정규 교육 세미나

ToBig's 9기 신현경

Neural Network 2

Contents

Unit 01 | 우리가 지금 뭐 배우는지..

Deep layer & More Neuron

보통 층이 깊을수록,
뉴런 수가 더 많아질수록
성능이 올라 간데!

Unit 01 | 우리가 지금 모 배우는지..

Deep layer & More Neuron

문제 발생!

Overfitting

Vanishing Gradient

Too Slow

Unit 01 | 우리가 지금 모 배우는지..

Deep layer & More Neuron

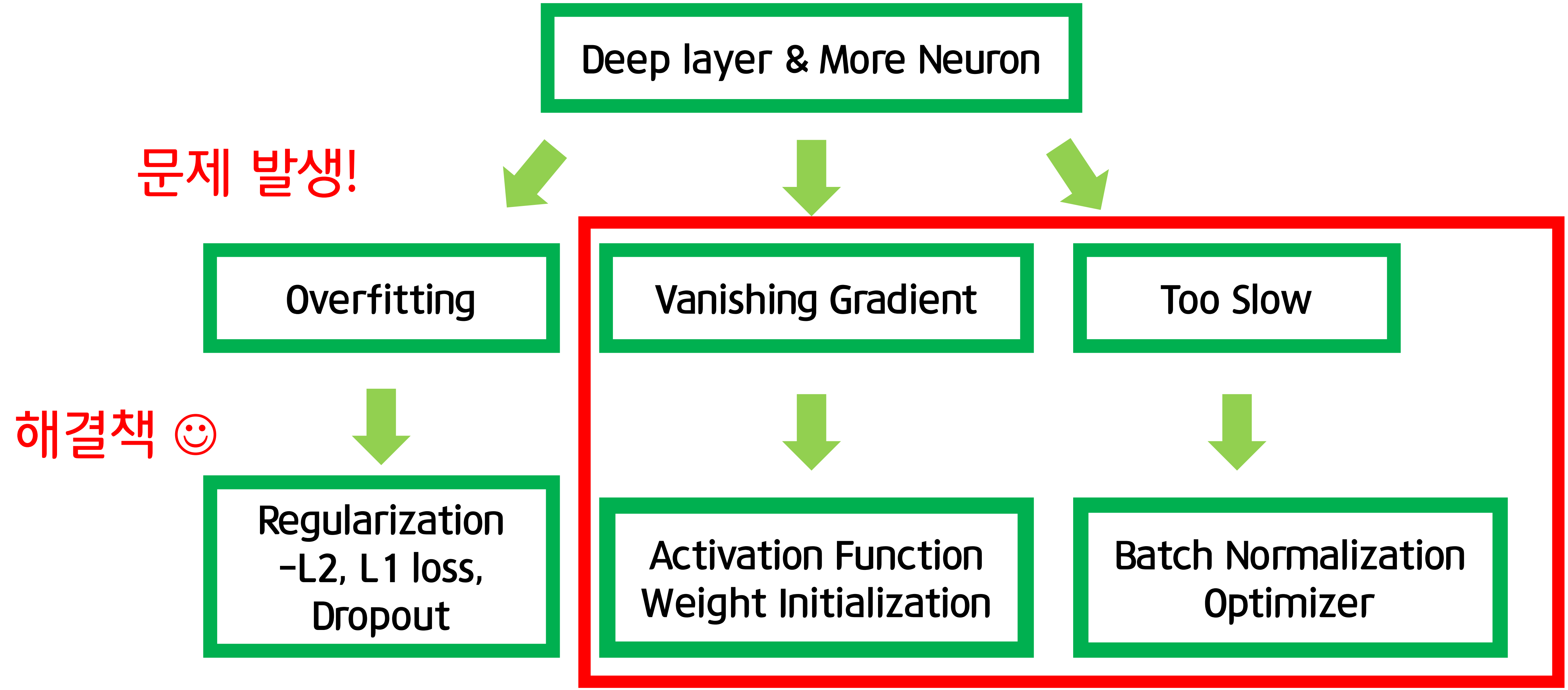
문제 발생!

- Overfitting
- Vanishing Gradient
- Too Slow

해결책 ☺

- Regularization
  -L2, L1 loss,
  Dropout
- Activation Function
  Weight Initialization
- Batch Normalization
  Optimizer

Unit 01 | 우리가 지금 모 배우는지..



Why use Activation Function?

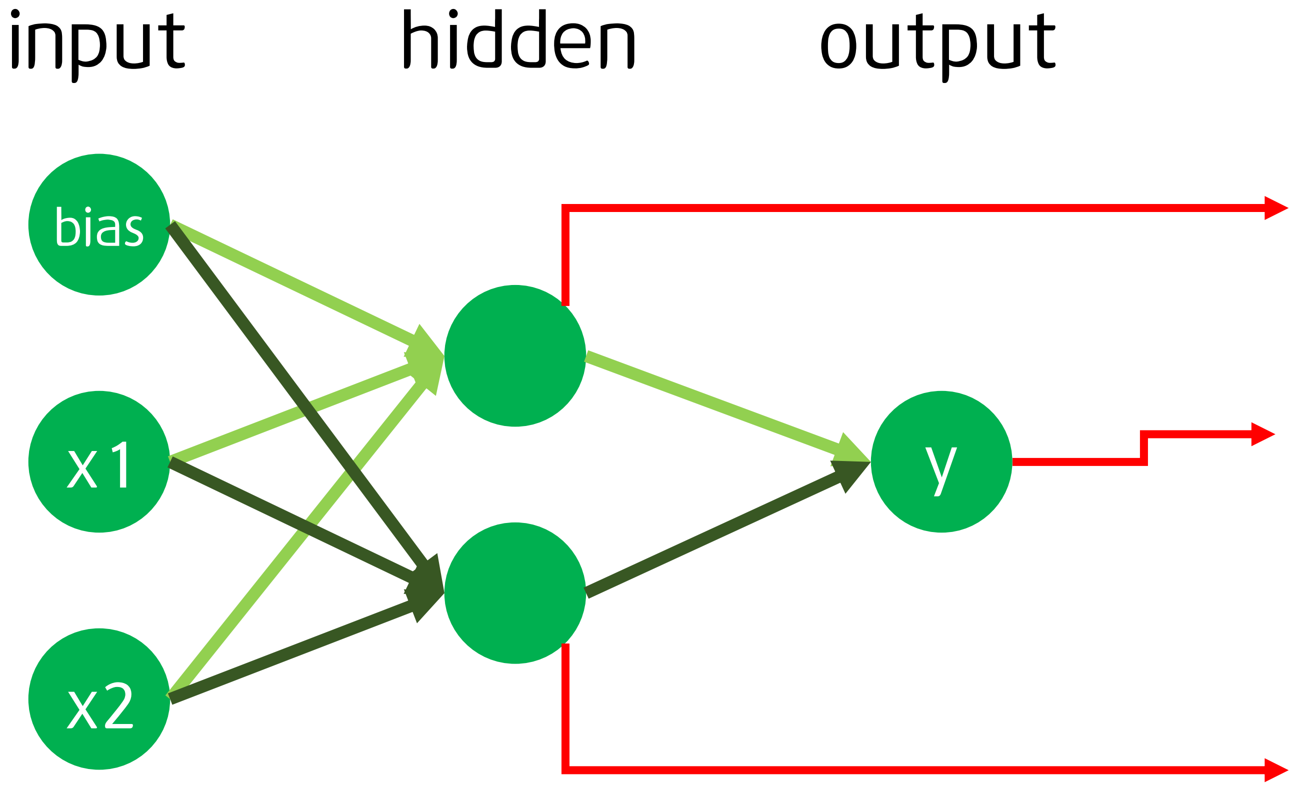
Why use Activation Function?

-> Activation Function을 쓰지 않으면

Linear regression과 똑같음.

(y = W * X + b 형태)

Unit 02 | Activation Function



x1*w11+x2*w12+b1

x1*(w11+w21)+
x2*(w12+w22)+
b1+b2

x1*w21+x2*w22+b2

Activation Function을 선형함수로 하면?

-> Linear regression과 똑같음.

(y = W * X + b 형태)

Unit 02 | Activation Function

Activation Function을 선형함수로 하면?

비선형 함수를 이용하자!

-> Linear regression과 똑같음.

(y = W * X + b 형태)

비선형 함수(non-linear function)

딥러닝의 핵심 내용은 선형 함수로 표현하지 못하던 비선형 영역을 표현

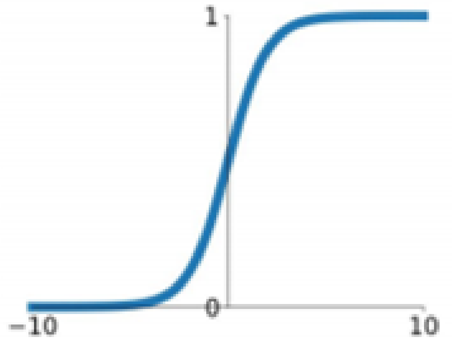
Unit 02 | Activation Function

비선형 함수(non-linear function) 중

대표로

Sigmoid

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



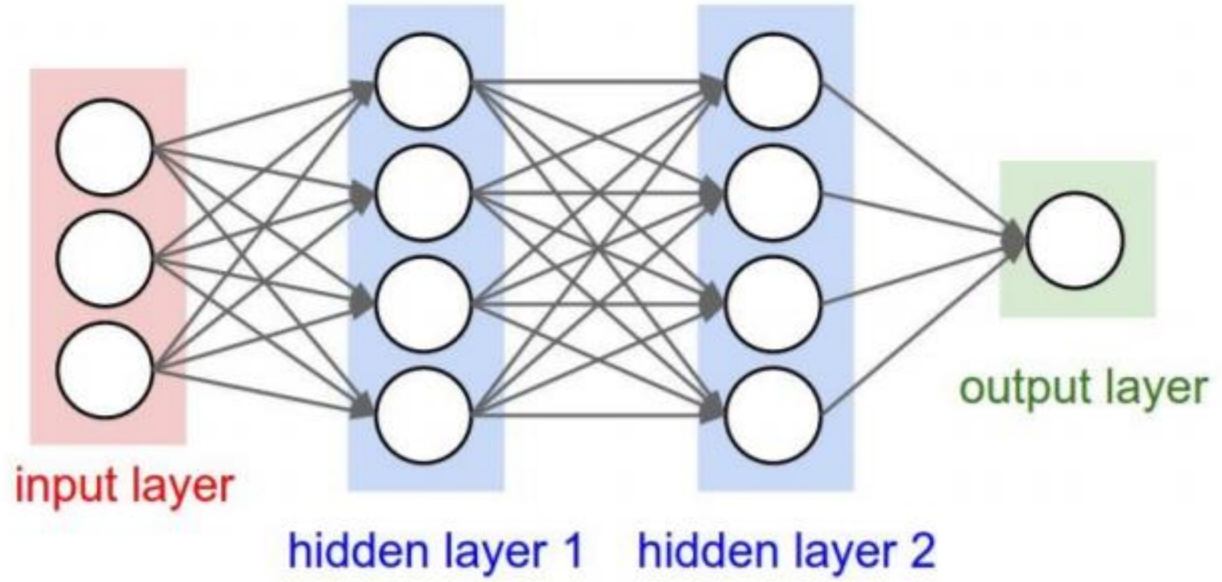
나는 시그모이드...!

Unit 02 | Activation Function

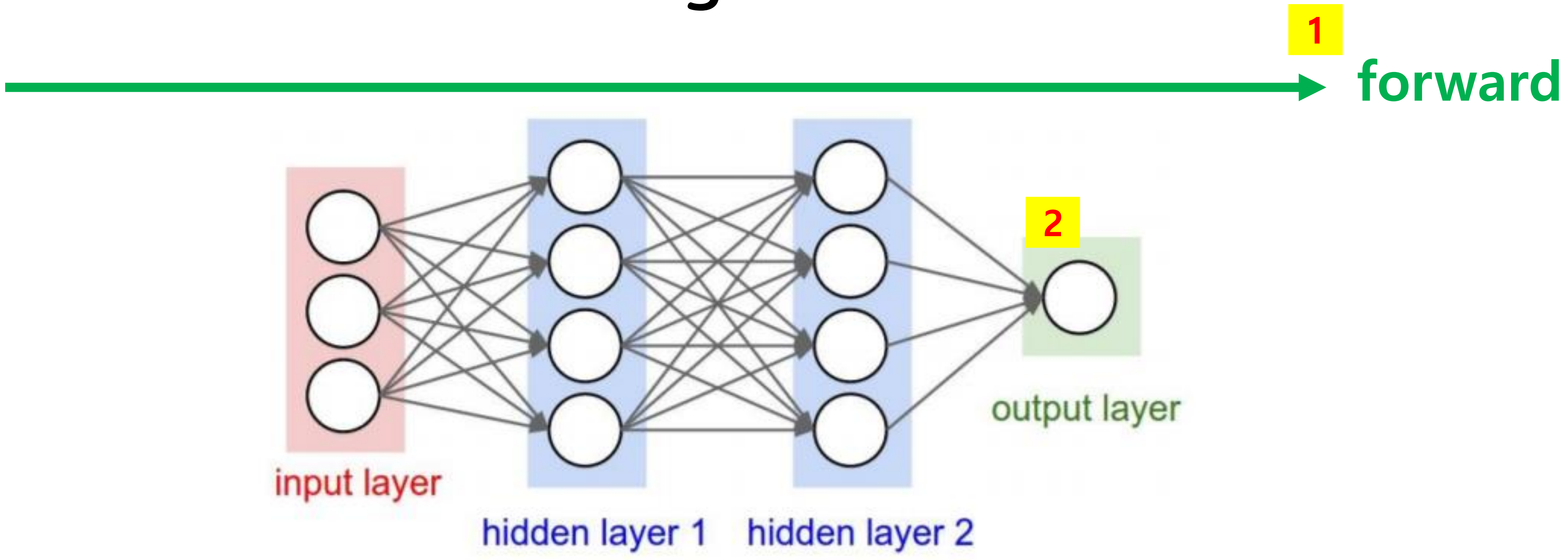
Training Process

1

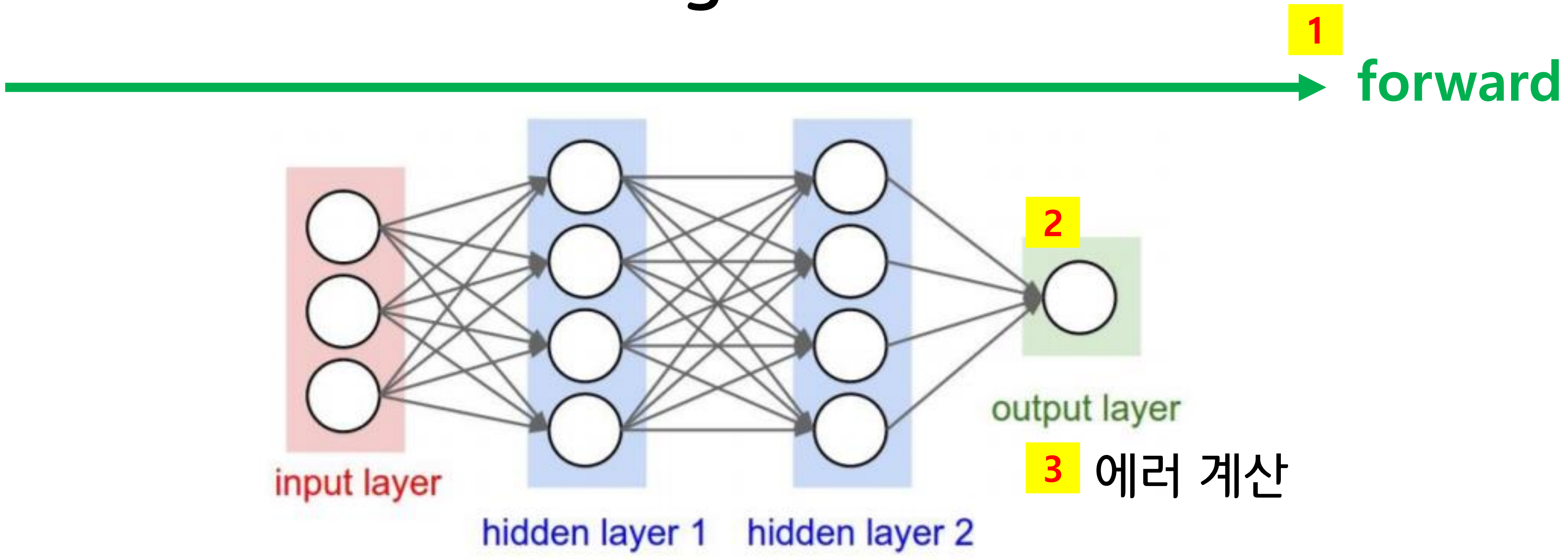
forward



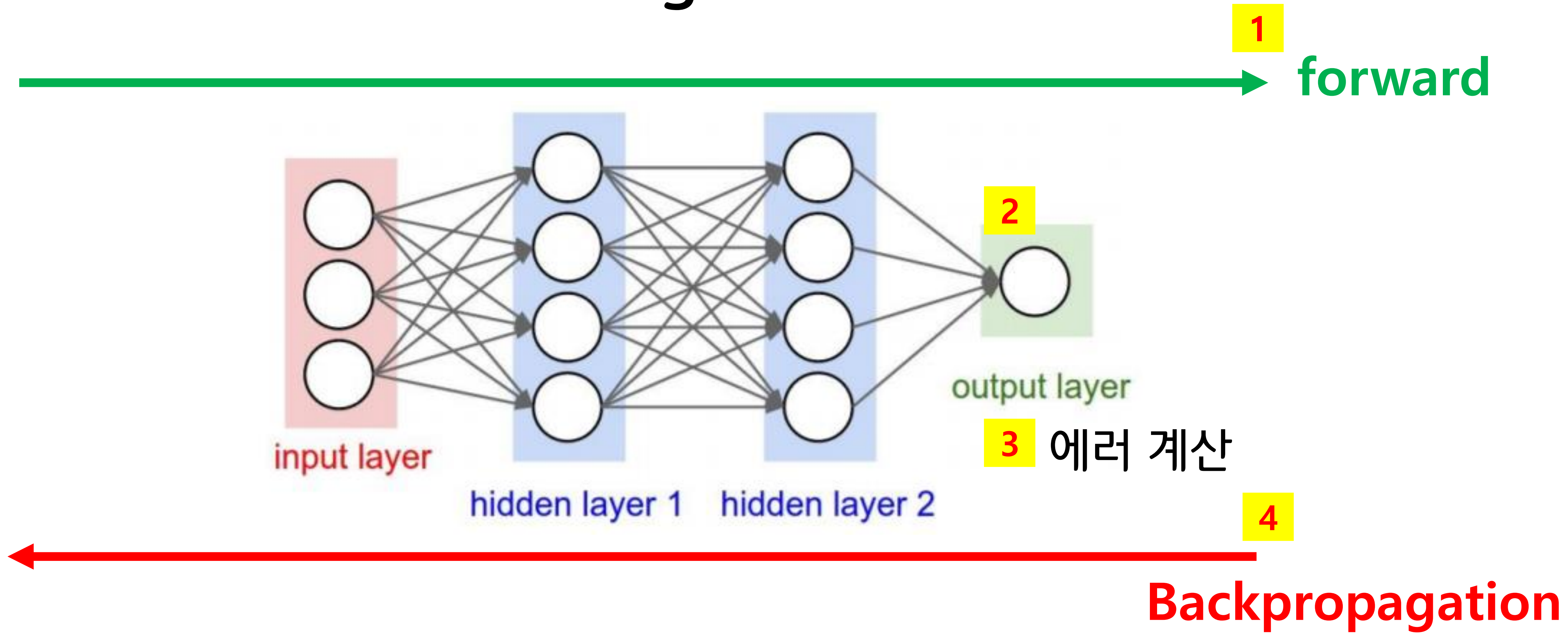
Training Process



Training Process

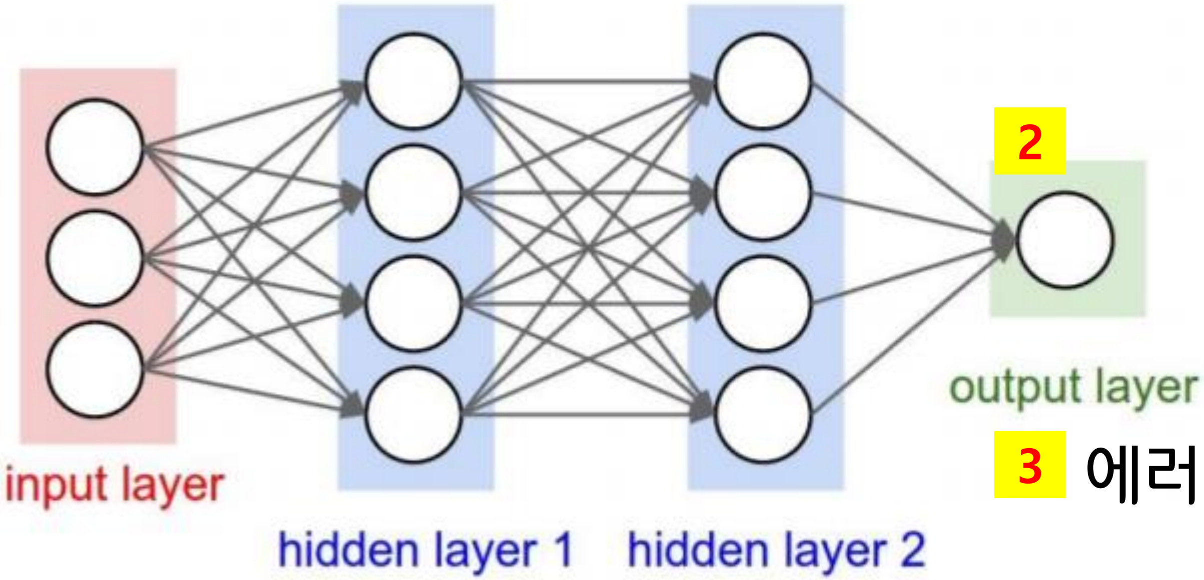


Training Process



Training Process

1 forward



2

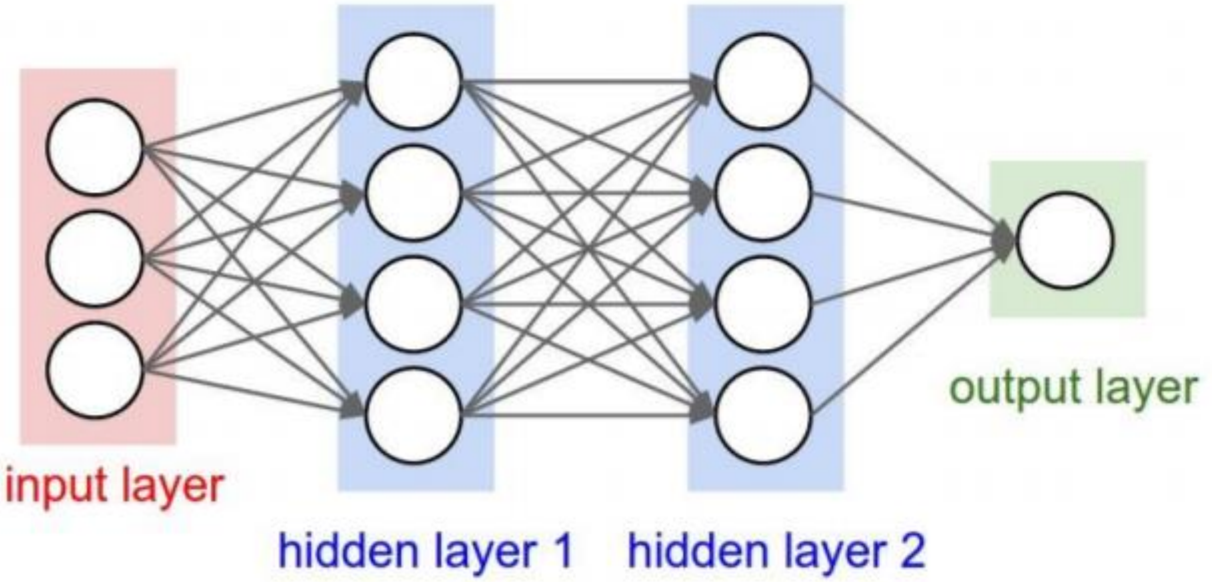
3 에러 계산

4 Backpropagation

5 Weight, bias 갱신

Vanishing Gradient Problem

= 사라지는 기울기..



4

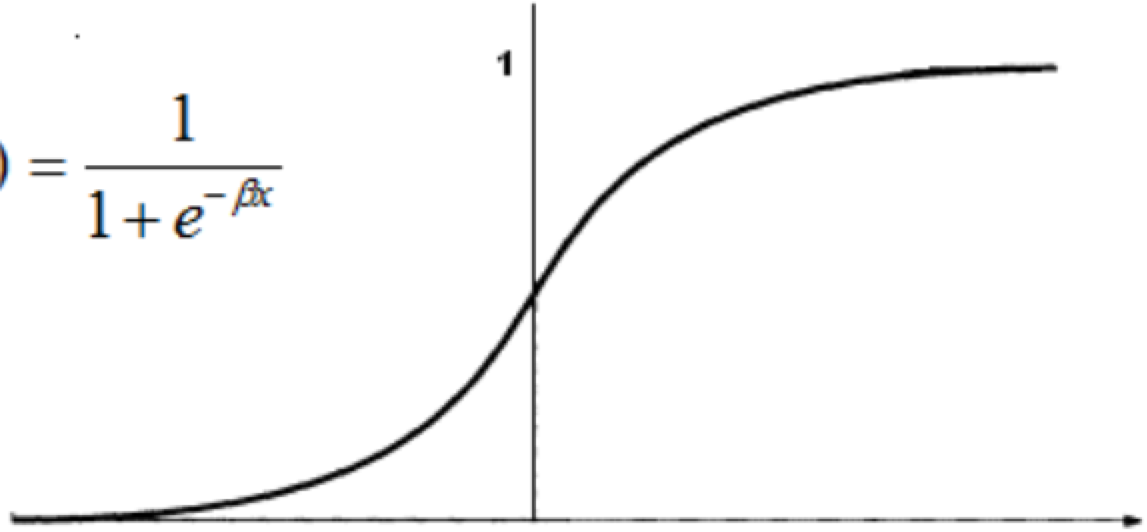
틀린 정도의 기울기 전달 = Backpropagation

Unit 02 | Activation Function

비선형 문제를 풀기 위해서 추가한 활성함수들, 하지만 모든 출력을 [0,1]사이이로 압축해버리는 sigmoid특성 때문에 기울기가 거의 사라져 버림.

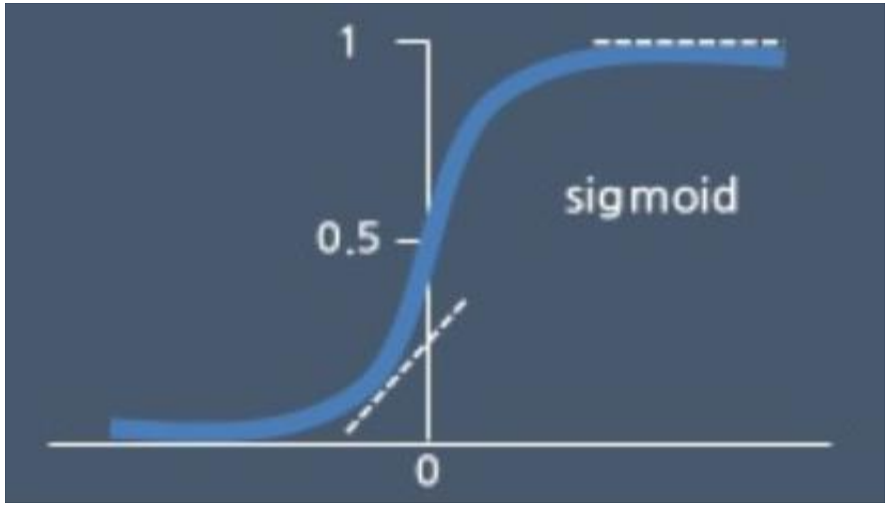
Sigmoid

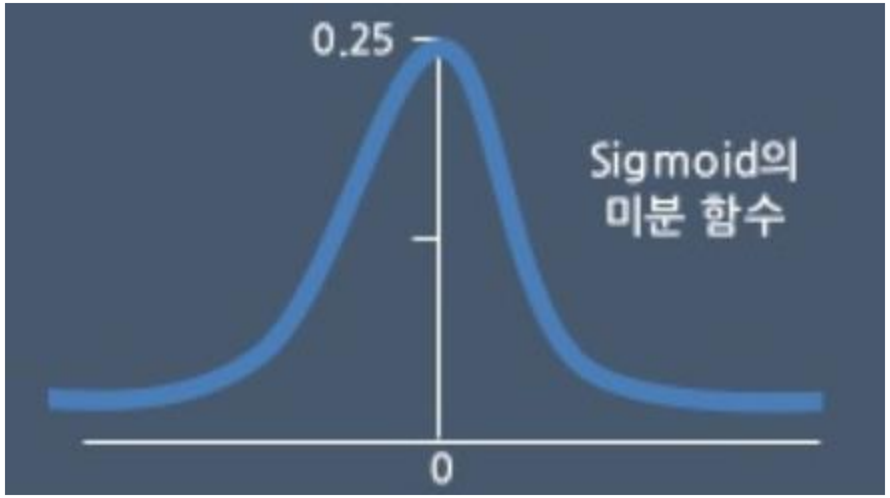
$$f(x) = \frac{1}{1+e^{-\beta x}}$$



Unit 02 | Activation Function

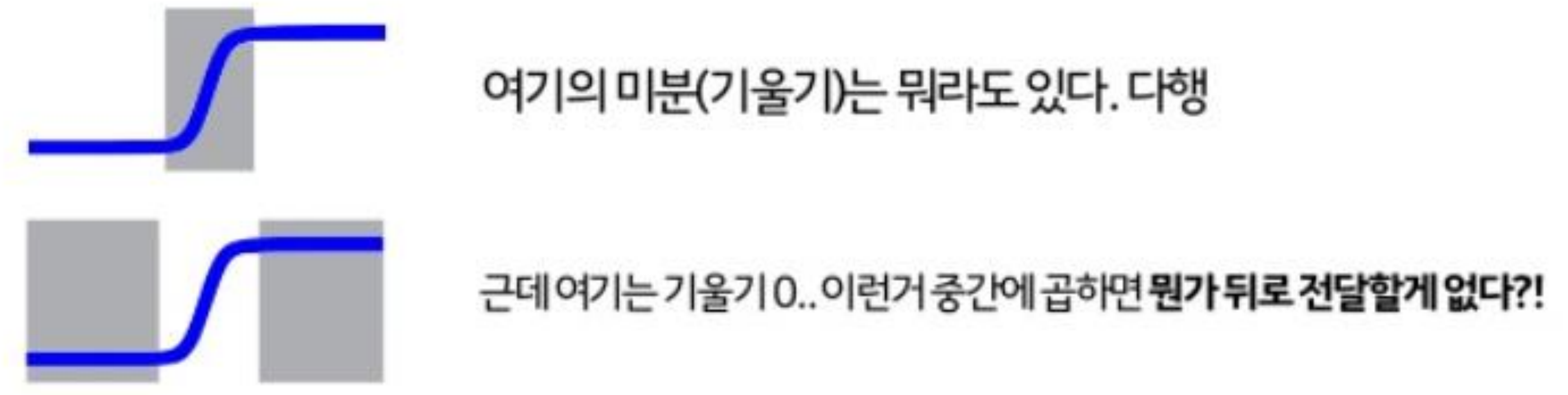
Vanishing Gradient로 인해 학습되는 양이 0에 가까워져, 학습이 더디게 진행되다가 오차를 더 줄이지 못하고 그 상태로 수렴





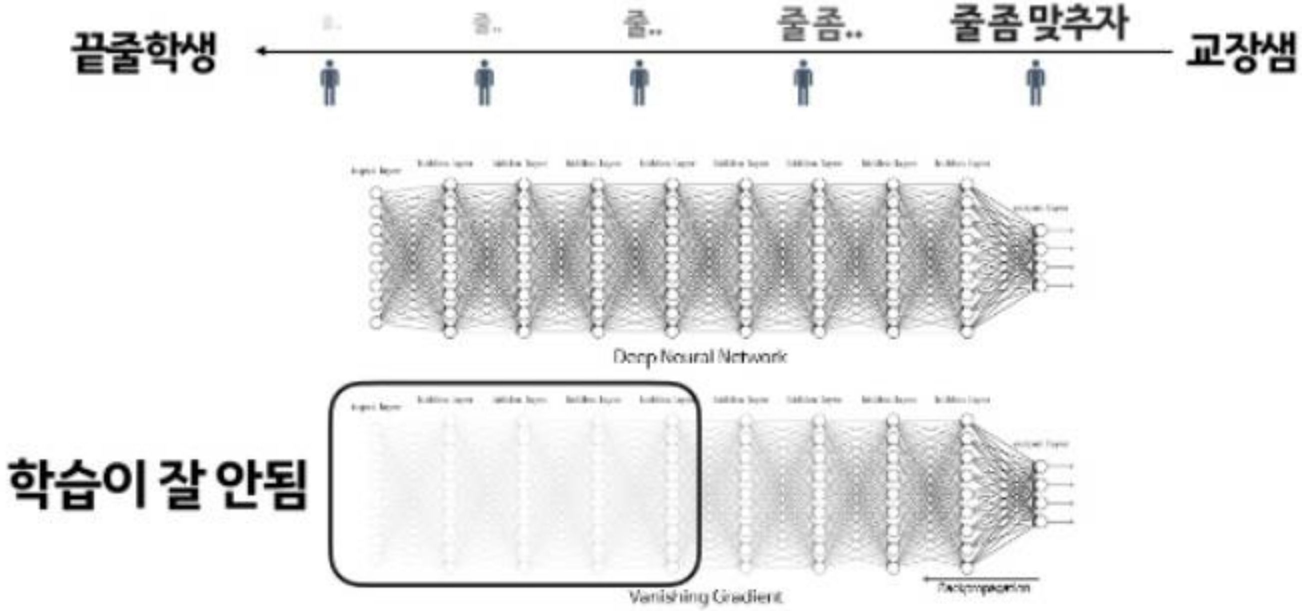
Unit 02 | Activation Function

Vanishing Gradient로 인해 학습되는 양이 0에 가까워져, 학습이 더디게 진행되다가 오차를 더 줄이지 못하고 그 상태로 수렴



Unit 02 | Activation Function

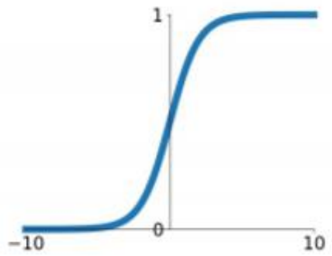
Vanishing Gradient로 인해 학습되는 양이 0에 가까워져, 학습이 더디게 진행되다가 오차를 더 줄이지 못하고 그 상태로 수렴



Activation Function 종류

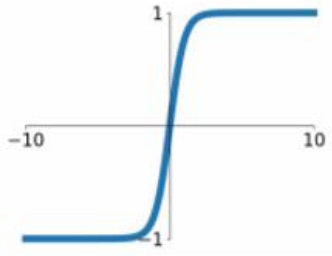
Sigmoid

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



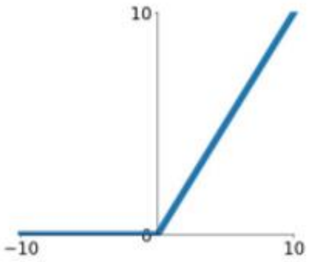
tanh

$\tanh(x)$



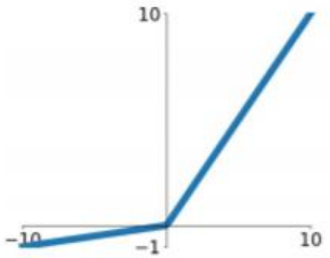
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

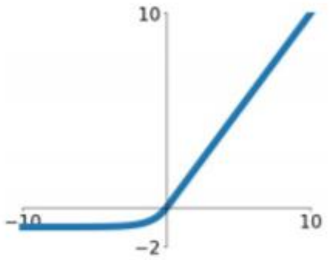


Maxout

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

ELU

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

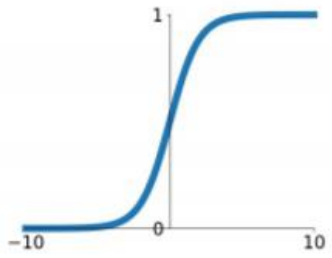


Unit 02 | Activation Function

Activation Function 종류

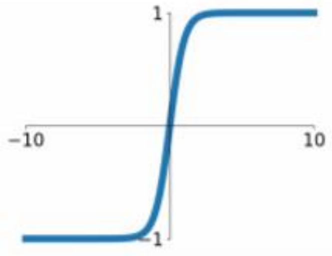
Sigmoid

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



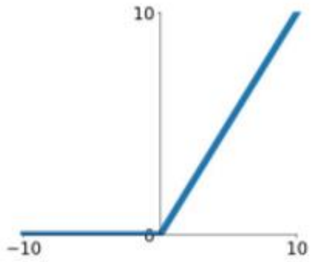
tanh

$\tanh(x)$



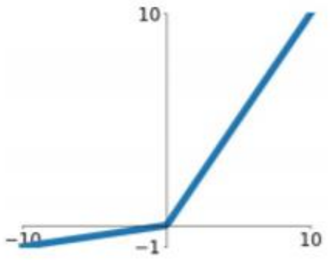
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

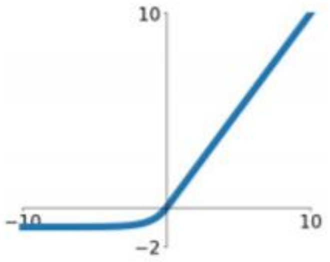


Maxout

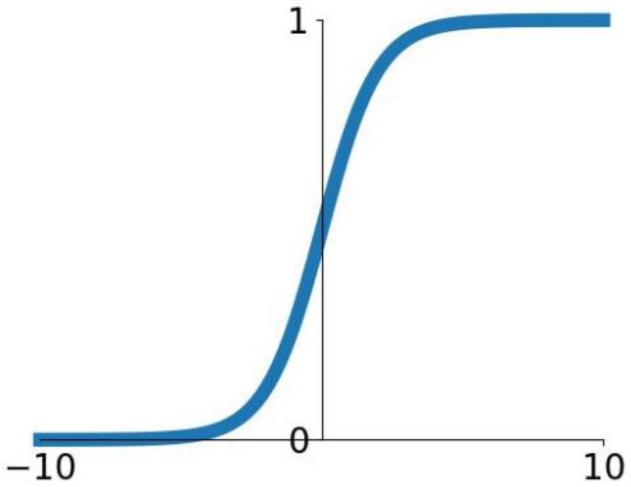
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

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



Unit 02 | Activation Function

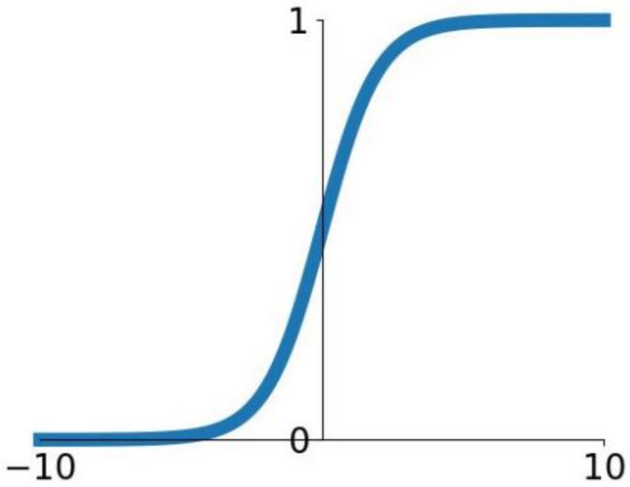


Sigmoid

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

Unit 02 | Activation Function



Sigmoid

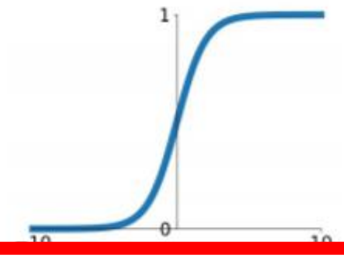
3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Activation Function 종류

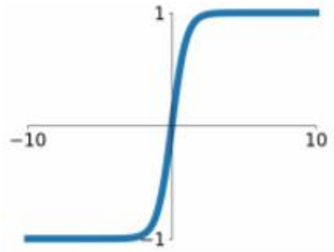
Sigmoid

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



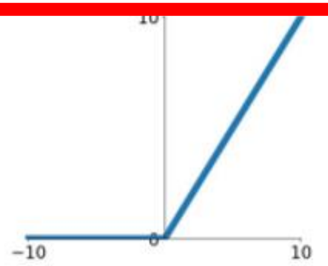
tanh

$\tanh(x)$



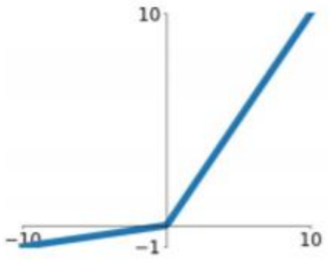
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

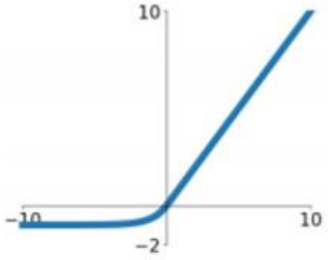


Maxout

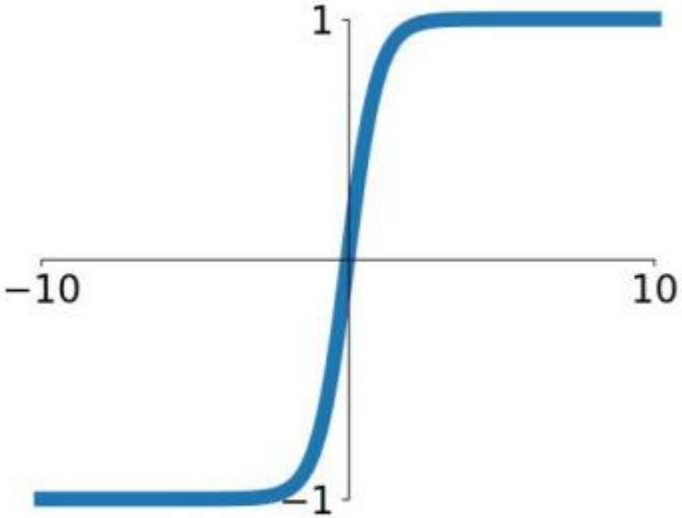
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

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



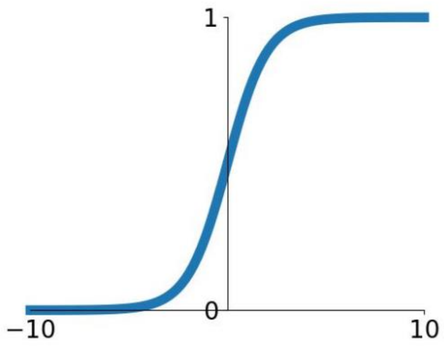
Unit 02 | Activation Function



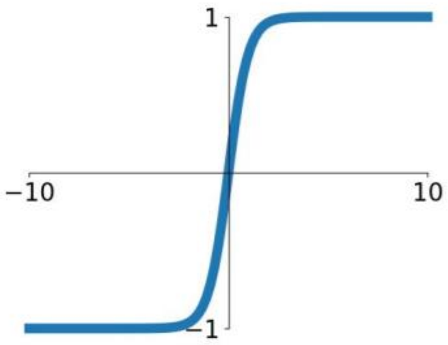
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

tanh(x)

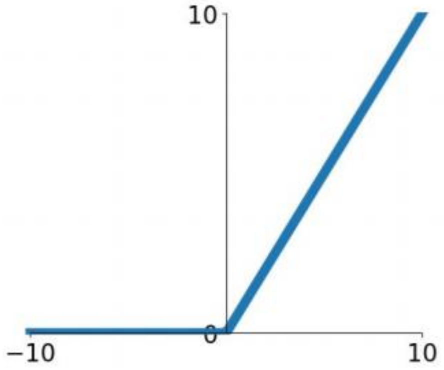
Vanishing Gradient Problem을 해결해줄 놈 등장..



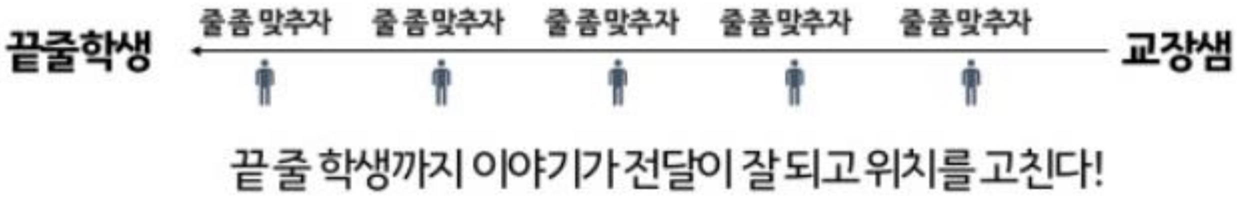
Sigmoid



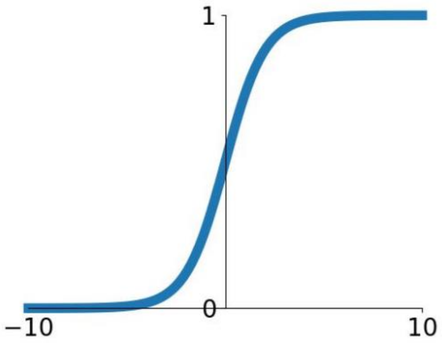
tanh(x)



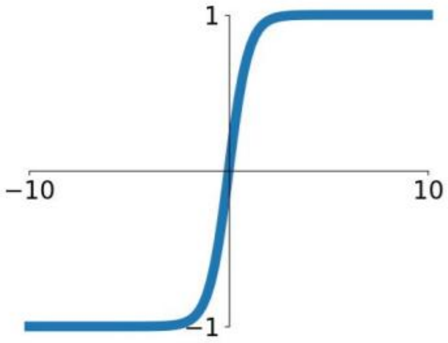
ReLU
(Rectified Linear Unit)



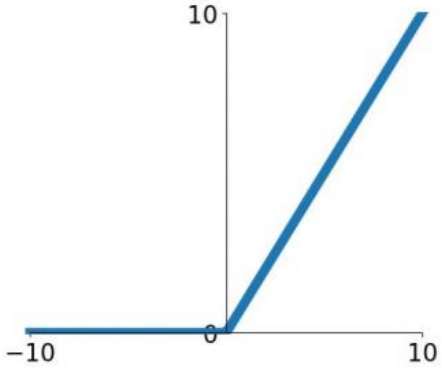
Vanishing Gradient Problem을 해결해줄 놈 등장..



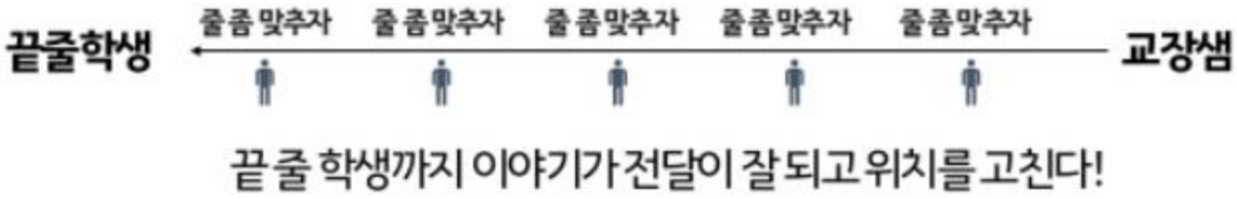
Sigmoid



tanh(x)



ReLU
(Rectified Linear Unit)

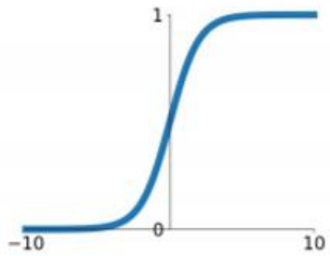


+ 수렴속도가 시그모이드류 함수 대비 6배 가량 빠름!

Activation Function 종류

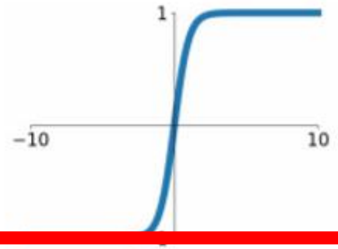
Sigmoid

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



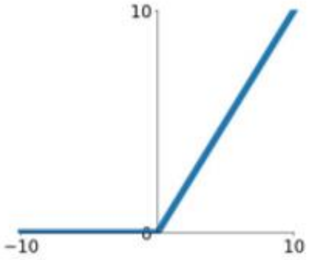
tanh

$\tanh(x)$



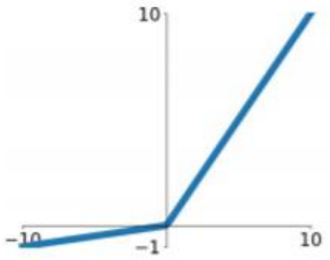
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

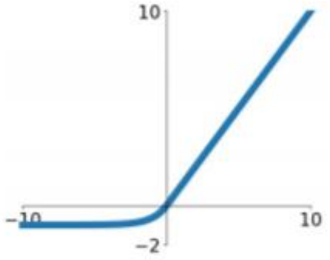


Maxout

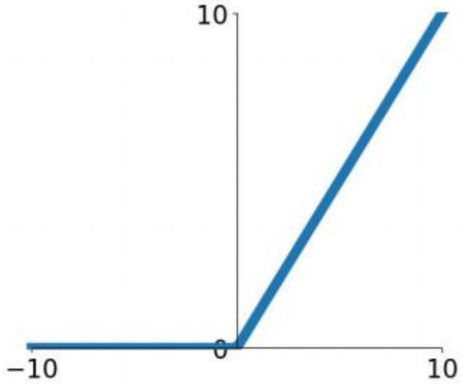
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

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



Unit 02 | Activation Function

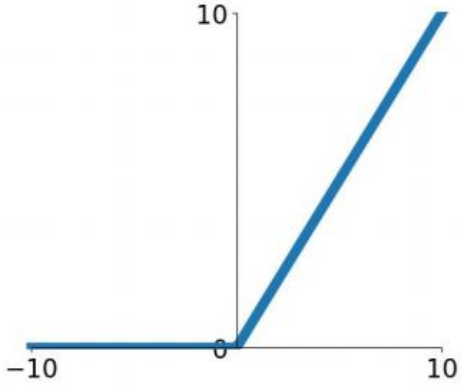


ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

Unit 02 | Activation Function

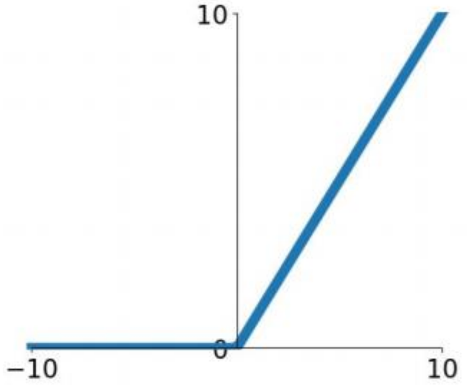


ReLU
(Rectified Linear Unit)

양의 구간에서의 미분 값은 1

$$F(x) = \begin{cases} x & (x > 0) \\ 0 & (x <= 0) \end{cases}$$

Unit 02 | Activation Function

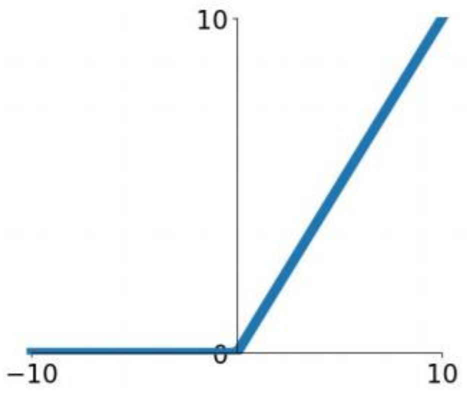


ReLU
(Rectified Linear Unit)

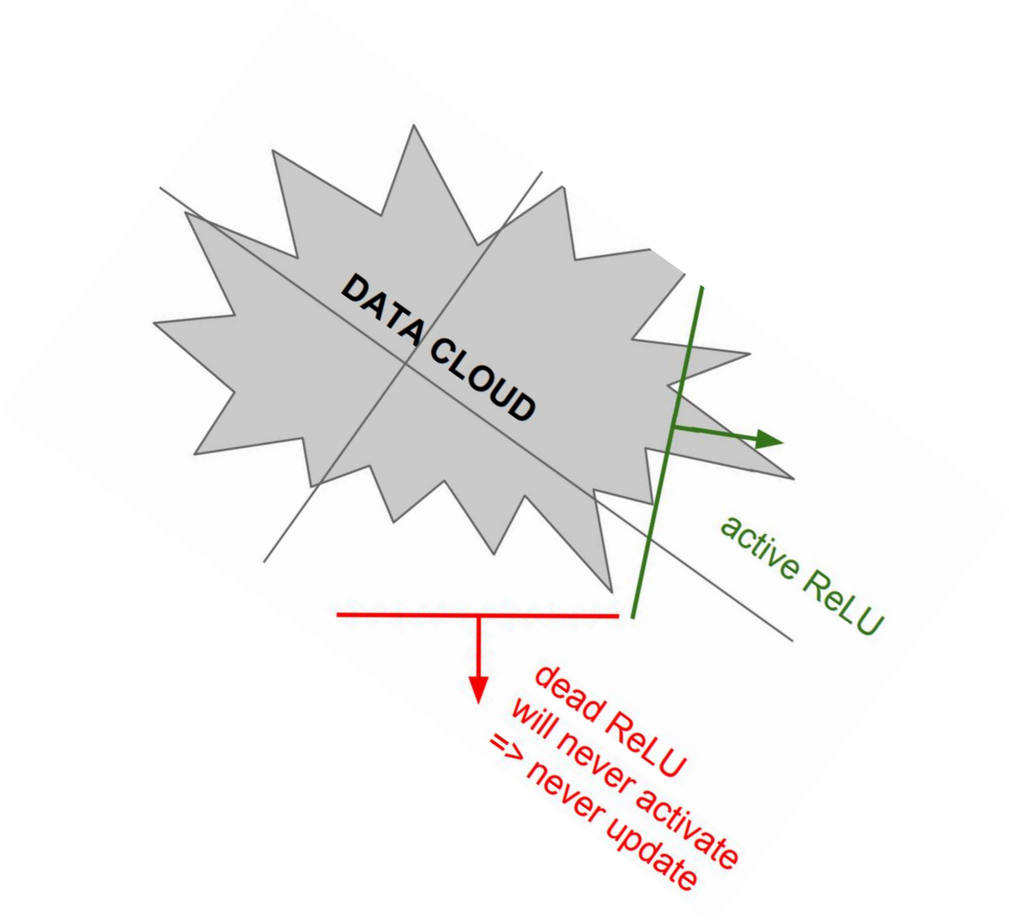
- Not zero-centered output
- An annoyance:

hint: what is the gradient when $x < 0$?

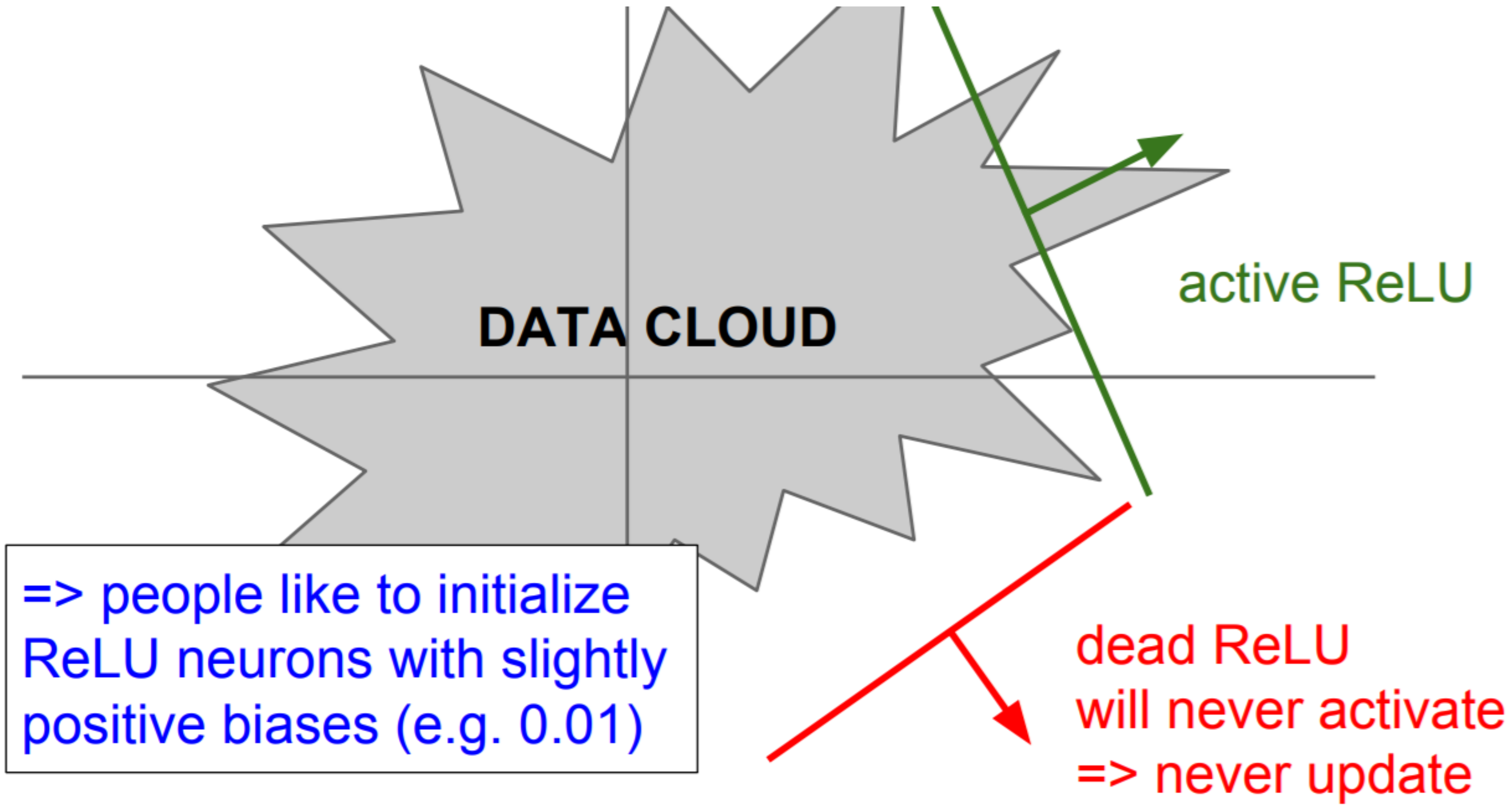
Unit 02 | Activation Function



ReLU
(Rectified Linear Unit)



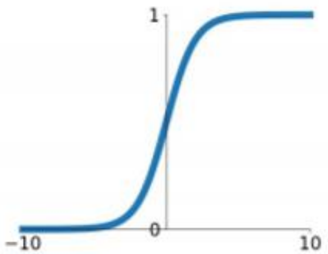
Unit 02 | Activation Function



Activation Function 종류

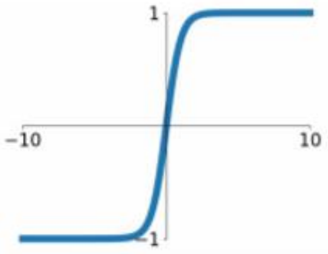
Sigmoid

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



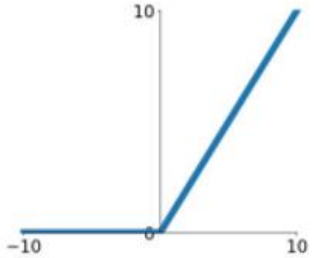
tanh

$$\tanh(x)$$



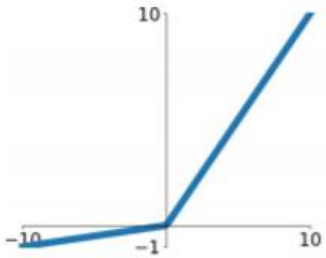
ReLU

$$\max(0, x)$$



Leaky ReLU

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

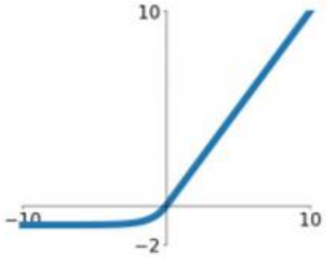


Maxout

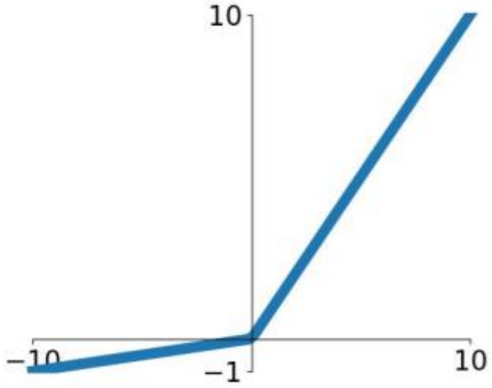
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Unit 02 | Activation Function

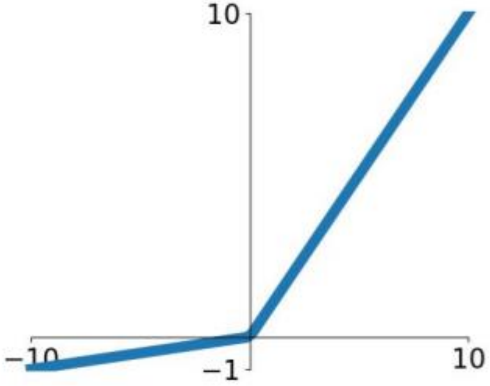


Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Unit 02 | Activation Function



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$\boxed{f(x) = \max(\alpha x, x)}$$

backprop into \alpha
(parameter)

Conclusion

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

Code

Activation Functions

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities (`sigmoid`, `tanh`, `elu`, `softplus`, and `softsign`), continuous but not everywhere differentiable functions (`relu`, `relu6`, `crelu` and `relu_x`), and random regularization (`dropout`).

All activation ops apply componentwise, and produce a tensor of the same shape as the input tensor.

- `tf.nn.relu`
- `tf.nn.relu6`
- `tf.nn.crelu`
- `tf.nn.elu`
- `tf.nn.softplus`
- `tf.nn.softsign`
- `tf.nn.dropout`
- `tf.nn.bias_add`
- `tf.sigmoid`
- `tf.tanh`

Unit 03 | Weight Initialization

**처음부터 정답과 비슷한 답을 내는 초기값을 설정한다면
당연히 학습이 빠르겠죠??**

Unit 03 | Weight Initialization

처음부터 정답과 비슷한 답을 내는 초기값을 설정한다면
당연히 학습이 빠르겠죠??

예전엔 초기값을 랜덤하게 설정했지만
이제는 Xavier나 He initialization을 사용합니다.

Unit 03 | Weight Initialization

Activation Function	Initialization	Code
Sigmoid	Xavier	np.random.randn(n_input, n_output) / sqrt(n_input)
ReLU	He	np.random.randn(n_input, n_output) / sqrt(n_input / 2)

Gaussian으로 초기화 하고,
Xavier Initialization: 인풋개수의 제곱근으로 나누기
He Initialization: 인풋개수의 절반의 제곱근으로 나누기

Unit 01 | 우리가 지금 모 배우는지..

Deep layer & More Neuron

문제 발생!

Overfitting

Vanishing Gradient

Too Slow

해결책 ☺

Regularization
-L2, L1 loss,
Dropout

Activation Function
Weight Initialization

Batch Normalization
Optimizer

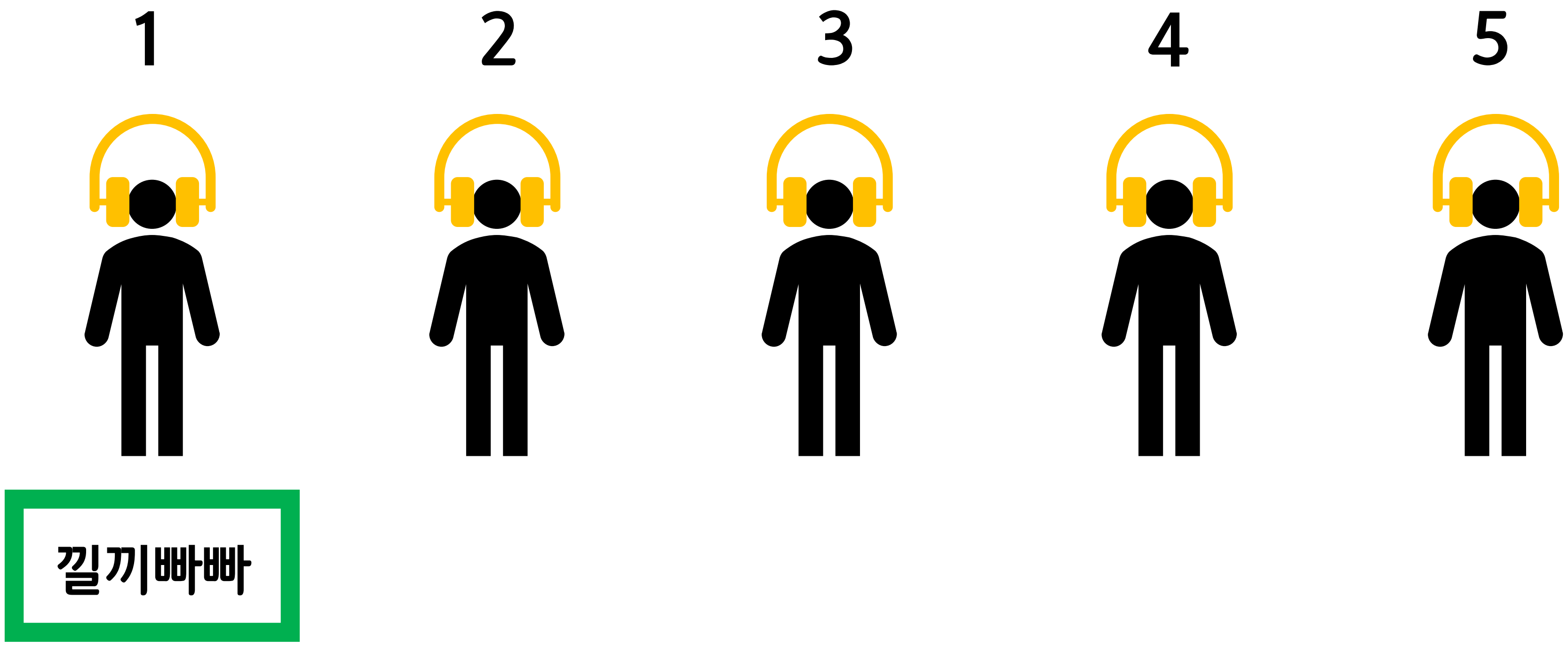
Batch Normalization VS Other Methods(Relu 함수, L-2, L-1, Dropout..)

네트워크망 내부 데이터를 아예 안정적으로 학습을 할 수 있도록 만들어 버림.

대게 데이터에 직접 손을 대지 않고 간접적으로 해결을 하는 방식

Internal Covariate Shift 현상

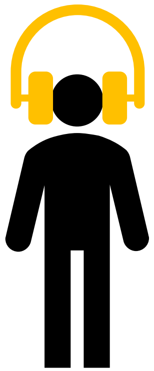
Internal Covariate Shift 현상



Internal Covariate Shift 현상

Internal Covariate Shift 현상

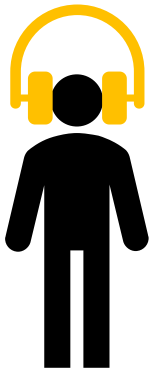
1



낄끼빠빠

레이어가 뒤로 갈수록
분포가 변화되어 결국
출력층에 안 좋은 영향
을 끼칠 수 있게 된다.

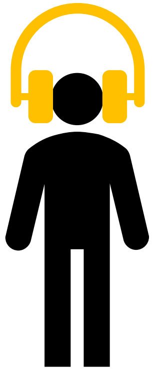
5



뛰뛰빵빵!

Internal Covariate Shift 현상

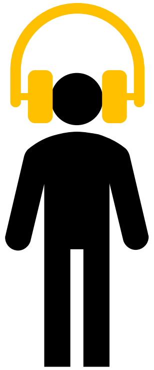
1



낄끼빠빠

즉, Network의 각 층이나 Activation마다 input의 distribution이 달라지는 현상

5



뛰뛰빵빵!

Unit 04 | Batch Normalization

Internal Covariate Shift 현상

1

5

이러한 현상을 방지하고자 각 네트워크망의 distribution을 같게 하는 방법으로 고안

레이어가 뒤로 갈수록

출력층에 안 좋은 영향을 끼칠 수 있게 된다.

낄끼빠빠

뛰뛰빵빵!

Internal Covariate Shift 현상을 막을 수 있는 방법

간단하게 각 층의 input의 distribution을 평균 0, 표준편차를 1인 input으로 normalize시키는 방법을 사용, 이 방법을 “whitening“이라고 함.

Internal Covariate Shift 현상을 막을 수 있는 방법

Whitening하기 위해 covariance matrix의 계산과 inverse계산이 필요하기 때문에 계산량이 증가, 일부 parameter들의 영향이 무시됨.

Whitening의 단점 보완 & Internal Covariance Shift를 줄이기 위한 접근

- 각각의 feature들이 이미 uncorrelated 되어있다고 가정하고, feature 각각에 대해서만 scalar 형태로 mean과 variance를 구하고 각각 normalize 한다.
- 단순히 mean과 variance를 0, 1로 고정시키는 것은 오히려 Activation function의 nonlinearity를 없앨 수 있다. 예를 들어 sigmoid activation의 입력이 평균 0, 분산 1이라면 출력 부분은 곡선보다는 직선 형태에 가까울 것이다. 또한, feature가 uncorrelated 되어있다는 가정에 의해 네트워크가 표현할 수 있는 것이 제한될 수 있다. 이 점들을 보완하기 위해, normalize된 값들에 scale factor (gamma)와 shift factor (beta)를 더해주고 이 변수들을 back-prop 과정에서 같이 train 시켜준다.
- training data 전체에 대해 mean과 variance를 구하는 것이 아니라, mini-batch 단위로 접근하여 계산한다. 현재 택한 mini-batch 안에서만 mean과 variance를 구해서, 이 값을 이용해서 normalize 한다.

Unit 04 | Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

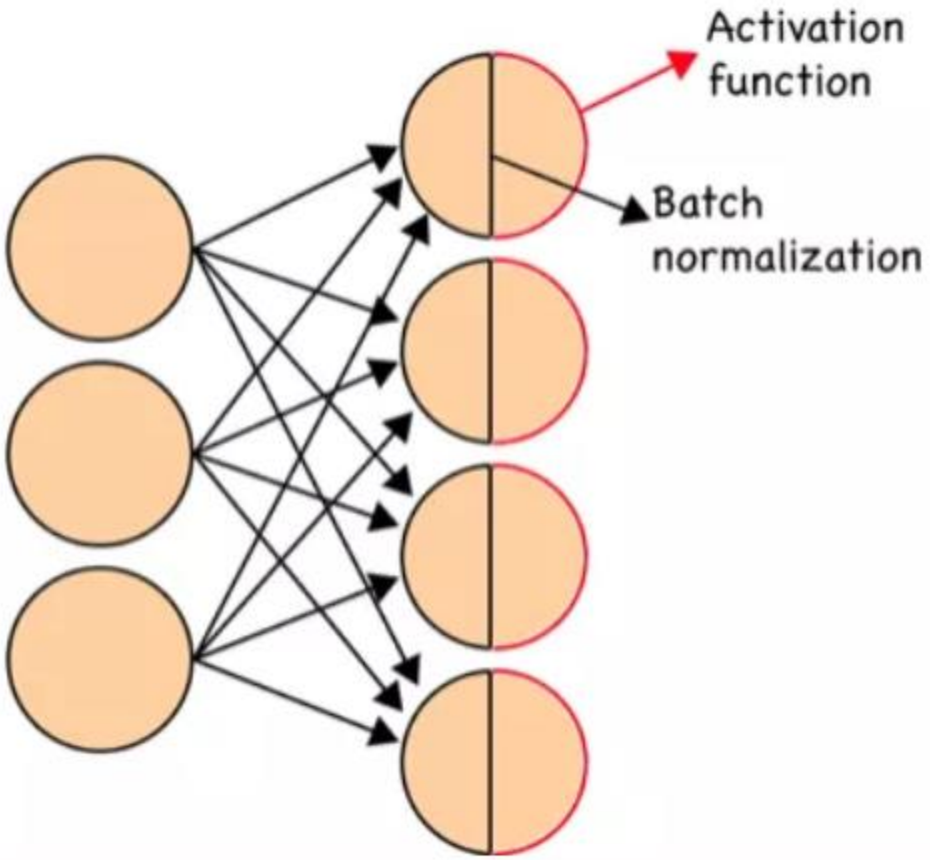
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Unit 04 | Batch Normalization



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

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

Unit 04 | Batch Normalization

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\mathrm{BN}}^{\mathrm{inf}}$

```
N_BN^tr ← N    // Training BN network
for k = 1 ... K do
    Add transformation y^(k) = BN_{γ^(k), β^(k)}(x^(k)) to N_BN^tr (Alg. 1)
    Modify each layer in N_BN^tr with input x^(k) to take y^(k) instead
end for
Train N_BN^tr to optimize the parameters Θ ∪ {γ^(k), β^(k)}_{k=1}^K
N_BN^inf ← N_BN^tr    // Inference BN network with frozen
                      // parameters
for k = 1 ... K do
    // For clarity, x ≡ x^(k), γ ≡ γ^(k), μ_B ≡ μ_B^(k), etc.
    Process multiple training mini-batches B, each of size m, and average over them:
```

$$E[x] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\mathrm{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

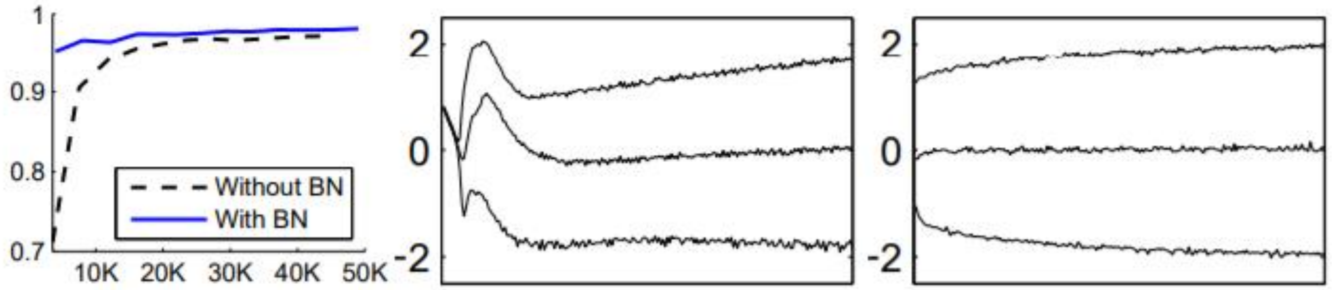
```
    In N_BN^inf, replace the transform y = BN_{γ,β}(x) with
```

$$y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\, E[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$$

```
end for
```

Algorithm 2: Training a Batch-Normalized Network

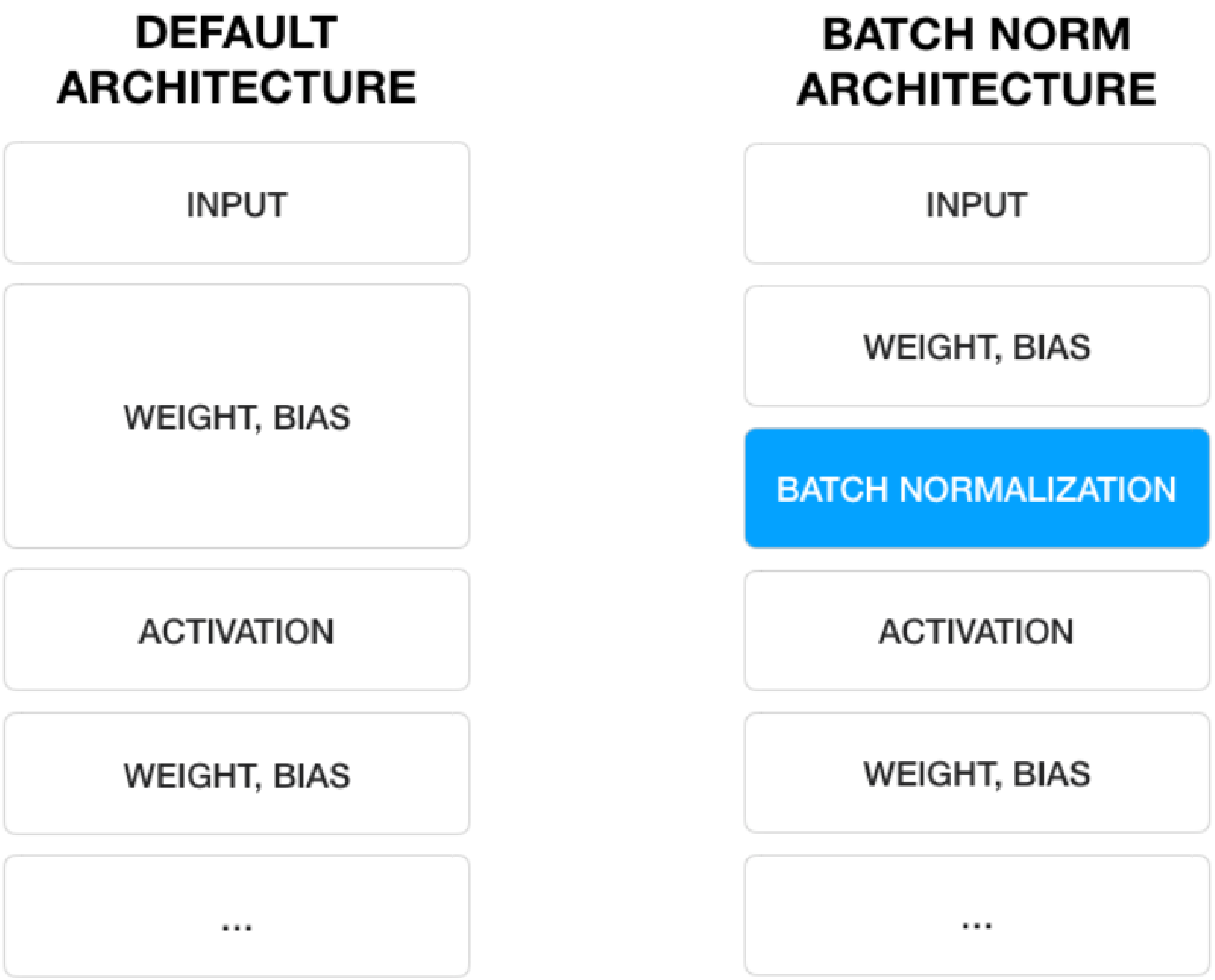
Unit 04 | Batch Normalization



(a) (b) Without BN (c) With BN

Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Unit 04 | Batch Normalization



Batch Normalization의 이점

- 신경망의 각 레이어의 분포를 같게 함으로써 좀 더 안정적인 학습이 가능
- 빠른 학습 속도
- 자체적인 Regularization

https://arxiv.org/pdf/1502.03167.pdf

Code

- `tf.nn.batch_normalization`
- `tf.nn.batch_norm_with_global_normalization`

기존 뉴럴넷이 가중치 parameter들을 최적화 하는 방법

“Gradient Descent”

Loss function의 현 가중치에서의 기울기(gradient)를 구해서 loss를 줄이는 방향으로 업데이트해 나간다.

Gradient Descent

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

learning rate :
한번에 얼마나 학습할지

gradient :
어떤 방향으로 학습할지

Gradient Descent

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

learning rate :
한번에 얼마나 학습할지

gradient :
어떤 방향으로 학습할지

근데 내가 트레이닝 데이터를 몇 억건 가지고 있다면…? 한 스텝 갈때마다 몇 억건 씩 계속 넣어??????????????????

Gradient Descent

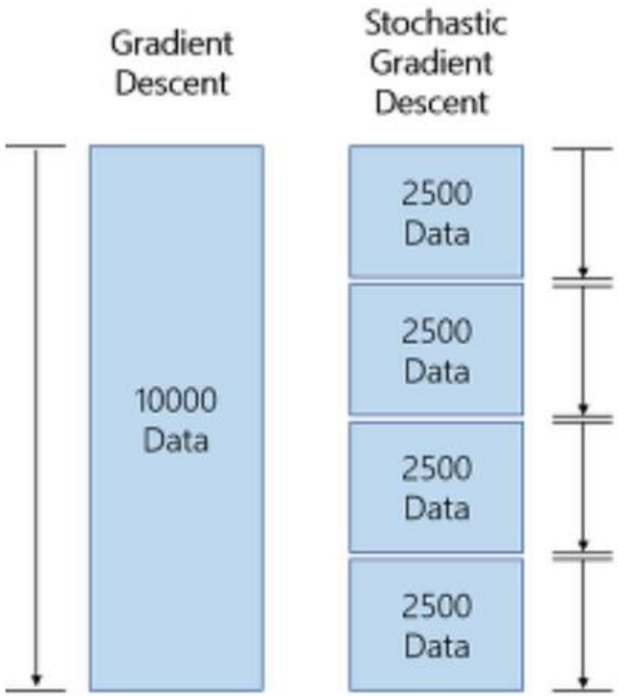
learning rate :
한번에 얼마나 학습할지

gradient :
어떤 방향으로 학습할지

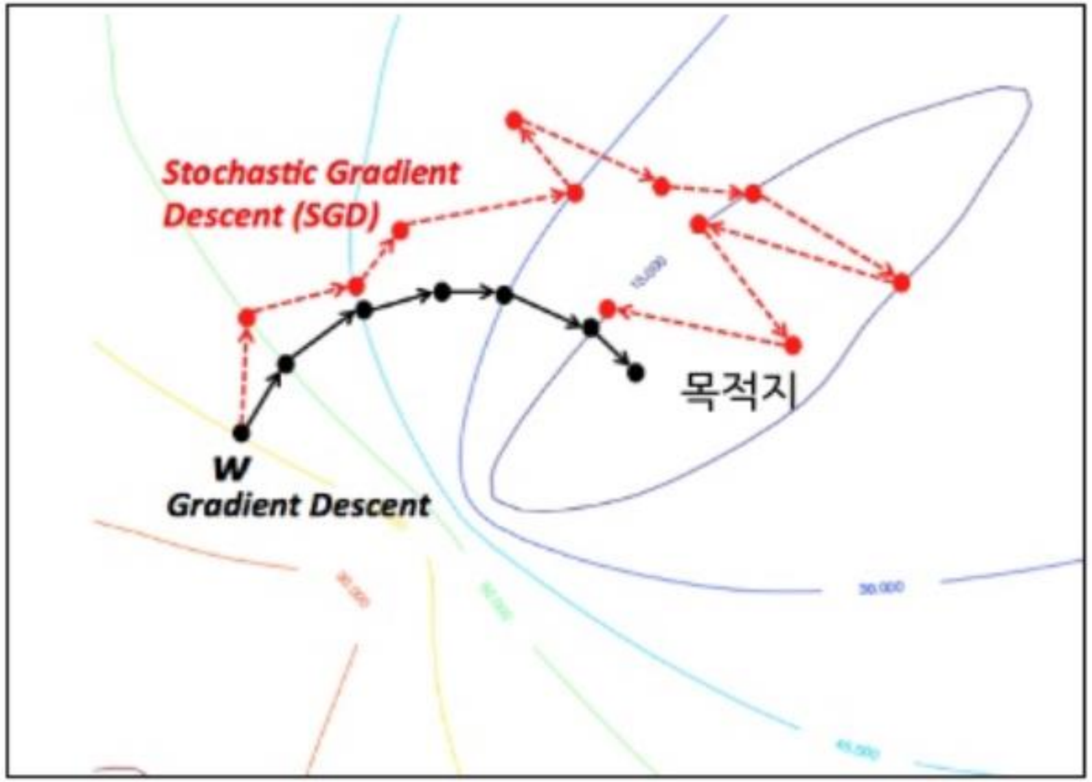
시간이 너무 오래 걸려..ㅠㅠ
GD보다 빠른 Optimizer은 없을까?

Stochastic Gradient Descent의 컨셉:

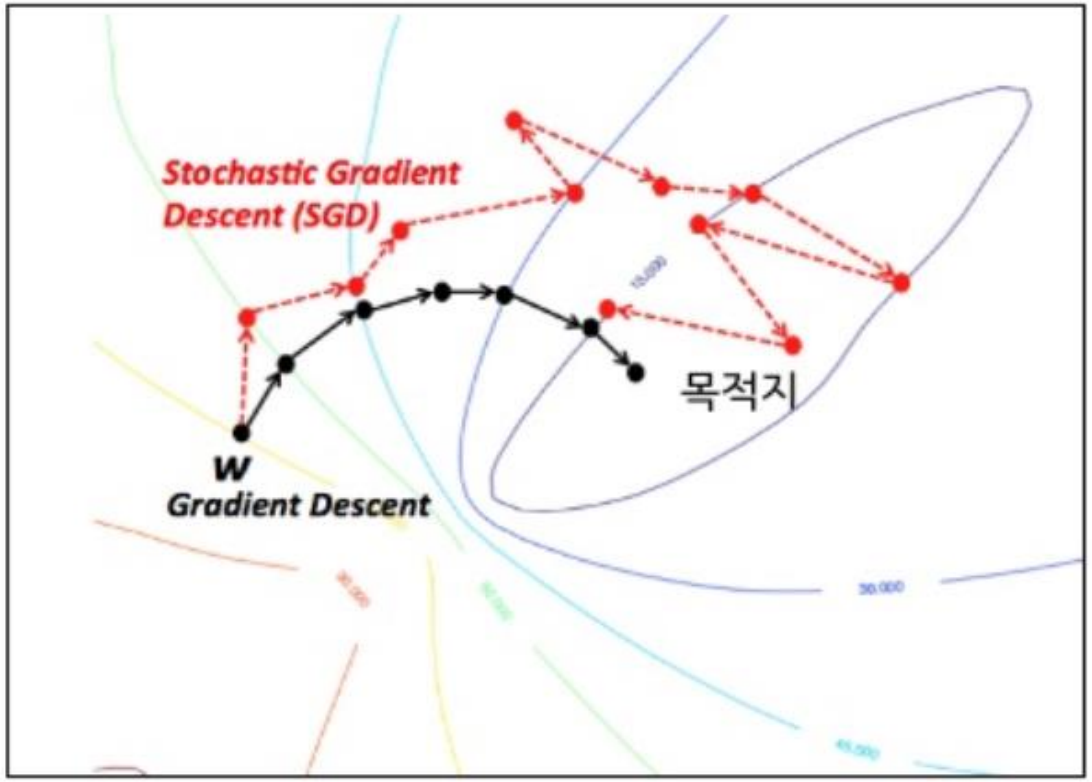
느린 완벽(GD의 컨셉)보다 조금만 훑어보고 일단 빨리 갑시다.



GD VS SGD



GD VS SGD



GD: 모든 걸 계산하는 것이 1시간 걸림
6스텝*1시간=6시간

SGD: 일부만 검토하는 것이 5분 걸림
11스텝*5분=55분<1시간

결론: 조금 헤매도 어쨌든 인근에 아주 빨리 갔다!

Unit 05 | Optimizer

문제점 : 스텝마다 batch로 전부 다 계산하려고 하니까 GD가 너무 오래 걸린다.

⬇

해결 : SGD로 mini-batch마다 움직여 같은 시간에 훨씬 더 많이 진행해서 해결!

Unit 05 | Optimizer

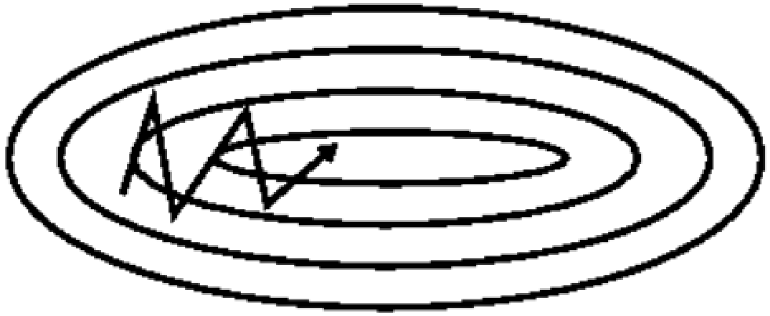
문제점 : 스텝마다 batch로 전부 다 계산하려고 하니까 GD가 너무 오래 걸린다.

근데 미니배치를 하다 보니 **진동현상** 발생!

해결 : SGD로 mini-batch마다 움직여 같은 시간에 훨씬 더 많이 진행해서 해결!

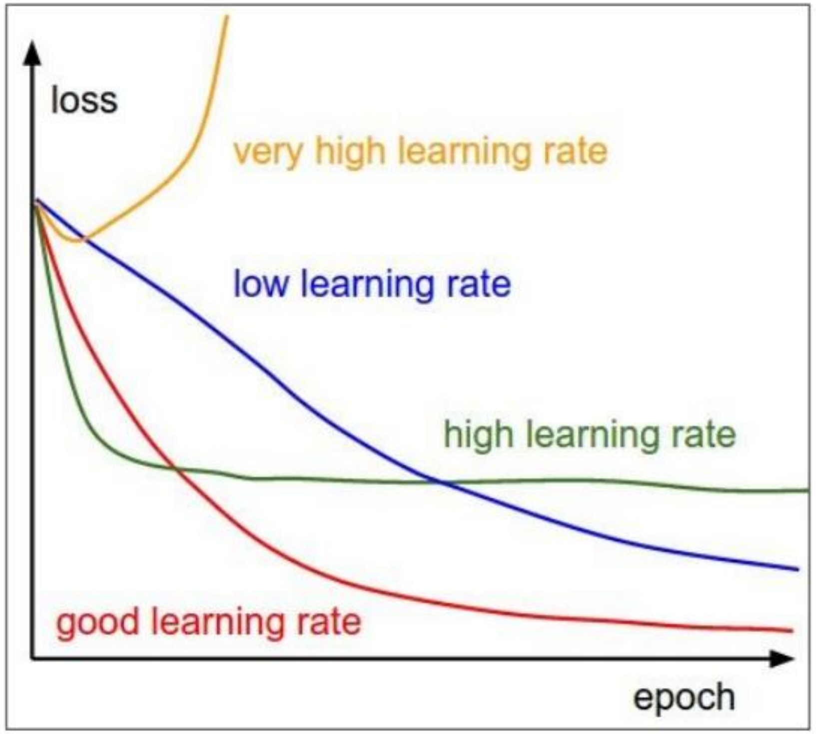
진동 현상 문제 (= Oscillation 현상)

딱 봐도 더 잘 갈 수 있는데 훨씬 더 헤매면서 간다.



훑기도 잘 훑으면서 좀 더 휙휙 더 좋은 **방향**으로 갈 수는 없을까?

스텝 사이즈(learning rate) 설정 문제



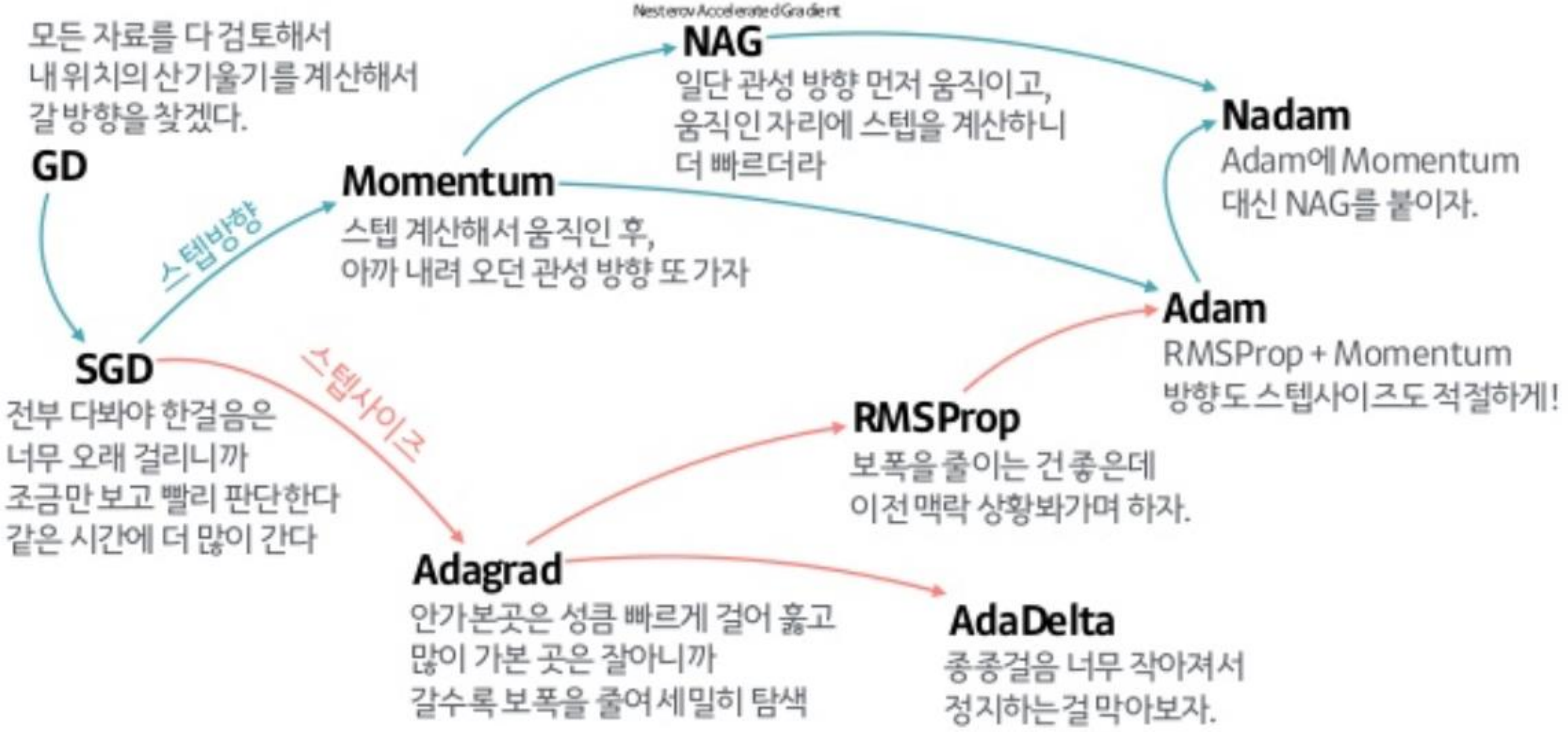
Learning rate가 너무 작으면 시간이 너무 오래 걸리고

Learning rate가 너무 크면 최적 값을 놓칠 수 있음.

따라서..

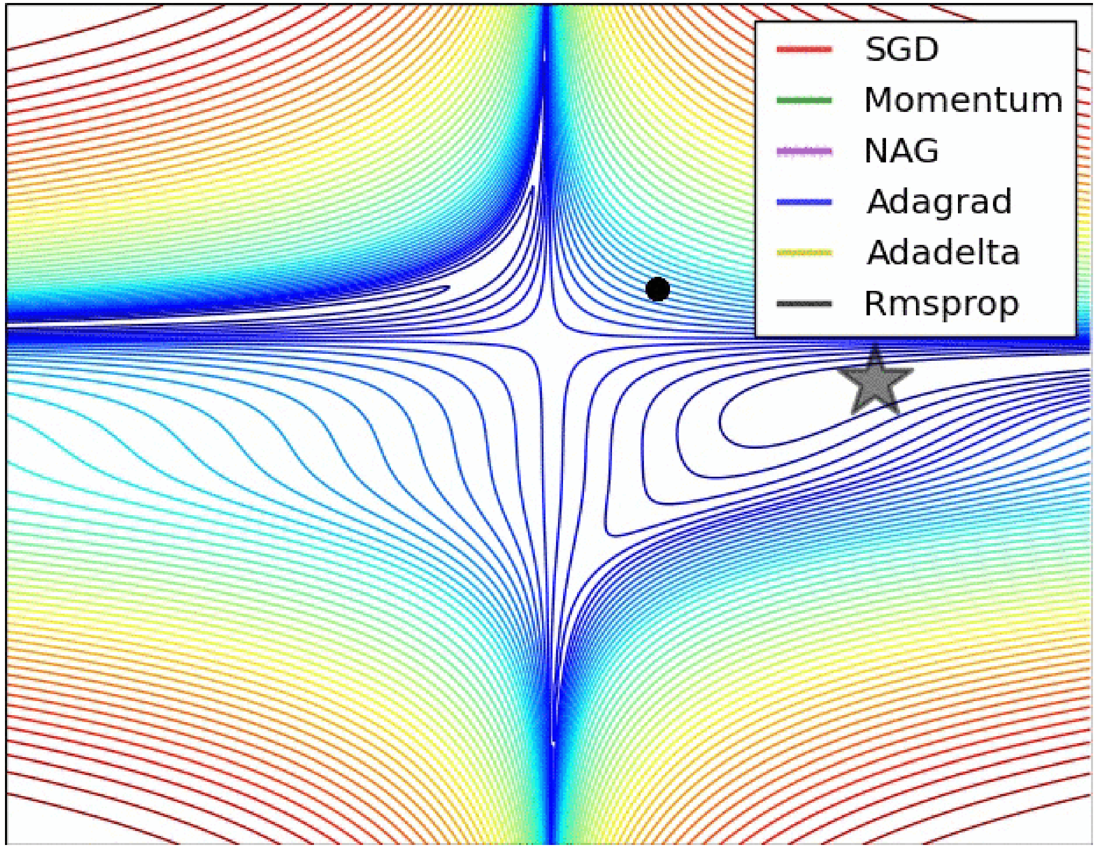
어느 방향으로 갈지,
얼마 learning rate(스텝 사이즈)로 설정할지
두 가지를 잘 잡아야 빠르게 타고 내려온다.

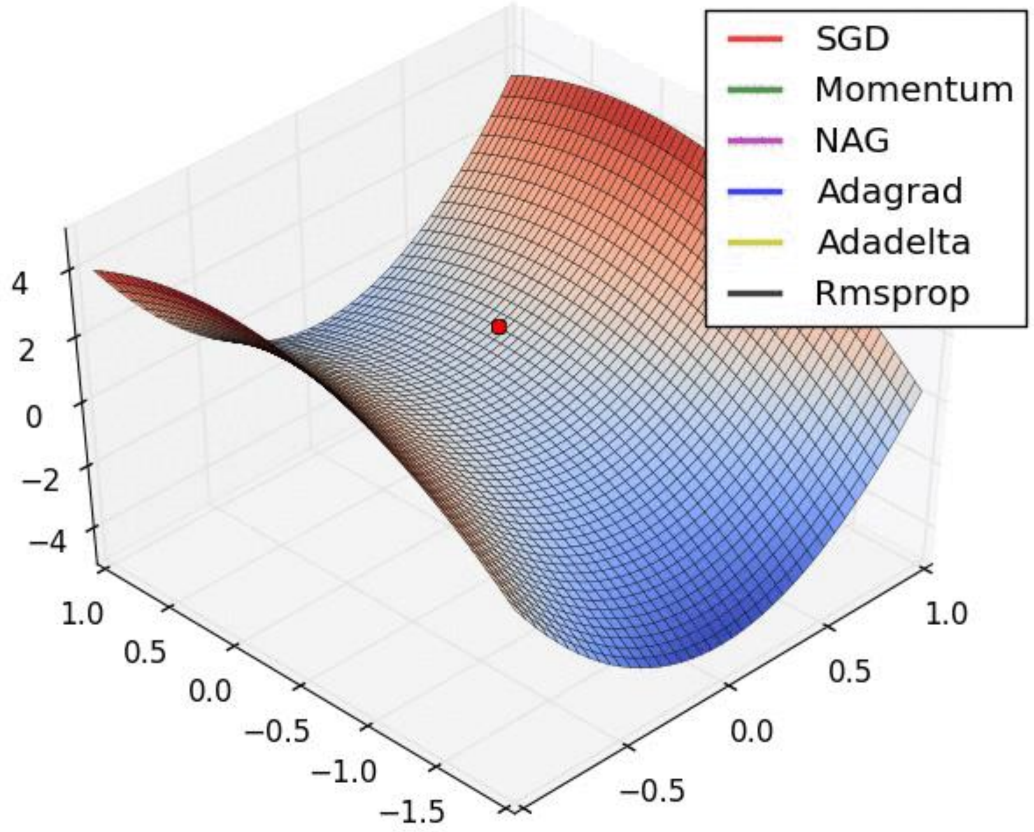
Optimizer 발전 과정 (feat. SGD를 더 개선한 후계자들..)



Unit 05 | Optimizer

Optimizer 발전 과정 (feat. SGD를 더 개선한 후계자들..)





Optimizer 발전 과정 (feat. SGD를 더 개선한 후계자들..)

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

learning rate :
한번에 얼마나 학습할지

gradient :
어떤 방향으로 학습할지

- Gradient 를 수정한 Momentum, Nag
- Learning Rate를 수정한 Adagrad, RMSProp, AdaDelta
- 이 두 종류의 장점을 합한 Adam, Nadam

Momentum

- SGD에서 계산된 gradient에 한 스텝 전의 gradient를 일정한 비율만큼 반영하여 사용한다.
- 원래의 gradient를 일정부분 유지하면서 새로운 gradient를 적용하여 관성 모멘트와 같은 효과를 주는 방식, 이를 통해 local minima를 더 빨리 빠져나올 수 있다.

Adagrad

- Learning rate를 normalization하여 서서히 감소시킨다.

RMSprop

- 모든 경사를 더하는 대신 지수이동평균을 사용한다. Non-stationary한 데이터를 학습시킬 때 많이 사용한다.

Adadelta

- RMSprop+Adagrad를 보정한 것이지만 실제로는 Adagrad가 더 학습을 잘 하는 경우가 많다.

Adam

- Adagrad와 비슷함.
- 0으로 편향된 것을 보정
- **최근 딥러닝에서 많이 사용된다.**

Unit 05 | Optimizer

문제점 : SGD가 빠른데 좀 헤맨다.

⬇

해결 : SGD의 개선된 버전을 골라 더 빠르고 더 정확하게!

http://shuuki4.github.io/deep%20learning/2016/05/20/Gradient-Descent-Algorithm-Overview.html

Unit 05 | Optimizer

Code

```
train = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(cost)
#train = tf.train.AdamOptimizer(learning_rate=0.01).minimize(cost) # Adam 부터 테스트하기

You never instantiate the Optimizer class itself, but instead instantiate one of the subclasses.

tf.train.Optimizer
tf.train.GradientDescentOptimizer
tf.train.AdadeltaOptimizer
tf.train.AdagradOptimizer
tf.train.AdagradDAOptimizer
tf.train.MomentumOptimizer
tf.train.AdamOptimizer
tf.train.FtrlOptimizer
tf.train.ProximalGradientDescentOptimizer
tf.train.ProximalAdagradOptimizer
tf.train.RMSPropOptimizer
```

< 과제 >

MNIST데이터 학습시키면서 적당한 Activation Function과 Weight Initialization, Optimizer등을 이용해 accuracy최대한 높이기

(accuracy높이려는 과정을 보여주세요.. 자세한 주석과 함께!)

< 과제 하면서 필요한 개념 >

Epoch: 학습 데이터 전체를 한 바퀴 도는 것

Ex) 데이터 전체 학습하는 일을 총 20번 하려면 for문을 돌리면 됩니다.

Batch_size: 학습 시 수만 개의 모든 데이터를 한꺼번에 학습하려면 너무 크므로 배치 크기로 나눠 미니배치를 사용

Minibatch: 전체 학습 데이터 셋을 batch_size크기로 쪼개서 학습

=> Epoch과 Batch_size를 같이 쓰려면 이중 for문이 필요하겠죠? ☺

Unit 06 | 과제

< 과제 하면서 필요한 개념 >

Q & A

들어주셔서 감사합니다.