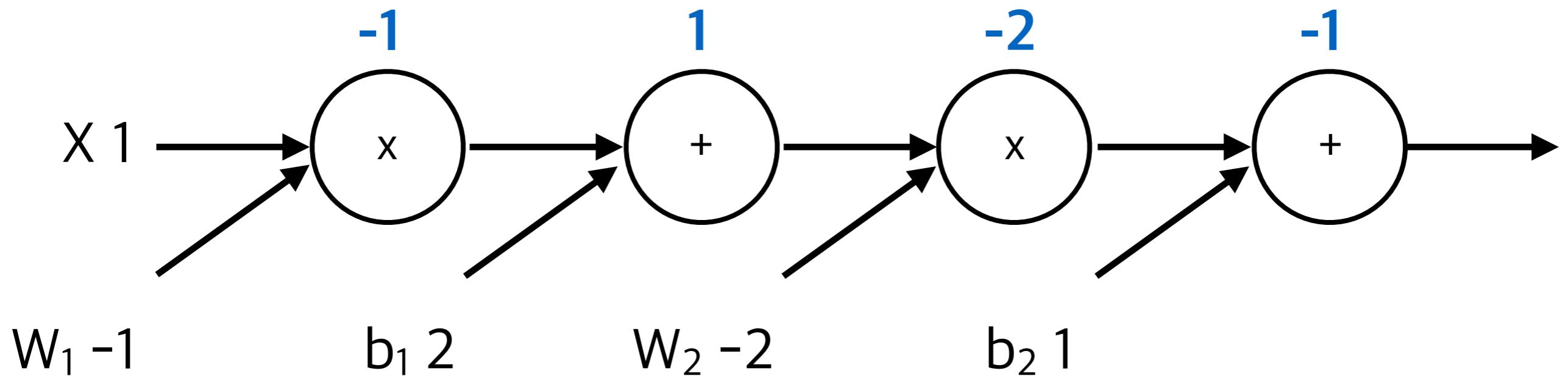
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



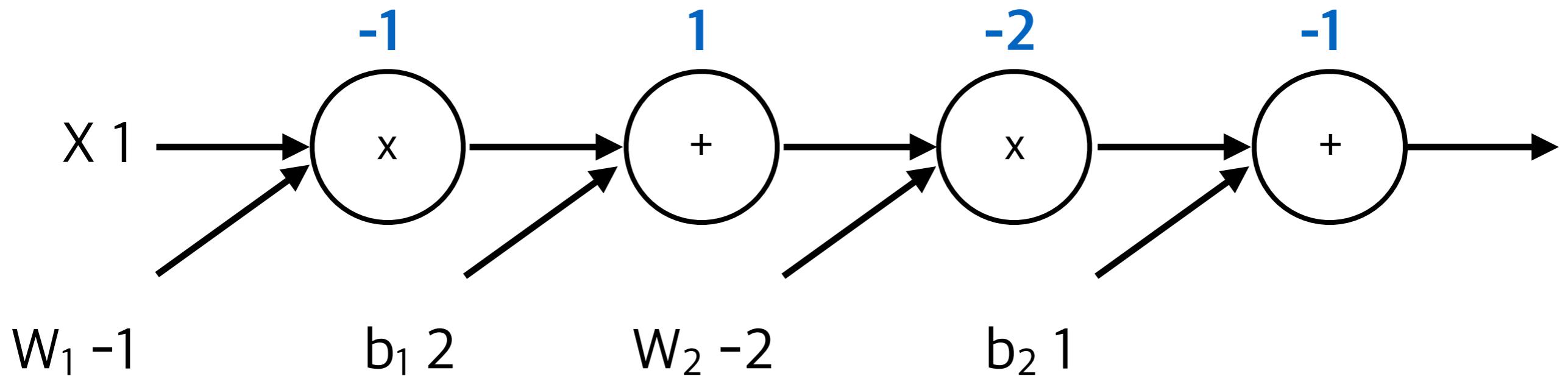
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



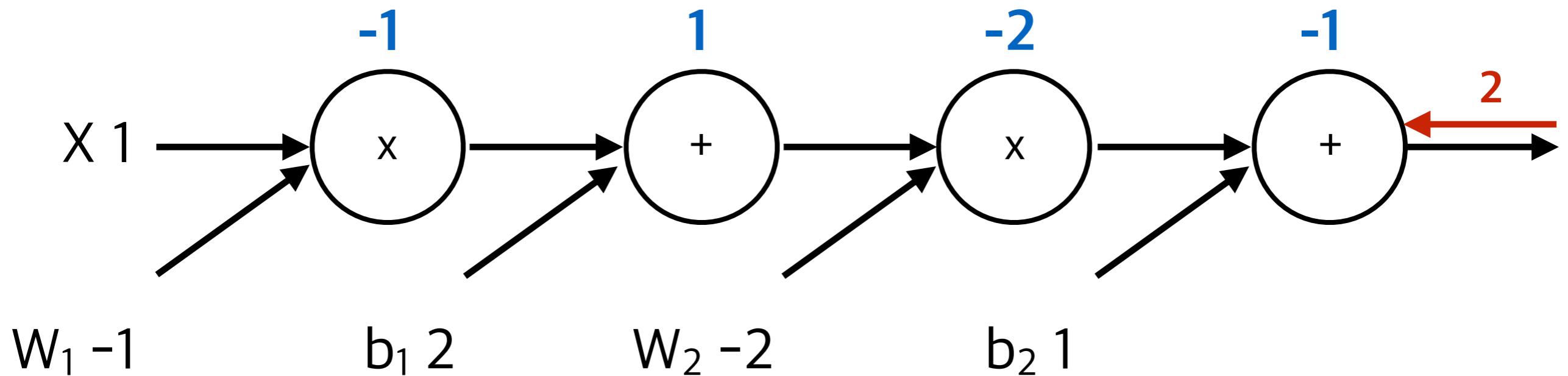
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



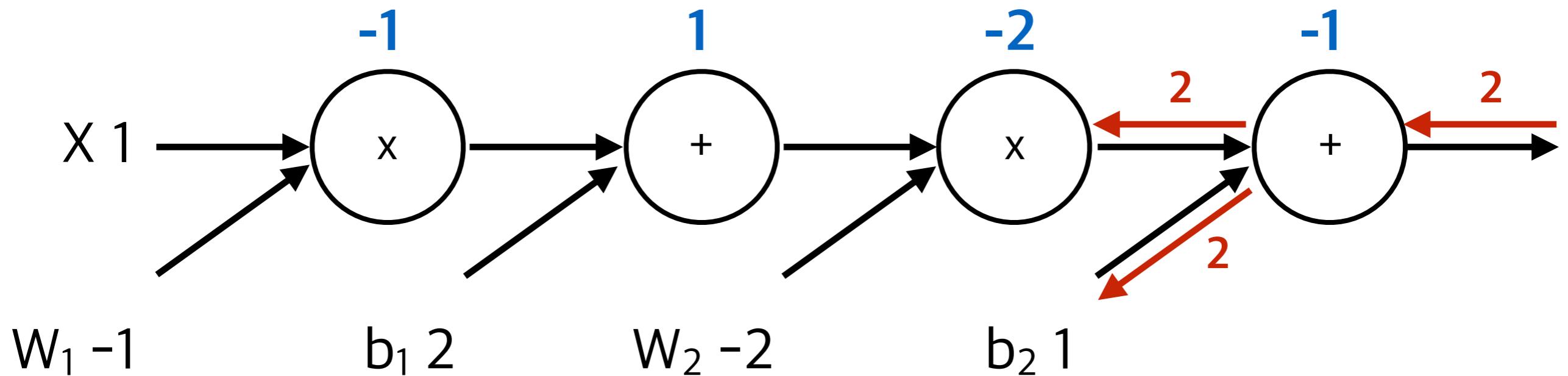
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



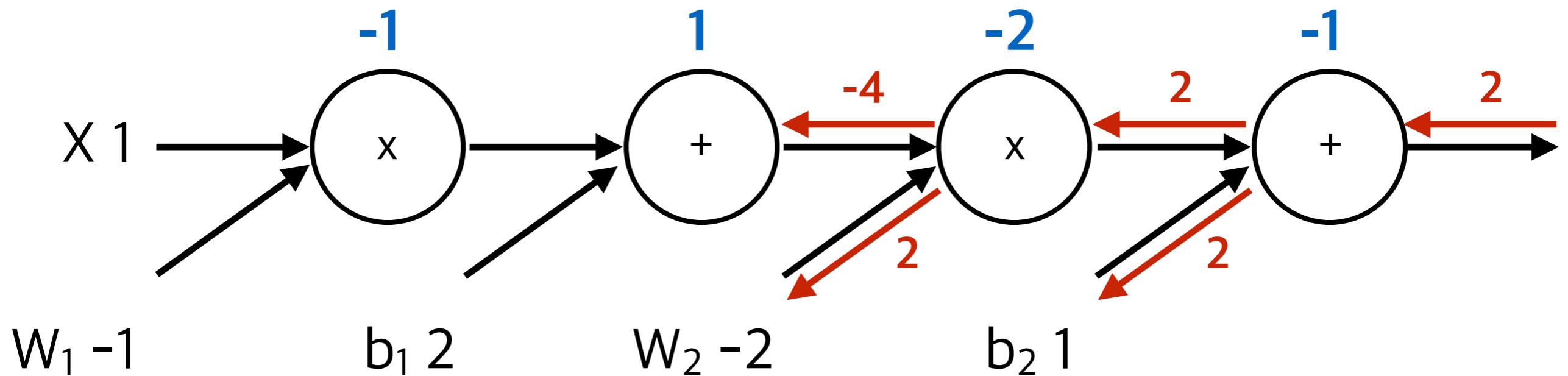
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



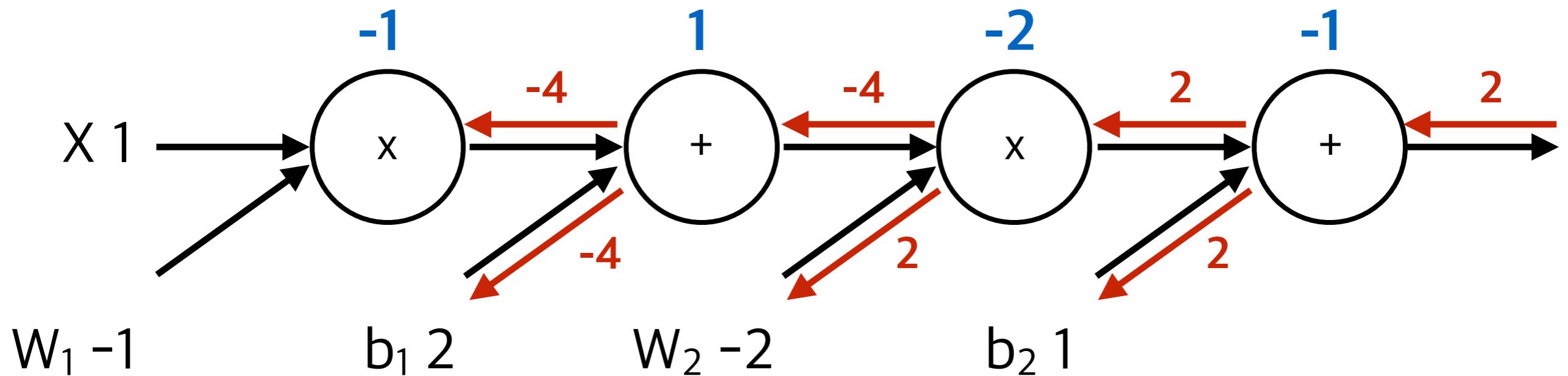
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



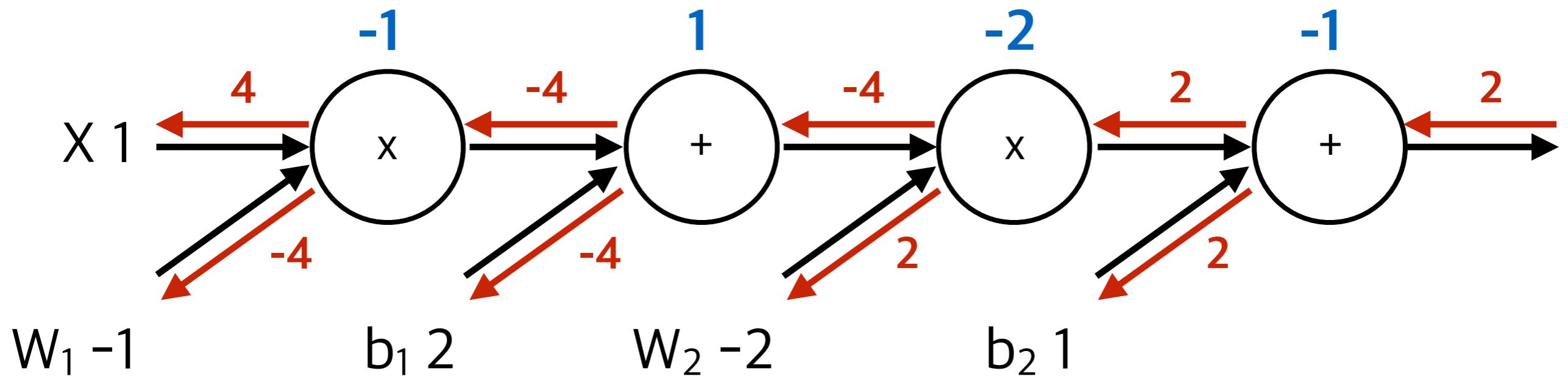
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



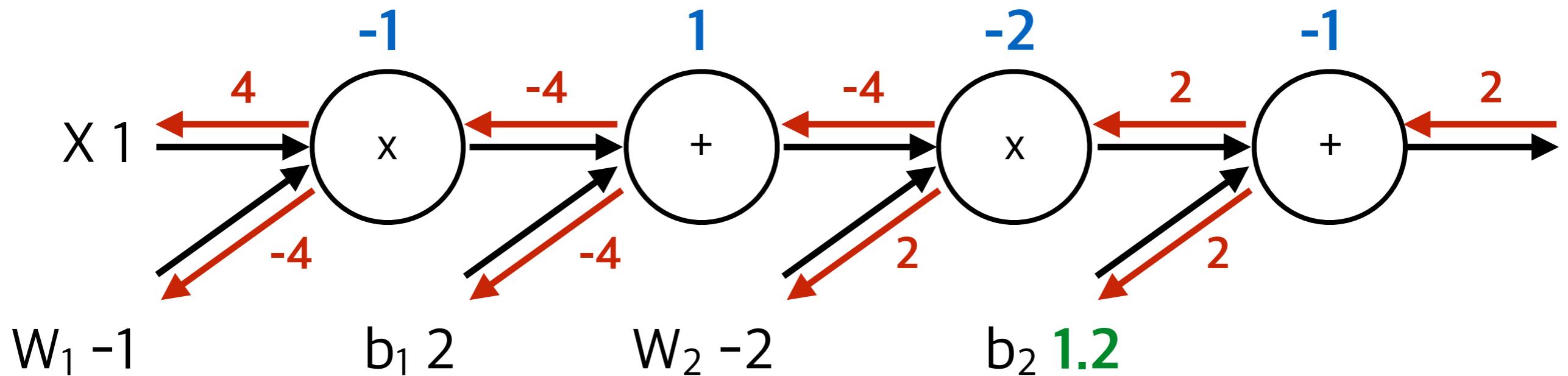
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



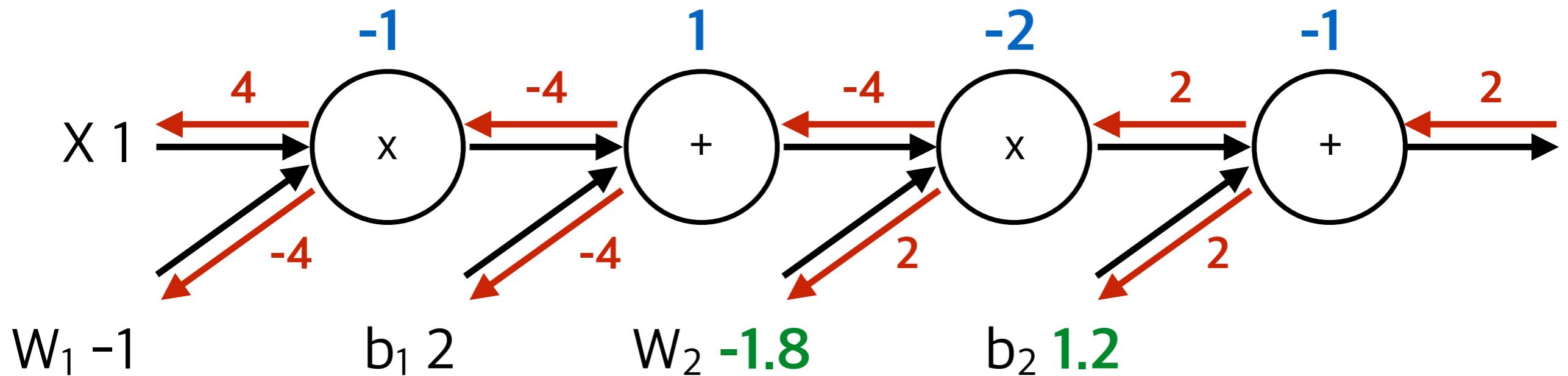
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



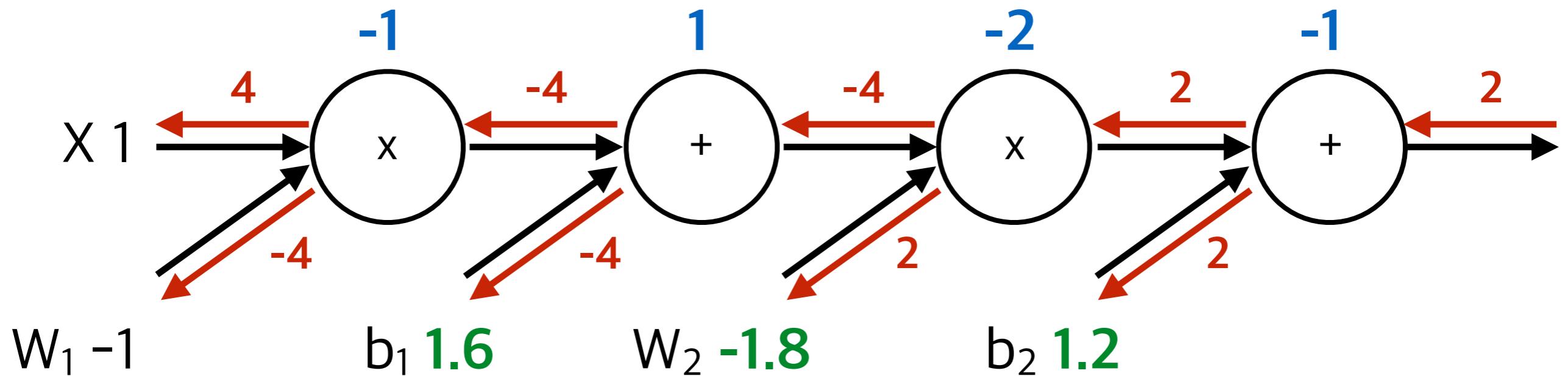
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



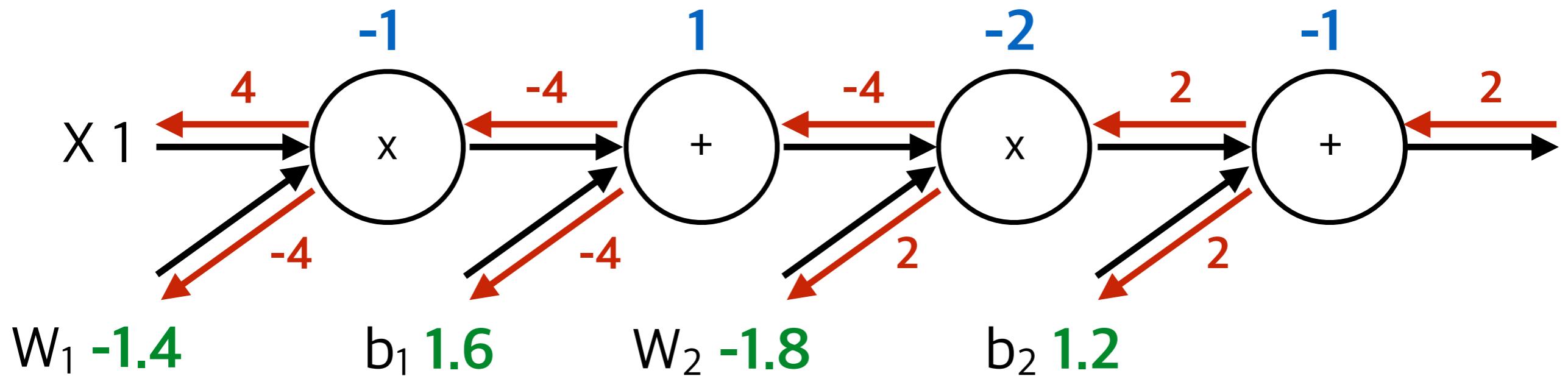
예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



예시

- 뉴럴 네트워크 $f = W_2 * (W_1 * X + b_1) + b_2$
 - 전파된 loss는 2, learning rate는 0.1로 가정



Gradient descent의 문제점

- 모든 데이터를 한꺼번에 보고 그라디언트를 계산
 - 모든 데이터에 대해 그라디언트를 계산하고 평균낸 뒤 파라미터 업데이트
 - 정확한 그라디언트를 계산할 수 있지만, 한 스텝에 딱 한 번 사용
 - 이러한 특성 때문에 Full-batch GD라고도 불림
 - 문제점:
 - 위 작업을 한번에 할만큼 메모리가 충분한가?
 - 시간 상으로도 매우 비효율적
-

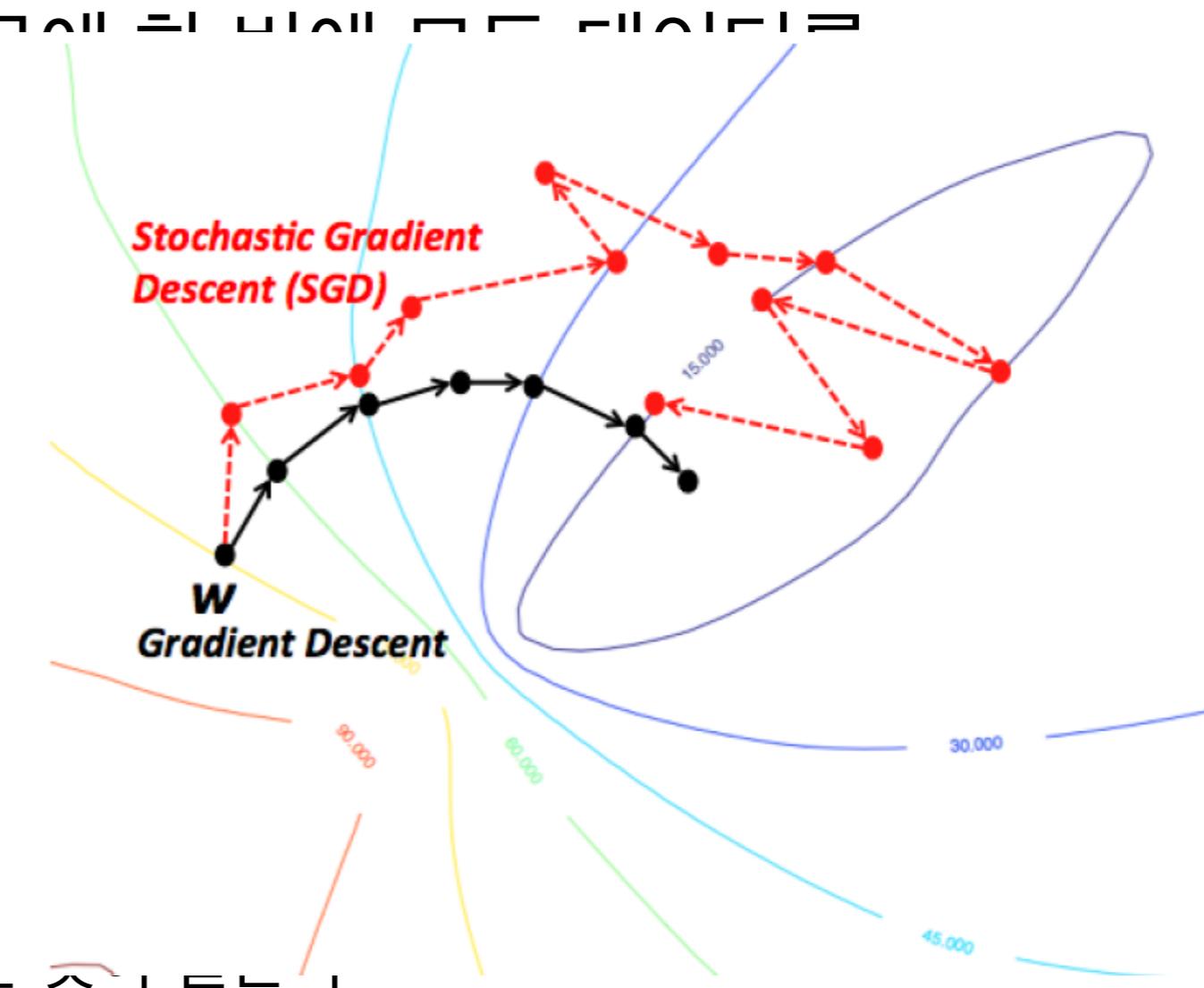
내경 2 Stochastic gradient descent

Stochastic gradient descent

- 학습 데이터가 매우 크기 때문에 한 번에 모든 데이터를 보고 계산하는 과정은 매우 비효율적
 - 데이터 몇개를 랜덤하게 뽑아서 그라디언트를 계산하자!
 - 이 때 뽑힌 데이터를 배치(batch)라고 부름
 - 그라디언트 계산은 배치들의 평균을 통해 내기 때문에 근사된 값을 도출
- Stochastic GD vs mini-batch GD
 - 사실 Stochastic GD는 배치 크기가 1인 경우만 해당
 - 배치 크기가 1보다 크다면 mini-batch GD라고 불러야 하지만.. 그냥 두 경우 모두 SGD라고 부르는 것이 일반적

Stochastic gradient descent

- 학습 데이터가 매우 크기 때문에 계산하는 과정은 매우
• 데이터 몇개를 랜덤하게 뽑아서 :
• 이 때 뽑힌 데이터를 **배치(batch)**
• 그라디언트 계산은 배치들의 평균
- Stochastic GD vs mini-batch
• 사실 Stochastic GD는 배치 크기
• 배치 크기가 1보다 크다면 mini-
그냥 두 경우 모두 SGD라고 부르



용어 정리

- **Learning rate**: 그라디언트를 통해 파라미터를 업데이트할 때 한 번에 얼마나 움직일지 결정하는 하이퍼파라미터
- **하이퍼파라미터**: 모델의 성능을 올리기 위해 선택해야 하는 세팅이며, 일반적으로 실험을 통해 최적의 하이퍼파라미터를 선정
- **배치(batch)**: SGD에서 한 번 학습 할 때 입력으로 들어오는 랜덤 샘플
- **Iteration(or step)**: 각 스텝은 서로 다른 배치를 입력으로 받고 학습
- **Epoch**: 모든 학습 데이터를 한 번 봤을 때 1 epoch
- 예) 배치크기 64, 데이터가 1000개일 때 한 epoch당 16 iteration 존재

네트워크 학습 수도 코드

```
while True:  
    batch_data = sample_batch_data()  
    loss = network.forward(batch_data)  
    grad_weight = network.backward()  
    weight -= learning_rate * grad_weight
```

네트워크 학습 수도 코드

```
while True:  
    batch_data = sample_batch_data()  
    loss = network.forward(batch_data)  
    grad_weight = network.backward()  
    weight -= learning_rate * grad_weight
```

학습 데이터로 부터 배치 샘플링

네트워크 학습 수도 코드

```
while True:  
    batch_data = sample_batch_data()  
    loss = network.forward(batch_data)  
    grad_weight = network.backward()  
    weight -= learning_rate * grad_weight
```

학습 데이터로 부터 배치 샘플링

forward/backward propagation

네트워크 학습 수도 코드

```
while True:
```

```
    batch_data = sample_batch_data()  
    loss = network.forward(batch_data)  
    grad_weight = network.backward()  
    weight -= learning_rate * grad_weight
```

학습 데이터로 부터 배치 샘플링

forward/backward propagation

backward propagation에서 계산된 그라디언트를
SGD 알고리즘을 통해 파라미터 업데이트

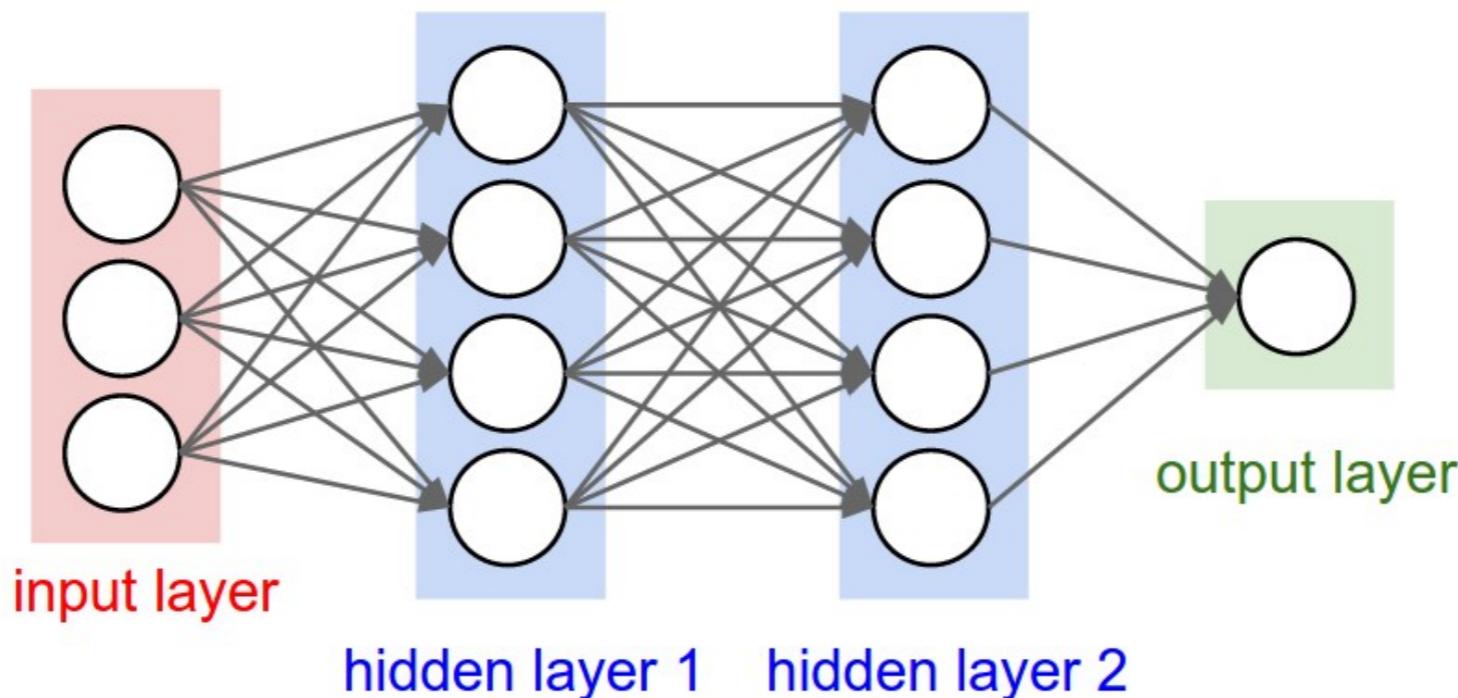
뉴럴넷의 역사

- Back propagation의 등장으로 뉴럴넷은 다시 각광받음
- 그러나.. 90년대 후반 다시 새로운 **빙하기** 도래
 - 뉴럴넷을 깊게 쌓아도 성능이 좋아지지 않는다! (과적합)
 - 매우 느린 연산 속도
 - SVM, 랜덤 포레스트 등의 매우 뛰어난 머신러닝 알고리즘 등장
- 2012년 **AlexNet**의 등장
 - 다시 시작된 뉴럴 네트워크 붐

뉴럴 네트워크 //

뉴럴 네트워크 정리

- **입력층, 은닉층, 출력층**으로 이루어진 멀티 레이어 퍼셉트론
- 각 층은 **활성함수**(activation function)가 존재
- 뉴럴 네트워크의 학습은 back propagation + SGD에 의해



활성 함수

- 활성 함수는 네트워크에 **비선형성**(nonlinearity)을 추가하기 위해 사용됨
 - 활성 함수 없이 레이어를 쌓은 네트워크는 1-레이어 네트워크와 동일함
 - 그래서 활성 함수는 비선형함수로 불리기도 함
 - 데모: <http://playground.tensorflow.org>

활성 함수의 종류

- 계단 함수
- 시그모이드 함수
- 하이퍼볼릭-탄젠트 함수
- ReLU
- ...

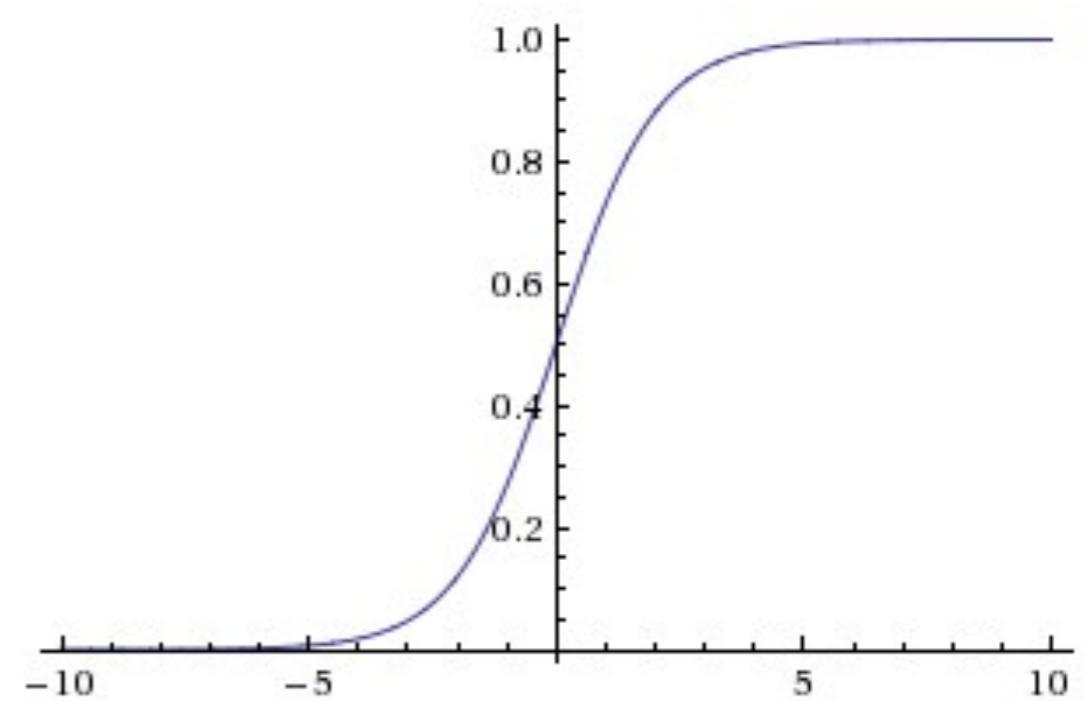
시그모이드

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

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

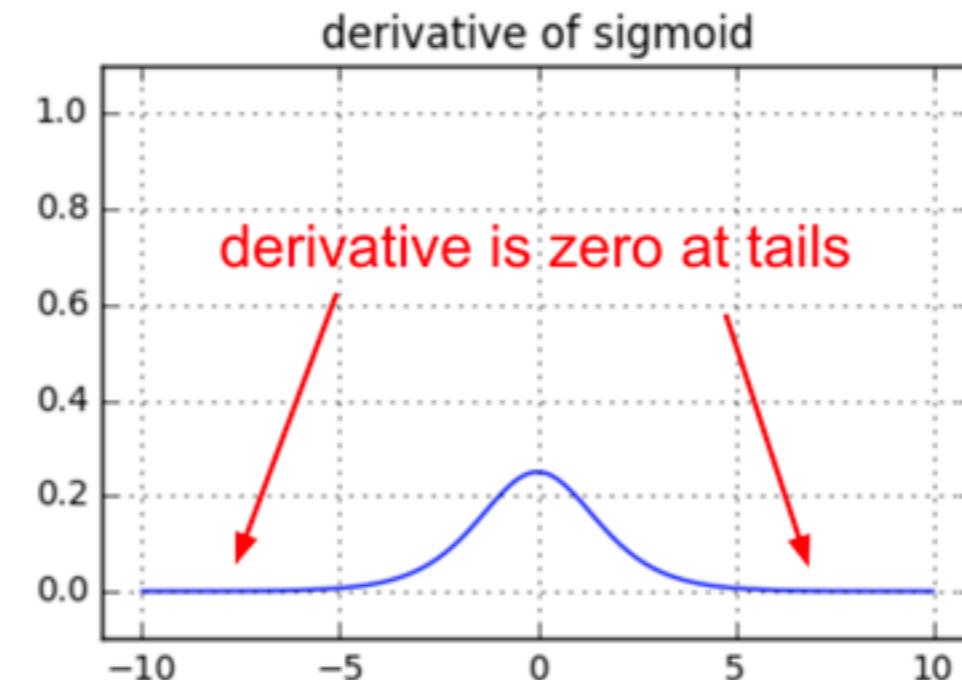
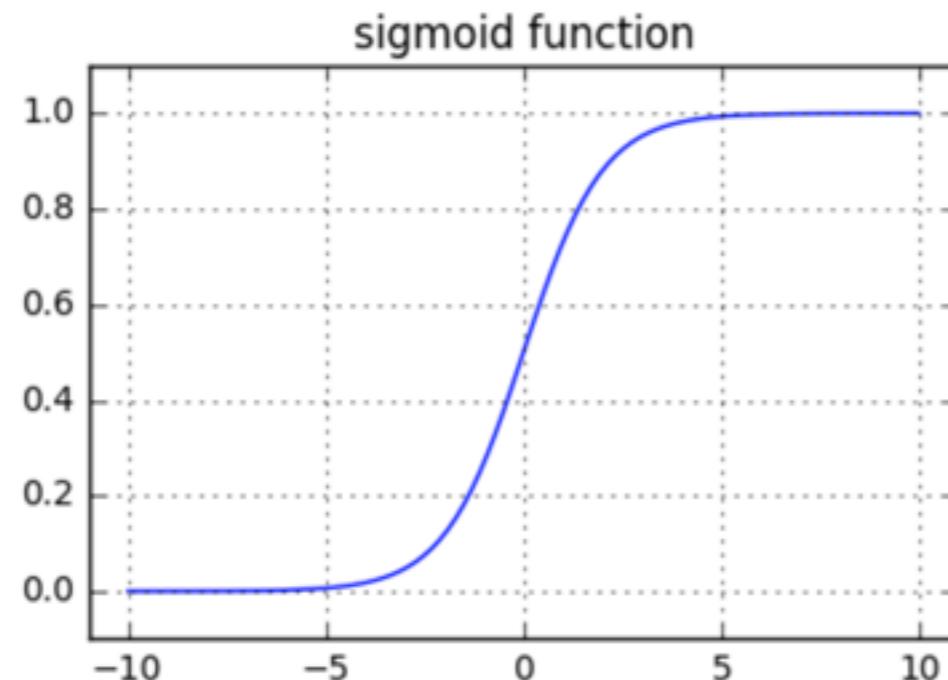
(-) 그라디언트가 죽는 현상이 발생
(gradient vanishing 문제)

- (-) 활성함수의 결과 값의 중심이 0이 아님
- (-) 계산이 복잡함 (지수함수 계산)



Gradient Vanishing

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

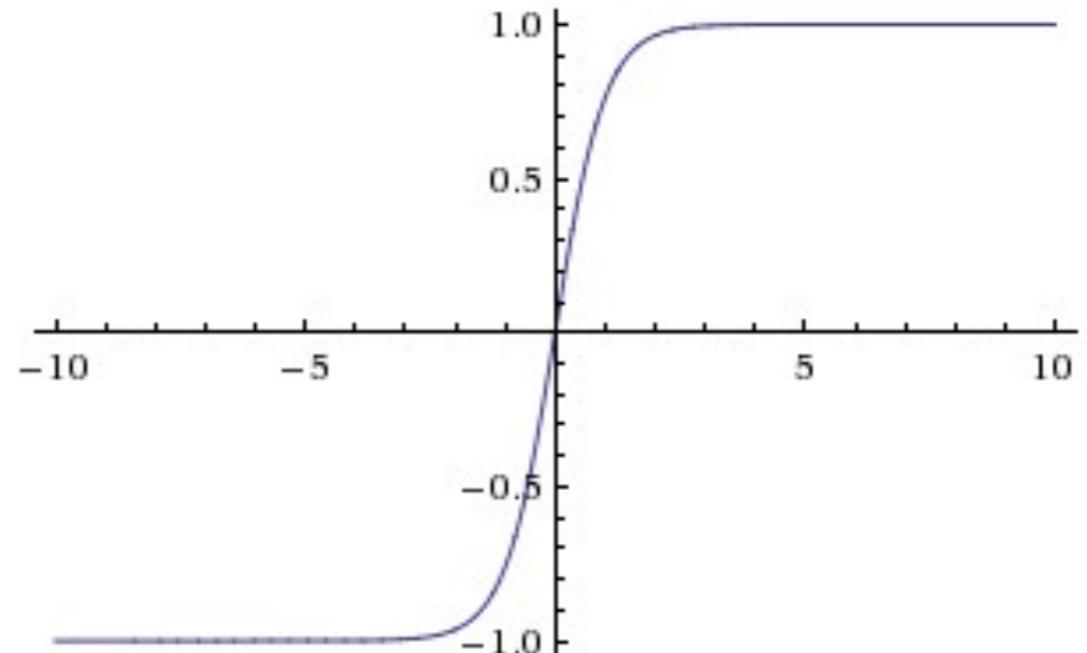


하이퍼볼릭-탄젠트 (tanh)

$$\tanh(x) = 2\sigma(2x) - 1$$

- 결과값이 [-1, 1] 사이로 제한됨
- 나머지 특성은 시그모이드와 비슷함
- 시그모이드 함수를 이용하여 유도 가능

- (-) 여전히 gradient vanishing
(+) 결과값 중심이 0



Rectified Linear Unit (ReLU)

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



- 최근 가장 많이 쓰이는 활성 함수

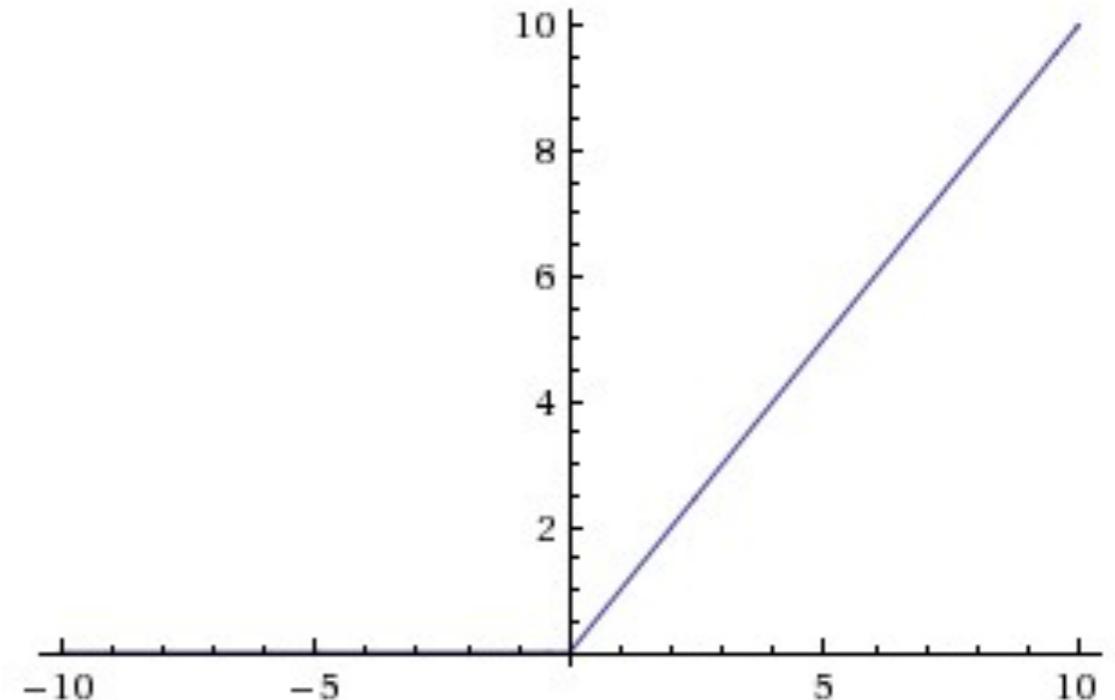
(+) 양 극단값이 포화되지 않음
(양수 지역은 선형적)

(+) 계산이 매우 효율적 (최대값 연산 1개)

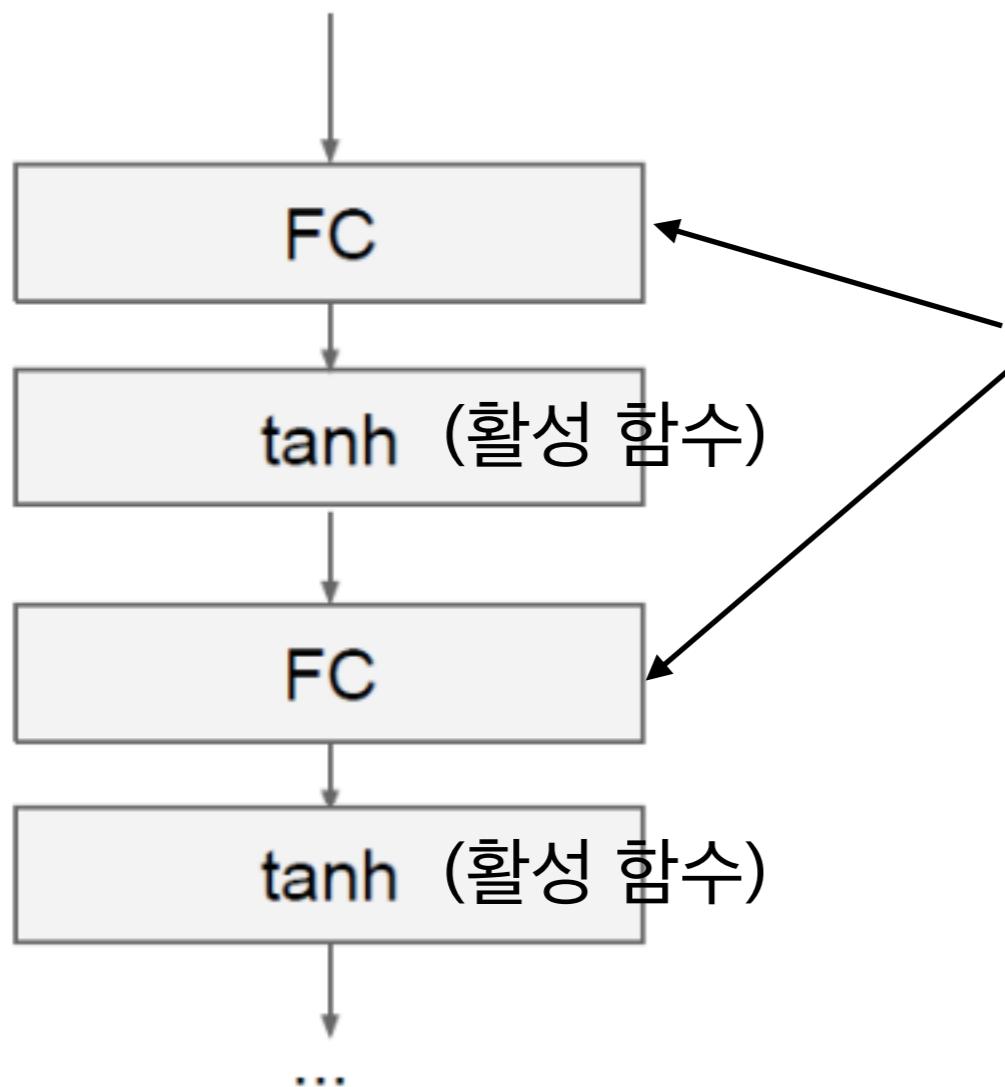
(+) 수렴속도가 시그모이드류 함수 대비
6배 정도 빠름

(-) 중심값이 0이 아님 (마이너한 문제)

(-) 입력값이 음수인 경우 항상 0을 출력함
(마찬가지로 파라미터 업데이트가 안됨)



일반적인 뉴럴 네트워크 구조



Fully-Connected 레이어 (층)

- 현재 레이어-다음 레이어의 뉴런들이 전부 연결되어 있기 때문에 불리는 이름
- MLP의 각 퍼셉트론 층과 같은 의미

Weight 초기화 전략

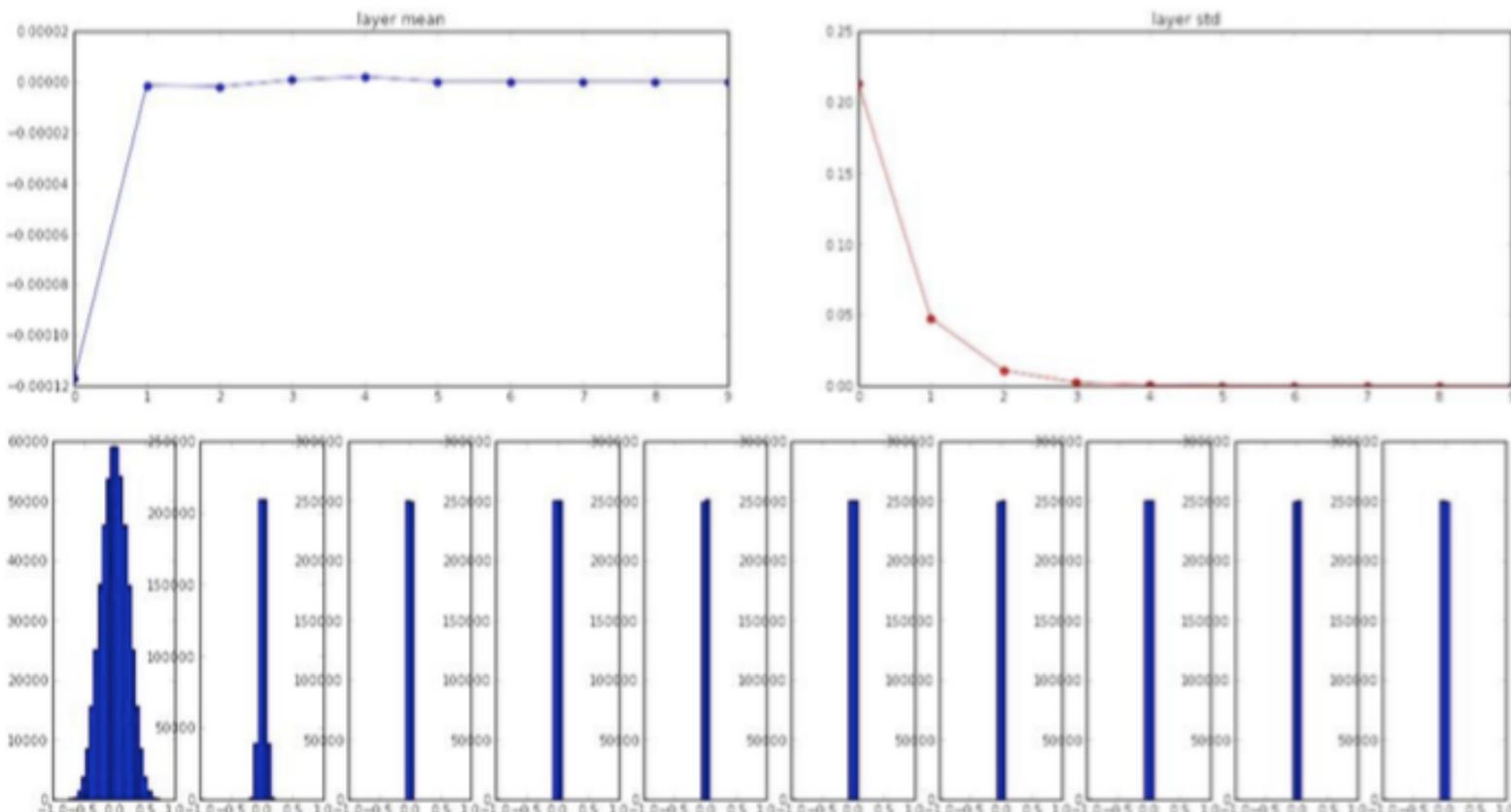
- 네트워크를 만들 때 weight의 초기화도 매우 중요한 요소
- 만약 weight를 모두 0으로 초기화 하면?
 - 모든 뉴런이 같은 그라디언트를 계산하게 되어 모든 weight 값이 똑같이 변하게 됨 (학습이 될리가 없어요..)
- 어쨌든 초기화 값은 0에 근접해야 함
 - 너무 큰 값으로 초기화 하면 학습시 그라디언트가 매우 커져서 폭발 (gradient exploding)

랜덤 값으로 초기화

- 예를 들어,
 - 실험 세팅: 레이어 10개와 tanh 활성 함수를 사용하는 네트워크
 - 모델1: 평균이 0, 표준편차는 0.01인 가우시안 분포에 의해
랜덤하게 초기화

랜덤 값으로 초기화

- 예를 들어,
 - 실험 세팅: 레이어 10개와 tanh 활성 함수를 사용하는 네트워크
 - 모델1: 평균이 0, 표준편차는 0.01인 가우시안 분포에 의해 랜덤하게 초기화

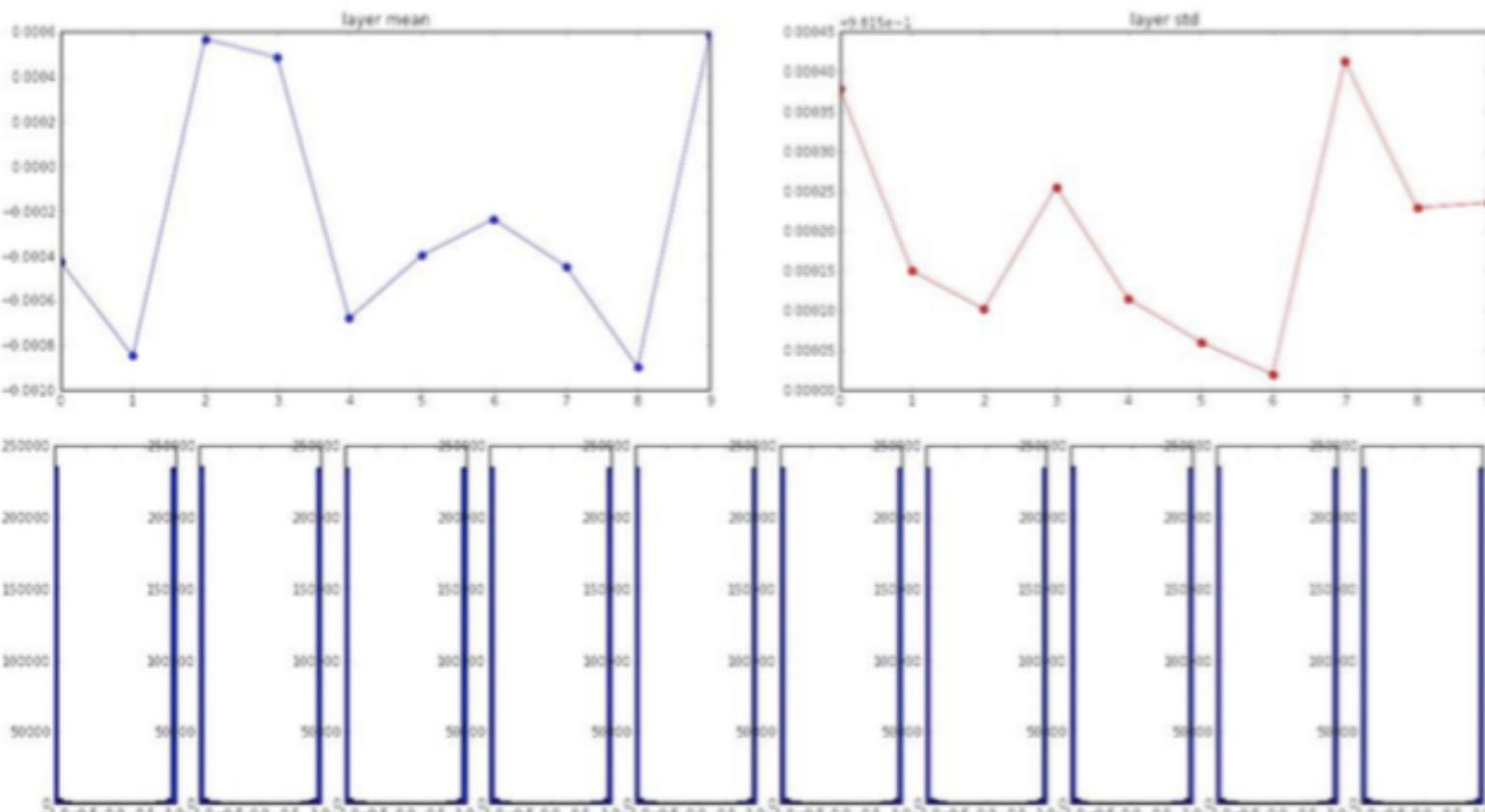


결과:

- 레이어를 통과할 수록 결과값이 0으로 수렴
- **backprop시 그라디언트가 0**

랜덤 값으로 초기화

- 예를 들어,
 - 실험 세팅: 레이어 10개와 tanh 활성 함수를 사용하는 네트워크
 - 모델2: 평균이 0, 표준편차는 1.0인 가우시안 분포에 의해 랜덤하게 초기화



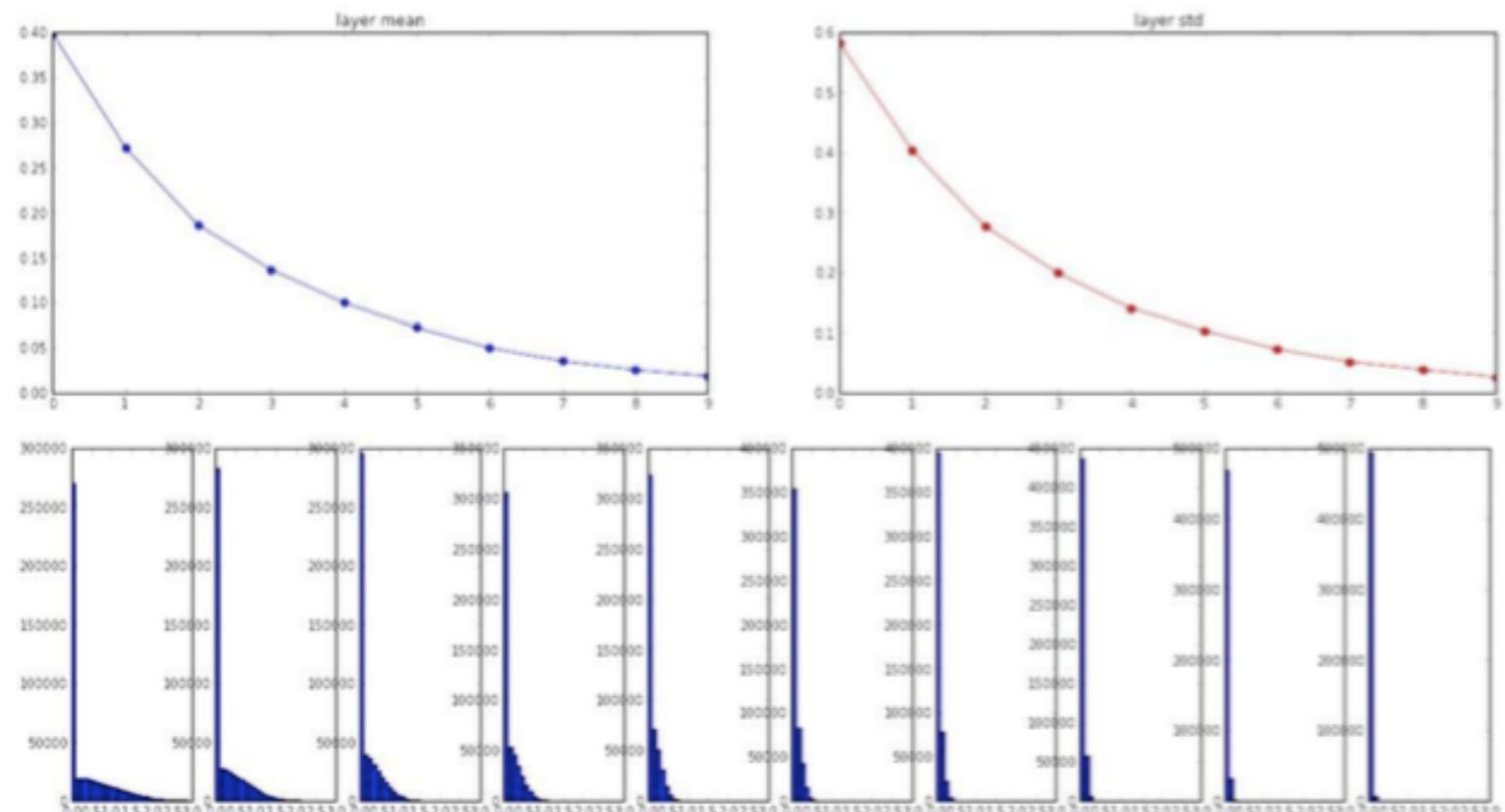
결과:

- 레이어를 통과할 수록 결과값이 -1, 1로 포화
- 시그모이드를 사용하면 그라디언트가 0에 가까워짐
(활성 함수에 들어오는 pre-activation이 양 극단)

Xavier 초기화

```
W = random.gaussian(n_input, n_output) / sqrt(n_input)
```

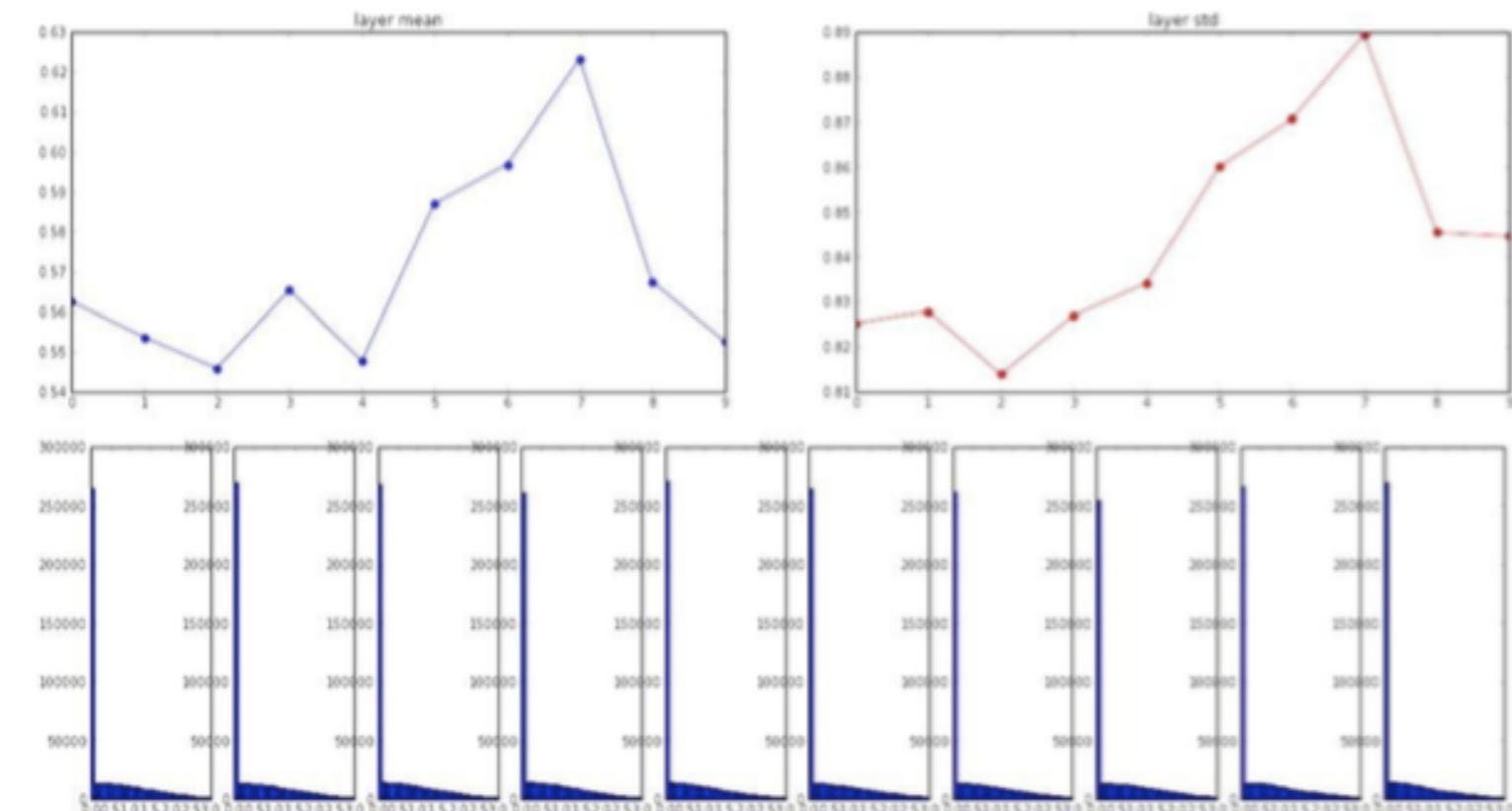
- Weight의 초기화를 이전 레이어의 뉴런 수에 맞게 조정
 - 뉴런 수가 많다면 weight 초기값을 낮춰 pre-activation이 너무 크지 않게
 - 반대의 경우 초기값을 높임
 - 하지만, 활성 함수를 고려하지 않은 방법 → ReLU를 사용하면 여전히 gradient vanishing 현상이 일어남



He 초기화

```
w = random.gaussian(n_input, n_output) / sqrt(n_input / 2)
```

- ReLU 활성 함수의 사용을 고려한 초기화 방법
 - Xavier와 비슷하나, 이전 레이어의 뉴런 개수의 절반만 고려
 - ReLU를 사용한다면 이 초기화 방법이 가장 무난한 방법

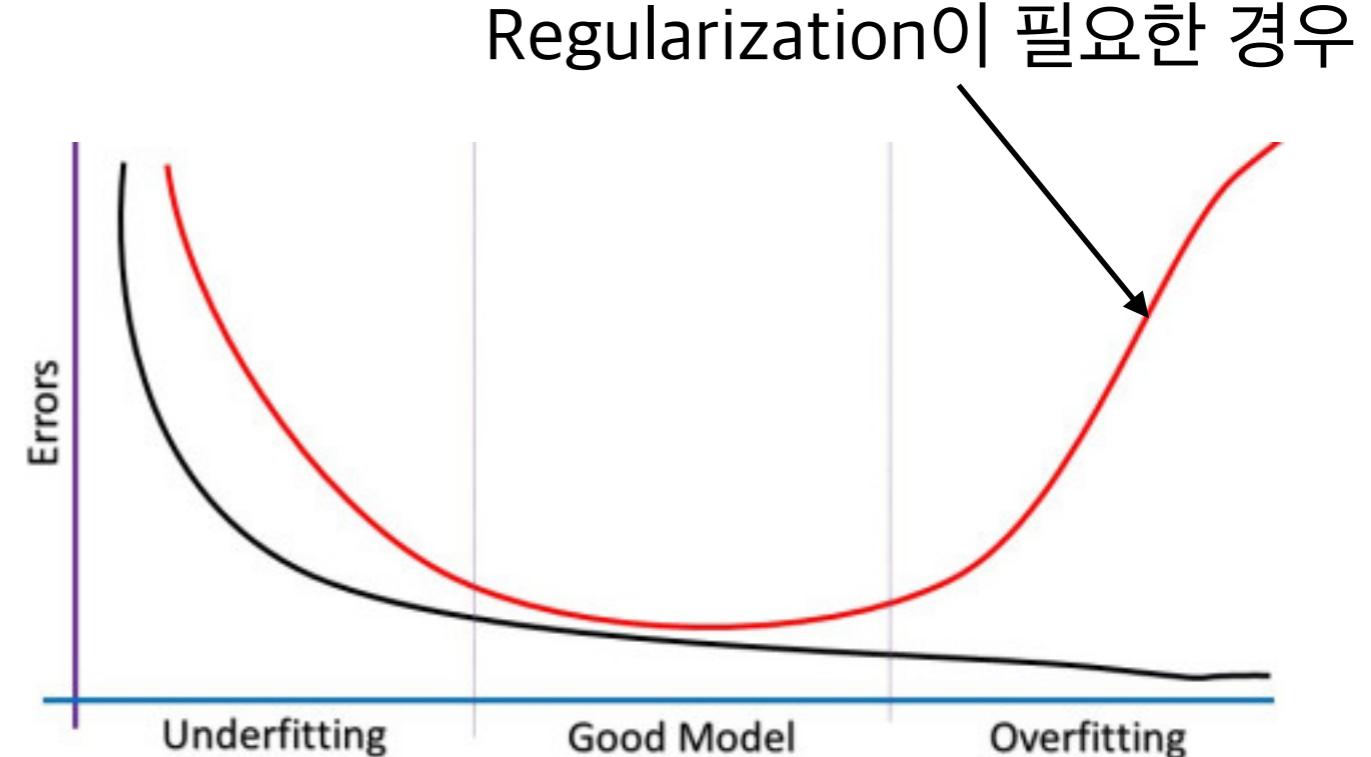


Bias 초기화

- Weight와 달리 정해진 룰이 크게 없음
 - 곱셈 연산이 수행되지 않아 gradient vanishing이 일어나지 않기 때문
 - 그냥 전부 0으로 초기화 하는 방법이 일반적임

Regularization

- 모델의 일반화 오류를 줄여 **과적합을 방지**하는 기법을 총칭
 - 머신러닝 특히 회귀 분석에서 매우 많이 쓰임
 - 딥러닝 또한 과적합을 방지하기 위해 regularization 기법을 사용함
- 자주 사용되는 regularization 기법
 - L1, L2 regularization
 - Dropout



L1/L2 Regularization

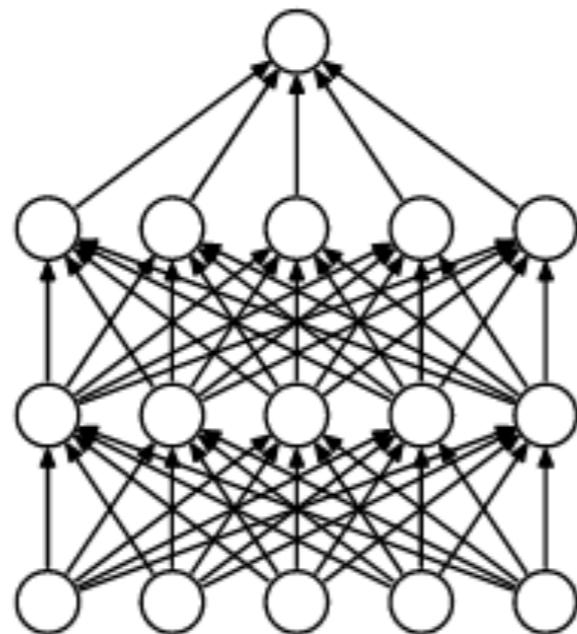
- 머신러닝에서 가장 일반적인 regularization 기법
- Loss 뒤에 weight에 대한 페널티 텀을 부여

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

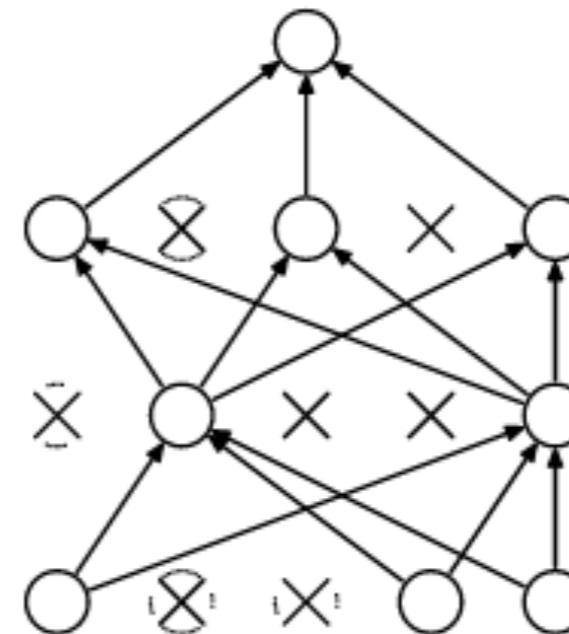
- L1: $R(W) = \sum_k \sum_l |W_{k,l}|$
- L2: $R(W) = \sum_k \sum_l W_{k,l}^2$
- Elastic: $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$
(L1과 L2의 혼합)

Dropout

- Forward-pass시 랜덤하게 뉴런의 연결을 끊어버리는 방법
 - 끊어진 길로는 정보(그라디언트)가 흐르지 않음
 - 간단한 구현: 뉴런의 결과값에 0을 곱하면 됨
 - 보통 0.5의 확률로 dropout을 적용



(a) Standard Neural Net



(b) After applying dropout.

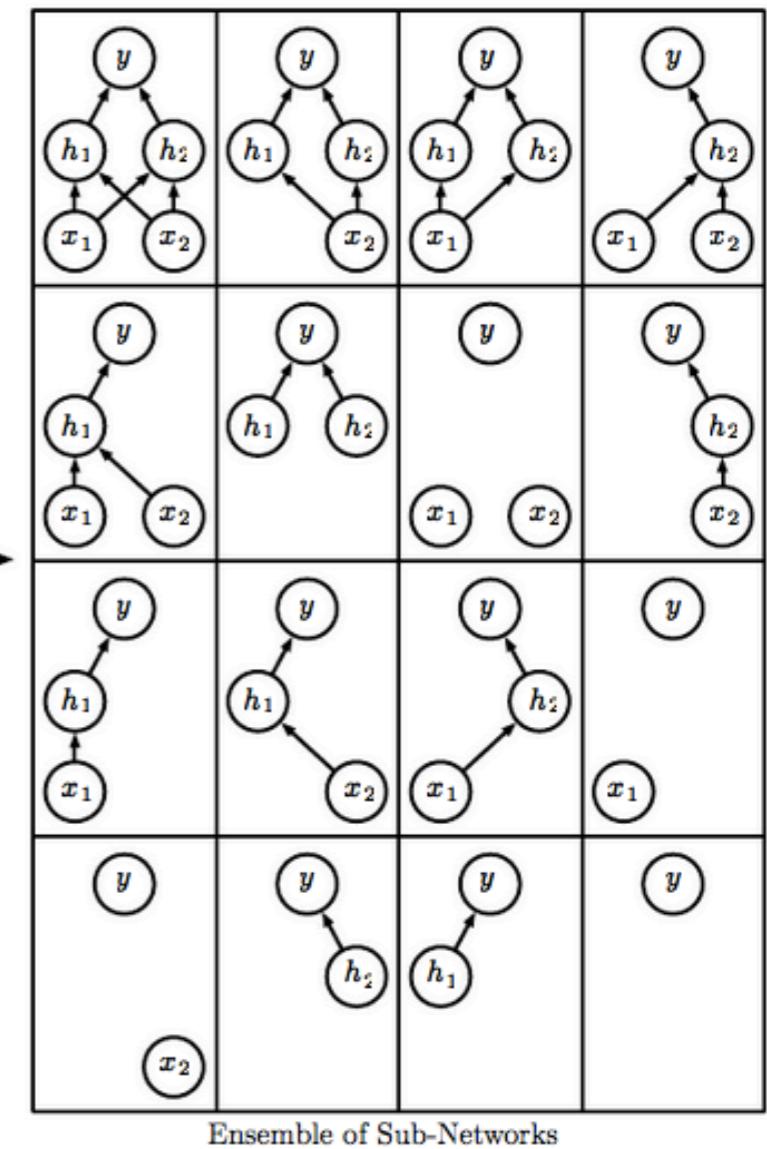
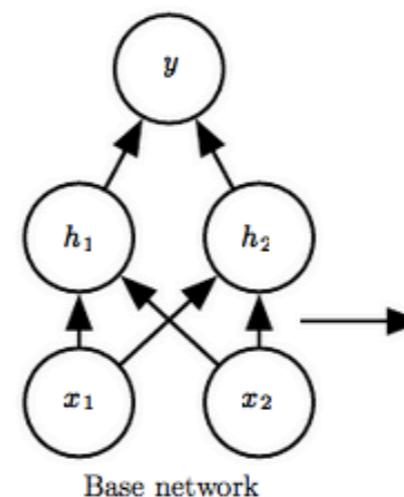
Dropout의 장점

- Dropout은 **양상블**의 근사(approximation)적인 방법

- 양상블: 여러 개의 모델을 혼합해서 일반화 에러를 줄임
(여러 모델이 예측한 결과값을 투표를 통해 최종 결과값으로 선정)

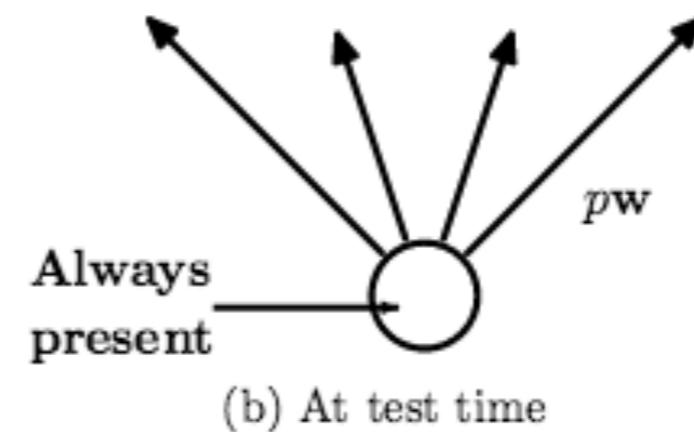
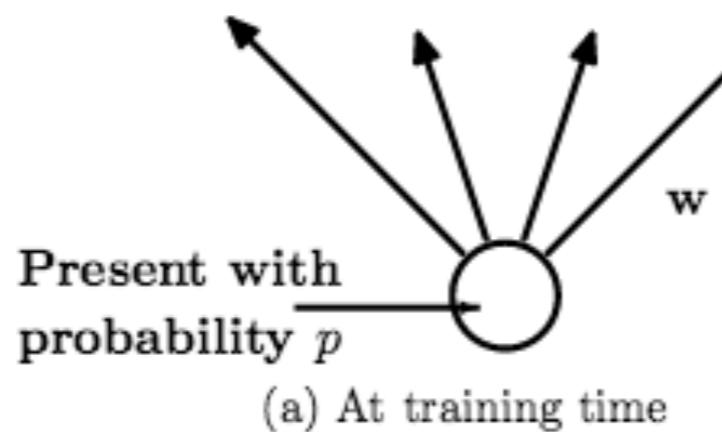
- 딥러닝은 학습이 오래걸리기 때문에 양상블은 비효율적

- Dropout에서 뉴런을 끊는 작업은 하나의 네트워크를 매우 많은 **서브 네트워크**로 만드는 작업과 유사함



Dropout: 테스트 단계

- 학습 단계에서는 뉴런을 제거하는 방법을 사용하지만,
테스트 단계는 다른 방법을 사용
 - 서브 네트워크를 전부 테스트해서 투표하는 방식은 너무 오래걸려서..
 - 대신 모든 뉴런을 활성화 시키고 각 뉴런의 결과값에 **확률 p 를 곱함**
 - 모든 서브 네트워크가 예측한 결과들의 **기대값**



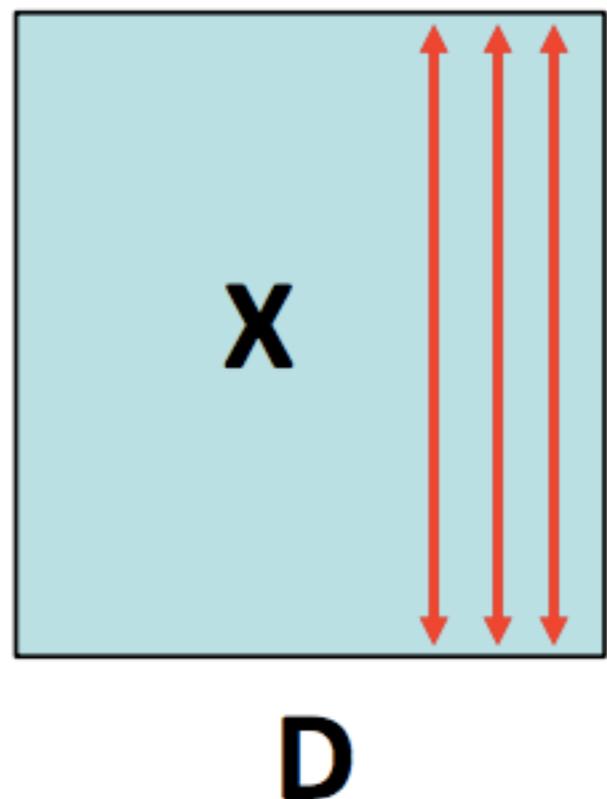
Gradient 문제 해결하기

- 지금까지 gradient vanishing을 풀기 위해 한 일
 - 활성 함수를 시그모이드 → ReLU로 교체
 - 초기화를 잘 하기
- Gradient exploding 문제는 learning rate를 낮게 조절
 - 문제는 낮은 learning rate에 의해 학습 수렴이 매우 느림

배치 정규화 (Batch Normalization)

- 다양한 초기화 전략은 레이어의 출력값을 **정규 분포**로 만들기 위해 한 작업
- 그럼 그냥 출력값을 정규화 하면 되지 않을까?
 - **배치 단위**로 정규화를 진행 (계산이 간편함)
 - 각각 차원을 독립으로 보고, 차원마다 정규화를 따로 진행

배치 정규화



1. 각 차원 단위로 평균과 분산을 구함
(차원은 서로 독립이라는 가정)
이 때 N은 배치 사이즈

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

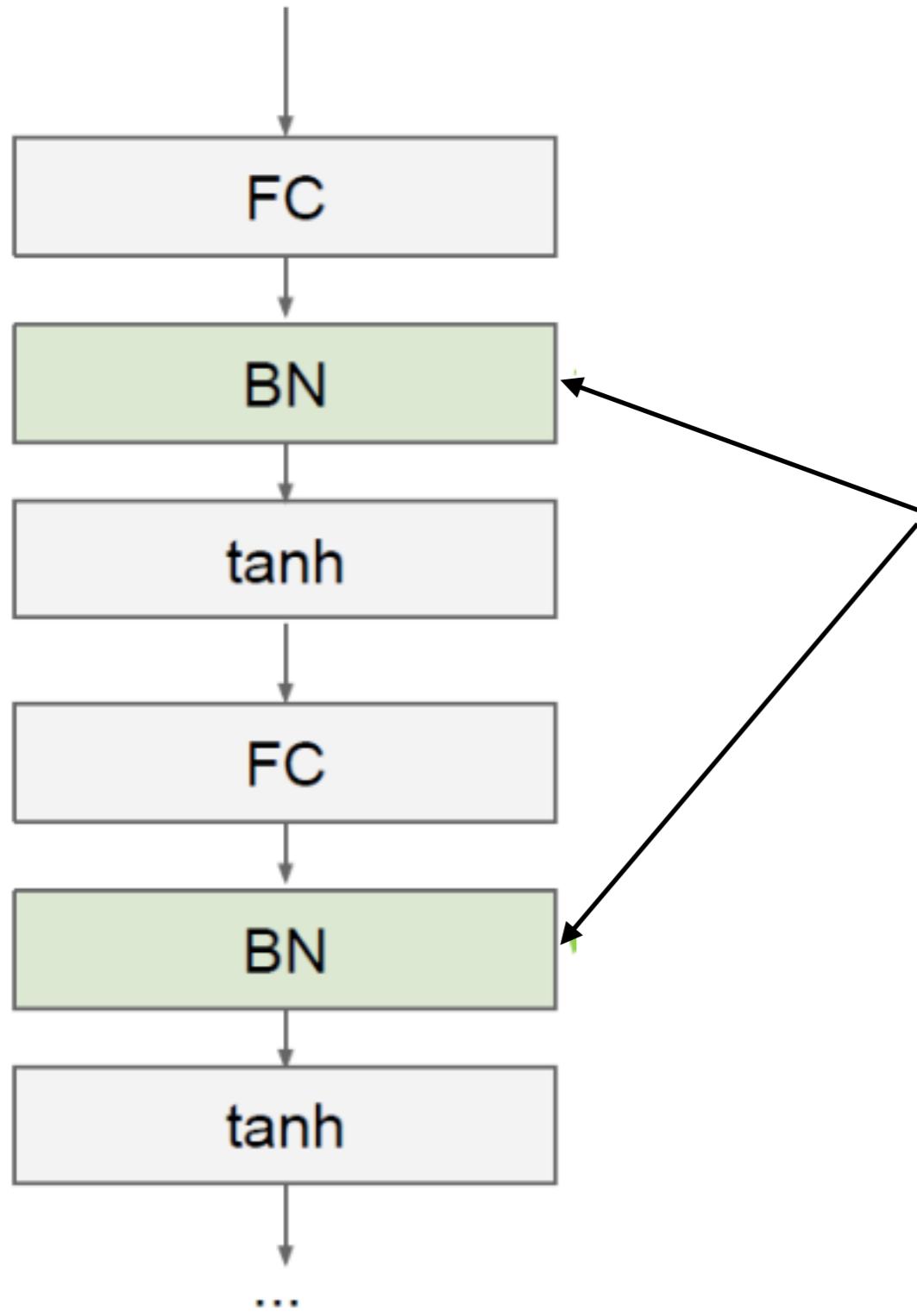
2. 정규화

배치 정규화

- 문제점:
 1. 평균과 분산을 0, 1로 고정하는 것은 비선형성을 깨뜨릴 수 있음
 2. 각 차원을 독립이라고 가정이 모델을 제한할 수 있음
- 해결책:
 - 정규화 된 값을 추가적인 파라미터를 사용하여 변형시킴

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \longrightarrow y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

배치 정규화



- 일반적으로 레이어를 거친 뒤, 배치 정규화를 수행하고 이를 활성 함수의 입력값으로 전달
- 배치 정규화를 일종의 레이어라 생각하고 구현 가능

배치 정규화의 장점

- Gradient vanishing 문제를 매우 잘 해결함
- Learning rate를 높게 주어도 학습을 잘 하기 때문에 수렴속도가 빠름
- 초기화 전략에 영향을 덜 받음 (물론 어느 정도 중요하지만)
- BN이 regularizer의 역할을 함
 - Dropout을 제거해도 비슷한 성능을 보이고, (항상 그렇지는 않음)
 - 일반적으로 성능이 상승!
 - 배치 단위로 정규화를 하면서 모델에 노이즈를 주는 것으로 해석 가능

배치 정규화: 테스트 단계

- Dropout처럼 학습/테스트 단계의 연산이 다름
- 학습 단계는 데이터가 배치 단위로 들어오기 때문에 배치의 평균, 분산을 구하는 것이 가능
- 테스트 단계에서는 배치 단위로 평균/분산 구하기가 어려움
 - 학습 단계에서 배치 단위의 평균/분산을 저장해놓기 (**이동 평균**)
 - 테스트 시에는 구해진 이동 평균을 사용하여 정규화

최적화 알고리즘

- 뉴럴 네트워크는 파라미터 최적화를 통해 학습됨
 - 기본적인 알고리즘은 Stochastic Gradient Descent (**SGD**)
- 다양한 SGD 파생 알고리즘 (Optimizer)
 - 모멘텀
 - RMSProp
 - Adam
 - ...
- 가장 많이 사용되는 Optimizer
 - SGD, 모멘텀, RMSProp, Adam ?

모멘텀

- 물리학 관점에서 영감을 받은 최적화 알고리즘
 - 공이 받는 힘은 공의 속도를 결정하고, 속도가 공의 위치를 결정
 - 관성의 법칙에 의해 평평한 공의 이전 속도가 현재 속도에 영향을 줌
 - 평평한 위치 (plateau)의 탈출이 더 빠름

```
# SGD  
x += -learning_rate * dx  
  
# 모멘텀 알고리즘  
vel = mu * vel - learning_rate * dx  
x += vel
```

모멘텀

- 물리학 관점에서 영감을 받은 최적화 알고리즘
 - 공이 받는 힘은 공의 속도를 결정하고, 속도가 공의 위치를 결정
 - 관성의 법칙에 의해 평평한 공의 이전 속도가 현재 속도에 영향을 줌
 - 평평한 위치 (plateau)의 탈출이 더 빠름

```
# SGD  
x += -learning_rate * dx
```

```
# 모멘텀 알고리즘  
vel = mu * vel - learning_rate * dx  
x += vel
```

속도 = 마찰력*이전 속도 - 현재 속도
마찰계수 (μ)는 주로 0.95, 0.99를 선택

RMSProp

- 빈번하게 신호가 들어오거나 높은 그라디언트가 들어오는 weight에 페널티 부여
 - 이동평균을 사용하여 learning rate가 급격하게 떨어지는 현상 방지
 - $1e-8$: 분모가 0이 되어 생기는 오류를 방지하기 위해
 - p : 이동평균을 계산할 때 필요한 하이퍼파라미터로 0.99 정도를 사용

```
# RMSProp
accm = p * accm + (1-p) * dx**2
x    += - learning_rate / sqrt(accm + 1e-8) * dx
```

RMSProp

- 빈번하게 신호가 들어오거나 높은 그라디언트가 들어오는 weight에 페널티 부여
 - 이동평균을 사용하여 learning rate가 급격하게 떨어지는 현상 방지
 - $1e-8$: 분모가 0이 되어 생기는 오류를 방지하기 위해
 - p : 이동평균을 계산할 때 필요한 하이퍼파라미터로 0.99 정도를 사용

```
# RMSProp
```

페널티 정도 계산

```
accm = p * accm + (1-p) * dx**2
x    += - learning_rate / sqrt(accm + 1e-8) * dx
```

RMSProp

- 빈번하게 신호가 들어오거나 높은 그라디언트가 들어오는 weight에 페널티 부여
 - 이동평균을 사용하여 learning rate가 급격하게 떨어지는 현상 방지
 - $1e-8$: 분모가 0이 되어 생기는 오류를 방지하기 위해
 - p : 이동평균을 계산할 때 필요한 하이퍼파라미터로 0.99 정도를 사용

```
# RMSProp
accm = p * accm + (1-p) * dx**2
x    += - learning_rate / sqrt(accm + 1e-8) * dx
```

페널티 정도 계산

weight에 페널티 부여

Adam

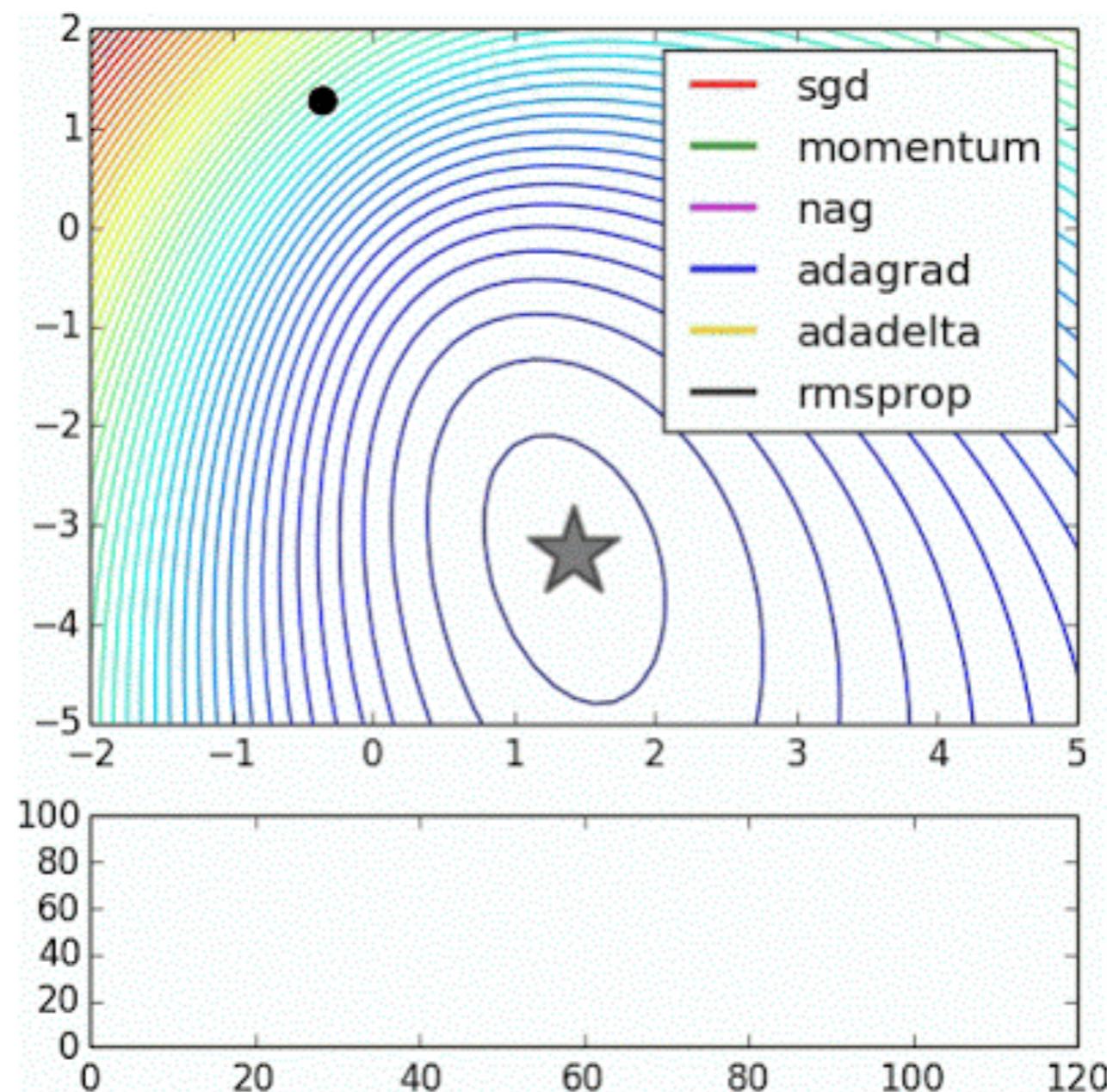
- RMSProp와 모멘텀을 결합한 알고리즘
 - 하이퍼파라미터로 beta_1, beta_2가 있음
 - 수렴 속도가 비교적 빠르기 때문에 최근 많이 사용되는 알고리즘

```
# Adam (실제로 사용할 때는 약간 변경된 버전을 사용)
```

```
m = beta_1*m + (1-beta_1)*dx
v = beta_2*v + (1-beta_2)*(dx**)
x += - learning_rate * m / (sqrt(v) + 1e-7)
```

Optimizer

Optimizer

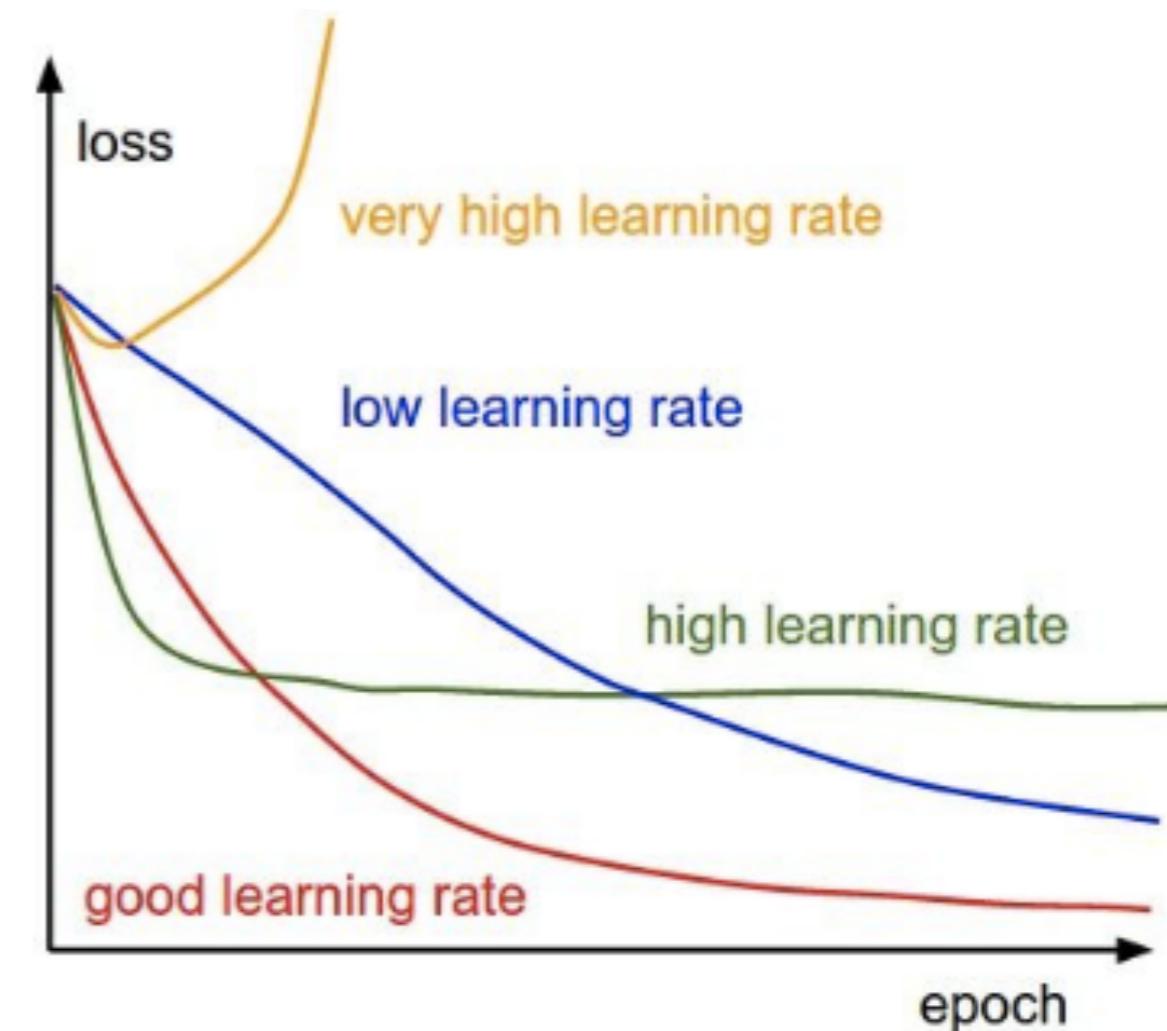


네트워크 구성 전략

- 최근 정형화 된 네트워크 구성 예시
 - 활성 함수는 **ReLU** (혹은 ReLU 베이스)
 - weight 초기화는 **He 초기화**, bias 초기화는 0
 - 처음과 마지막을 제외한 모든 레이어에 **배치 정규화** 사용
 - 배치 정규화를 사용했다면 dropout은 선택 사항
 - Optimizer는 **Adam** 이 최근 많이 사용되고 있음 (빠른 초기 수렴 때문)

네트워크 학습 전략

- 최적의 Learning rate?

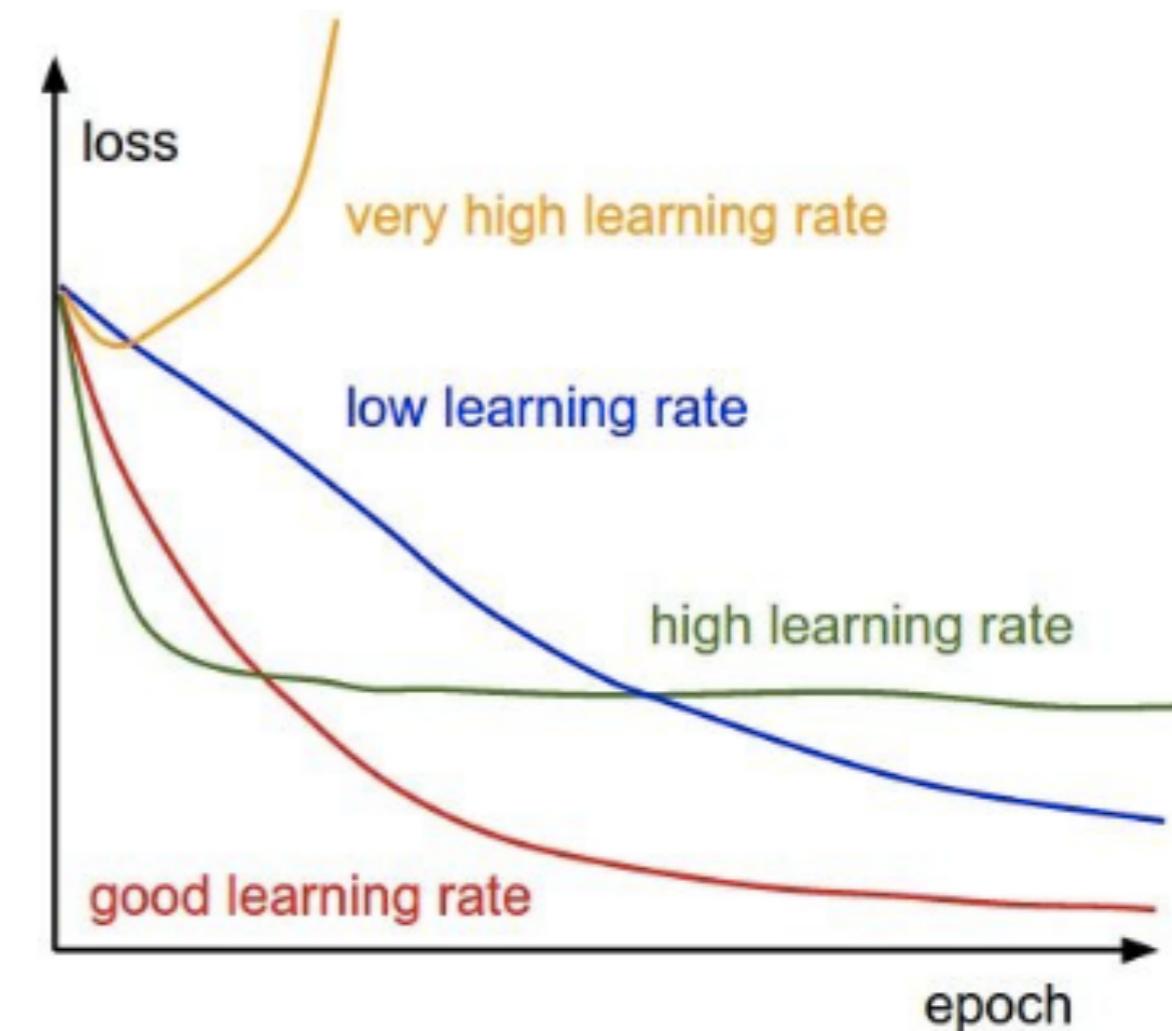


네트워크 학습 전략

- 최적의 Learning rate?

학습 초기 높은 learning rate

학습을 진행하면서 learning rate 낮추기
(learning rate decay)



네트워크 학습 전략

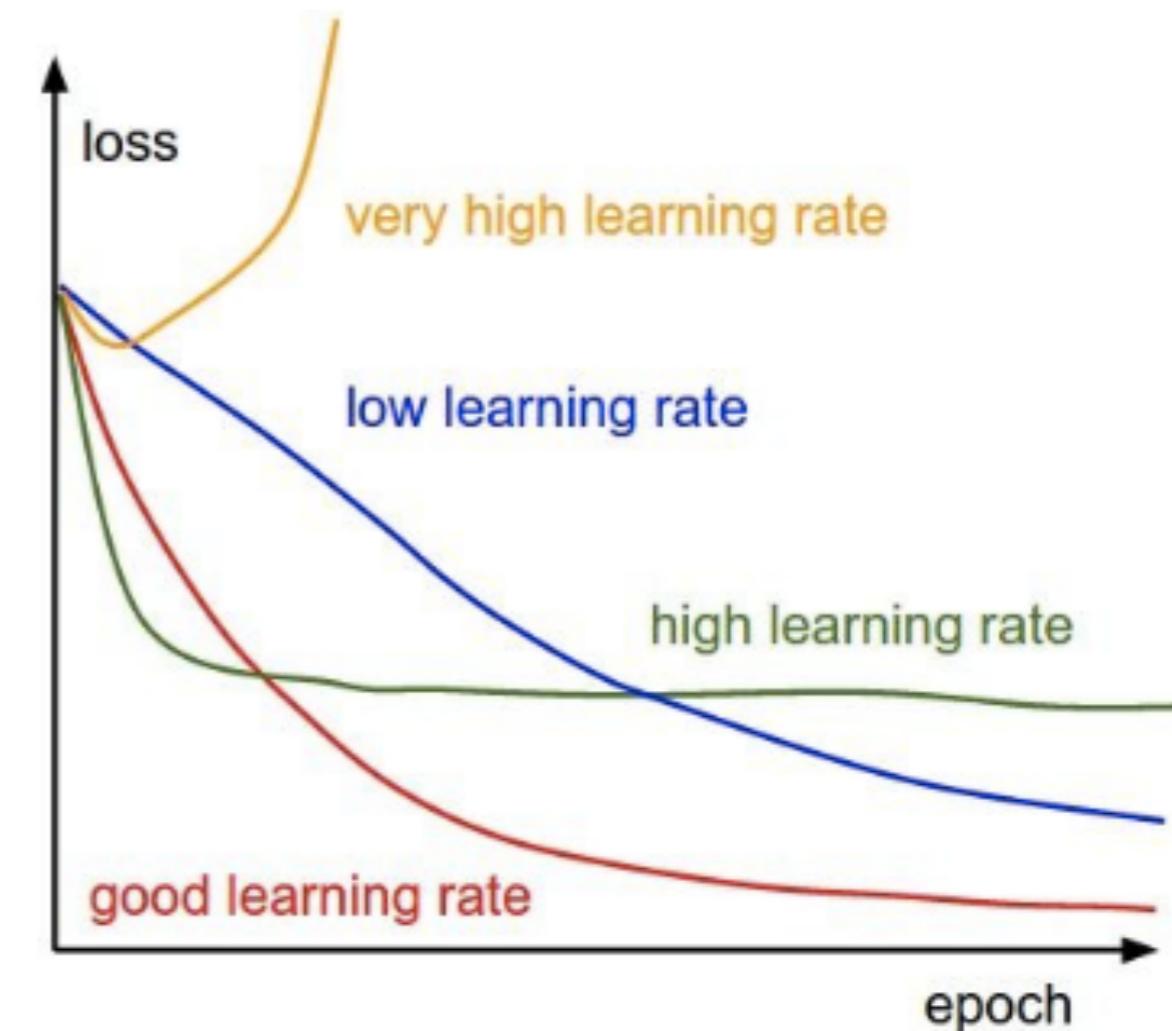
- 최적의 Learning rate?

학습 초기 높은 learning rate

학습을 진행하면서 learning rate 낮추기
(learning rate decay)

Decay 전략

- 일정 epoch 마다 줄이기 (절반, 1/10 등)
- 매 스텝 마다 점진적으로 감소
- 테스트 성능이 좋아지지 않으면 감소

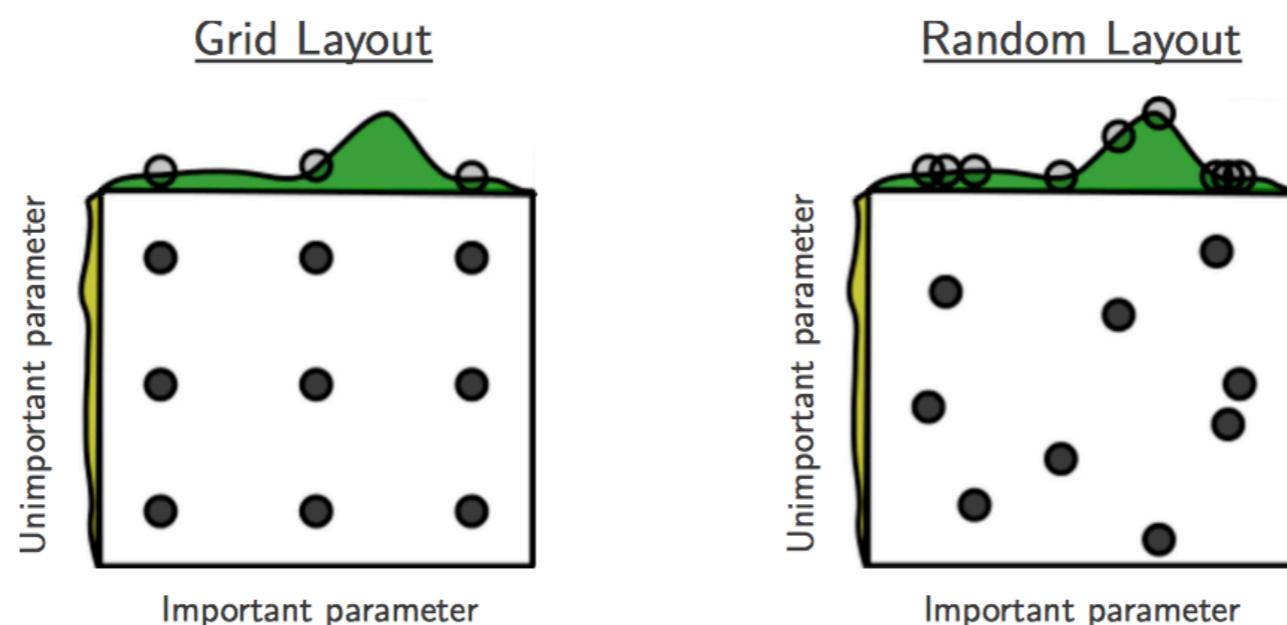


네트워크 학습 전략

- 하이퍼파라미터 튜닝
 - 최적의 하이퍼파라미터는 네트워크의 성능에 직결되는 중요한 세팅값
 - 제일 중요한 하이퍼파라미터는 learning rate?
- 하이퍼파라미터 튜닝 하는법
 - 휴리스틱하게
 - (노가다)

그리드 탐색 vs 랜덤 탐색

- 그리드 탐색 vs 랜덤 탐색
 - **그리드 탐색**: 일정한 간격으로 하이퍼파라미터를 설정하고 모델 학습 최적의 하이퍼파라미터 값을 찾지 못할 수 있음
 - **랜덤 탐색**: 랜덤하게 하이퍼파라미터 값을 설정
- 랜덤 탐색이 더 좋은 방법? 그러나 엄청 큰 차이는 없는듯



그리드 탐색 vs 랜덤 탐색

```
# 그리드 탐색
lr_candidates = [0.001, 0.002, 0.003]
for lr in lr_candidates:
    network.train(lr=lr)
    accuracy = network.test()
```

```
# 랜덤 탐색
for i in range(3):
    lr = 10**random.randint(-3, -1) # 0.001 ~ 0.1
    network.train(lr=lr)
    accuracy = network.test()
```

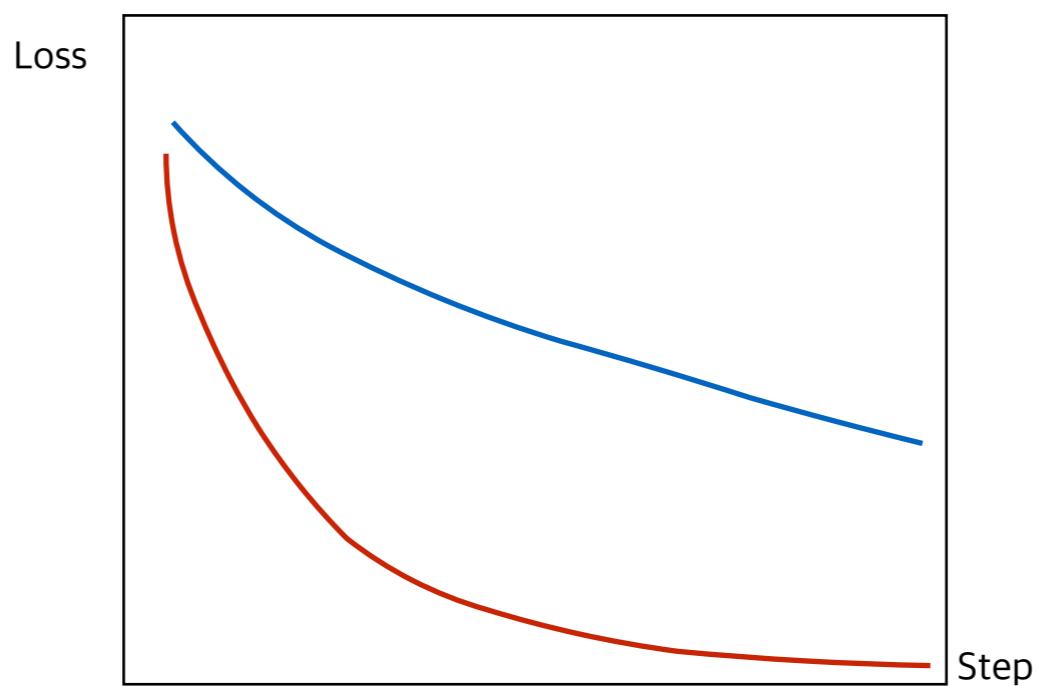
일반적으로 로그공간 상에서 랜덤값 선택



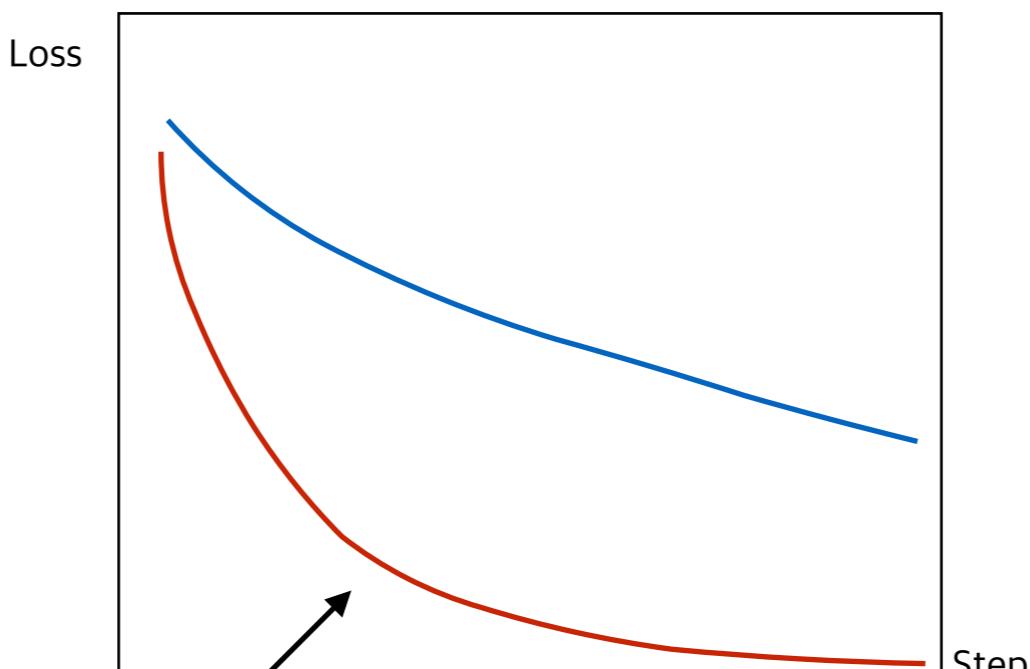
하이퍼파라미터 튜닝

- 하이퍼파라미터를 어떻게 최적화 시킬까?
 1. 처음에는 범위를 대충 줄여나가기 ($0.1 \sim 0.00001 \rightarrow 0.01 \sim 0.0001$)
 - epoch을 약간만 (5~10?) 돌려서 간단하게 확인만 하는 작업
 2. 범위를 대충 좁힌 뒤 정밀하게 하이퍼파라미터 탐색
 - 처음보다 더 오래 학습
 - 일반적으로 learning rate의 초기 범위는 $0.01 \sim 0.0001$ 이 적당한듯?
(네트워크에 따라, 상황에 따라 변동이 있음)

Loss 그려보기

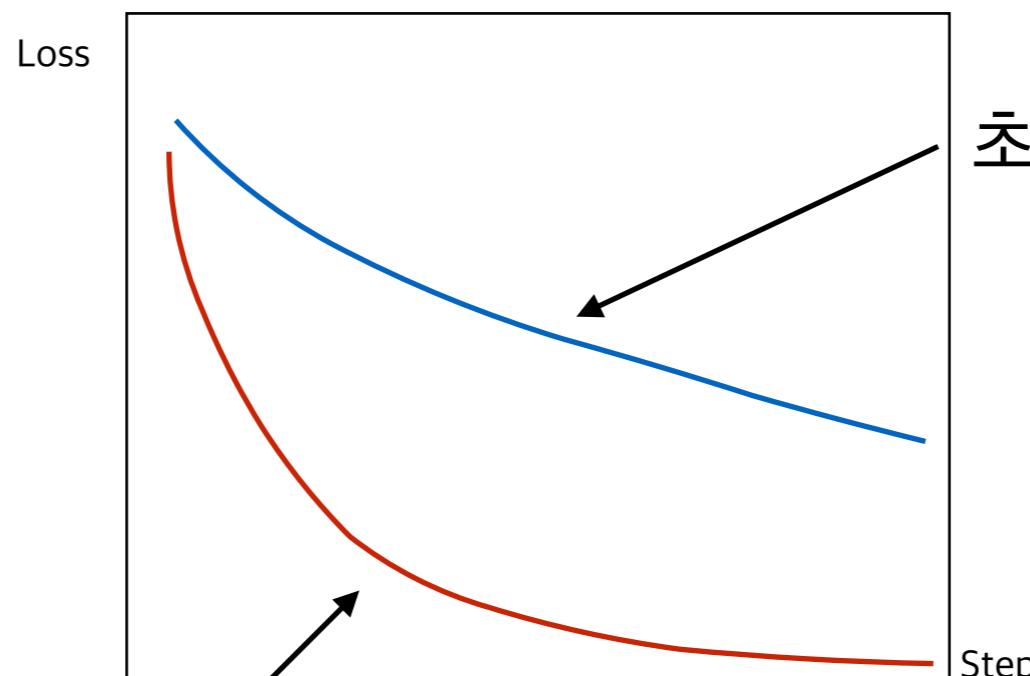


LOSS 그려보기



일반적인 Loss 커브 형태

LOSS 그려보기

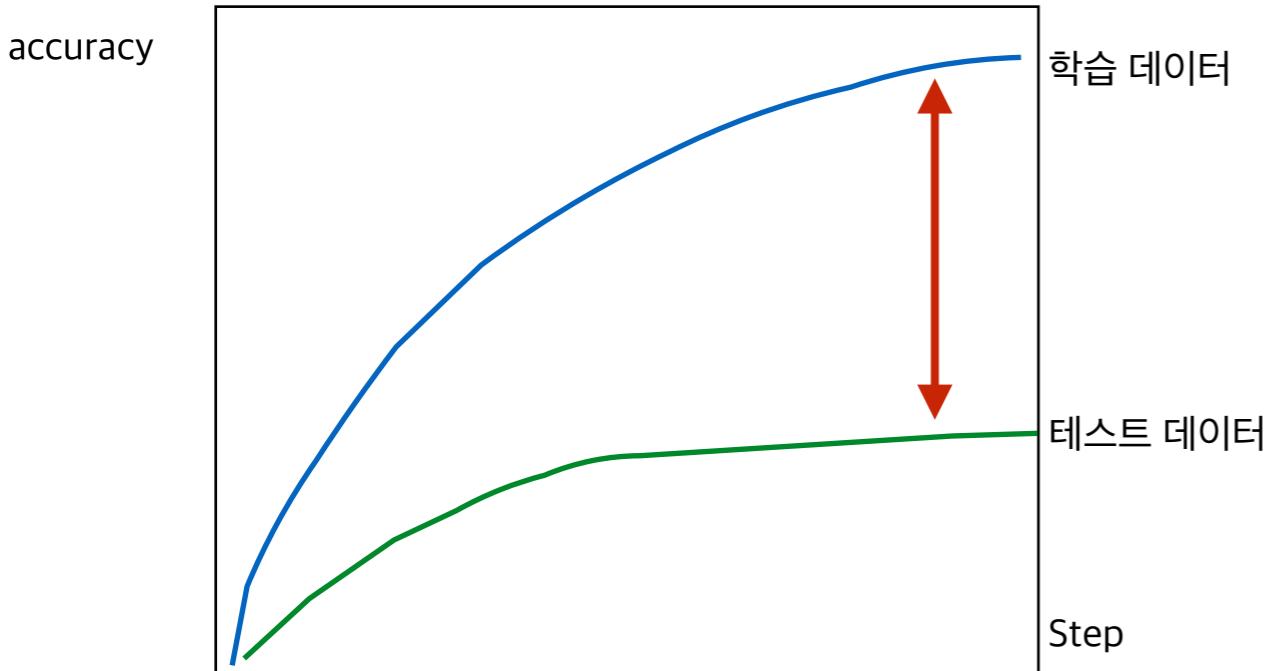


일반적인 Loss 커브 형태

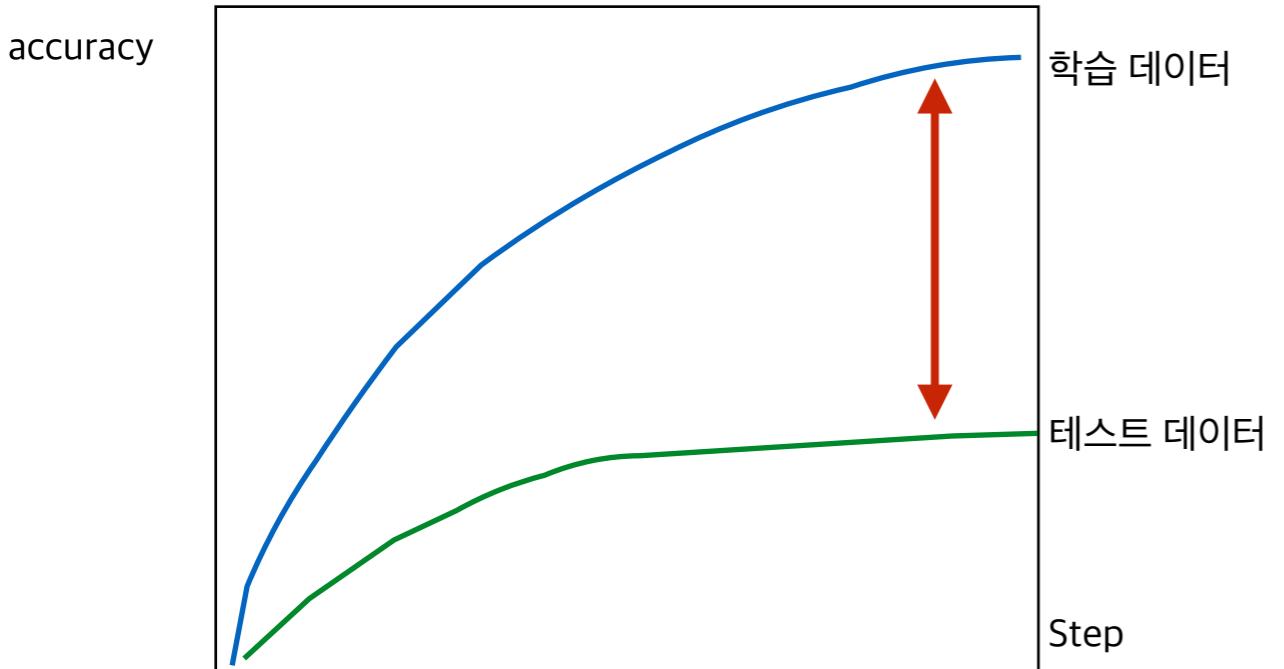
초기 learning rate가 작은 경우

Accuracy 그래프 그려보기

Accuracy 그래프 그려보기



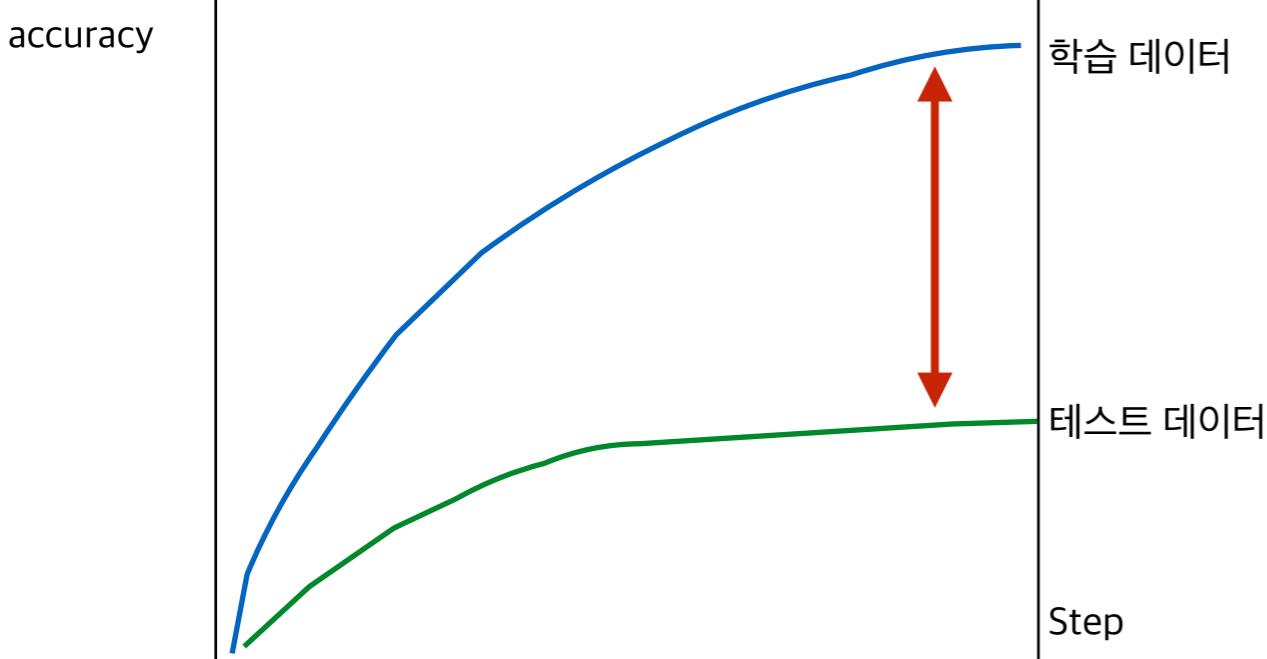
Accuracy 그래프 그려보기



Overfit!

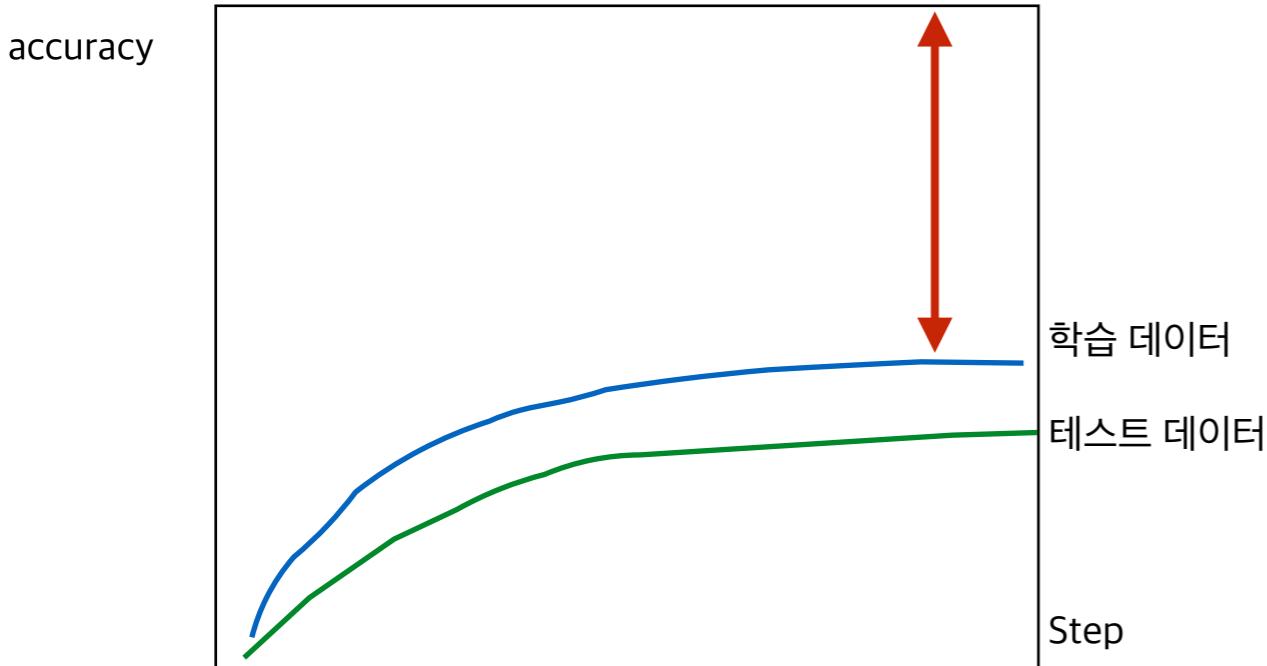
- **데이터 늘리기**
(안된다면 data augmentation 라도..)
- 모델이 데이터 수에 비해 너무 큰 것은 아닌가?
(모델 크기 줄여보기)
- Regularizer 추가 (dropout 등)

Accuracy 그래프 그려보기

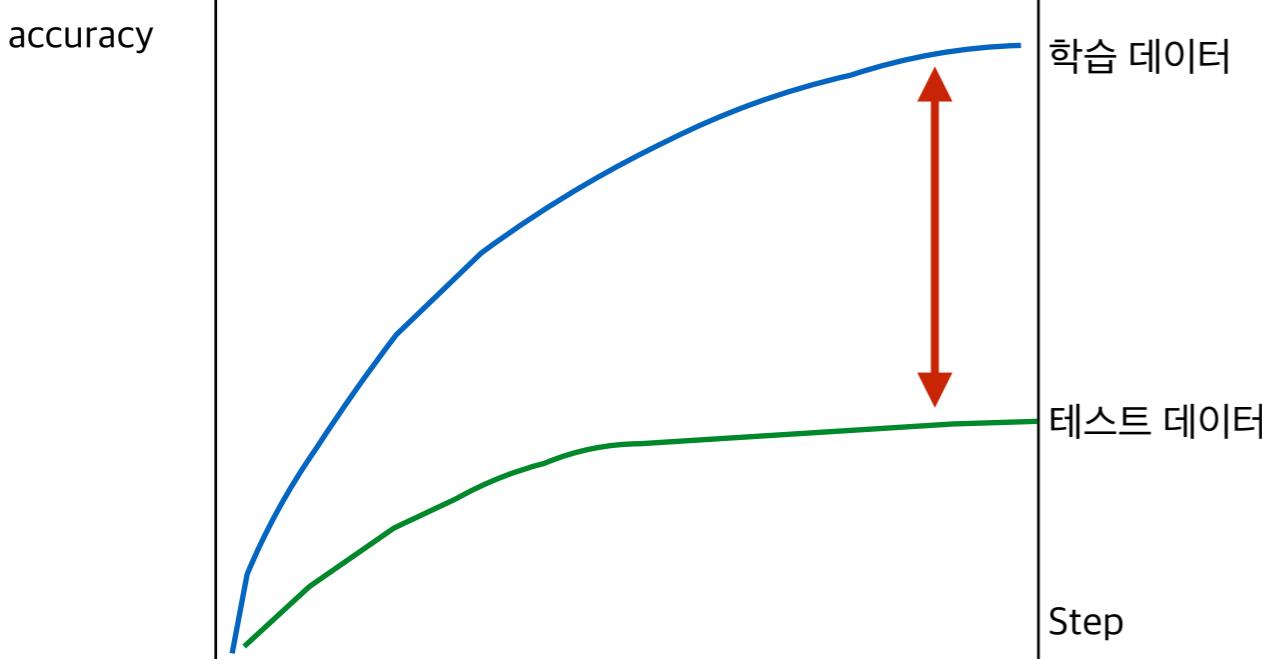


Overfit!

- **데이터 늘리기**
(안된다면 data augmentation 라도..)
- 모델이 데이터 수에 비해 너무 큰 것은 아닌가?
(모델 크기 줄여보기)
- Regularizer 추가 (dropout 등)

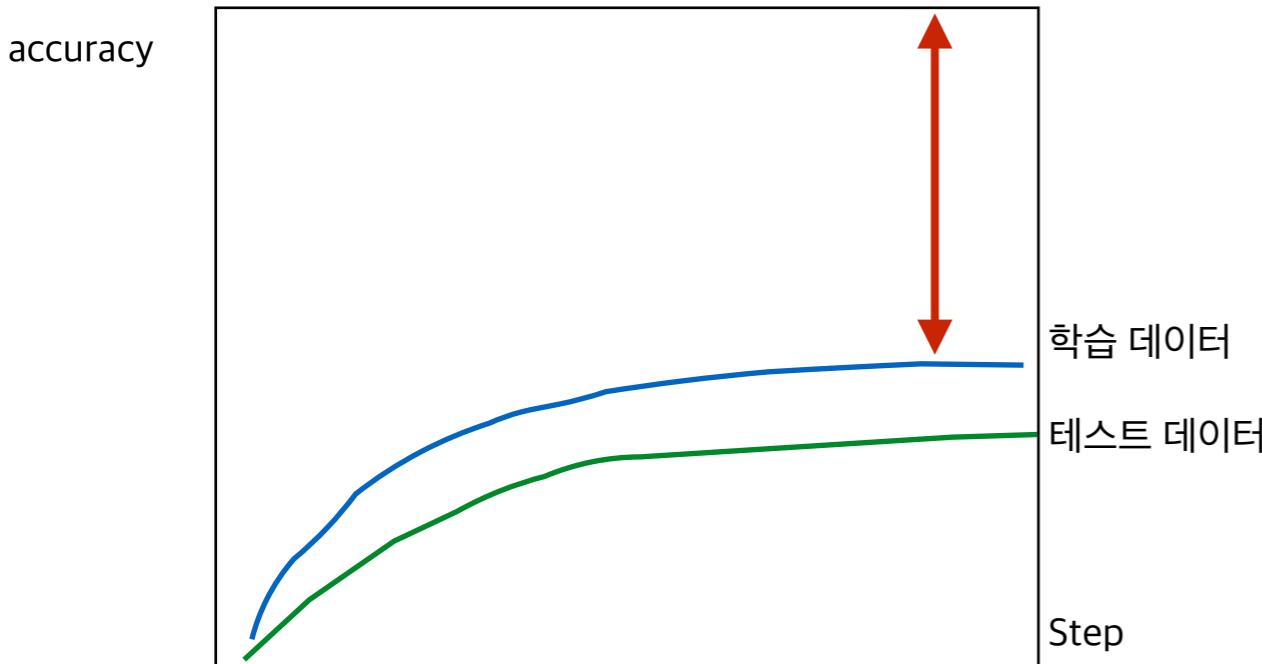


Accuracy 그래프 그려보기



Overfit!

- **데이터 늘리기**
(안된다면 data augmentation 라도..)
- 모델이 데이터 수에 비해 너무 큰 것은 아닌가?
(모델 크기 줄여보기)
- Regularizer 추가 (dropout 등)



Underfit!

- **모델 크기 키우기**
- 학습을 더 오래 시키기

Summary

- 활성 함수는 시그모이드 계열과 ReLU가 있음
 - Gradient vanishing을 해결하기 위해 ReLU를 권장
- Weight 초기화는 랜덤 / bias 초기화는 일반적으로 0으로
 - Xavier / He 초기화를 권장
- Dropout / 배치 정규화
 - 특수한 경우가 아니면 배치 정규화를 사용하는 것이 좋음