딥러닝기초

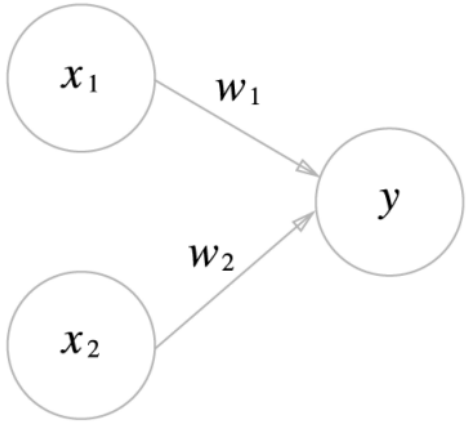
BOAZ BASE 분석 세션

퍼셉트론

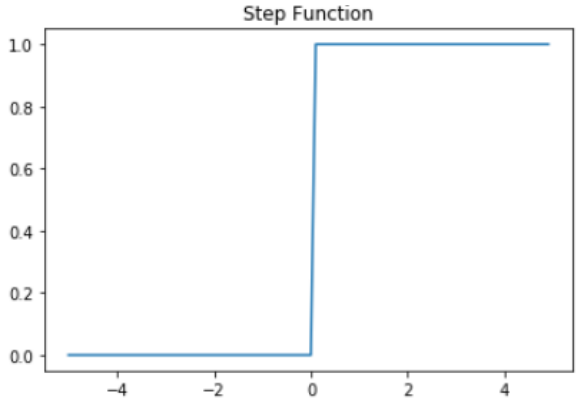
퍼셉트론이란?

입력 값에 대해 가중치를 적용한 뒤, 결과를 전달하는 방식

다수의 신호 입력



하나의 신호 출력



$$y = \begin{cases} 0 & (w_1 x_1 + w_2 x_2 \leq \theta) \\ 1 & (w_1 x_1 + w_2 x_2 > \theta) \end{cases}$$

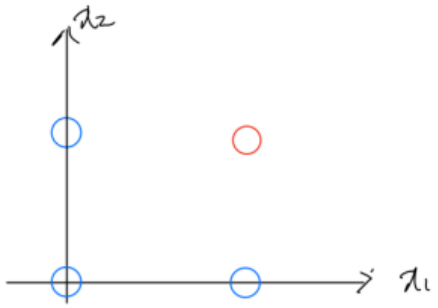
신호의 합이 임계치보다 클 때만 출력

단순한 논리회로

적절하게 매개변수만 조절해주면 모두 표현 가능하다!

AND

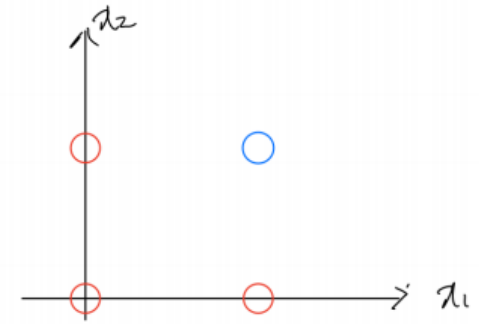
x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1



$(w_1, w_2, \theta) = (0.5, 0.5, 0.7)$

NAND

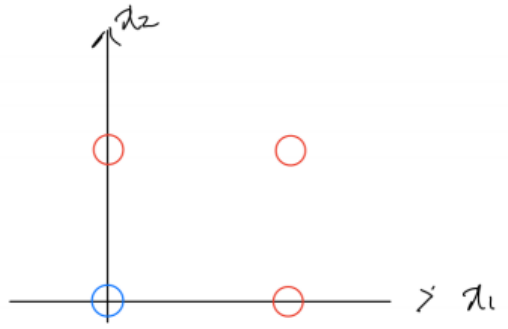
x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0



$(w_1, w_2, \theta) = (-0.5, -0.5, -0.7)$

OR

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

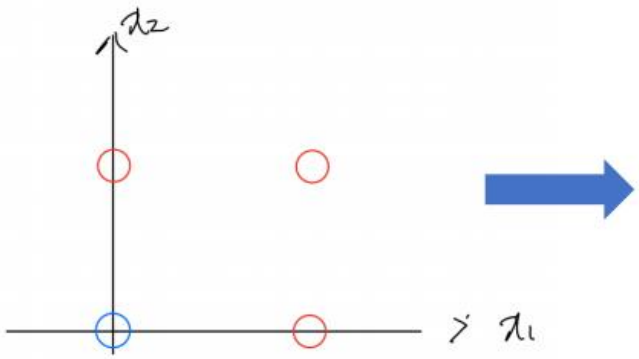


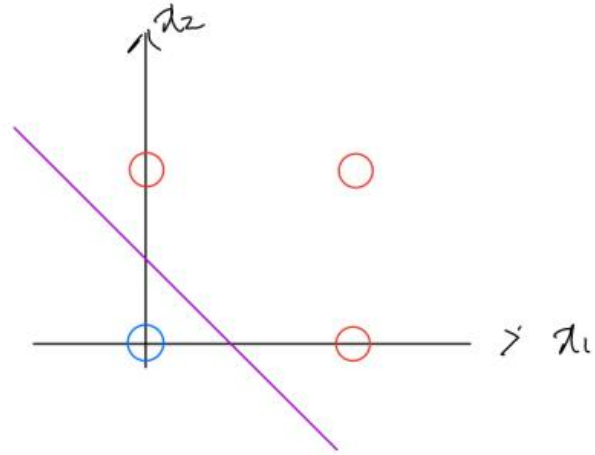
$(w_1, w_2, \theta) = (0.5, 0.5, 0.2)$

퍼셉트론의 한계

OR

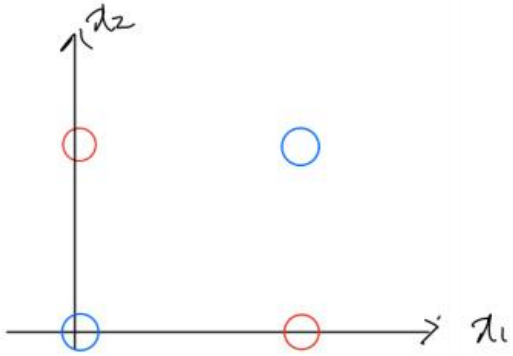
x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1





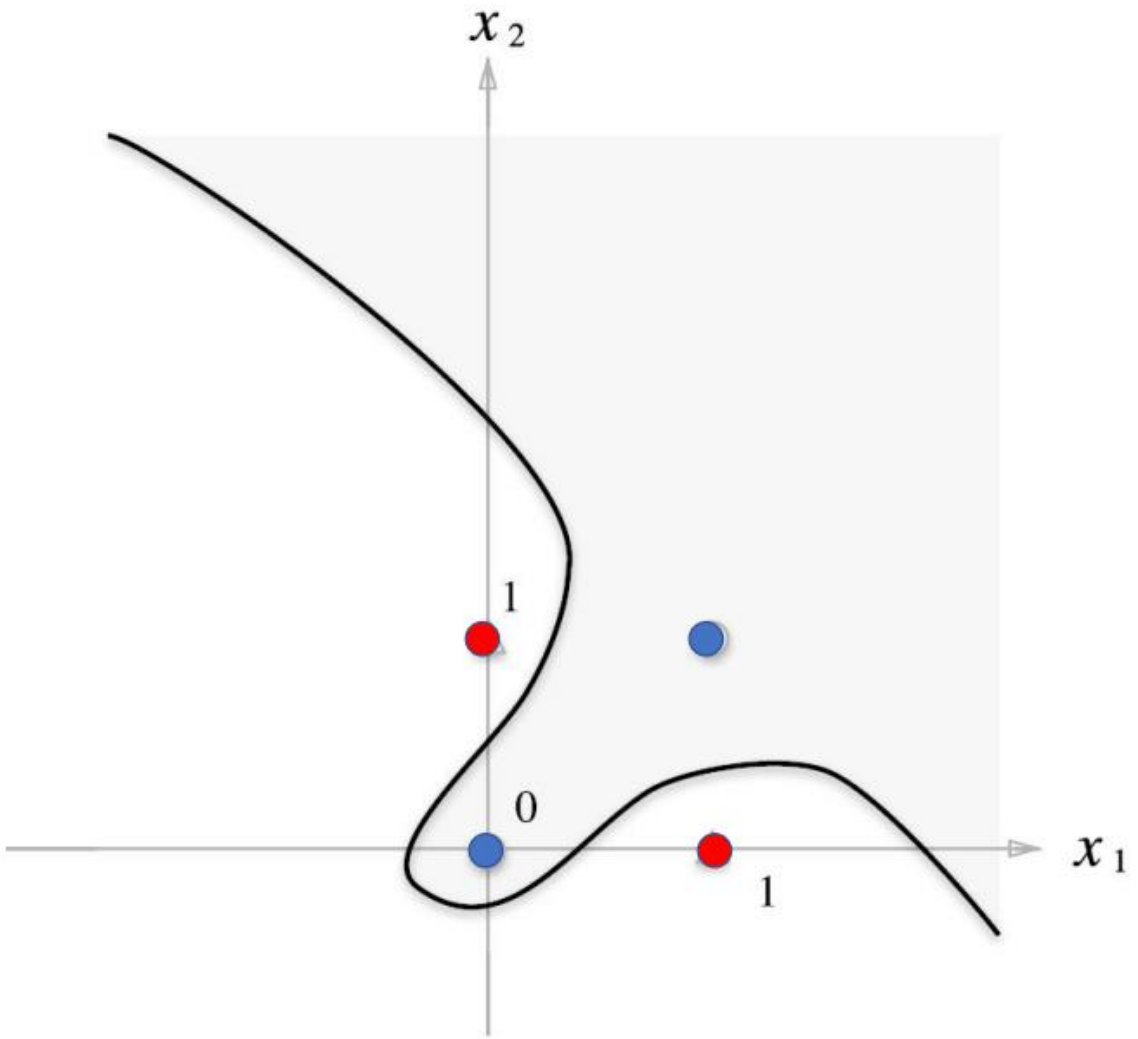
XOR

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



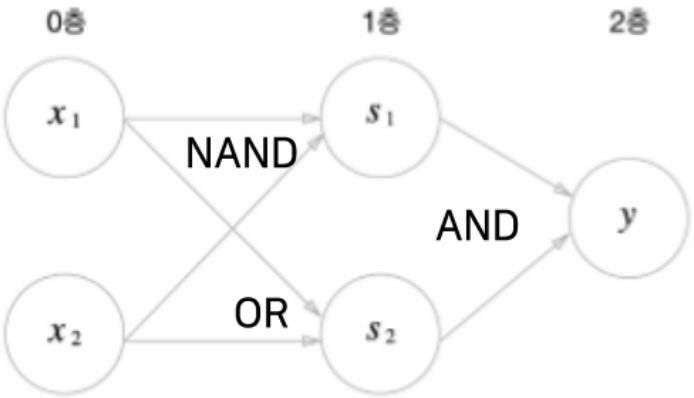
$$y = \begin{cases} 0 & (-0.5 + x_1 + x_2 \leq 0) \\ 1 & (-0.5 + x_1 + x_2 > 0) \end{cases}$$

퍼셉트론의 한계



다층 퍼셉트론

- 퍼셉트론 여러개를 이어 붙여 층을 쌓는다!

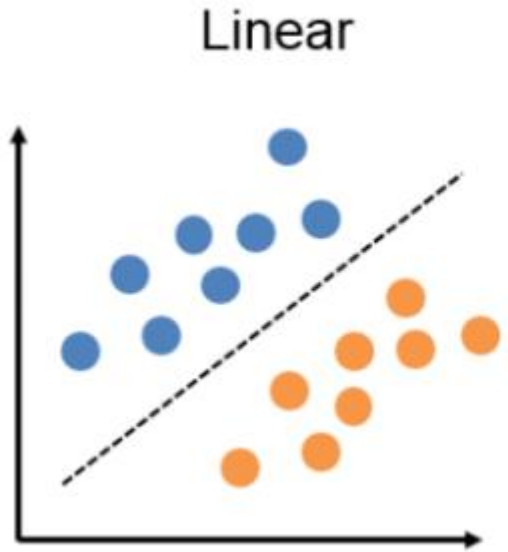


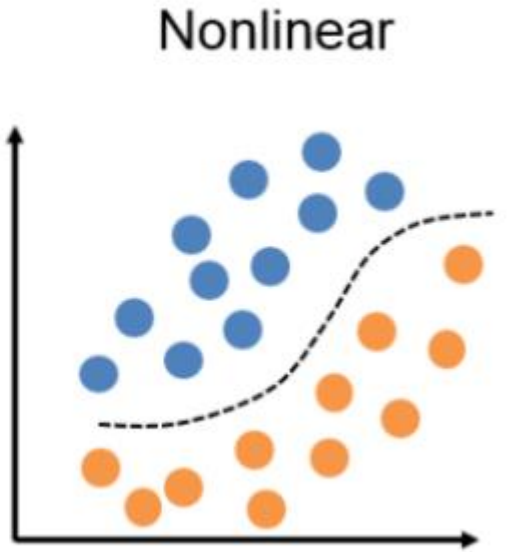
x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

선형 분류만으로는 풀지 못했던 문제를 비선형적으로 해결

다층 퍼셉트론

- 다층 레이어
- 선형성 극복
- 다양한 결과 가능

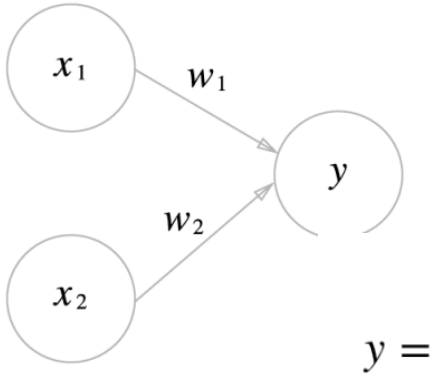




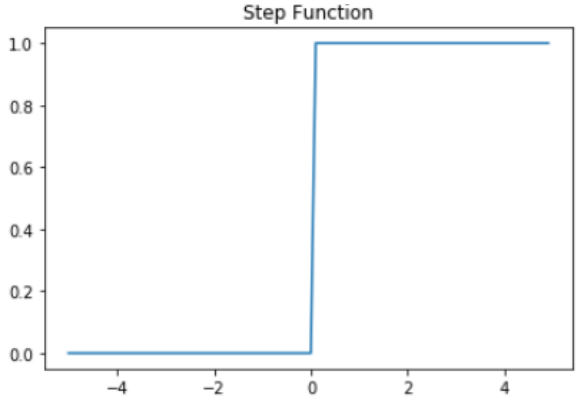
신경망

퍼셉트론에서 신경망으로

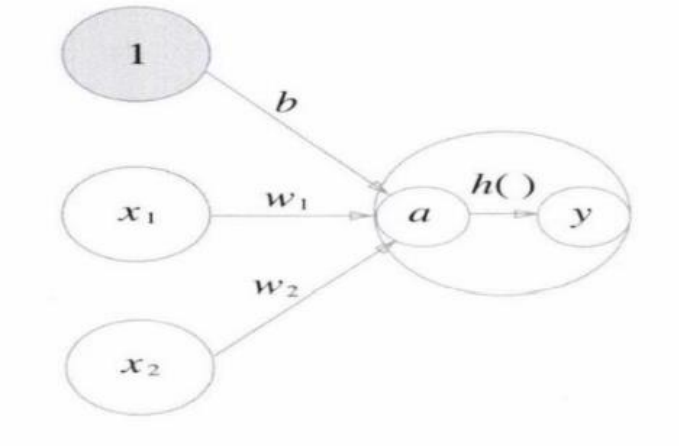
퍼셉트론



$$y = \begin{cases} 0 & (w_1 x_1 + w_2 x_2 \leq \theta) \\ 1 & (w_1 x_1 + w_2 x_2 > \theta) \end{cases}$$



신경망



$$y = h(b + w_1 x_1 + w_2 x_2)$$

$$h(x) = \begin{cases} 0 & (x \leq 0) \\ 1 & (x > 0) \end{cases}$$

활성화 함수!

활성화 함수

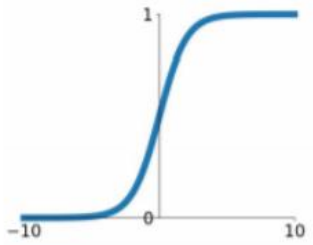
노드에 들어오는 값들에 대해 다음 레이어로 전달하기 전 비선형 함수를 사용

사용 이유 : 선형 함수를 사용할 경우 층을 깊게하는 의미가 줄어든다.

딥러닝에서 활성화 함수는 비선형 함수라고 생각하면 된다.

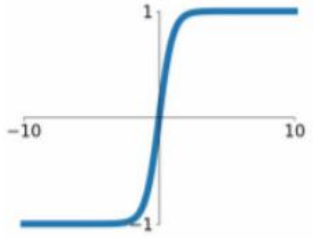
Sigmoid

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



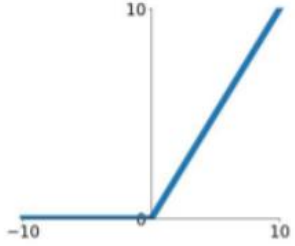
tanh

$$\tanh(x)$$



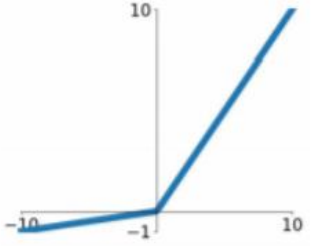
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

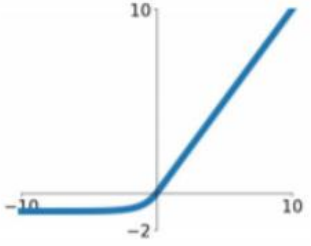


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

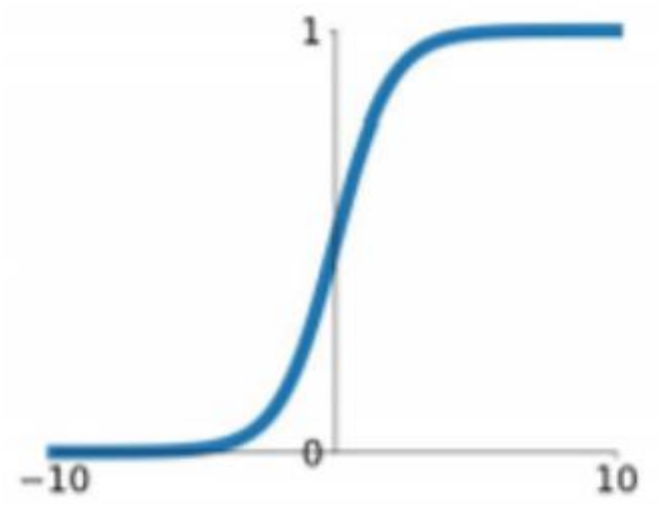
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



활성화 함수

- sigmoid



Sigmoid

Sigmoid 함수

- 결과 값이 (0,1)사이로 출력
- 극단의 값은 대부분 0 또는 1
- 신경망 초기에 사용

단점

- Gradient Vanishing 발생
- 함수 값 중심이 0이 아님 -> zig zag 현상
- Exp 사용으로 비용 많이 발생

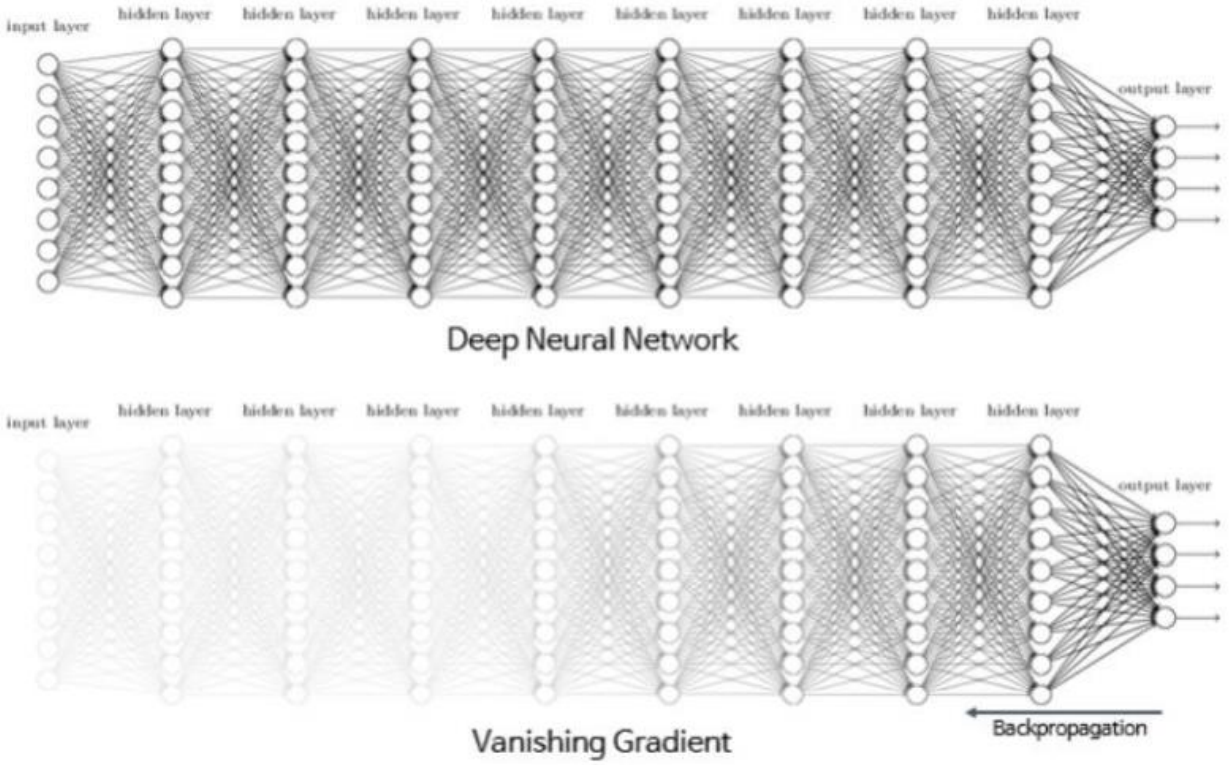
활성화 함수

- Gradient Vanishing? Gradient 항이 사라지는 문제

$$w^{+} = w - \eta * \frac{\partial E}{\partial w}$$

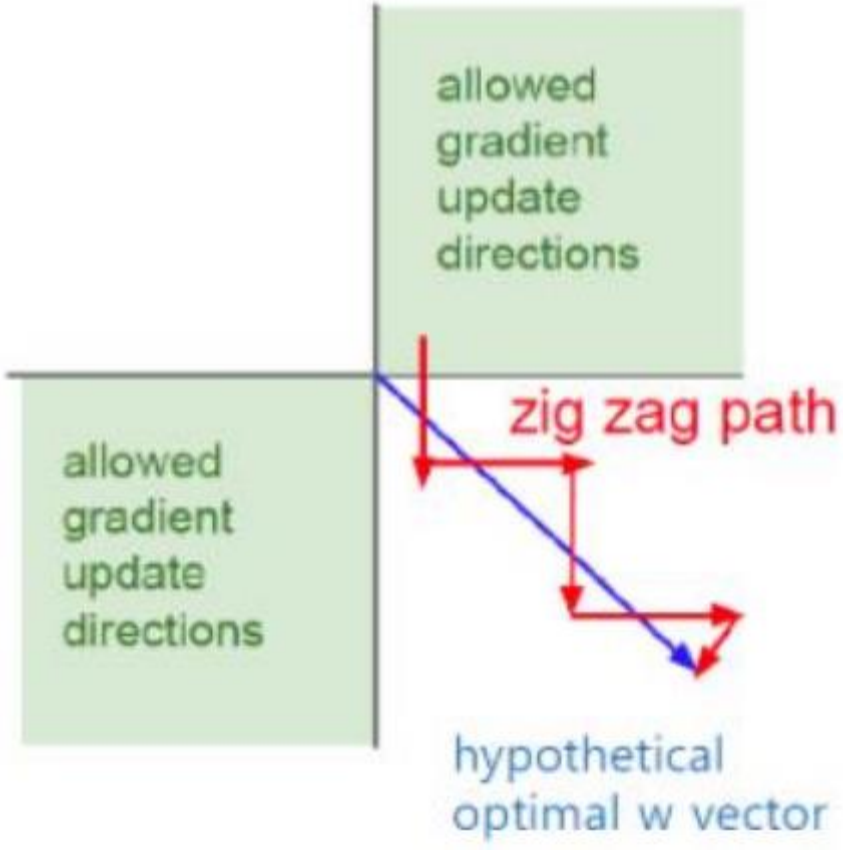
learning rate : 한번에 얼마나 학습할지

gradient : 어떤 방향으로 학습할지



활성화 함수

- 지그재그 현상?

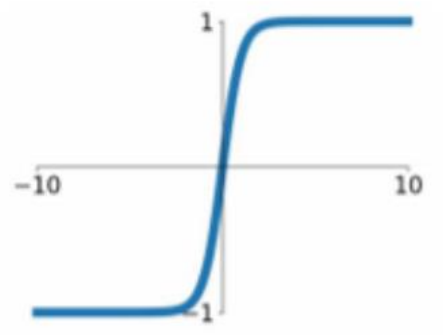


함수 값이 모두 양수

-> Loss가 가장 낮은 지점을 찾아 가는데 정확한 방향으로 가지 못하고 지그재그로 수렴

활성화 함수

- tanh(x)



tanh(x)

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

tahn 함수

- Sigmoid와 유사
- 결과 값이 (-1,1)사이로 출력
- Zigzag현상이 덜하다

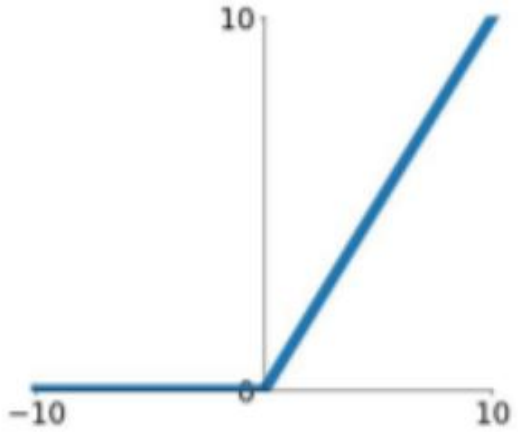
단점

- Gradient Vanishing 발생

활성화 함수

- ReLU

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$



ReLU

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

ReLU 함수

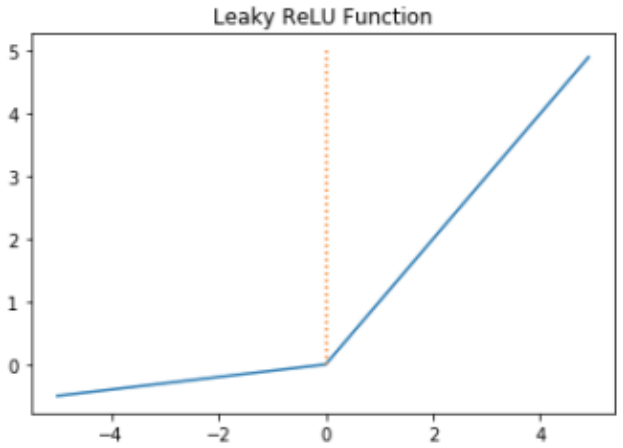
- 현재 가장 많이 사용하는 함수
- 학습이 빠름
- 연산 비용이 크지않고 구현이 간단

단점

- zig zag 현상 발생
- X<0 값들로 뉴런이 죽을 수도 있음

활성화 함수

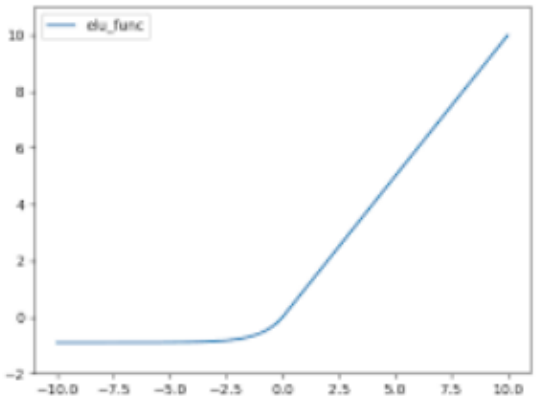
Leaky ReLU



$$\max(0.01, x)$$

ReLU 음수부분 개선

ELU



$$\begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$

ReLU 특성 공유
gradient vanishing 극복

Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

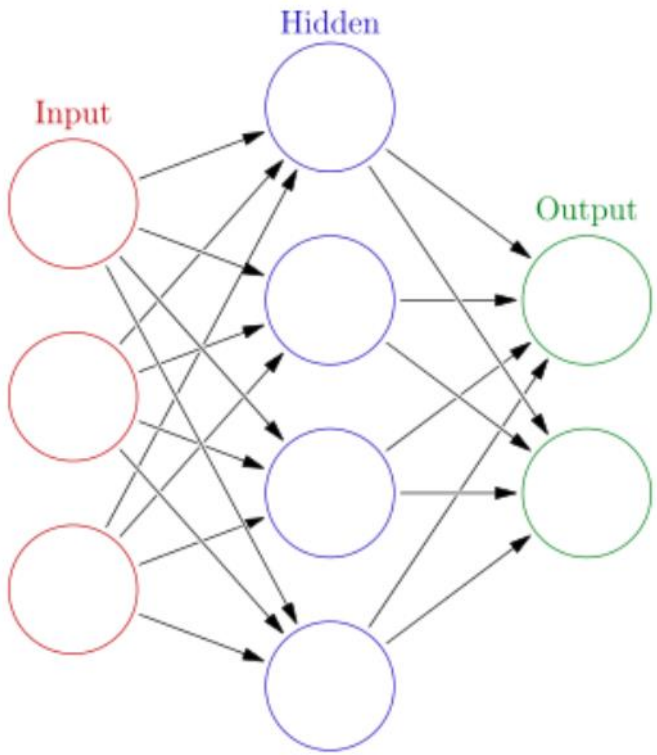
연결된 두 뉴런 값 중 큰 값 이용
연산량이 많이 필요

신경망 학습

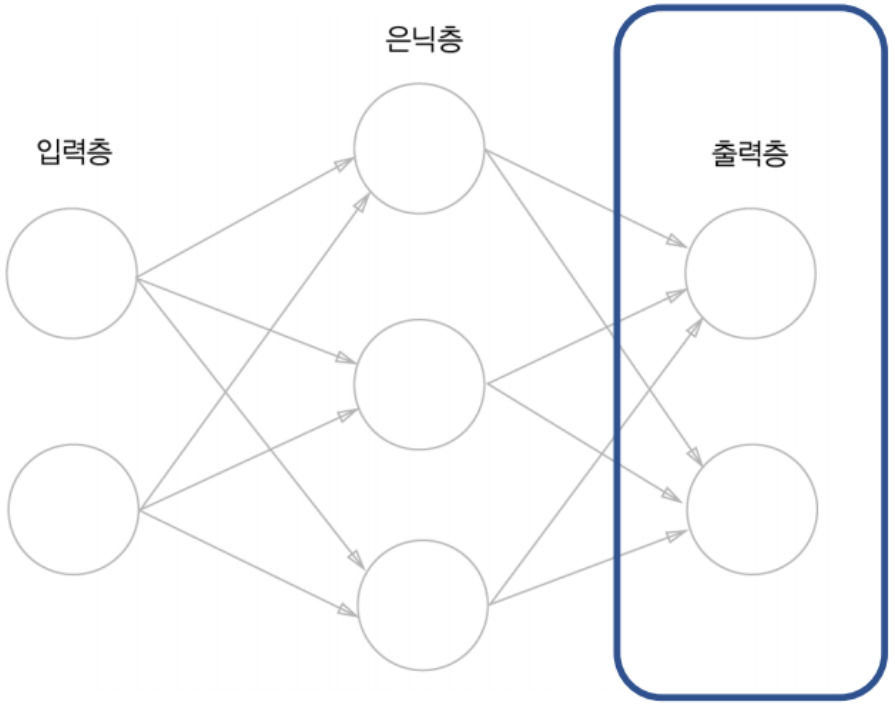
신경망 학습

구성

- 입력층 (Input)
- 은닉층 (Hidden Layer)
- 출력층 (Output)



신경망 학습



회귀 =항등 함수

입력 그대로 출력

분류

Binary > **sigmoid** 사용

$$h(x) = \frac{1}{1 + \exp(-x)}$$

Multi_class > **softmax** 사용

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

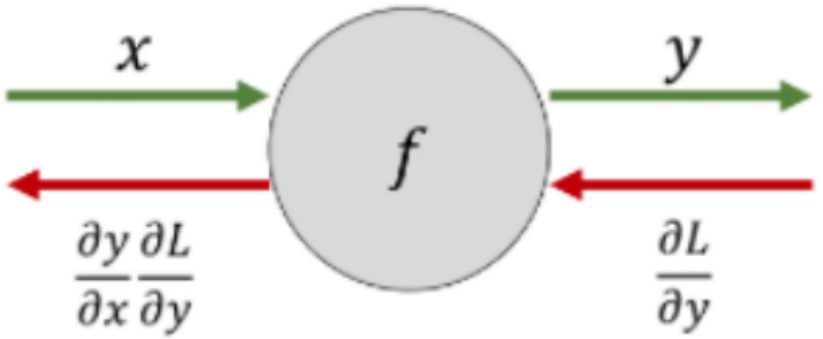
신경망 학습

Forward propagation -> computational node를 순서대로 방문하면서 loss값을 계산한다!

- Input에서 Output 방향으로 결과 값을 내보내는 과정

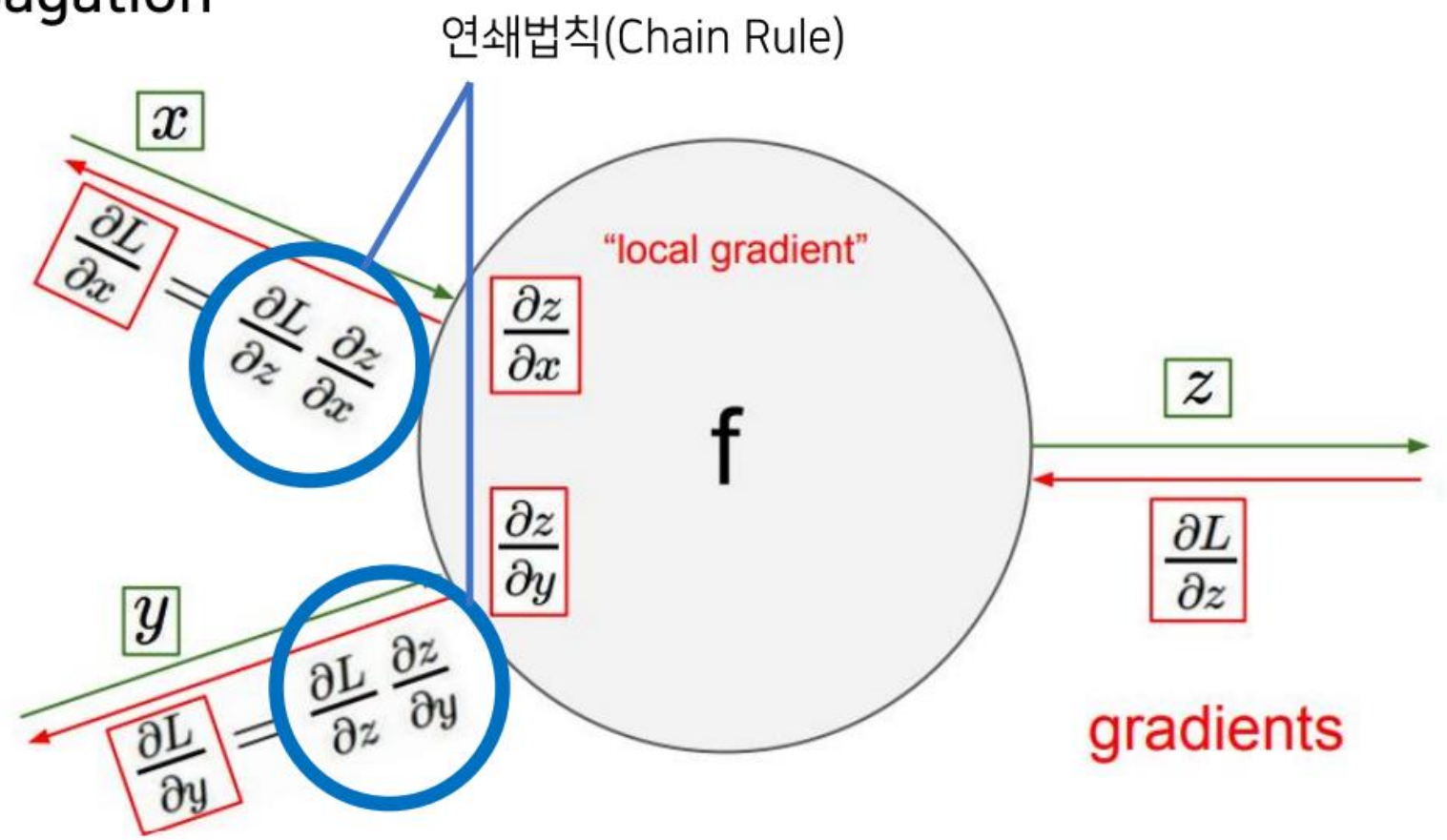
Backpropagation ->loss값을 최소화하는 가중치 w를 찾기 위해 backpropagation을 한다!

- 결과 값을 통해 역으로 Input 방향으로 오차를 다시 보내며 가중치 업데이트



신경망 학습

Backpropagation



신경망 학습

Backpropagation: a simple example

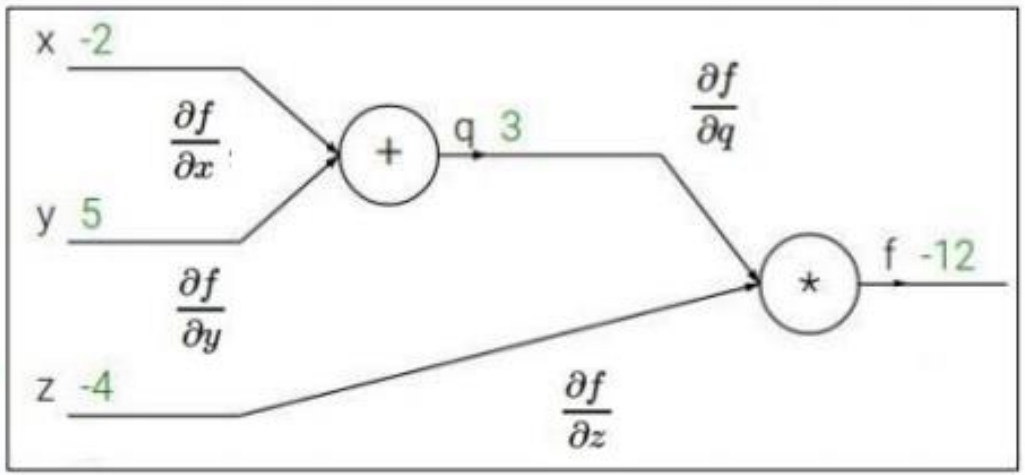
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

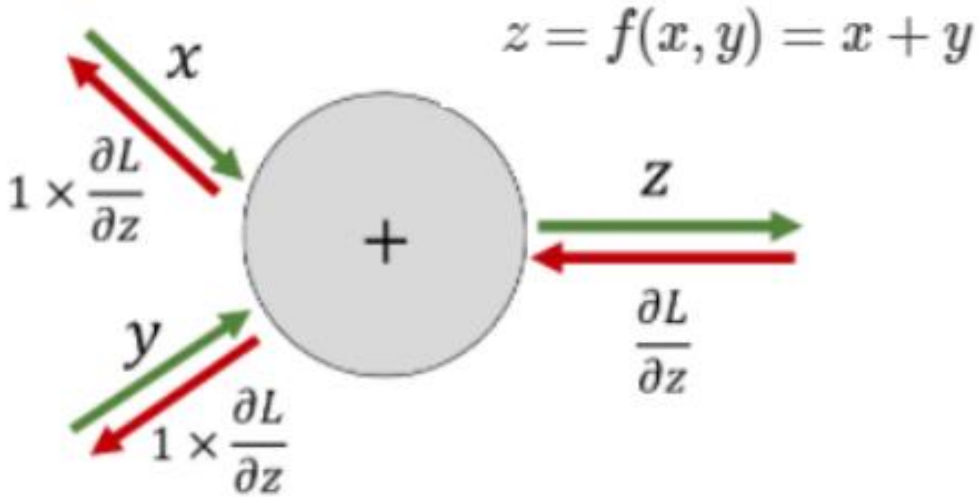
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



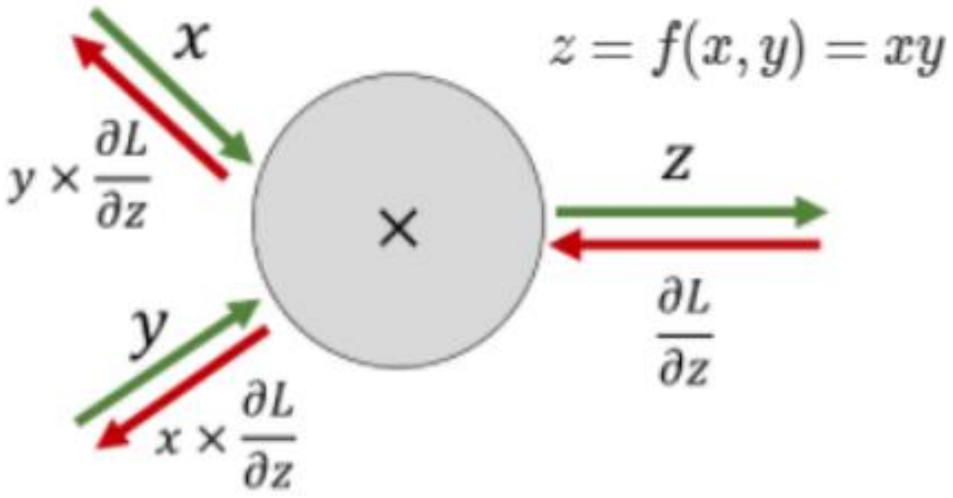
신경망 학습

덧셈 노드



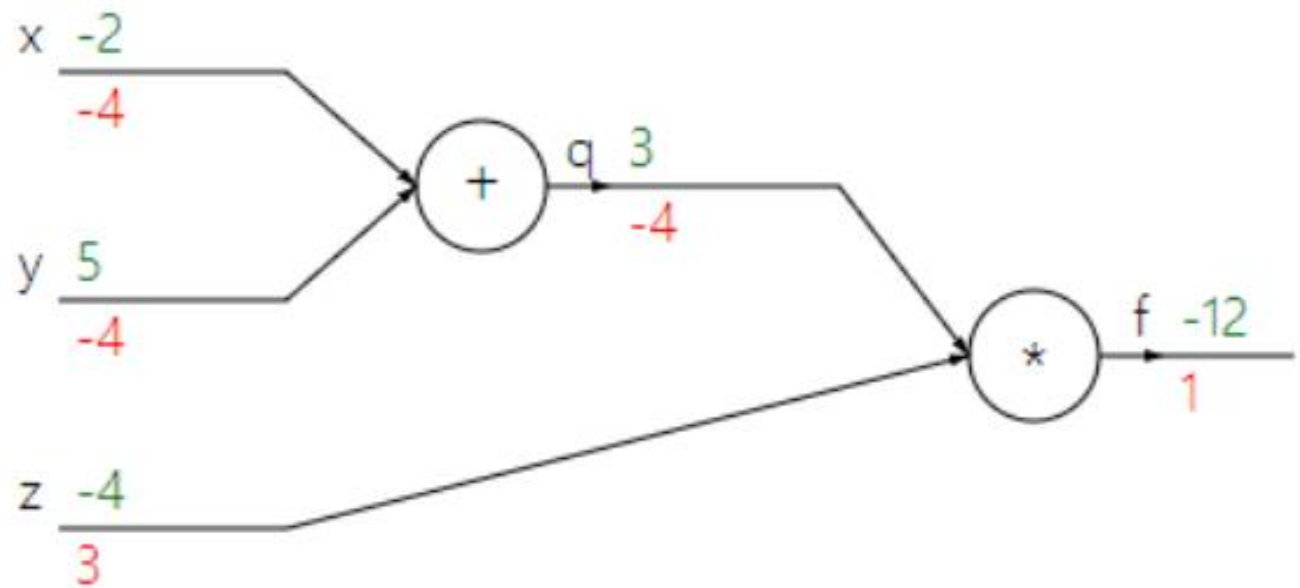
$$\frac{\partial z}{\partial x} = \frac{\partial (x+y)}{\partial x} = 1 \quad \frac{\partial z}{\partial y} = \frac{\partial (x+y)}{\partial y} = 1$$

곱셈 노드



$$\frac{\partial z}{\partial x} = \frac{\partial (xy)}{\partial x} = y \quad \frac{\partial z}{\partial y} = \frac{\partial (xy)}{\partial y} = x$$

신경망 학습

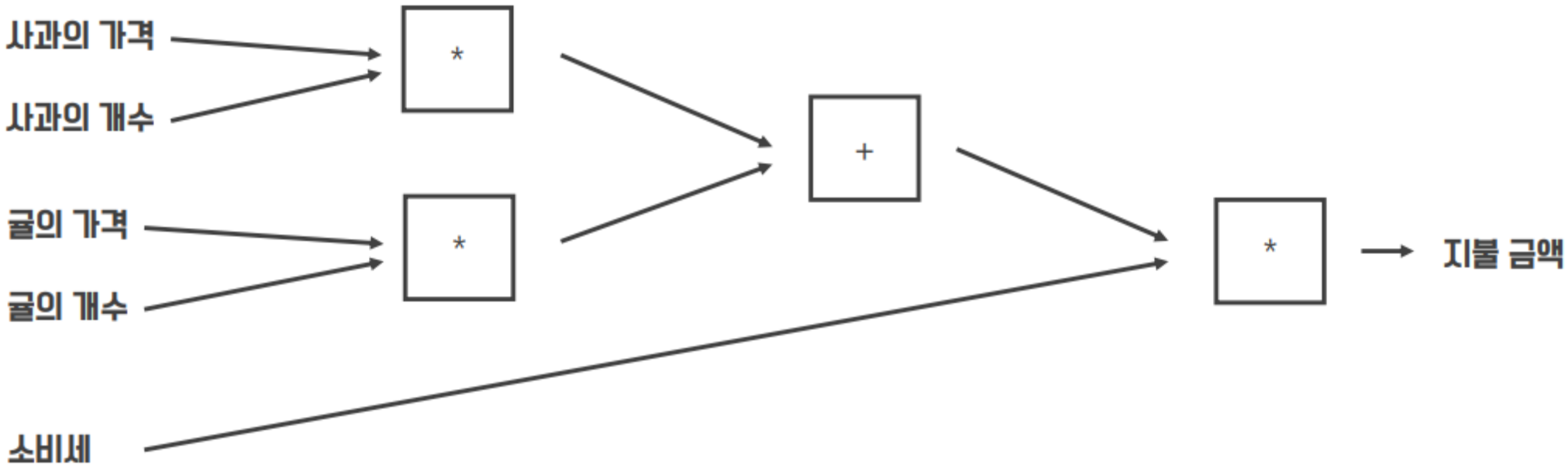


신경망 학습 - 예제

- 예제 : 한 개당 100원인 사과 2개와 한 개당 150원인 귤 3개를 샀습니다. 소비세가 10%라면 지불해야하는 금액을 구해주세요.

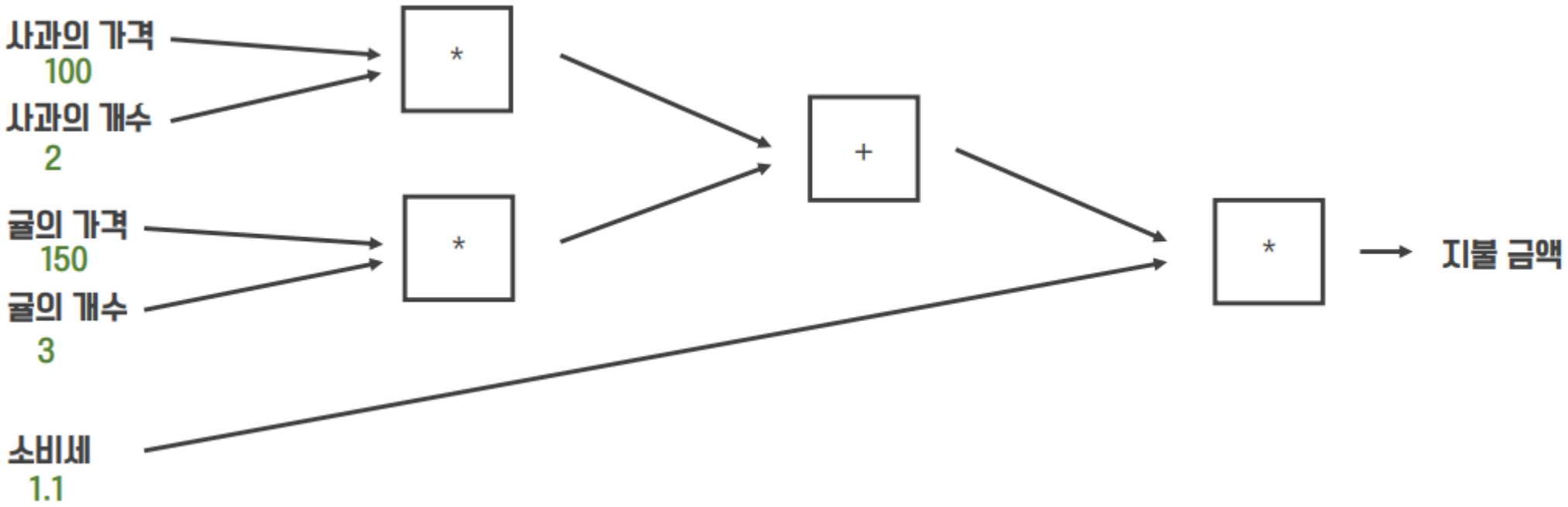
신경망 학습 - 예제

1. Computational graph를 그린다



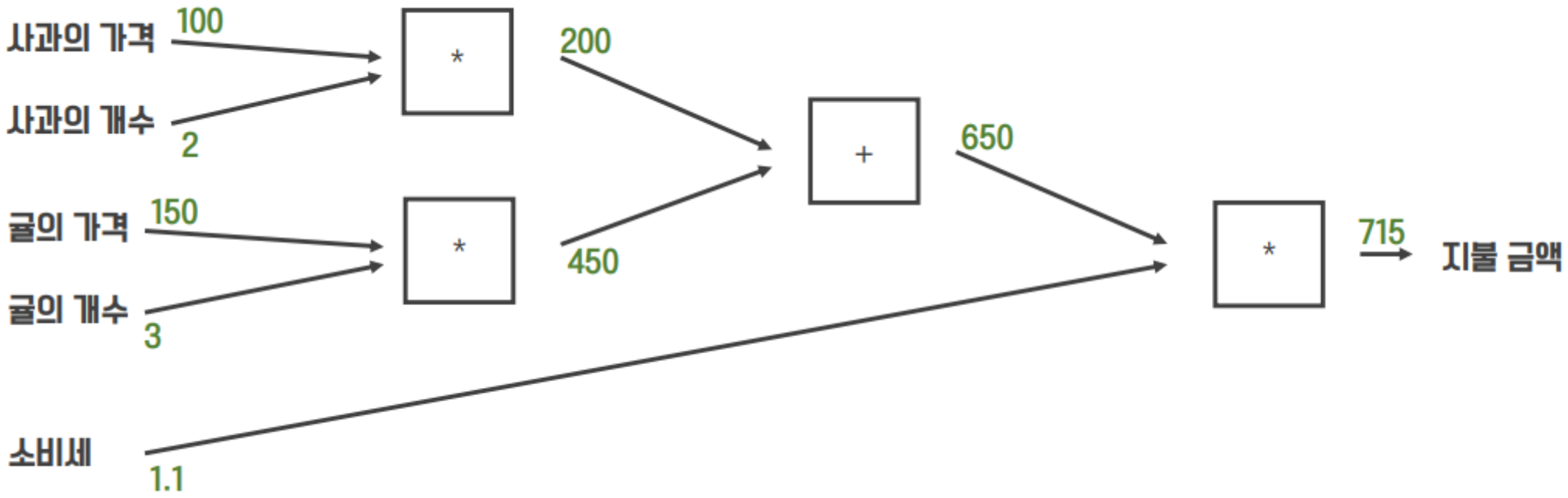
신경망 학습 - 예제

2. Forward Propagation



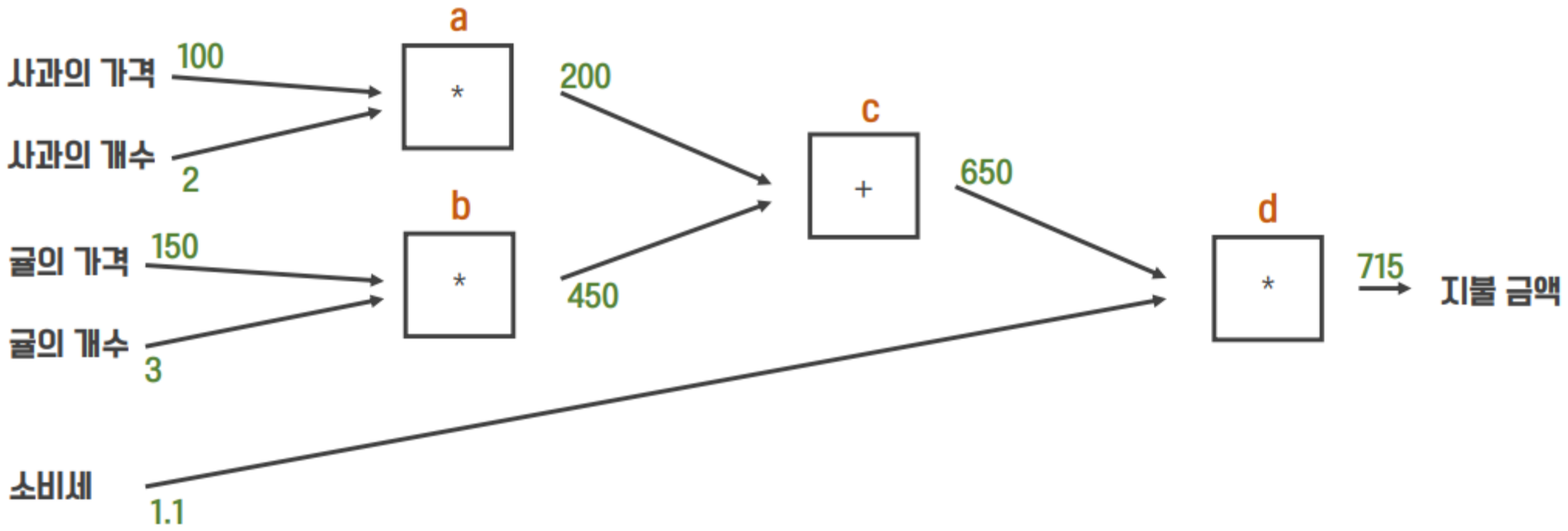
신경망 학습 - 예제

2. Forward Propagation



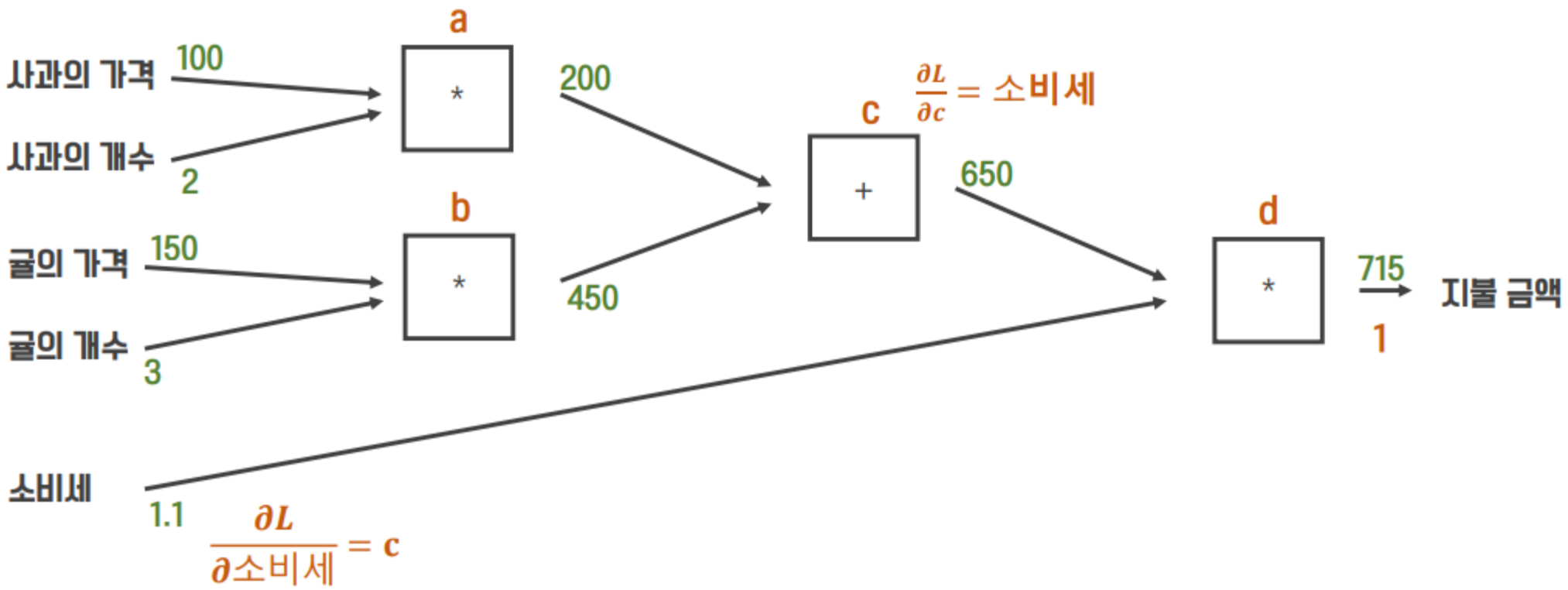
신경망 학습 - 예제

3. BackPropagation



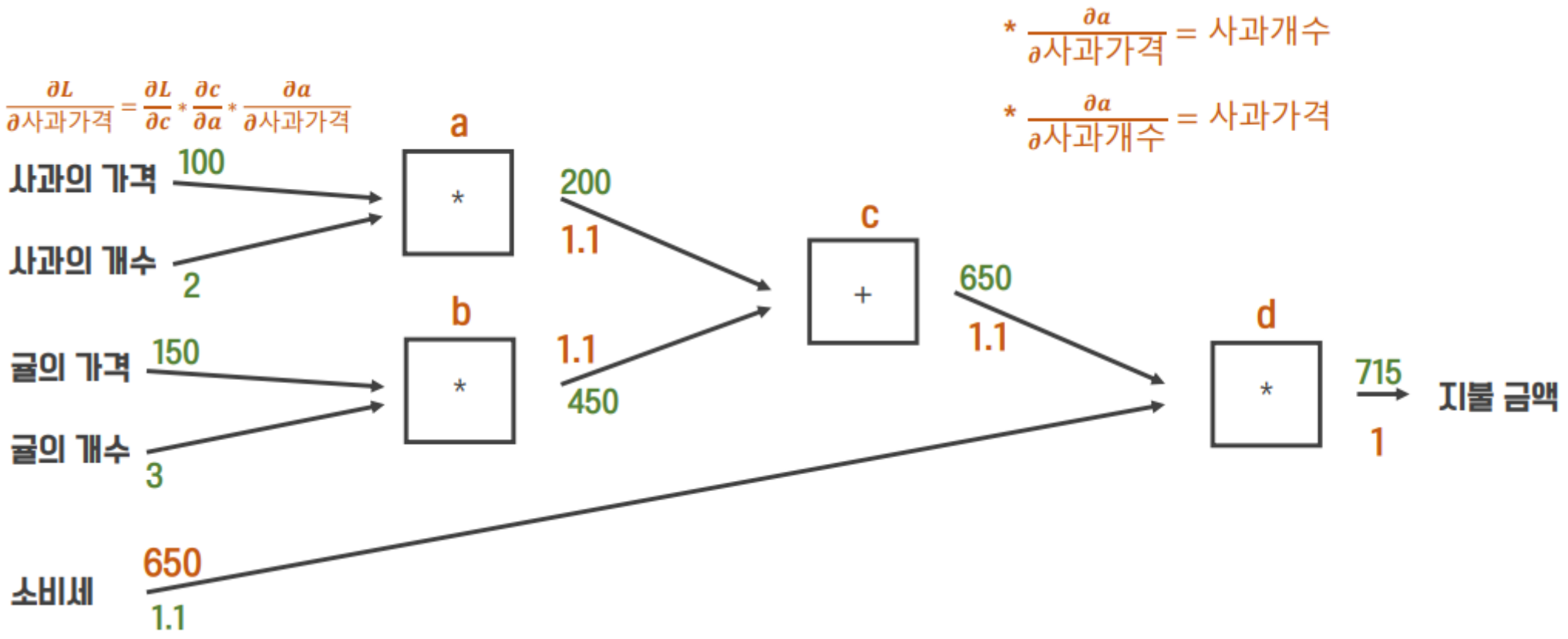
신경망 학습 - 예제

3. BackPropagation



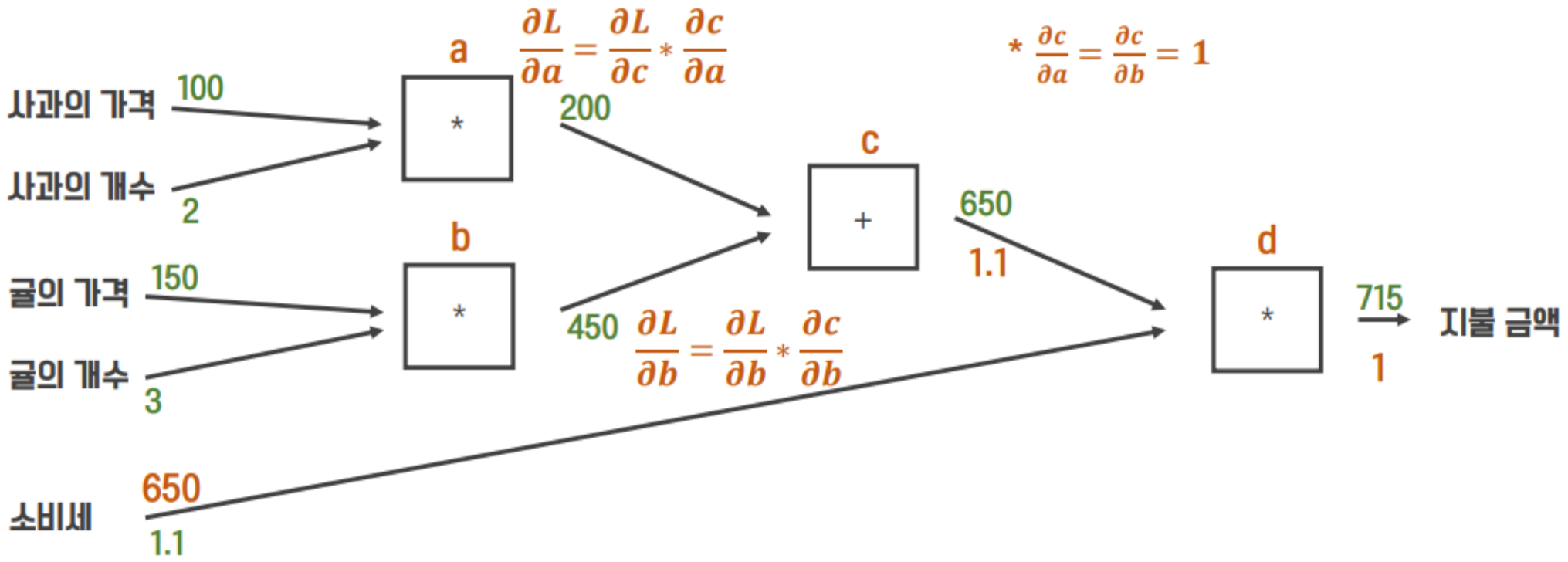
신경망 학습 - 예제

3. BackPropagation



신경망 학습 - 예제

3. BackPropagation



신경망 학습

BackPropagation

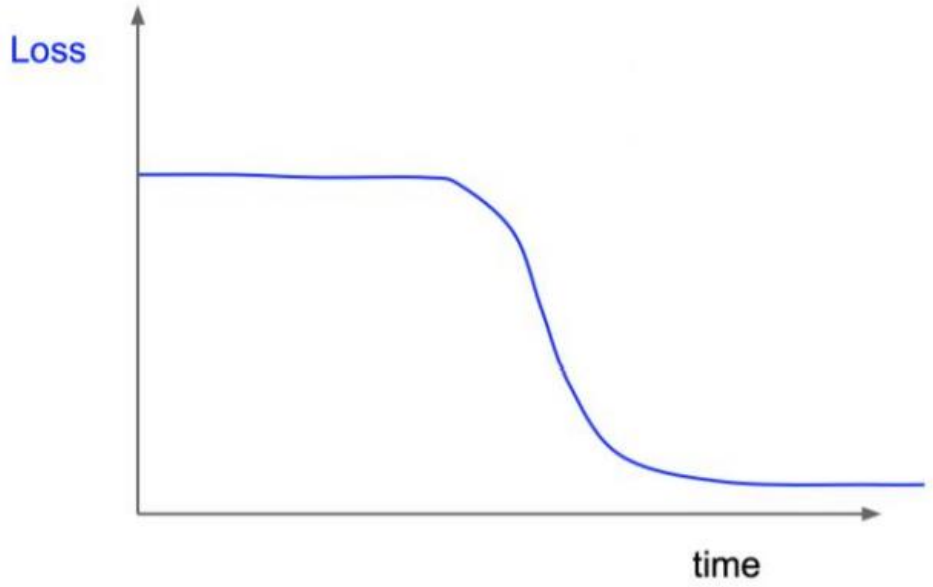
내가 뽑고자 하는 target값과 실제 모델이 계산한 output이 얼마나 차이가 나는지 구한 후 그 오차값을 다시 뒤로 전파해가면서 각 노드가 가지고 있는 변수들을 갱신하는 알고리즘인 것이다.

Weight initialization

weight initialization

초기 가중치 설정 (weight initialization)

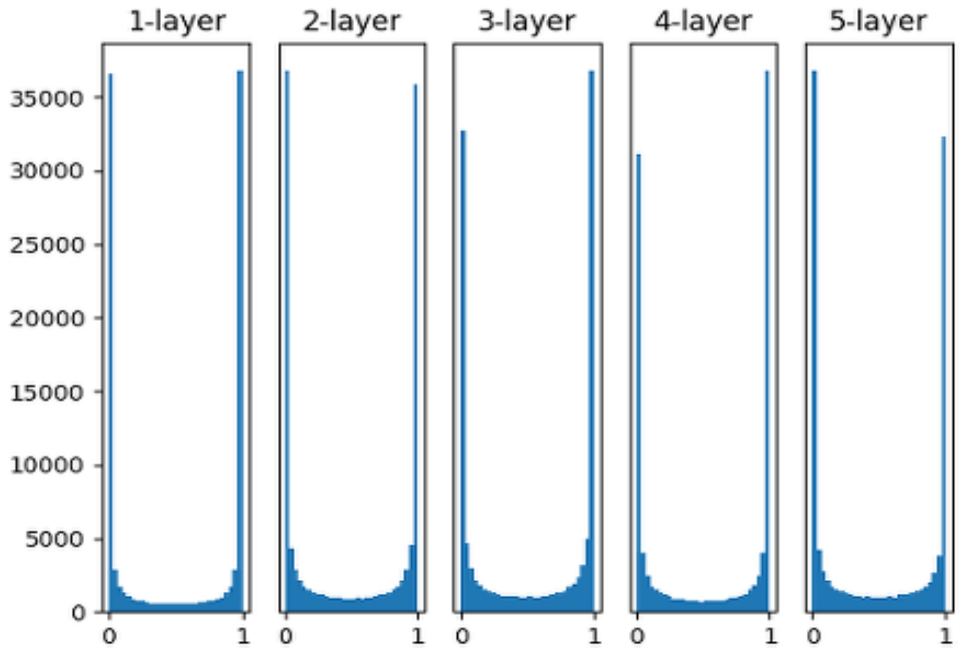
- 딥러닝 학습에 있어 초기 가중치 설정은 매우 중요한 역할
- 가중치를 잘못 설정할 경우 기울기 소실 문제나 표현력의 한계를 갖는 등 여러 문제를 야기하게 된다.
- 또한 딥러닝의 학습의 문제가 non-convex 이기 때문에 초기값을 잘못 설정할 경우 local minimum에 수렴할 가능성이 커지게 된다.



weight initialization

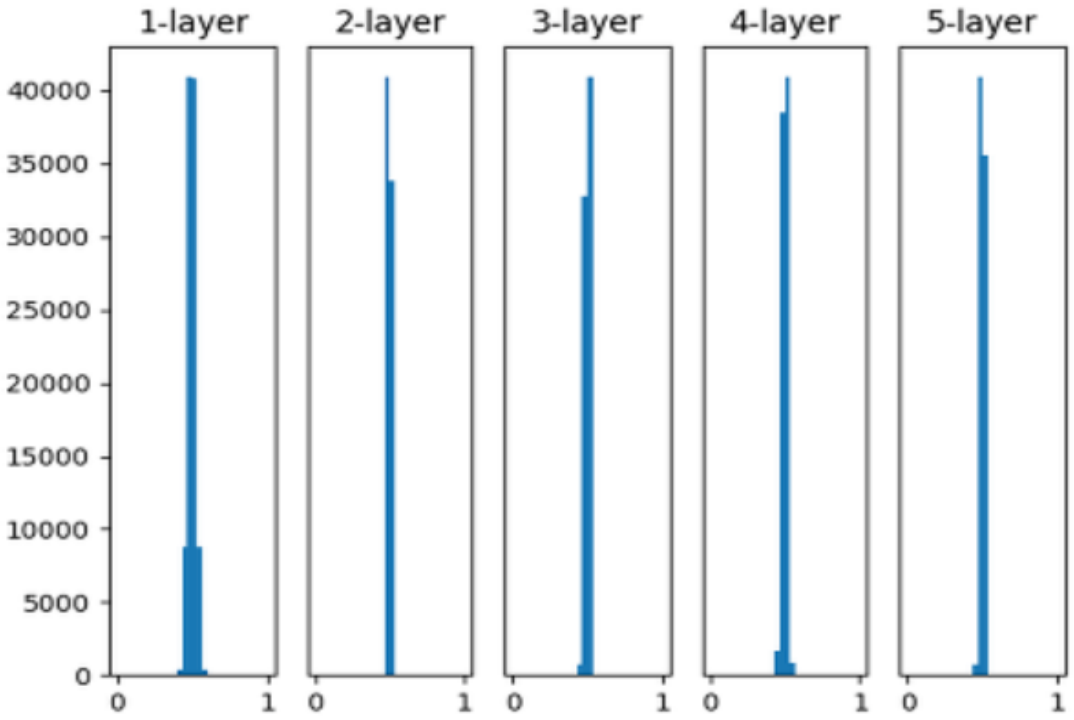
ex) sigmoid

- 표준 편차가 1일 경우, 값이 0과 1에 분포
- Gradient Vanishing 문제



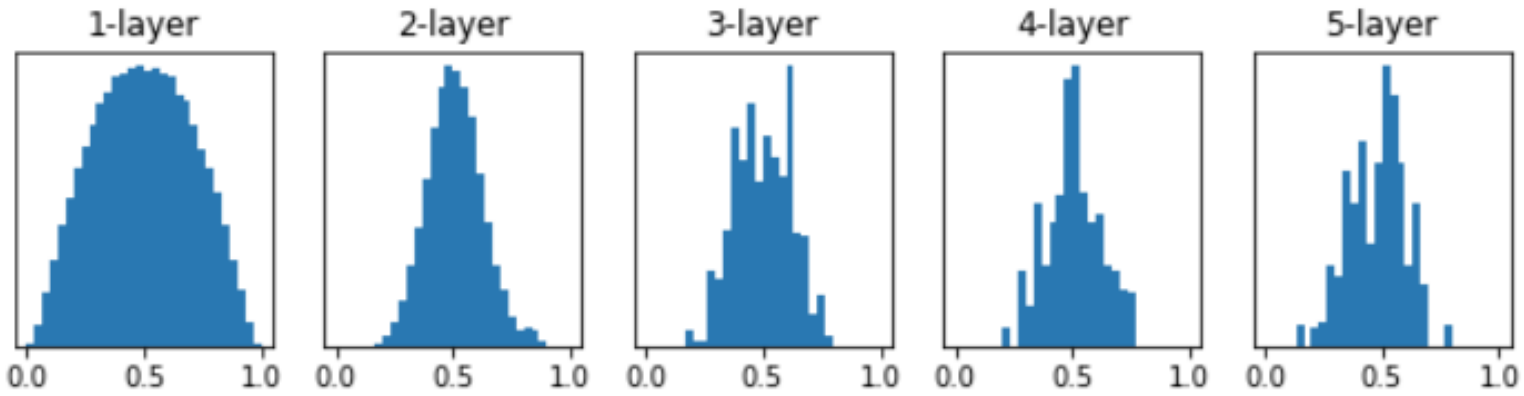
weight initialization

ex) sigmoid
- 표준편차를 줄였을 경우
- 값이 0.5에 치우쳐져 모두 같은 값 출력



weight initialization

- 1. Xavier Initialization

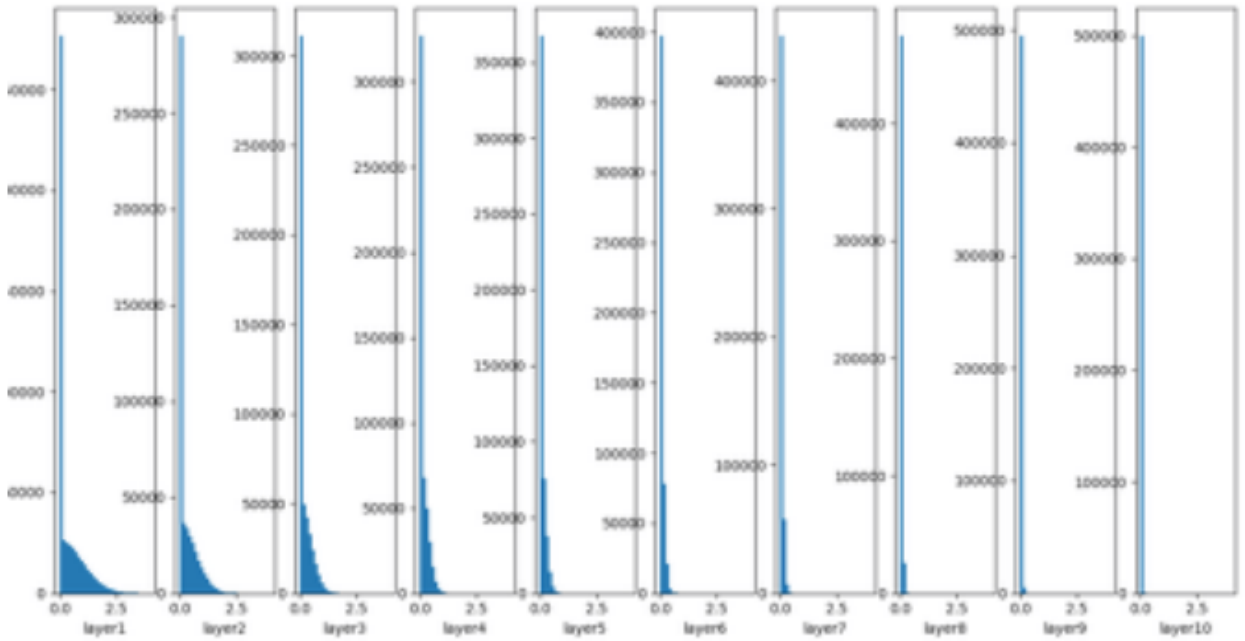


표준 정규 분포를 입력 개수의 표준편차로 나누기
np.random.randn(n_input, n_output)/sqrt(n_input)

Sigmoid 또는 tanh 함수 사용시 주로 사용

weight initialization

- 1. Xavier Initialization



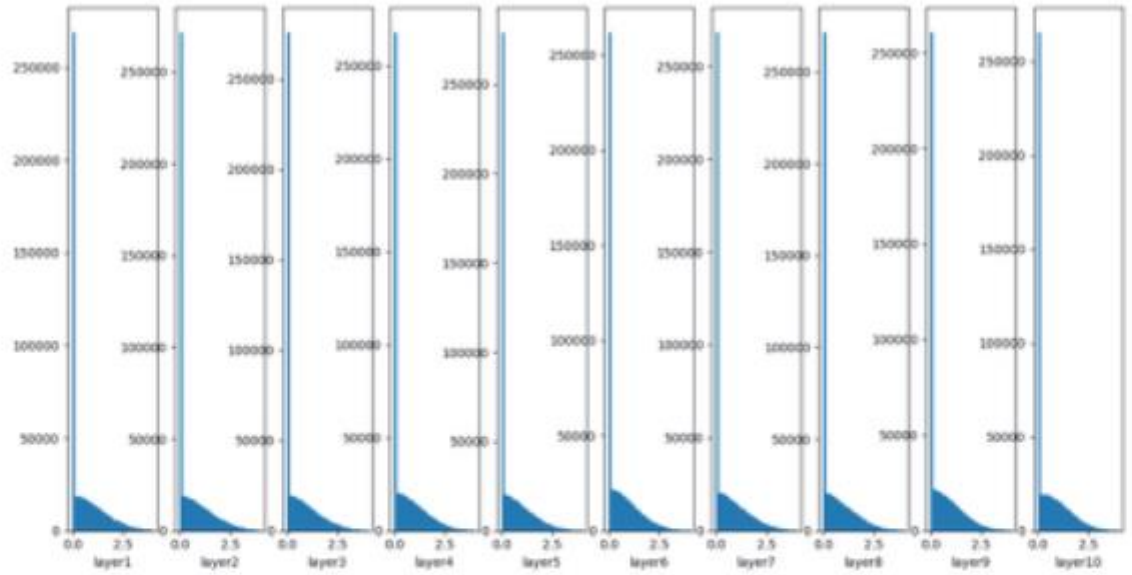
ReLU함수
+
Xavier Initialization

↓

점차 0으로 수렴

weight initialization

- 2.He Initialization

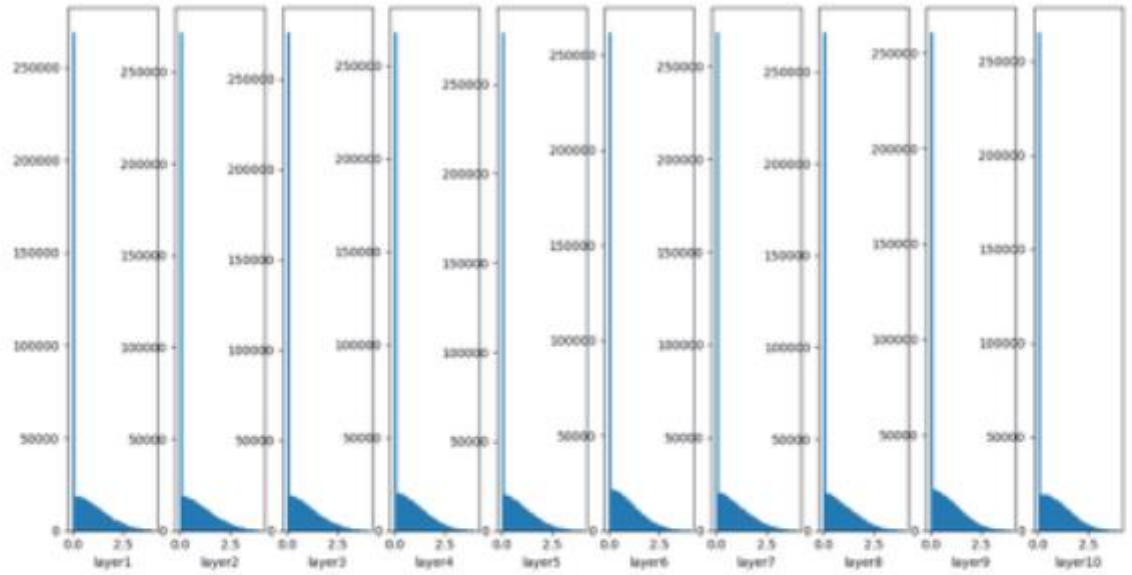


표준 정규 분포를 입력 개수의 절반의 제곱근으로 나누기
np.random.randn(n_input, n_output)/sqrt(n_input/2)

ReLu계열 함수에 사용

weight initialization

- 2.He Initialization

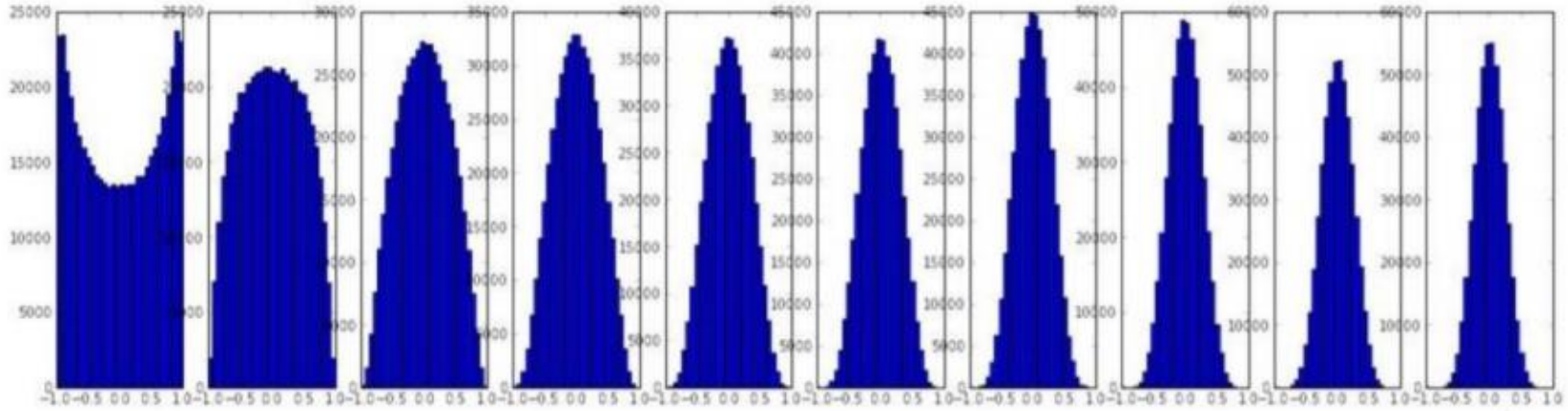


표준 정규 분포를 입력 개수의 절반의 제곱근으로 나누기
np.random.randn(n_input, n_output)/sqrt(n_input/2)

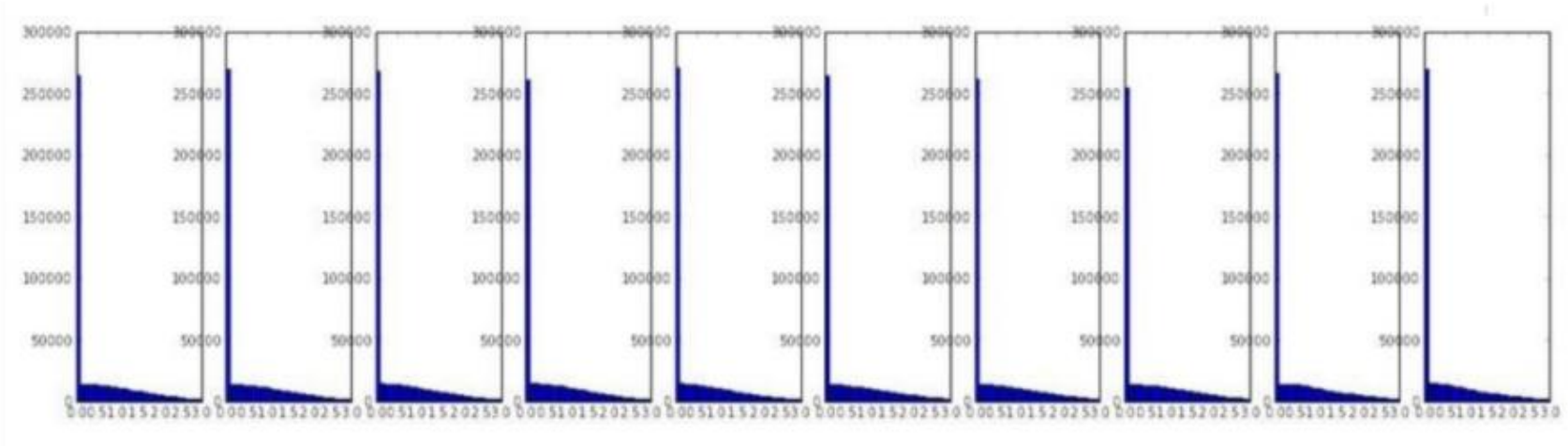
ReLu계열 함수에 사용

weight initialization

Xavier Initializer
나머지 sigmoid tanh



He Initializer
RELU family들



bias initialization

- 가중치 초기화 뿐만 아니라 편향(bias) 초기값 또한 초기값 설정 또한 중요하다.
- 보통의 경우에는 Bias는 0으로 초기화 하는 것이 일반적이다. ReLU의 경우 0.01과 같은 작은 값으로 b를 초기화 하는 것이 좋다는 보고도 있지만 모든 경우는 아니라 일반적으로는 0으로 초기화 하는 것이 효율적이다.

Loss Function

Loss Function

손실함수(Loss function)

- 신경망이 학습할 수 있도록 해주는 지표
- 모델의 출력 값과 사용자가 원하는 출력 값의 차이, 즉 오차를 의미

목적

- 손실함수 값을 가능한 작게 하는 매개변수 값을 찾는다.
- 정확도는 매개변수 변화에 둔감하고 변화가 있더라도 불연속적으로 변화하기 때문에 미분 불가능해 손실함수 사용

Loss Function

평균 제곱 오차
(Mean Square Error)

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

- 계산 간편해서 가장 많이 사용

교차 엔트로피 오차
(Cross Entropy Error)

$$E = -\sum_{k} t_k \log y_k$$

- 분류 문제에서 사용

Loss Function

그 외 손실함수

Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

Hinge losses for "maximum-margin" classification

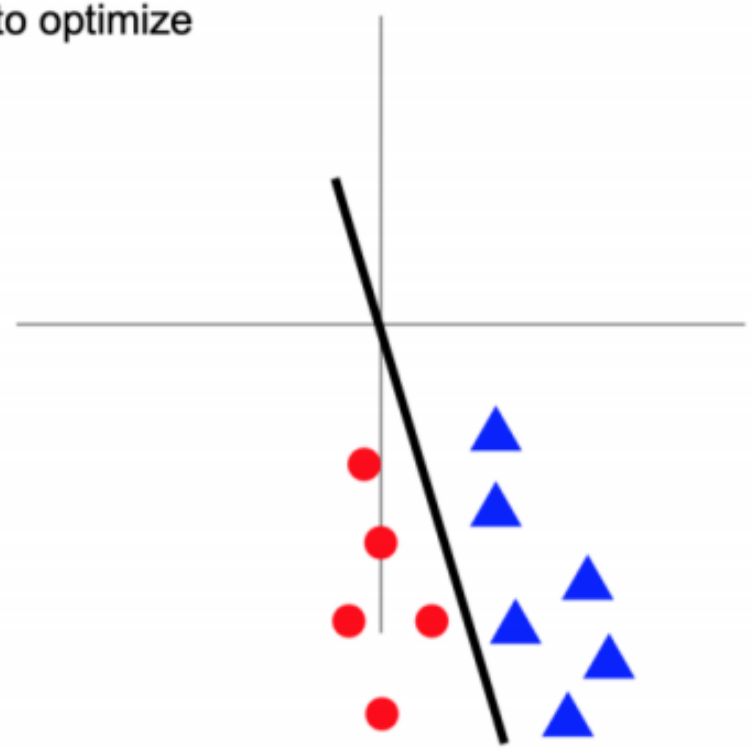
- Hinge class
- SquaredHinge class
- CategoricalHinge class
- hinge function
- squared_hinge function
- categorical_hinge function

Batch Normalization

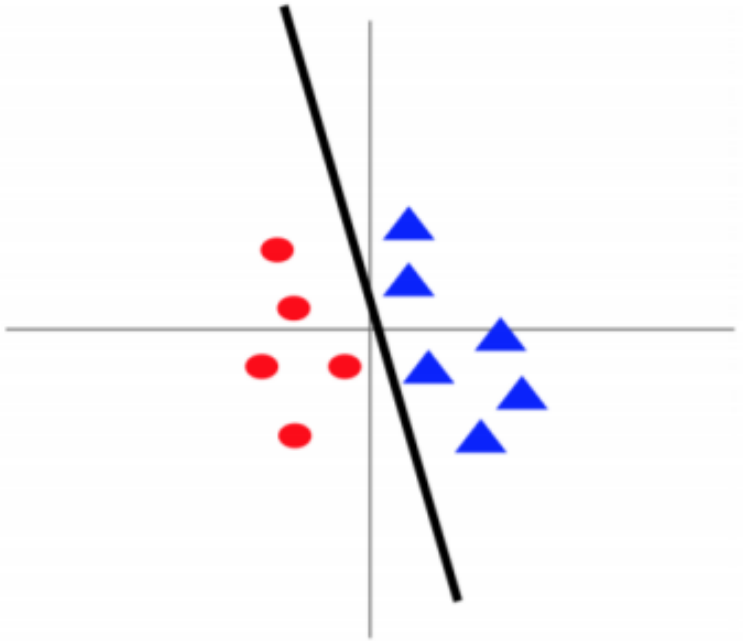
Batch Normalization

정규화 이유 : 영향력의 균일화

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Batch Normalization

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Optimizer

Optimizer

- 학습 최적화 (Optimizer)
- 손실함수(loss)의 값의 최저점 찾기
- 학습 속도 빠르고 안정적인 방향 찾기

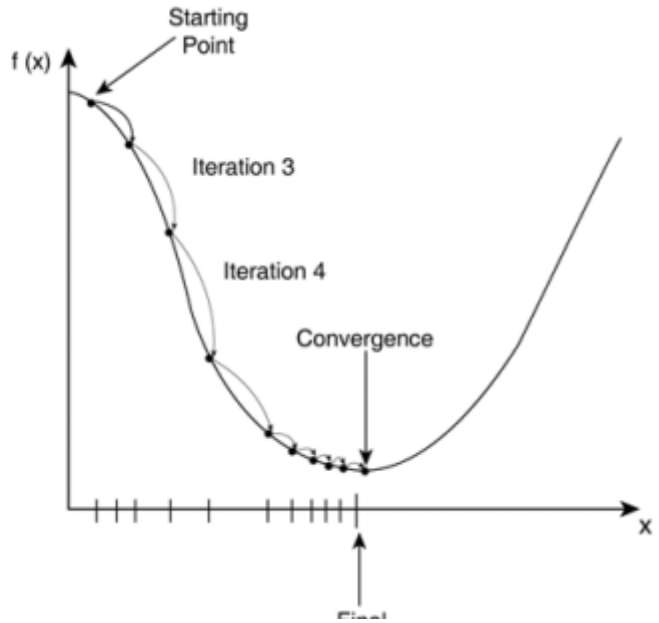
Optimizer

Gradient Descent

$$w^{+} = w - \eta * \frac{\partial E}{\partial w}$$

learning rate :
한번에 얼마나 학습할지

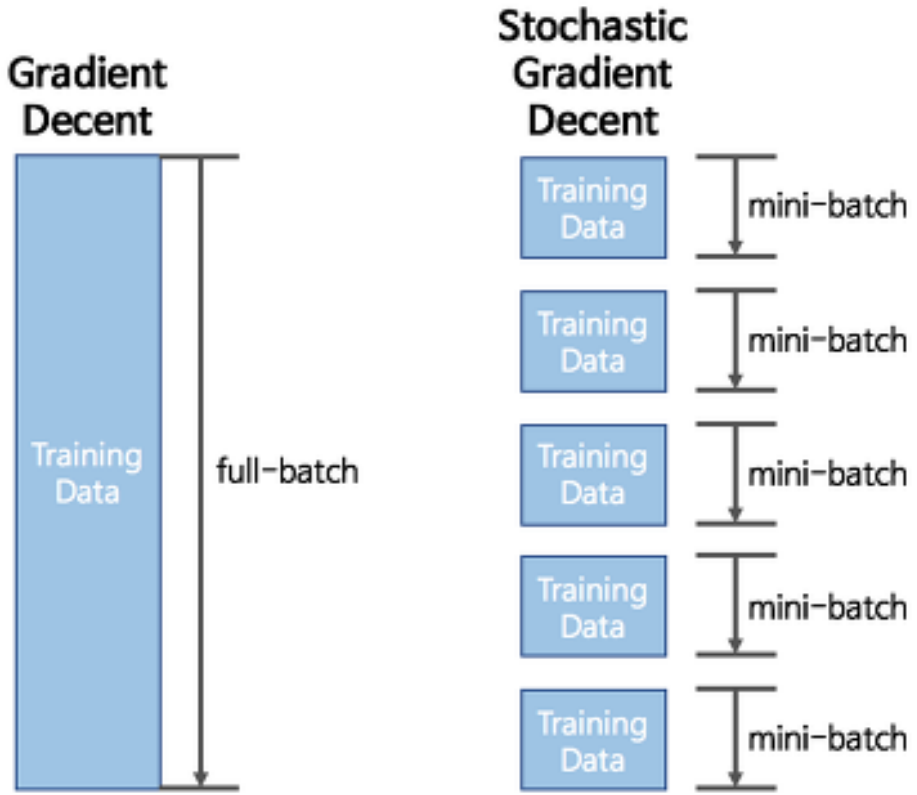
gradient :
어떤 방향으로 학습할지



- 오차함수의 낮은 지점을 찾아가는 최적화 방법
- 낮은 쪽의 방향을 찾기 위해 오차함수를 현재 위치에서 미분

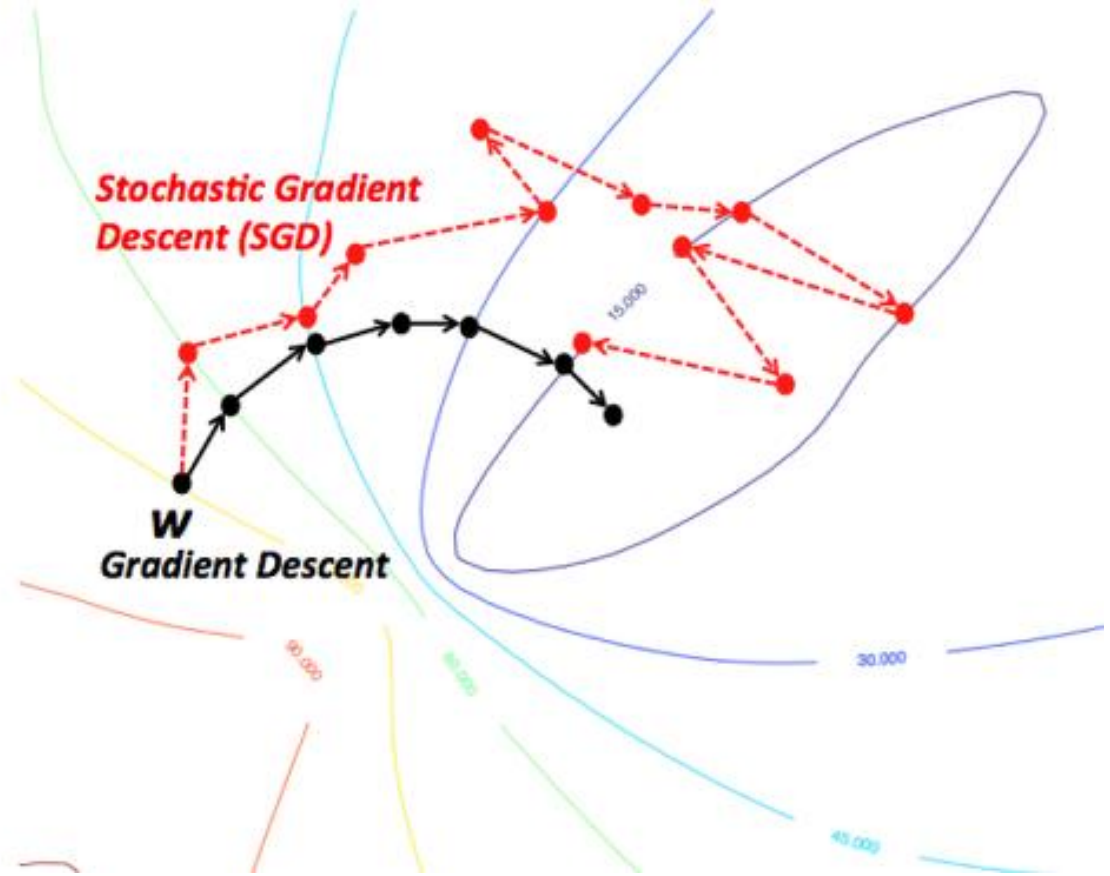
Optimizer

- SGD(Stochastic Gradient Descent)
- 조금만 훑어보고(Mini batch) 빠르게 가자!



Optimizer

- SGD(Stochastic Gradient Descent)
- 조금만 훑어보고(Mini batch) 빠르게 가자!

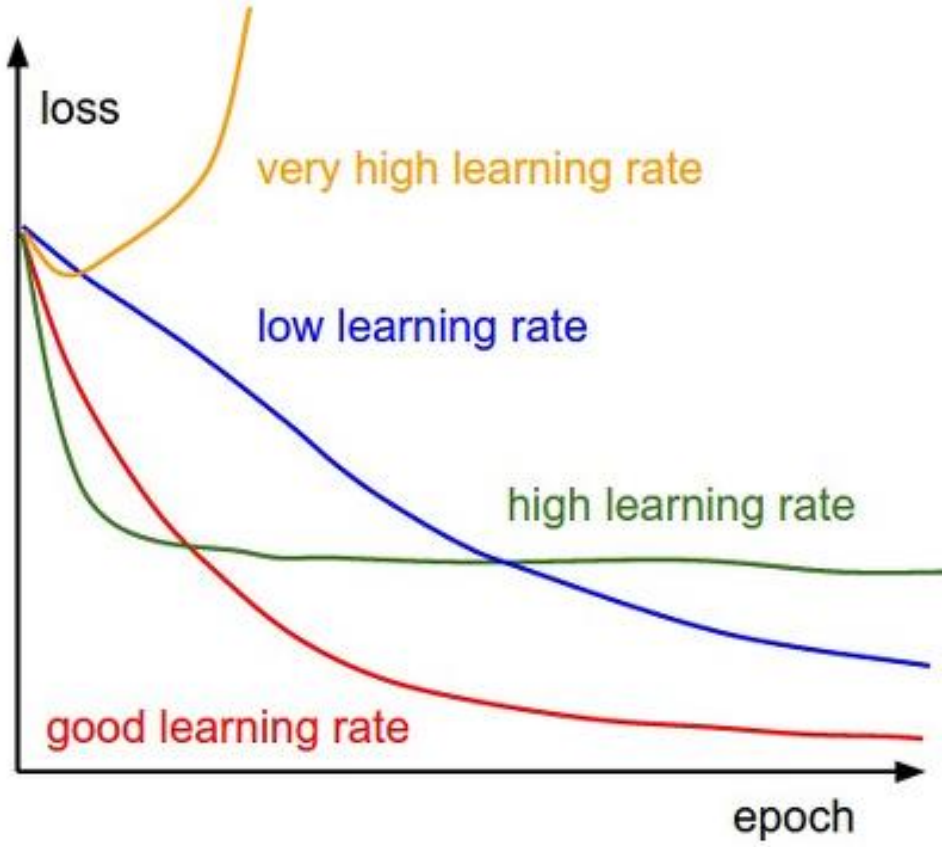


Optimizer

- GD
  - 모든 데이터를 계산한다 => 소요시간 1시간
  - 최적의 한스텝을 나아간다.
  - 6 스텝 * 1시간 = 6시간
  - 확실한데 너무 느리다.
- SGD
  - 일부 데이터만 계산한다 => 소요시간 5분
  - 빠르게 전진한다.
  - 10 스텝 * 5분 => 50분
  - 조금 헤메지만 그래도 빠르게 간다!

Optimizer

- 스텝의 사이즈(learning rate)
- 한걸음 나아가기 위한 보폭이 낮으면 학습하는데 오래 걸리고, 너무 크면 최적의 값을 찾지 못하는 문제



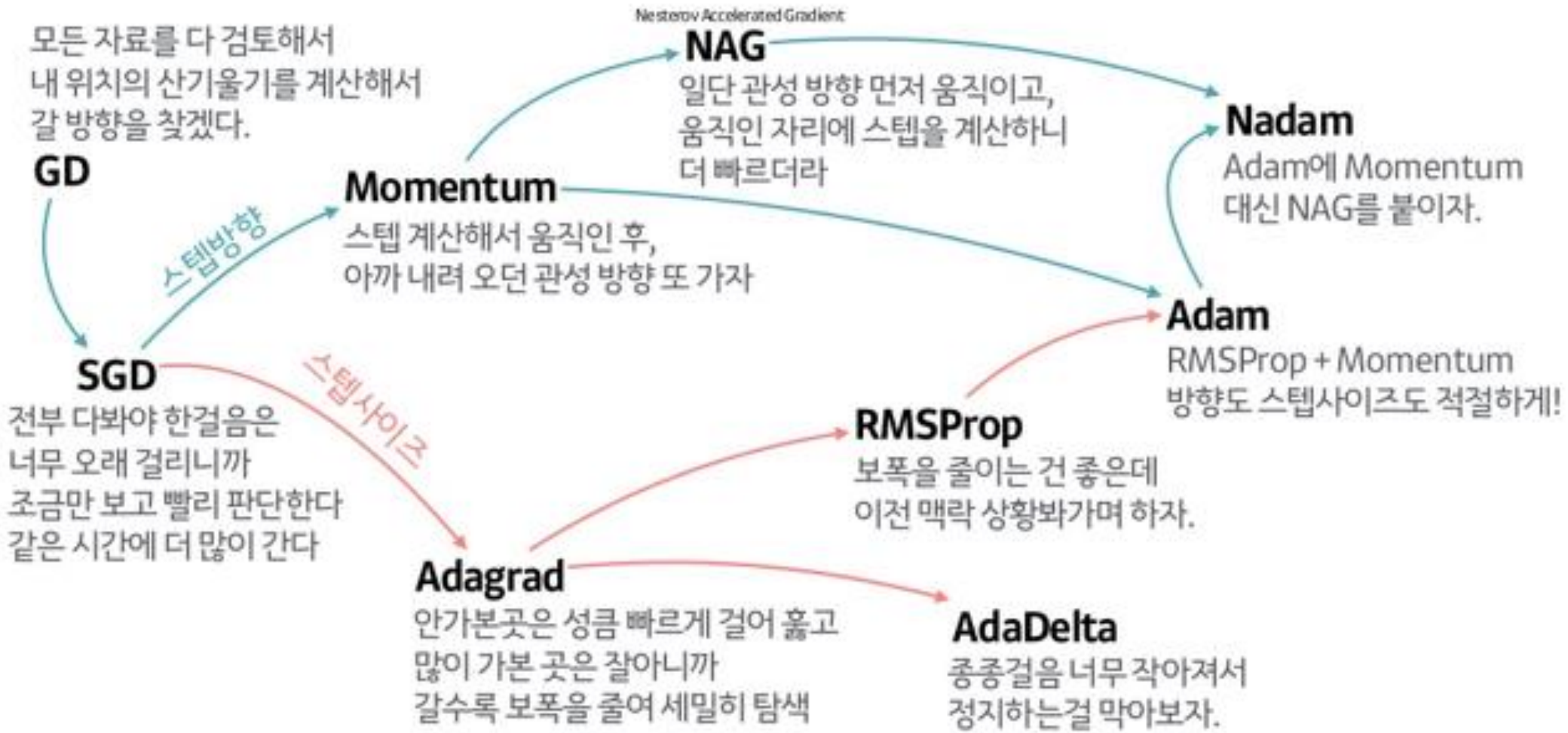
Optimizer

- Gradient Descent
  최적을 값을 찾아가는 것은 정확하지만 너무 느리다.
- Stochastic Gradient Descent를 통해서 빠르게 찾을 수 있도록 발전
  그러나 최적의 값을 찾아가는 방향이 뒤죽 박죽이고, 한 스텝 나아가기 위한 사이즈를 정하기 어렵다.

방향과 스텝 사이즈를 고려하는 새로운 Optimizer들이 많이 나왔다.

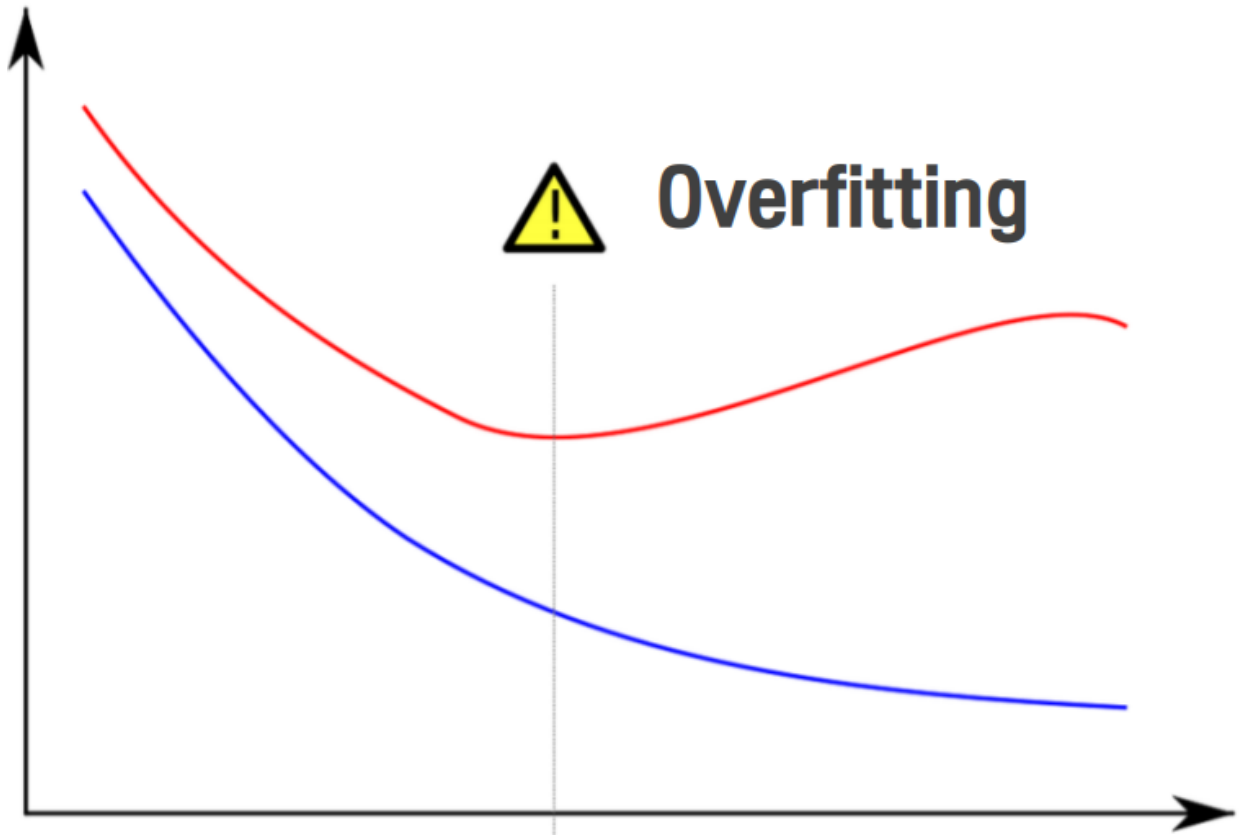
- 방향성
  - Momentum
  - NAG
- 스텝 사이즈
  - Adagrad
  - RMSPop
  - AdaDelta
- 방향성 + 스텝사이즈
  - Adam
  - Nadam

Optimizer


모든 자료를 다 검토해서
내 위치의 산기울기를 계산해서
갈 방향을 찾겠다.
GD
스텝방향
SGD
전부 다봐야 한걸음은
너무 오래 걸리니까
조금만 보고 빨리 판단한다
같은 시간에 더 많이 간다
Momentum
스텝 계산해서 움직인 후,
아까 내려 오던 관성 방향 또 가자
Nesterov Accelerated Gradient
NAG
일단 관성 방향 먼저 움직이고,
움직인 자리에 스텝을 계산하니
더 빠르더라
Nadam
Adam에 Momentum
대신 NAG를 붙이자.
Adam
RMSProp + Momentum
방향도 스텝사이즈도 적절하게!
스텝사이즈
RMSProp
보폭을 줄이는 건 좋은데
이전 맥락 상황봐가며 하자.
Adagrad
안가본곳은 성큼 빠르게 걸어 홅고
많이 가본 곳은 잘아니까
갈수록 보폭을 줄여 세밀히 탐색
AdaDelta
종종걸음 너무 작아져서
정지하는걸 막아보자.

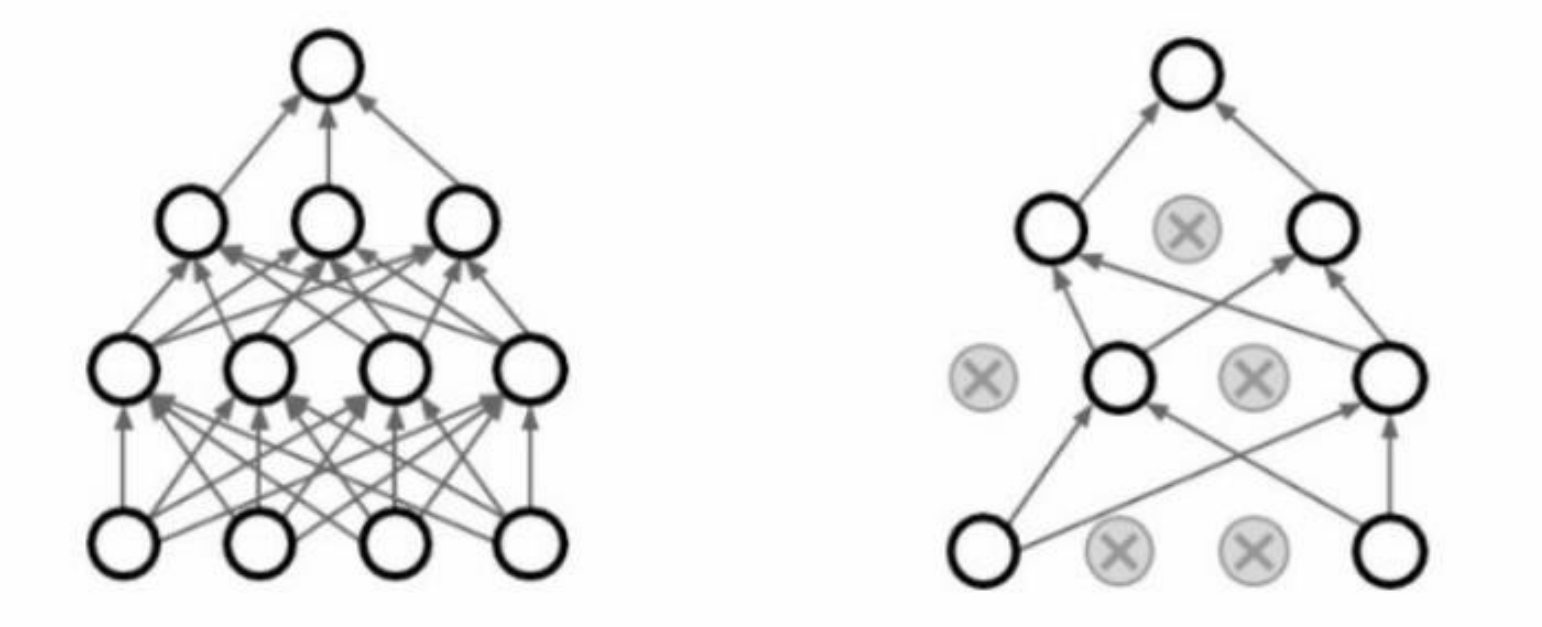
Regularizer - Dropout

오버피팅



Regularizer - Dropout

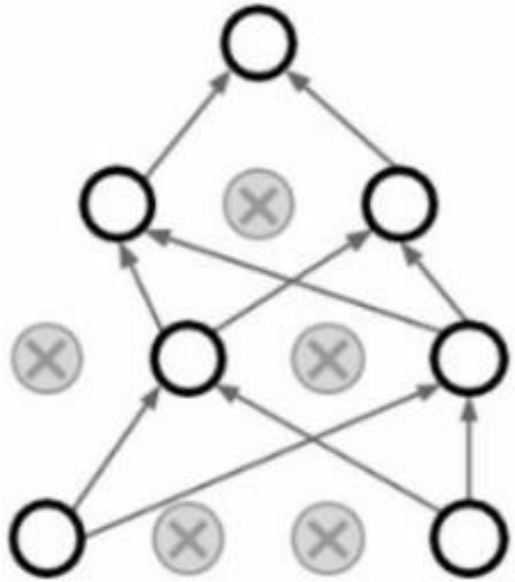
Drop Out : 일정 노드를 랜덤하게 버려 버리기 Forward pass시에 일부 뉴런의 출력을 0으로 만들어 버림 (할 때마다 0이 되는 뉴런은 바껴!)



Regularizer - Dropout

Dropout이 왜 좋은데?

- 뉴런 하나당 하나의 특성만을 학습하는데 랜덤으로 노드를 죽이게 되면 살아남은 노드들이하나의 노드에만 의존 하지 않고, 다른 특징까지 학습하려 하면서 overfitting을 막을 수 있음
- - forward pass 시마다 랜덤하게 dropout 시키기 때문에 앙상블의 효과를 볼 수 있음

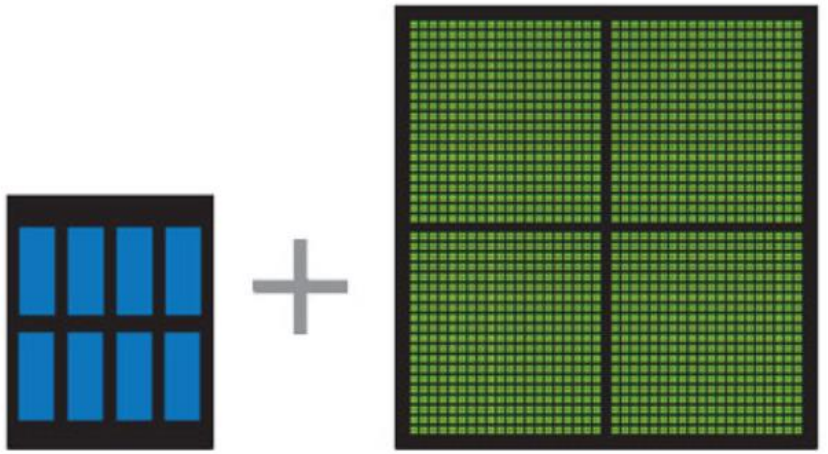


CPU vs GPU

CPU vs GPU

- CPU(Central Processing Unit) : 컴퓨터의 모든 프로그램의 동작을 제어하는 장치
- GPU(Graphical Processing Unit) : 약자로 빠른 시간에 많은 연산을 수행해야 하는 그래픽 관련 처리를 담당하기 위해 만들어진 장치

직렬 처리에 최적화된 몇 개의 코어로 구성

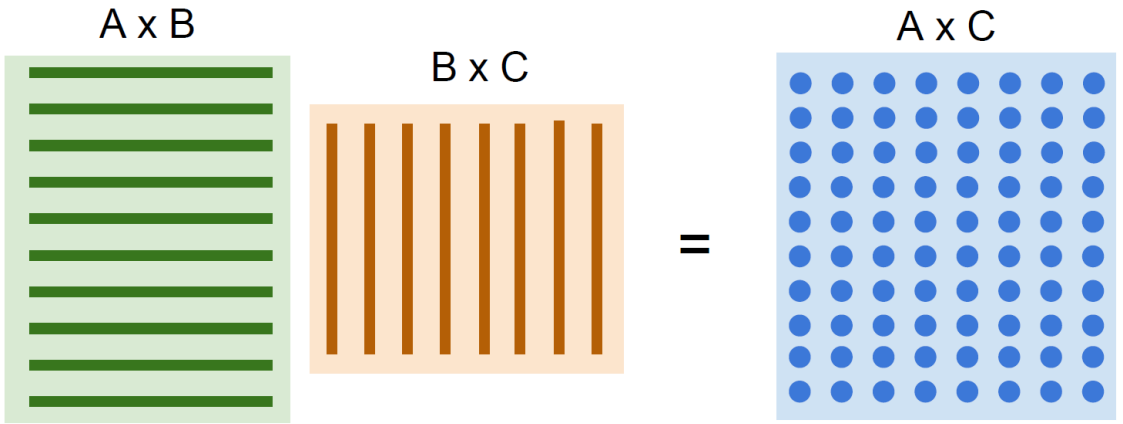


병렬 처리용으로 설계된 수천개의 보다 소형이고 효율적인 코어로 구성

CPU vs GPU

GPU가 머신러닝이나 딥러닝에 자주 쓰이는 이유

- 굉장히 많은 코어 수 : 코어 수가 많기 때문에 GPU는 병렬 처리에 특화
- 딥러닝에 자주 쓰이는 행렬의 곱셈 연산 같은 경우 병렬 처리를 활용할 경우 연산 시간을 획기적으로 단축가능
- 따라서 클락 사이클이 느리더라도 코어수가 CPU보다 훨씬 많은 GPU가 선호됨



CPU vs GPU

	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

딥러닝의 응용 분야

딥러닝 응용 분야

컴퓨터비전

자연어처리

음성

강화학습

기타 특정 분야에 딥러닝을 적용 : 네트워크, 로봇, 금융 등