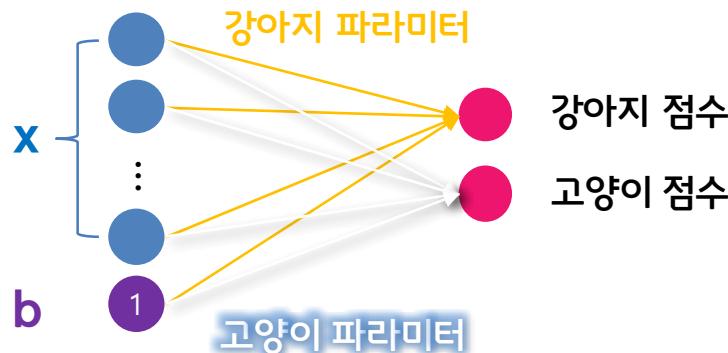


# 지난시간 Review

모두의연구소  
박은수 Research Director

# 돌아보기 ...



선형분류기

강아지 파라미터

고양이 파라미터

0.2	0.1	...	0.3	0.7
0.7	0.8	...	0.9	0.4

**W**

32x32x3 영상 =  
3,072 픽셀



$$2 \times 1 \quad f(x, W) = Wx + b \quad 2 \times 1$$

**3,072x1**

**2x3,072**

**X**

155
200
...
110
78

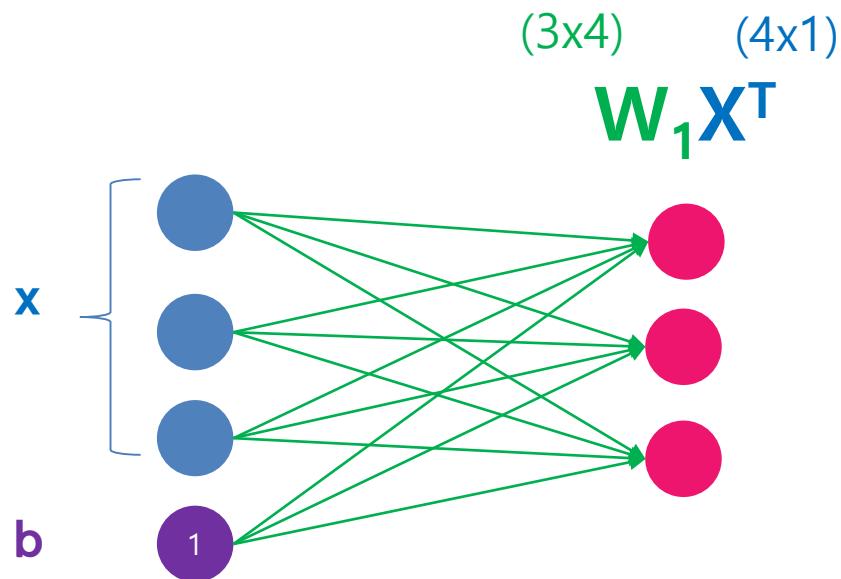
0.1
0.2

**b**

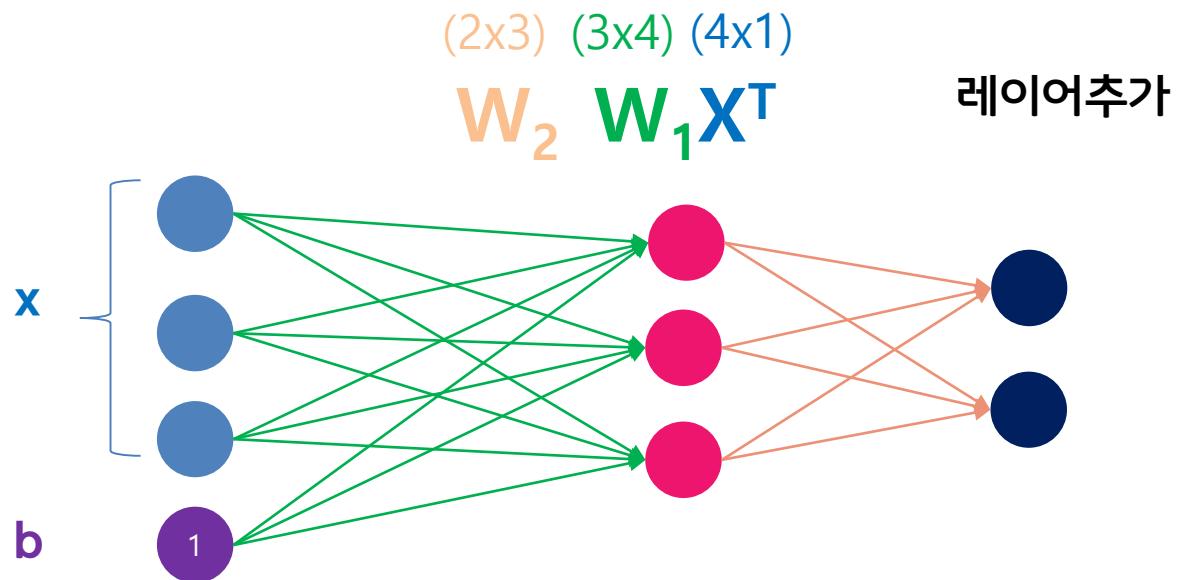
0.3
0.7

강아지 점수  
고양이 점수

# 돌아보기 ..



# 돌아보기 ..



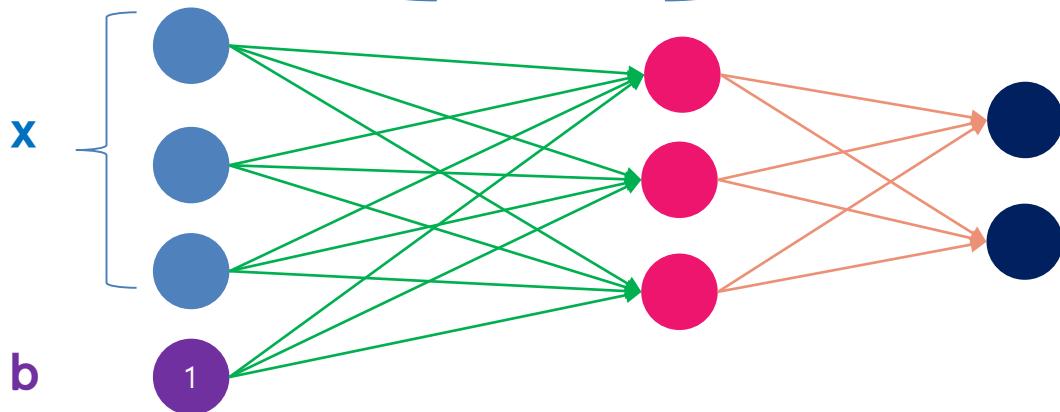
# 돌아보기 ..

핫 ~

$$(2 \times 3) \quad (3 \times 4) \quad (2 \times 4)$$

$$W_2 \quad W_1 = W$$

$$\begin{bmatrix} (2 \times 3) & (3 \times 4) \\ W_2 & W_1 \end{bmatrix} X^T \quad (4 \times 1)$$

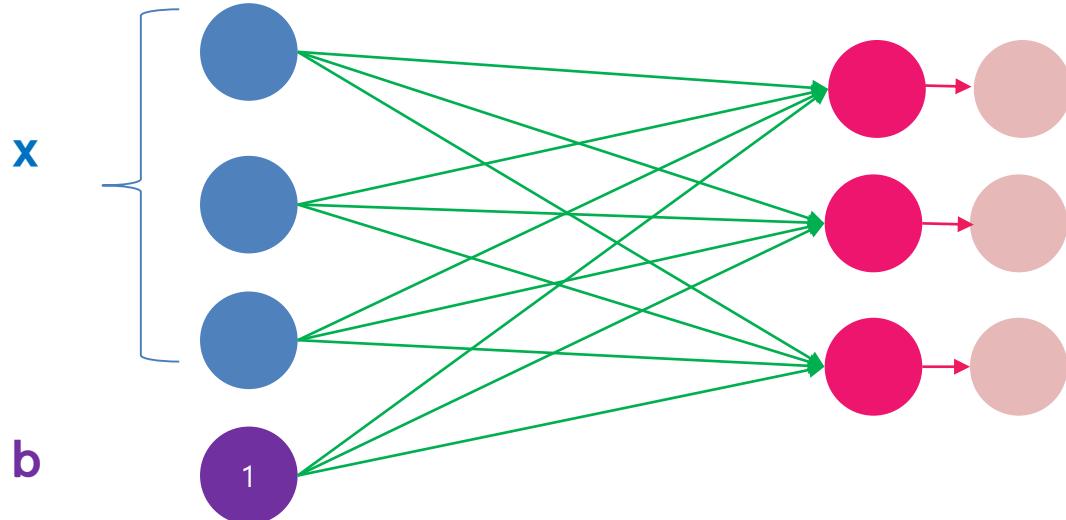


- 두개의 레이어가 하나로 표현
- 레이어를 쌓는 효과가 없어짐

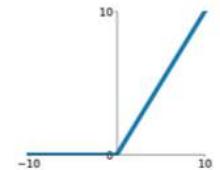
비선형 연산이 필요

# 돌아보기 ..

$$\max(0, W_1 X^T)$$



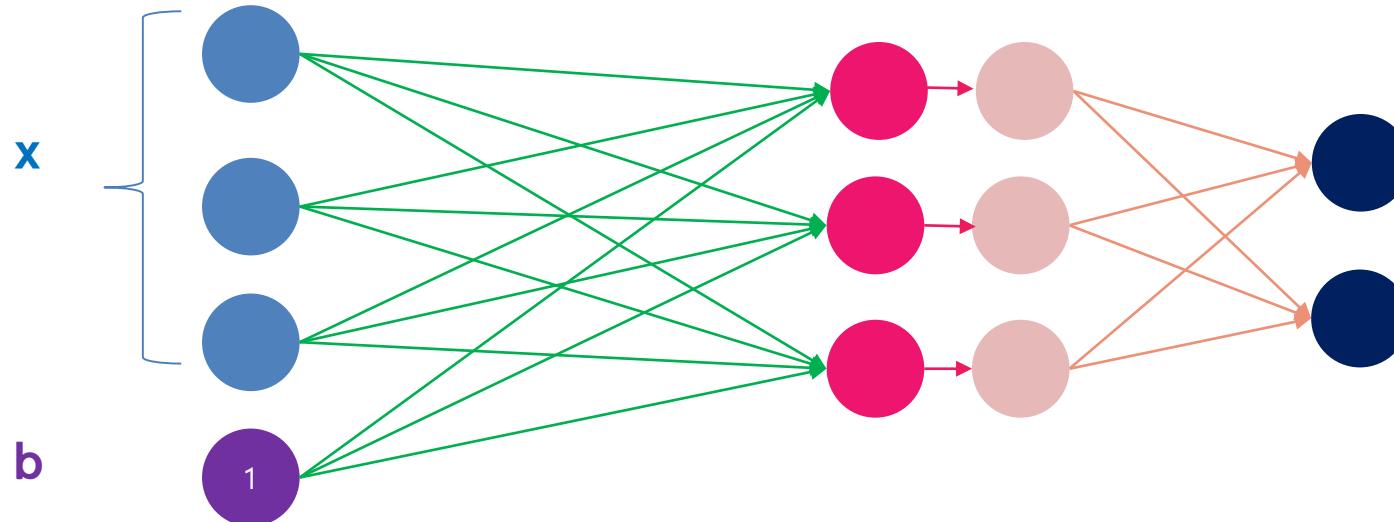
**ReLU**  
 $\max(0, x)$



# 돌아보기 ..

레이어 추가

$$W_2 \times \max(0, W_1 X^T)$$

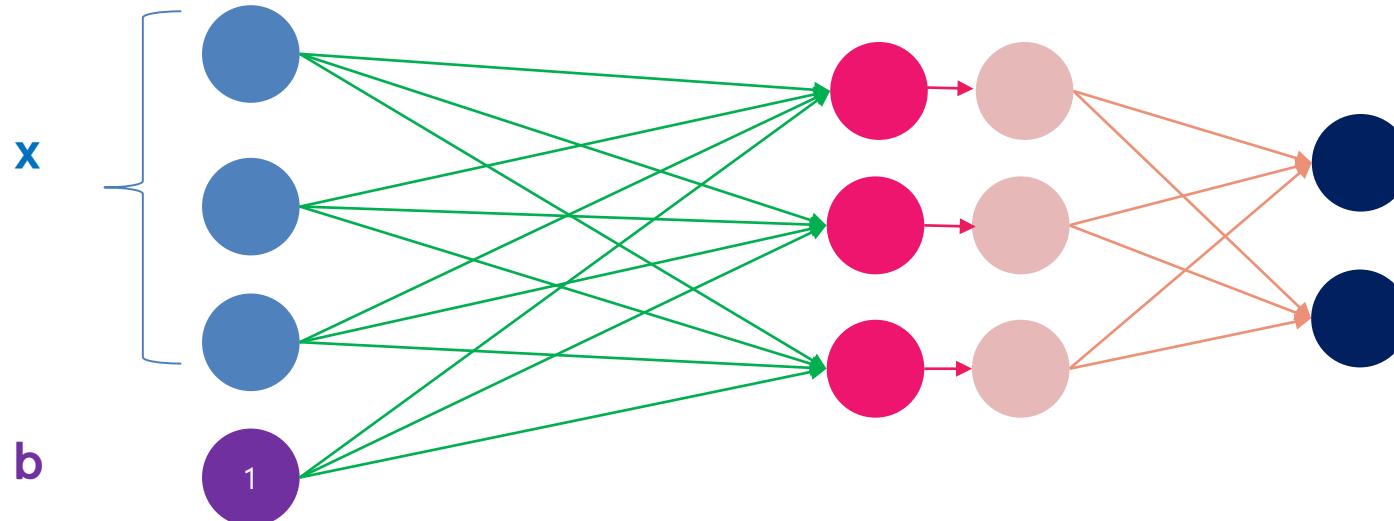


# 돌아보기 ..

## 1. Score function

레이어 추가

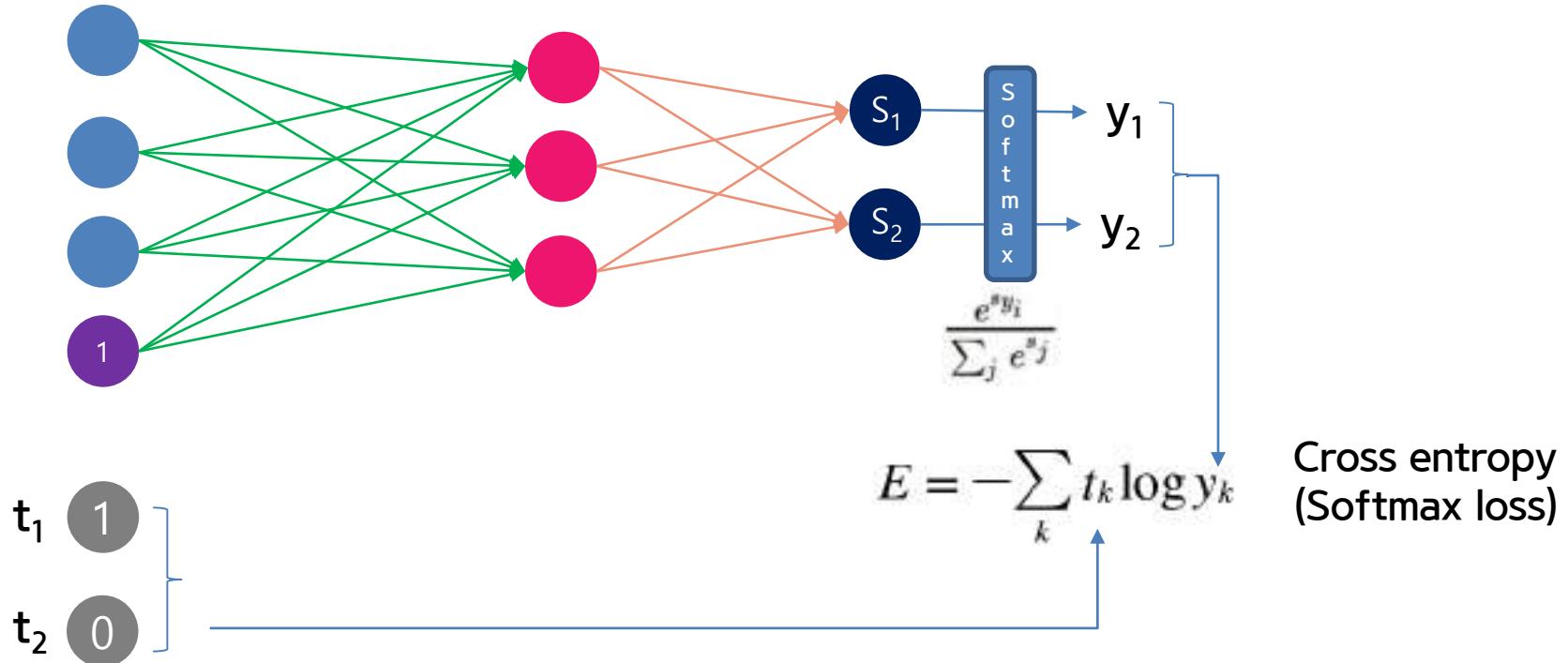
$$W_2 \times \max(0, W_1 X^T)$$



# 돌아보기 ..

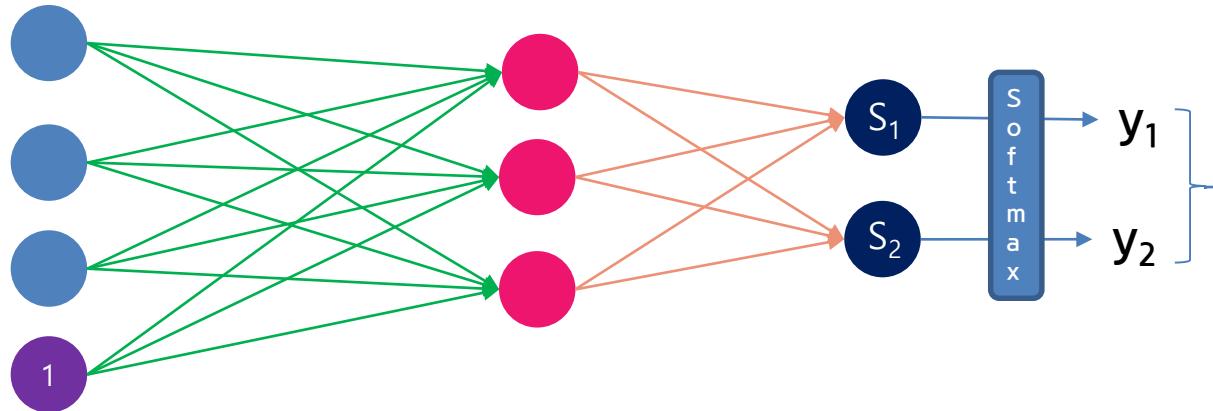
1. Score function
2. Loss function

## Loss function



# 돌아보기 ..

1. Score function
2. Loss function
3. Optimization



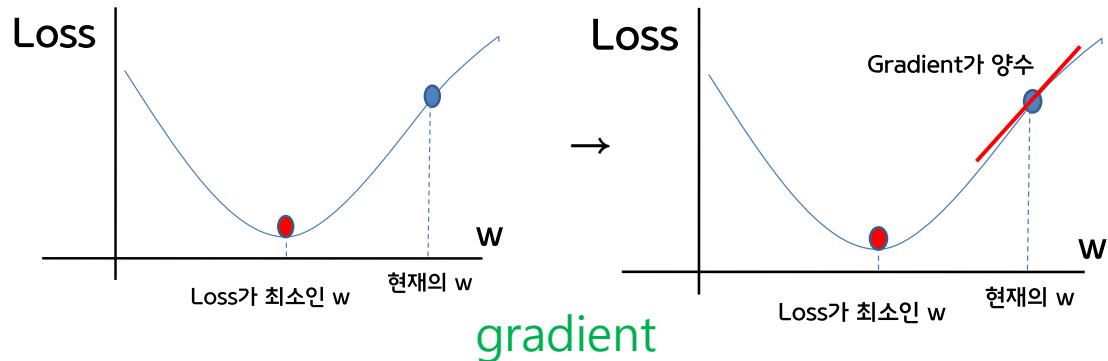
Loss를 줄이는 방법

Gradient Descent

$$Loss \leftarrow E = -\sum_k t_k \log y_k$$

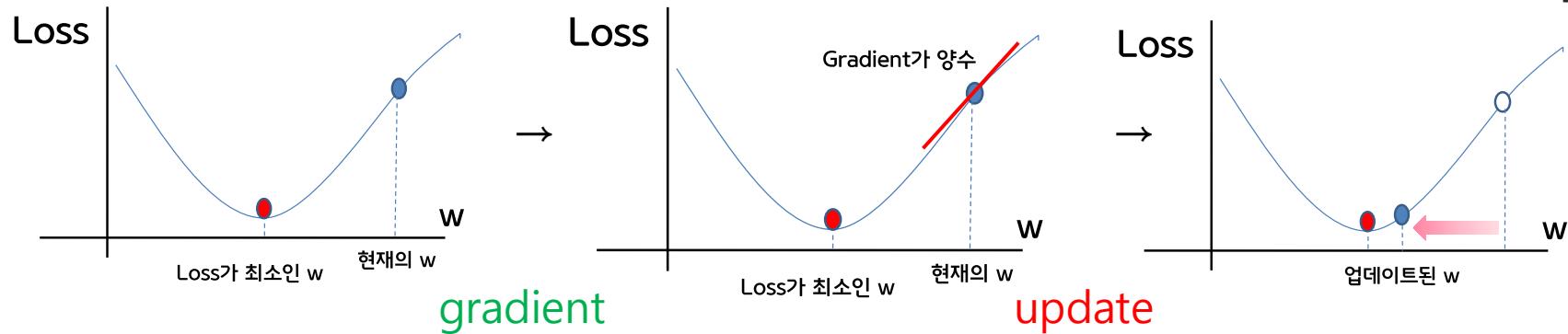
Cross entropy  
(Softmax loss)

# Gradient Descent



$$w = w - \eta \frac{\partial L}{\partial w}$$

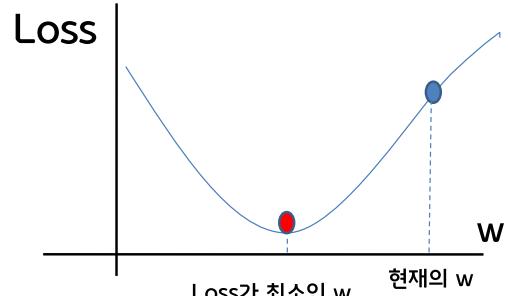
# Gradient Descent



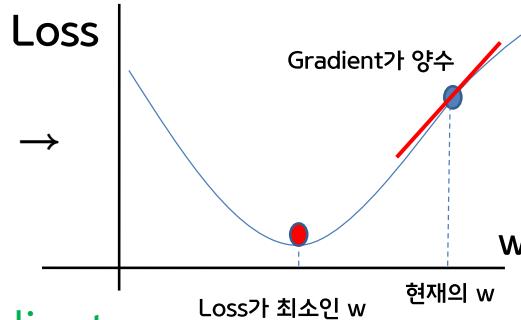
$$w = w - \eta \frac{\partial L}{\partial w}$$

# Gradient Descent :

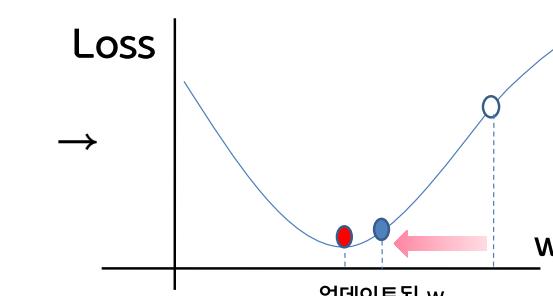
$$w = w - \eta \frac{\partial L}{\partial w}$$



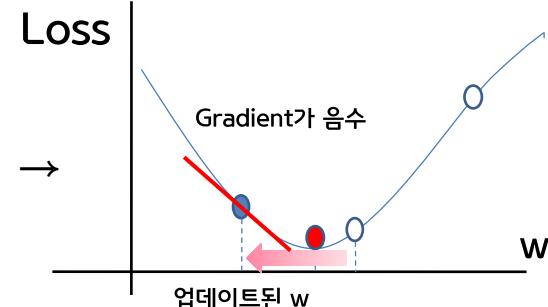
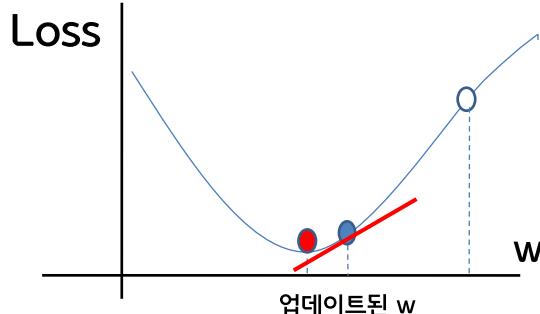
gradient



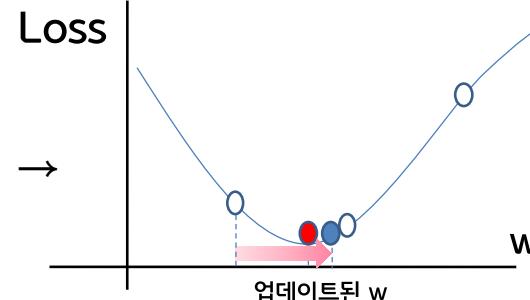
Loss가 최소인  $w$



update



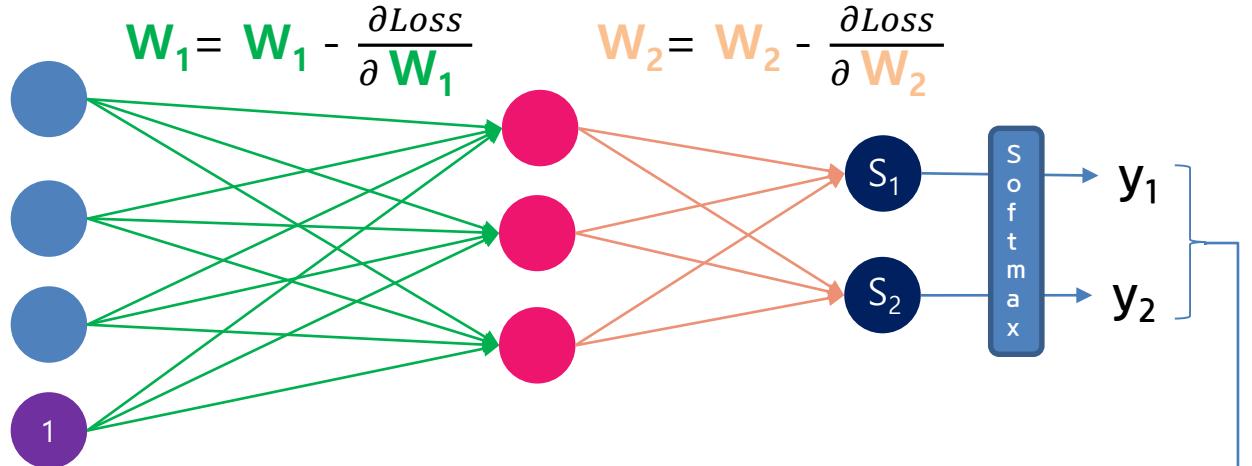
업데이트된  $w$



업데이트된  $w$

1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



Loss를 줄이는 방법

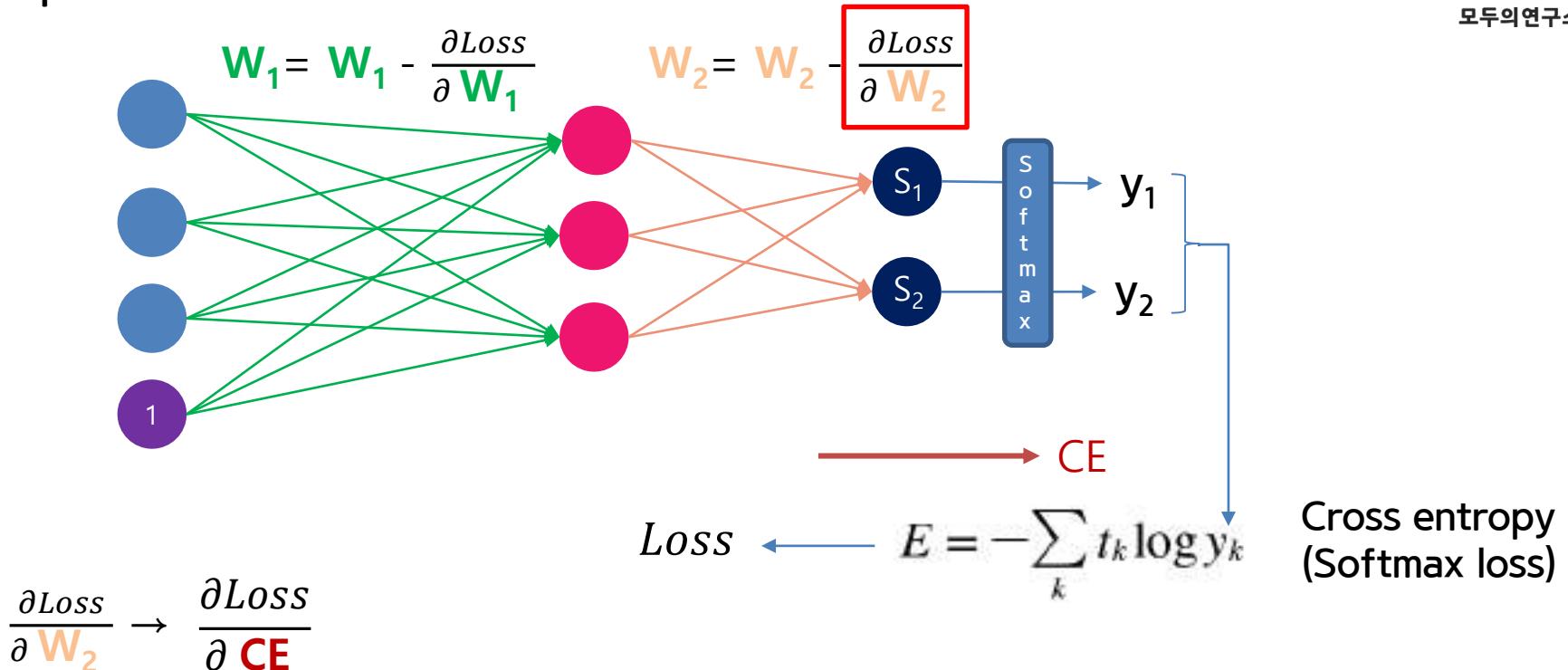
$$Loss \leftarrow E = -\sum_k t_k \log y_k$$

Cross entropy  
(Softmax loss)

Gradient 계산방법 : Backpropagation

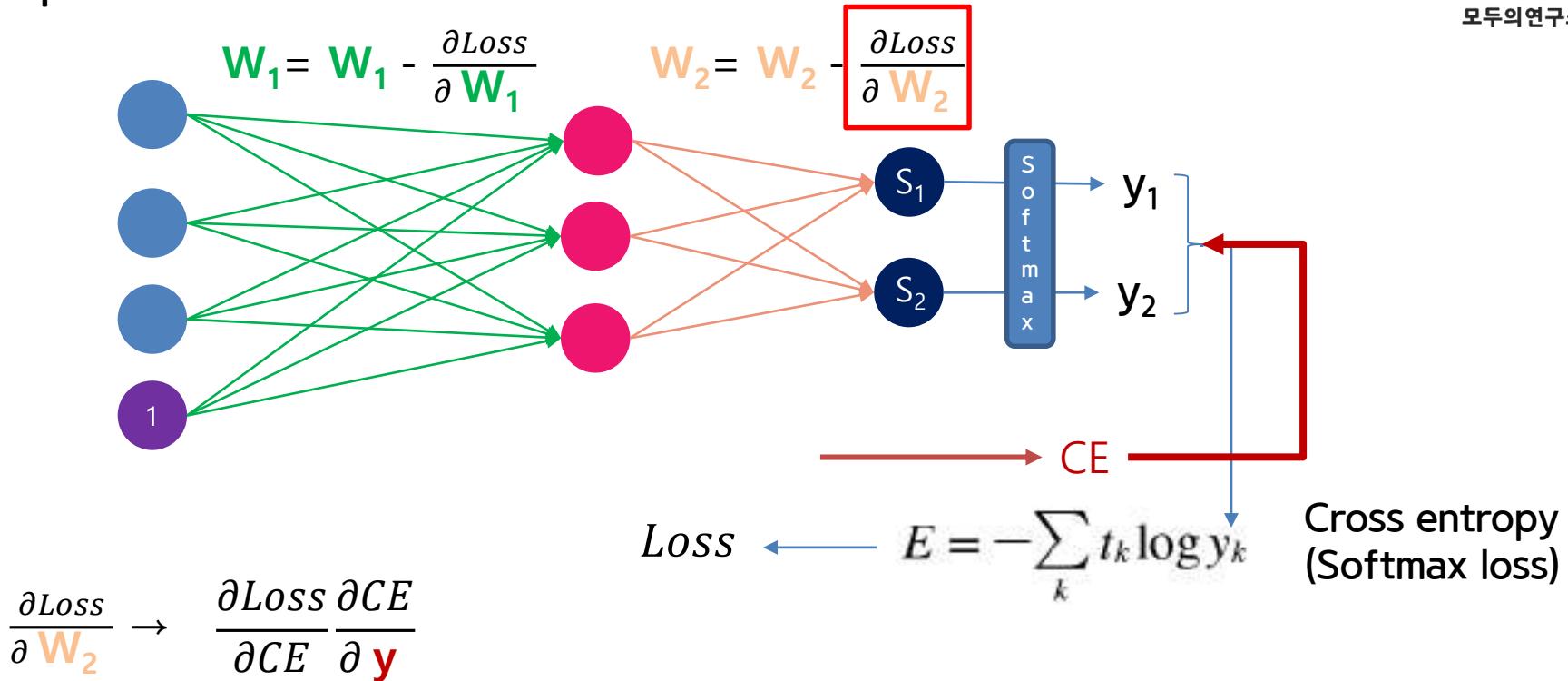
1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



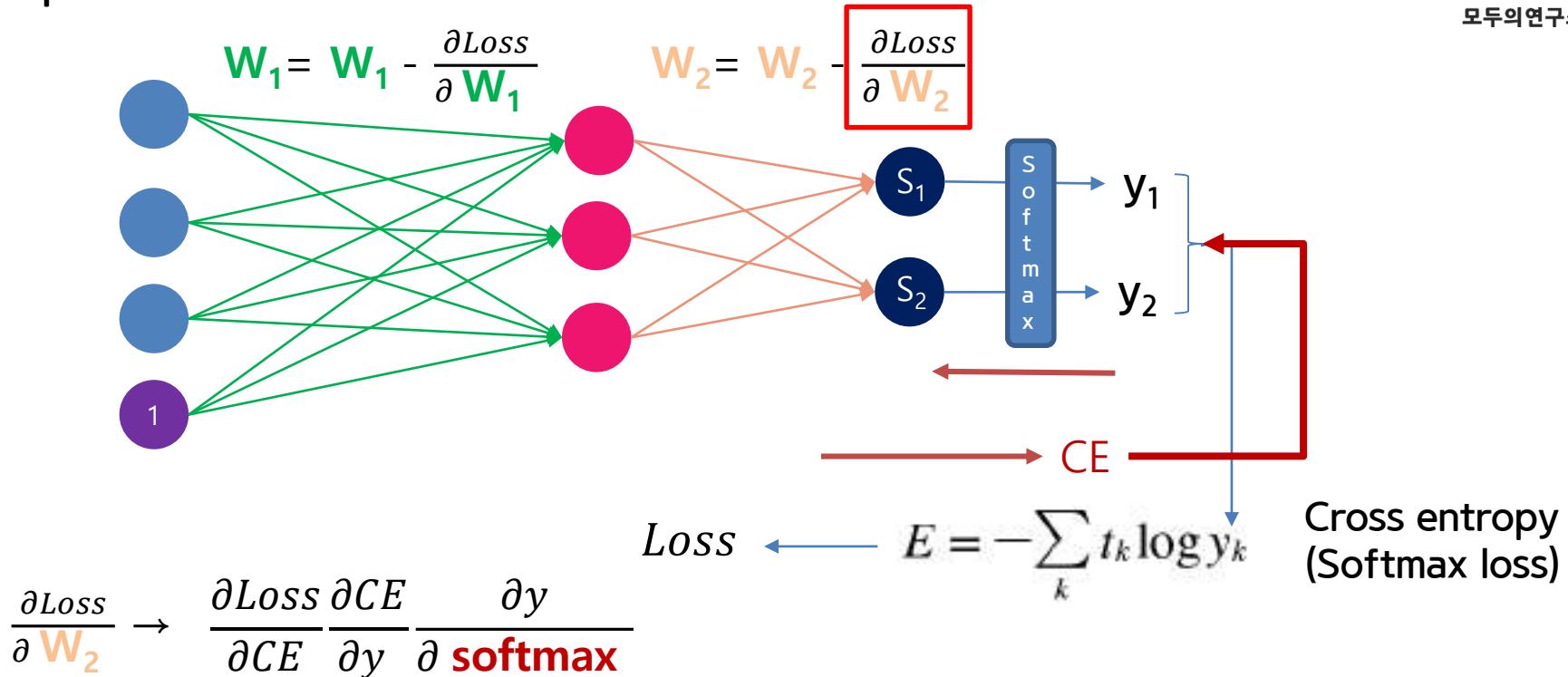
1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



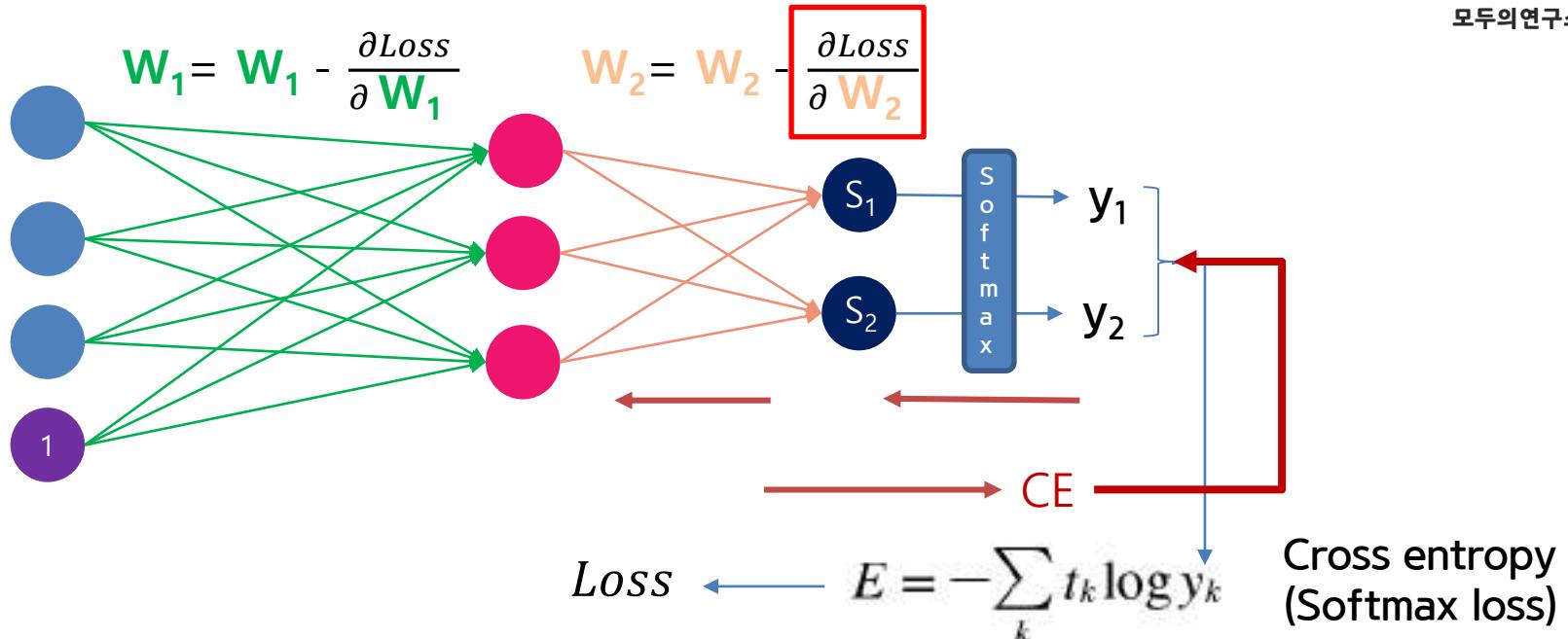
1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



1. Score function
2. Loss function
3. Optimization

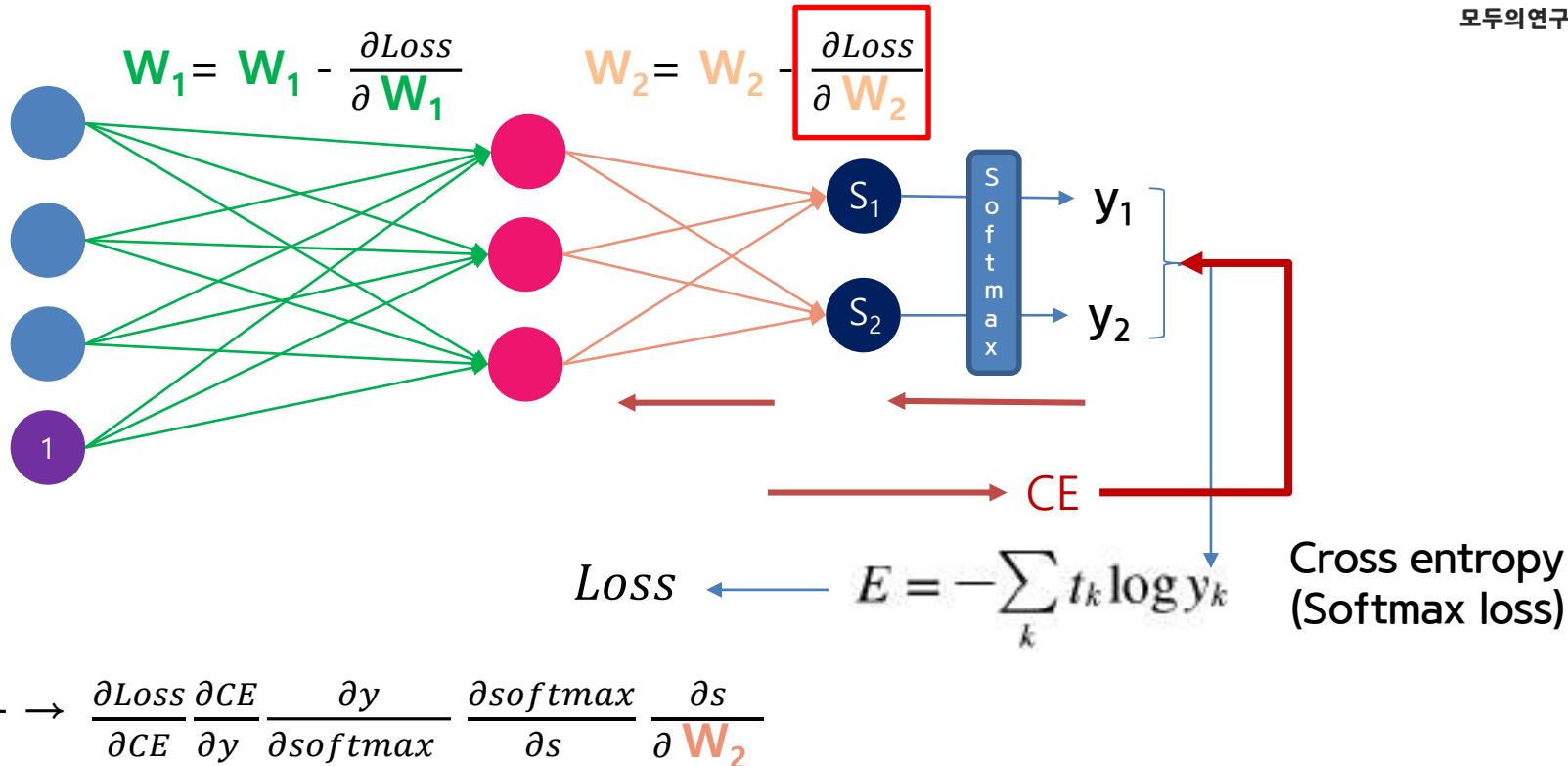
# 돌아보기 ..



$$\frac{\partial Loss}{\partial W_2} \rightarrow \frac{\partial Loss}{\partial CE} \frac{\partial CE}{\partial y} \frac{\partial y}{\partial softmax} \frac{\partial softmax}{\partial S}$$

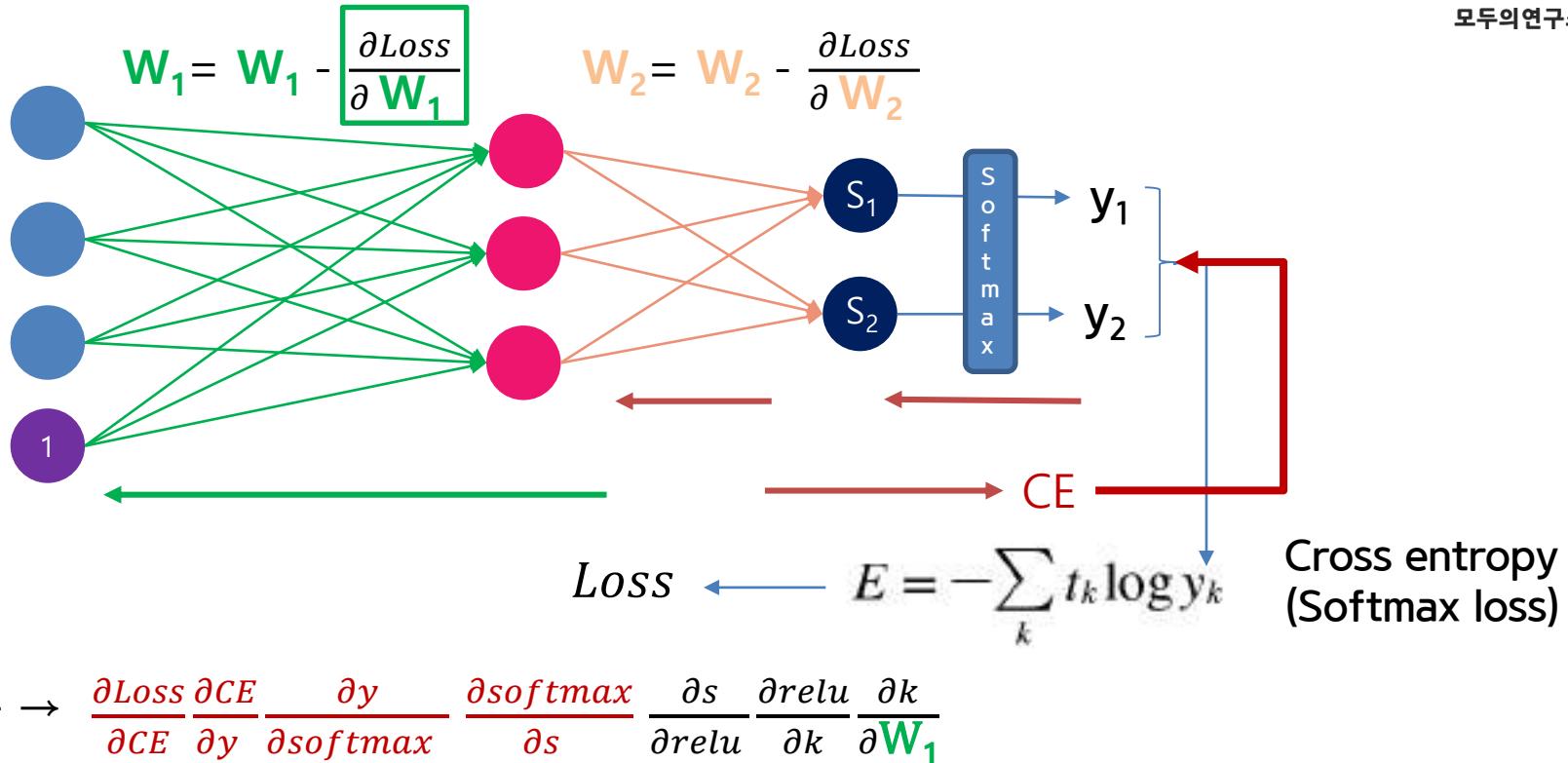
1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



1. Score function
2. Loss function
3. Optimization

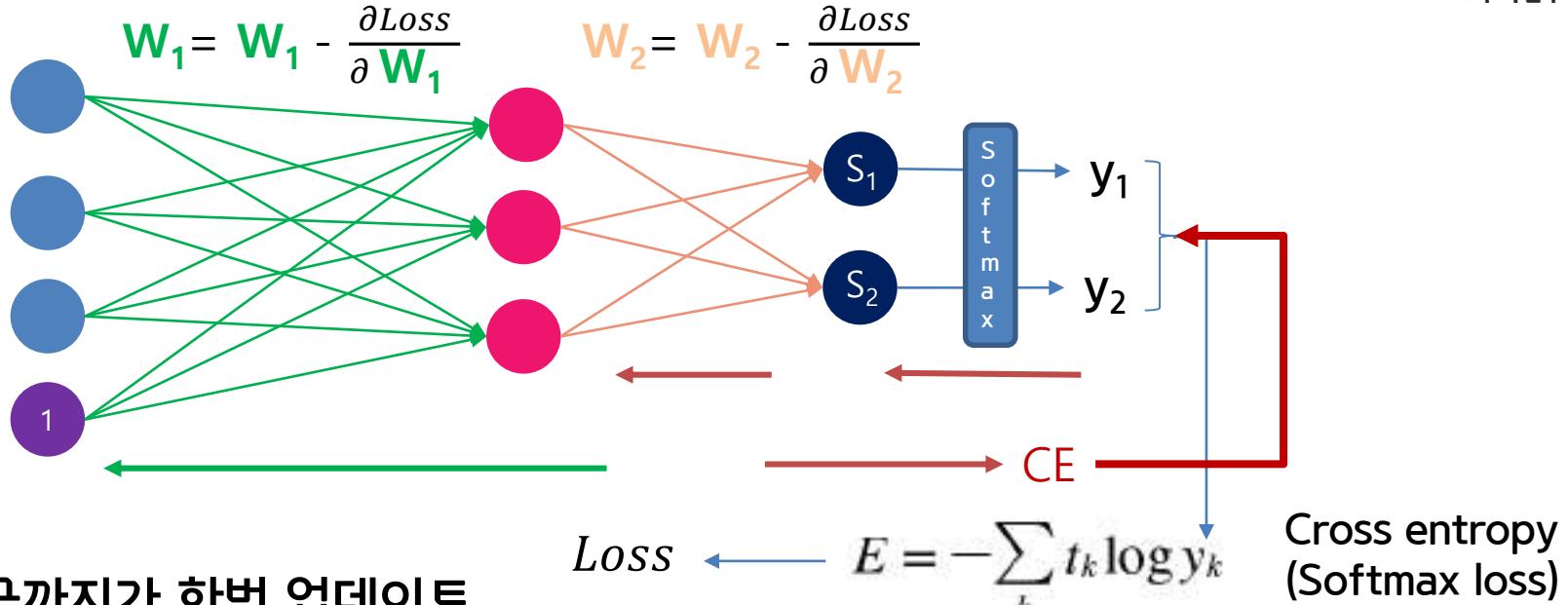
# 돌아보기 ..



$$\frac{\partial Loss}{\partial W_1} \rightarrow \frac{\partial Loss}{\partial CE} \frac{\partial CE}{\partial y} \frac{\partial y}{\partial softmax} \frac{\partial softmax}{\partial s} \frac{\partial s}{\partial relu} \frac{\partial relu}{\partial k} \frac{\partial k}{\partial W_1}$$

1. Score function
2. Loss function
3. Optimization

# 돌아보기 ..



- 지금까지가 한번 업데이트
- 모든 이미지에 대해서 반복

# Training Neural Networks

모두의연구소  
박은수 Research Director

# Gradient Descent



우리가 지금까지 Optimization에서 초점을 둔 부분

$$w = w - \eta \frac{\partial L}{\partial w}$$

Gradient 의 계산

# Gradient Descent



이제 부터는 Optimizer를 공부해 봅니다

$$w = w - \eta \frac{\partial L}{\partial w}$$

목적 : 그레디언트를 좀 더 효율적으로 변형한 뒤 업데이트

- Optimizer : 최적화를 행하는 자

# 매개변수 갱신

- 확률적 경사하강법 복습 (Stochastic Gradient Descent; SGD)

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

- /common/optimizer.py

```
1 # coding: utf-8
2 import numpy as np
3
4 class SGD:
5
6     """확률적 경사 하강법 (Stochastic Gradient Descent)"""
7
8     def __init__(self, lr=0.01):
9         self.lr = lr    Learning rate
10
11    def update(self, params, grads):
12        for key in params.keys():
13            params[key] -= self.lr * grads[key]
```

# 매개변수 갱신

- 기본 Optimizer : SGD
- common/optimizer.py

```
1 network = TwoLayerNet(...)
2 optimizer = SGD()
3
4 for i in range(10000):
5     ...
6     x_batch, t_batch = get_mini_batch(...) # 미니배치
7     grads = network.gradient(x_batch, t_batch)
8     params = network.params
9     optimizer.update(params, grads)
10    ...
```

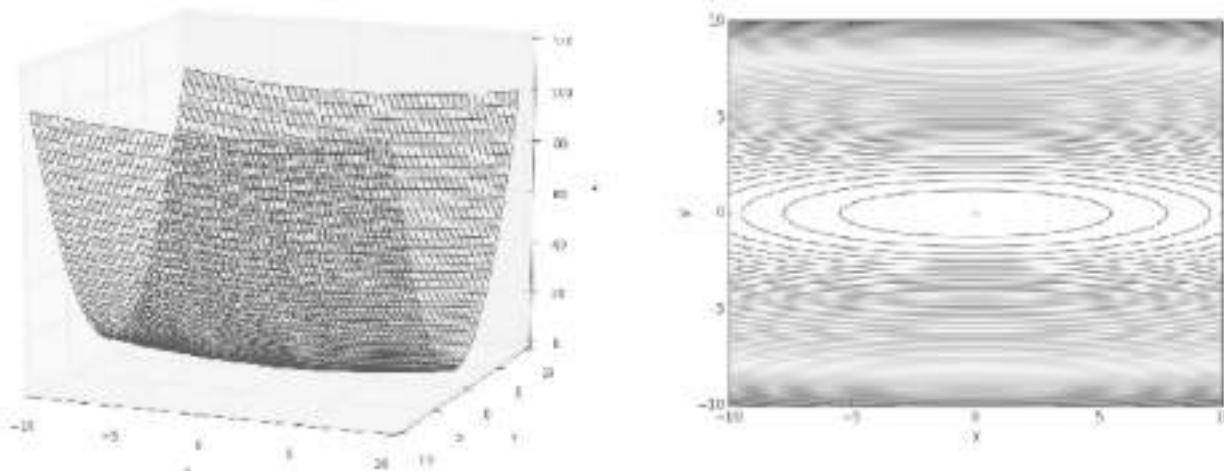
Gradient만을 변형할 것이기 때문에 Network와 별개로 존재할 수 있습니다.

\*대부분의 딥러닝 프레임워크는 이렇게 구성되어 있습니다

# 매개변수 갱신

- SGD의 단점

$$f(x,y) = \frac{1}{20}x^2 + y^2$$

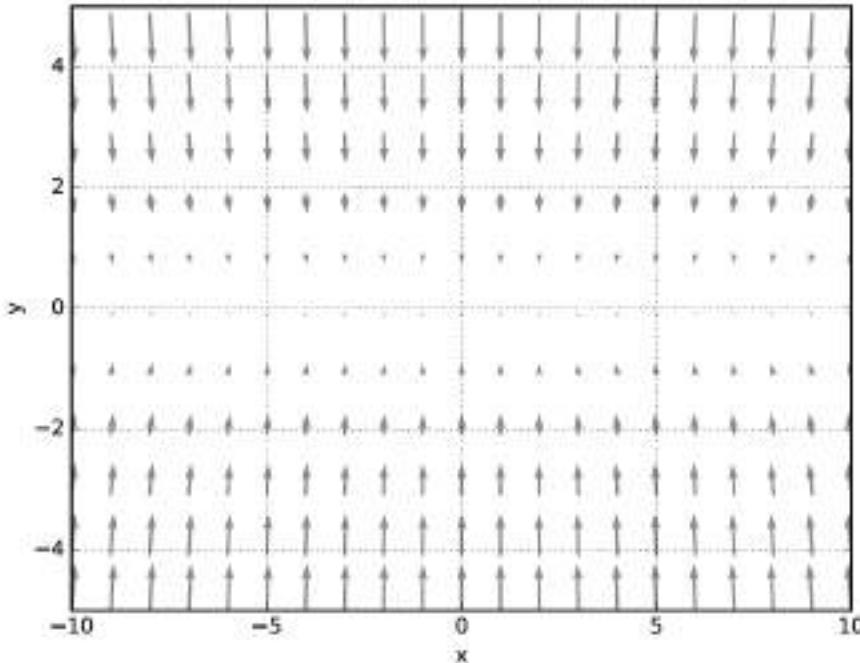


기울기를 그려보면

# 매개변수 갱신

- SGD의 단점

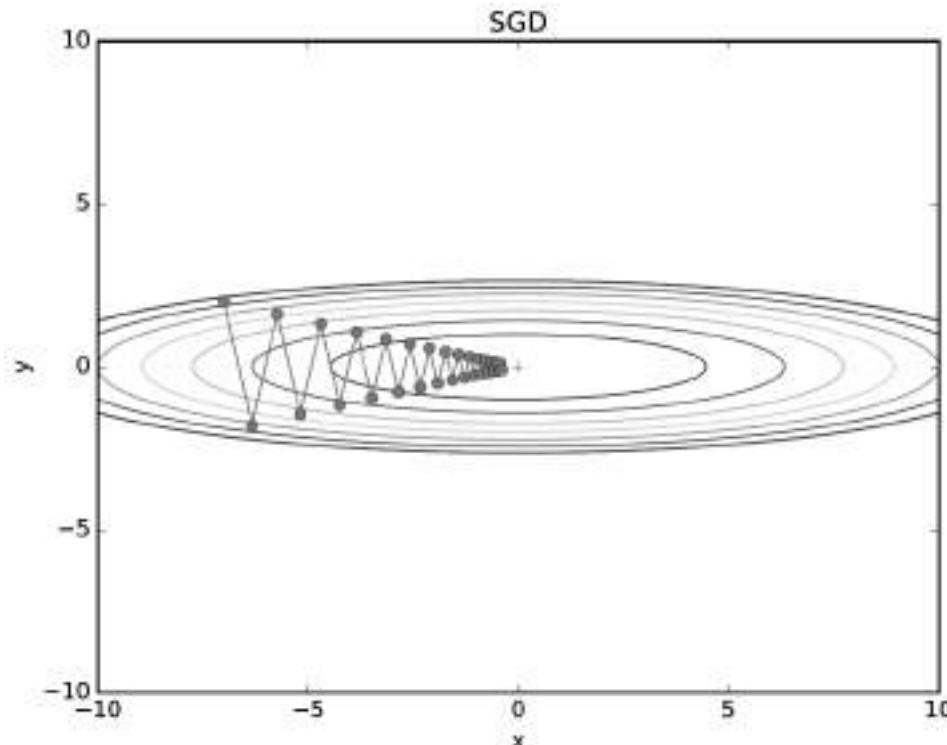
$$f(x, y) = \frac{1}{20}x^2 + y^2$$



- y축 방향은 크고 x축 방향은 작습니다
- 최소값은 (0,0)이지만 대부분 (0,0)을 가르키지 못합니다
- $(x, y) = (-7, 2)$ 에서 시작해서 SGD를 수행해 봅니다

# 매개변수 갱신

- SGD의 단점



- 탐색경로가 비효율적입니다
- y축 기울기는 크고 x축 기울기는 매우 작고
- 제대로 최솟값을 가르키지 못하고 있어서입니다

개선이  
필요합니다

# 매개변수 갱신

- Optimizer 1 : 모멘텀 (Momentum)
  - ‘운동량’을 뜻하는 단어로 물리와 관계가 있습니다

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

→ 기울기 방향으로 힘을 받아 물체가 가속된다는 물리 법칙을 나타냄

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

- $\mathbf{W}$  : 갱신 할 가중치
- $\frac{\partial L}{\partial \mathbf{W}}$  : 손실함수 기울기
- $\eta$  : 학습율
- $\mathbf{v}$  : 물리에서의 속도(velocity)

# 매개변수 갱신

- 모멘텀 (Momentum)

- ‘운동량’을 뜻하는 단어로 물리와 관계가 있습니다

물리에서의 지면 마찰이나 공기 저항에 해당합니다. 보통 0.9

$$v \leftarrow \alpha v - \eta \frac{\partial L}{\partial w}$$

기울기 방향으로 힘을 받아 물체가 가속 된다는 물리 법칙을 나타냄

$$W \leftarrow W + v$$

- $W$  : 갱신 할 가중치
- $\frac{\partial L}{\partial w}$  : 손실함수 기울기
- $\eta$  : 학습율
- $v$  : 물리에서의 속도(velocity)



# 매개변수 갱신

- 모멘텀 (Momentum)
  - 구현

```

16 class Momentum:
17     """모멘텀 SGD"""
18
19     def __init__(self, lr=0.01, momentum=0.9):
20         self.lr = lr
21         self.momentum = momentum
22         self.v = None
23
24     def update(self, params, grads):
25         if self.v is None:
26             self.v = {}
27             for key, val in params.items():
28                 self.v[key] = np.zeros_like(val) 초기값 0
29
30         for key in params.keys():
31             self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
32             params[key] += self.v[key]
33

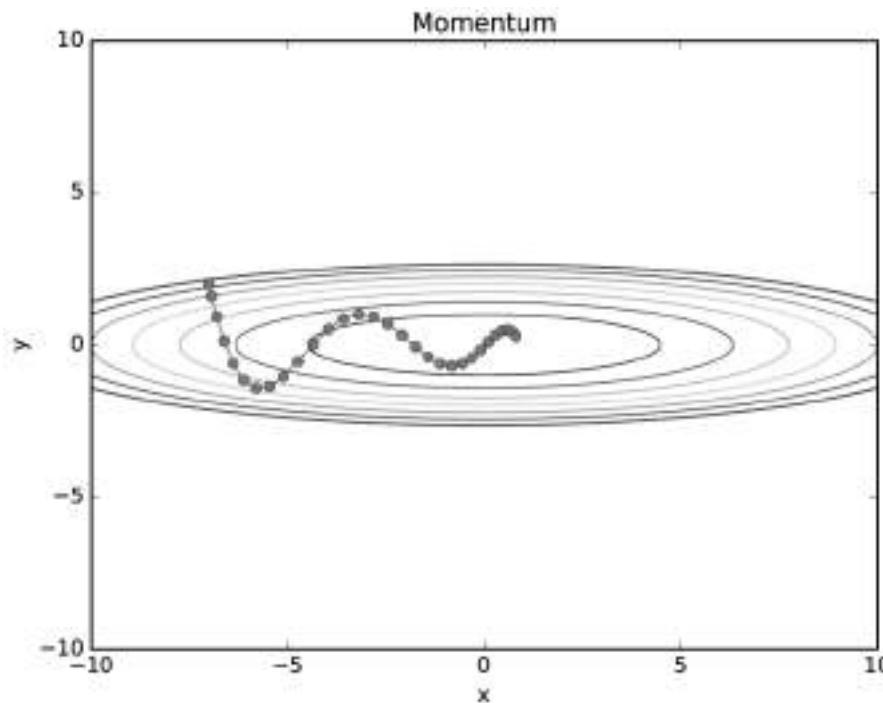
```

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

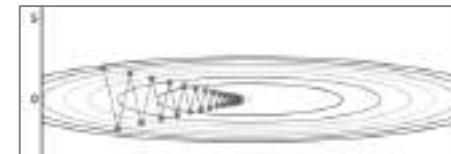
$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

# 매개변수 갱신

- 모멘텀 (Momentum)



- 공이 그릇 바닥으로 구르듯 움직입니다
- 전체적으로 지그재그가 SGD에 비해 덜합니다



# 매개변수 갱신

- 모멘텀 (Momentum)
  - 갑자기 물리 얘기 나오고 공 굴러가고 대체 뭔 소리에요?

# 매개변수 갱신

- 모멘텀 (Momentum)

- 갑자기 물리 얘기 나오고 공 굴러가고 대체 뭔 소리에요?

- $v$ 의 변화를 살펴 봅시다
  - $v_0$ 는 0부터 시작

$$v \leftarrow \alpha v - \eta \frac{\partial L}{\partial W}$$

$$W \leftarrow W + v$$

관측 K\_i로 변경  
(그레이디언트임)

업데이트 1)  $v_1 \leftarrow \alpha * 0 - K_o : -K_o$

업데이트 2)  $v_2 \leftarrow \alpha v_1 - K_1 : -\alpha K_o - K_1$

업데이트 3)  $v_3 \leftarrow \alpha v_2 - K_2 : -\alpha^2 K_o - \alpha K_1 - K_2$

업데이트 4)  $v_4 \leftarrow \alpha v_3 - K_3 : -\alpha^3 K_o - \alpha^2 K_1 - \alpha K_2 - K_3$

⋮

- 1)  $W \leftarrow W + v_1$
- 2)  $W \leftarrow W + v_2$
- 3)  $W \leftarrow W + v_3$
- 4)  $W \leftarrow W + v_4$

# 매개변수 갱신

- 모멘텀 (Momentum)

$$\begin{aligned} \mathbf{v} &\leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \\ \mathbf{W} &\leftarrow \mathbf{W} + \mathbf{v} \end{aligned}$$

관측 K\_i로 변경  
(그레이디언트임)

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_o : -K_o$

업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_o - K_1$

업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_o - \alpha K_1 - K_2$

업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_o - \alpha^2 K_1 - \alpha K_2 - K_3$

⋮

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

일반적인 경우처럼  $\alpha = 0.9$ 로 하면  $\alpha^2 = 0.81$ ,  $\alpha^3 = 0.729$



기존의 업데이트 값을 계속 반영하지만 점점 지수적으로 줄입니다

물리에서의 지면 마찰이  
나 공기 저항에 해당합니  
다. 보통 0.9

$$\begin{aligned} \mathbf{v} &\leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \\ \mathbf{W} &\leftarrow \mathbf{W} + \mathbf{v} \end{aligned}$$

기울기 방향으로 힘  
을 받아 물체가 가  
속된다는 물리 법칙  
을 나타냄

# 매개변수 갱신

- 모멘텀 (Momentum)

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

관측  
K<sub>i</sub>로 변경  
(그레이디언트임)

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_o : -K_o$

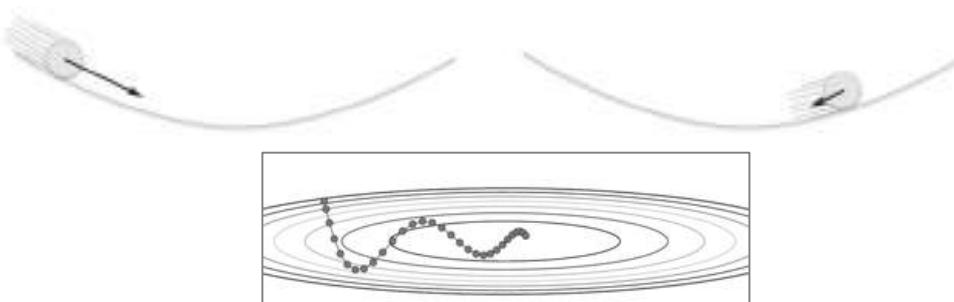
업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_o - K_1$

업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_o - \alpha K_1 - K_2$

업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_o - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

- 지난 업데이트를 기억하고 이를 업데이트에 반영함과 동시에 과거의 값들의 영향력은 줄여 나갑니다
- 업데이트 부호가 바뀌면 그 영향력이 줄어 들게 됩니다



# 매개변수 갱신

- AdaGrad
  - AdaGrad는 ‘각각의’ 매개변수에 맞게 ‘맞춤형’으로 매개변수를 갱신합니다
    - 적응적(adaptive)으로 학습률을 조절하면서 학습을 진행

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

← 기존 기울기값을 제곱하여  
계속 더함

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

- $\odot$  : 행렬의 원소별 곱셈을 의미함

# 매개변수 갱신

- AdaGrad
  - AdaGrad는 ‘각각의’ 매개변수에 맞게 ‘맞춤형’으로 매개변수를 갱신합니다

```
59 class AdaGrad:  
60     """AdaGrad"""  
61  
62     def __init__(self, lr=0.01):  
63         self.lr = lr  
64         self.h = None  
65  
66     def update(self, params, grads):  
67         if self.h is None:  
68             self.h = {}  
69             for key, val in params.items():  
70                 self.h[key] = np.zeros_like(val)  
71  
72         for key in params.keys():  
73             self.h[key] += grads[key] * grads[key]  
74             params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

# 매개변수 갱신

- Adagrad

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \boxed{\frac{\partial L}{\partial \mathbf{W}}} \rightarrow K_i \text{로 변경 (Gradient임)}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \boxed{\frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}} \quad \text{업데이트를 관측 해봅시다}$$

업데이트 1)  $\frac{1}{\sqrt{K_0^2}} K_0$

- 일변수가 아니라 실제로는 벡터로 작용해서 각 매개 변수마다 업데이트 되는 크기가 다름을 잊지 마세요

업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_0^2}} K_1$

- 업데이트를 어느 정도 안정된 값으로 하는 정규화 속성이 존재합니다

업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_0^2}} K_3$

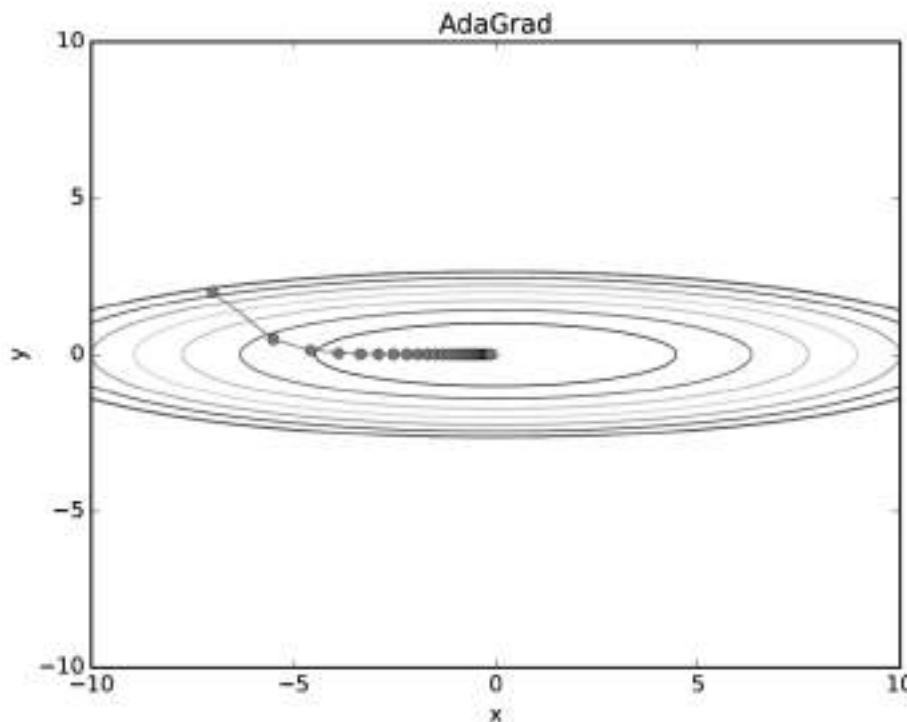
- 시간이 지날 수록 업데이트 값이 계속 줄어들어 나중에는 업데이트가 발생하지 않게 됩니다

업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_0^2}} K_4$

- 해결책 -> RMSprop**

# 매개변수 갱신

- Adagrad



- 지그재그가 줄어들고 어느 정도 안정적으로 최솟값을 향해 갑니다

# 매개변수 갱신

- RMSProp

## RMSProp update

[Tieleman and Hinton, 2012]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

# 매개변수 갱신

- RMSProp

## rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.

- The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?

- rmsprop: Keep a moving average of the squared gradient for each weight

$$\text{MeanSquare}(w, t) = 0.9 \text{ MeanSquare}(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w}(t) \right)^2$$

- Dividing the gradient by  $\sqrt{\text{MeanSquare}(w, t)}$  makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in  
Geoff Hinton's Coursera  
class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \boxed{\frac{\partial L}{\partial \mathbf{W}}} \right) \rightarrow \mathbf{K}_i \text{로 변경}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\mathbf{h}_1 = (1 - a) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha \mathbf{h}_1 + (1 - a) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha \mathbf{h}_2 + (1 - a) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right) \rightarrow \mathbf{K}_i \text{로 변경}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$  가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha \mathbf{h}_1 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha \mathbf{h}_2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha \mathbf{h}_1 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha \mathbf{h}_2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

기존 그레디언트 값들의 누적하면서 영향력을 지속적으로 줄임

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$\mathbf{K}_i$ 로 변경

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha(1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha \mathbf{h}_2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \boxed{\frac{\partial L}{\partial \mathbf{W}}} \right) \rightarrow \mathbf{K}_i \text{로 변경}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha(1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha \mathbf{h}_2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$\mathbf{K}_i$ 로 변경

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha (1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha (\alpha (1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2) + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \boxed{\frac{\partial L}{\partial \mathbf{W}}} \right) \rightarrow \mathbf{K}_i \text{로 변경}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha(1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha^2(1 - \alpha) \mathbf{K}_1^2 + \alpha(1 - \alpha) \mathbf{K}_2^2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha \mathbf{h}_3 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$\mathbf{K}_i$ 로 변경

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha(1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha^2(1 - \alpha) \mathbf{K}_1^2 + \alpha(1 - \alpha) \mathbf{K}_2^2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha(\alpha^2(1 - \alpha) \mathbf{K}_1^2 + \alpha(1 - \alpha) \mathbf{K}_2^2 + (1 - \alpha) \mathbf{K}_3^2) + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- RMSProp

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

$\alpha$ 가 1보다 작다면(보통 0.9) : 모멘텀과 같이 기존값들을 반영하면서 그 영향력을 지수적으로 줄인다

업데이트 1)  $\mathbf{h}_1 = (1 - \alpha) \mathbf{K}_1^2$

업데이트 2)  $\mathbf{h}_2 = \alpha(1 - \alpha) \mathbf{K}_1^2 + (1 - \alpha) \mathbf{K}_2^2$

업데이트 3)  $\mathbf{h}_3 = \alpha^2(1 - \alpha) \mathbf{K}_1^2 + \alpha(1 - \alpha) \mathbf{K}_2^2 + (1 - \alpha) \mathbf{K}_3^2$

업데이트 4)  $\mathbf{h}_4 = \alpha^3(1 - \alpha) \mathbf{K}_1^2 + \alpha^2(1 - \alpha) \mathbf{K}_2^2 + \alpha(1 - \alpha) \mathbf{K}_3^2 + (1 - \alpha) \mathbf{K}_4^2$

# 매개변수 갱신

- Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

업데이트 1)  $v_1 \leftarrow \alpha * 0 - K_o : -K_o$   
 업데이트 2)  $v_2 \leftarrow \alpha v_1 - K_1 : -\alpha K_o - K_1$   
 업데이트 3)  $v_3 \leftarrow \alpha v_2 - K_2 : -\alpha^2 K_o - \alpha K_1 - K_2$   
 업데이트 4)  $v_4 \leftarrow \alpha v_3 - K_3 : -\alpha^3 K_o - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + v_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + v_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + v_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + v_4$

- Adagrad

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\frac{1}{\sqrt{K_o^2}} K_o$   
 업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_o^2}} K_1$   
 업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_o^2}} K_3$   
 업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_o^2}} K_4$

- RMSprop

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) (\frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}})$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $h_1 = (1 - a) \mathbf{K}_1^2$   
 업데이트 2)  $h_2 = a(1 - a) \mathbf{K}_1^2 + (1 - a) \mathbf{K}_2^2$   
 업데이트 3)  $h_3 = a^2(1 - a) \mathbf{K}_1^2 + a(1 - a) \mathbf{K}_2^2 + (1 - a) \mathbf{K}_3^2$   
 업데이트 4)  $h_4 = a^3(1 - a) \mathbf{K}_1^2 + a^2(1 - a) \mathbf{K}_2^2 + a(1 - a) \mathbf{K}_3^2 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

## Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_0 : -K_0$   
 업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_0 - K_1$   
 업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_0 - \alpha K_1 - K_2$   
 업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_0 - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

## Adagrad

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\frac{1}{\sqrt{K_0^2}} K_0$   
 업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_0^2}} K_1$   
 업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_0^2}} K_3$   
 업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_0^2}} K_4$

## RMSprop

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) (\frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}})$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\mathbf{h}_1 = (1 - a) \mathbf{K}_1^2$   
 업데이트 2)  $\mathbf{h}_2 = a(1 - a) \mathbf{K}_1^2 + (1 - a) \mathbf{K}_2^2$   
 업데이트 3)  $\mathbf{h}_3 = a^2(1 - a) \mathbf{K}_1^2 + a(1 - a) \mathbf{K}_2^2 + (1 - a) \mathbf{K}_3^2$   
 업데이트 4)  $\mathbf{h}_4 = a^3(1 - a) \mathbf{K}_1^2 + a^2(1 - a) \mathbf{K}_2^2 + a(1 - a) \mathbf{K}_3^2 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

- Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_0 : -K_0$   
 업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_0 - K_1$   
 업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_0 - \alpha K_1 - K_2$   
 업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_0 - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

- Adagrad

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\frac{1}{\sqrt{K_0^2}} K_0$   
 업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_0^2}} K_1$   
 업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_0^2}} K_3$   
 업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_0^2}} K_4$

Gradient  
Normalization  
스러운 방법

- RMSprop

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\mathbf{h}_1 = (1 - a) \mathbf{K}_1^2$   
 업데이트 2)  $\mathbf{h}_2 = a(1 - a) \mathbf{K}_1^2 + (1 - a) \mathbf{K}_2^2$   
 업데이트 3)  $\mathbf{h}_3 = a^2(1 - a) \mathbf{K}_1^2 + a(1 - a) \mathbf{K}_2^2 + (1 - a) \mathbf{K}_3^2$   
 업데이트 4)  $\mathbf{h}_4 = a^3(1 - a) \mathbf{K}_1^2 + a^2(1 - a) \mathbf{K}_2^2 + a(1 - a) \mathbf{K}_3^2 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

## Momentum

$$\begin{aligned} \mathbf{v} &\leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \\ \mathbf{W} &\leftarrow \mathbf{W} + \mathbf{v} \end{aligned}$$

## Gradient 이동누적 스러운 방법

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_0 : -K_0$   
 업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_0 - K_1$   
 업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_0 - \alpha K_1 - K_2$   
 업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_0 - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

## Adagrad

$$\begin{aligned} \mathbf{h} &\leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \\ \mathbf{W} &\leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \end{aligned}$$

업데이트 1)  $\frac{1}{\sqrt{K_0^2}} K_0$   
 업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_0^2}} K_1$   
 업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_0^2}} K_3$   
 업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_0^2}} K_4$

## Gradient Normalization 스러운 방법

## RMSprop

$$\begin{aligned} \mathbf{h} &\leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right) \\ \mathbf{W} &\leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \end{aligned}$$

업데이트 1)  $\mathbf{h}_1 = (1 - a) \mathbf{K}_1^2$   
 업데이트 2)  $\mathbf{h}_2 = a(1 - a) \mathbf{K}_1^2 + (1 - a) \mathbf{K}_2^2$   
 업데이트 3)  $\mathbf{h}_3 = a^2(1 - a) \mathbf{K}_1^2 + a(1 - a) \mathbf{K}_2^2 + (1 - a) \mathbf{K}_3^2$   
 업데이트 4)  $\mathbf{h}_4 = a^3(1 - a) \mathbf{K}_1^2 + a^2(1 - a) \mathbf{K}_2^2 + a(1 - a) \mathbf{K}_3^2 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

- Momentum

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

업데이트 1)  $\mathbf{v}_1 \leftarrow \alpha * 0 - K_0 : -K_0$   
 업데이트 2)  $\mathbf{v}_2 \leftarrow \alpha \mathbf{v}_1 - K_1 : -\alpha K_0 - K_1$   
 업데이트 3)  $\mathbf{v}_3 \leftarrow \alpha \mathbf{v}_2 - K_2 : -\alpha^2 K_0 - \alpha K_1 - K_2$   
 업데이트 4)  $\mathbf{v}_4 \leftarrow \alpha \mathbf{v}_3 - K_3 : -\alpha^3 K_0 - \alpha^2 K_1 - \alpha K_2 - K_3$

- 1)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_1$
- 2)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_2$
- 3)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_3$
- 4)  $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}_4$

- Adagrad

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\frac{1}{\sqrt{K_0^2}} K_0$   
 업데이트 2)  $\frac{1}{\sqrt{K_1^2 + K_0^2}} K_1$   
 업데이트 3)  $\frac{1}{\sqrt{K_3^2 + K_1^2 + K_0^2}} K_3$   
 업데이트 4)  $\frac{1}{\sqrt{K_4^2 + K_3^2 + K_1^2 + K_0^2}} K_4$

두 방법의 같이쓰자  
Adam

- RMSprop

$$\mathbf{h} \leftarrow \alpha \mathbf{h} + (1 - \alpha) \left( \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \right)$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}}$$

업데이트 1)  $\mathbf{h}_1 = (1 - a) \mathbf{K}_1^2$   
 업데이트 2)  $\mathbf{h}_2 = a(1 - a) \mathbf{K}_1^2 + (1 - a) \mathbf{K}_2^2$   
 업데이트 3)  $\mathbf{h}_3 = a^2(1 - a) \mathbf{K}_1^2 + a(1 - a) \mathbf{K}_2^2 + (1 - a) \mathbf{K}_3^2$   
 업데이트 4)  $\mathbf{h}_4 = a^3(1 - a) \mathbf{K}_1^2 + a^2(1 - a) \mathbf{K}_2^2 + a(1 - a) \mathbf{K}_3^2 + (1 - a) \mathbf{K}_4^2$

# 매개변수 갱신

- Adam
  - RMSProp + 모멘텀

```

99  class Adam:
100
101     """Adam (http://arxiv.org/abs/1412.6980v8)"""
102
103     def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
104         self.lr = lr
105         self.beta1 = beta1
106         self.beta2 = beta2
107         self.iter = 0
108         self.m = None
109         self.v = None
110
111     def update(self, params, grads):
112         if self.m is None:
113             self.m, self.v = {}, {}
114             for key, val in params.items():
115                 self.m[key] = np.zeros_like(val)
116                 self.v[key] = np.zeros_like(val)
117
118             self.iter += 1
119             lr_t = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)
120
121             for key in params.keys():
122                 #self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
123                 #self.v[key] = self.beta2*self.v[key] + (1-self.beta2)*(grads[key]**2)
124                 self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
125                 self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

```

# 매개변수 갱신

- Adam
  - RMSProp + 모멘텀

## Adam update

(incomplete, but close)

[Kingma and Ba, 2014]

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning rate * m / (np.sqrt(v) + 1e-7)
```

momentum

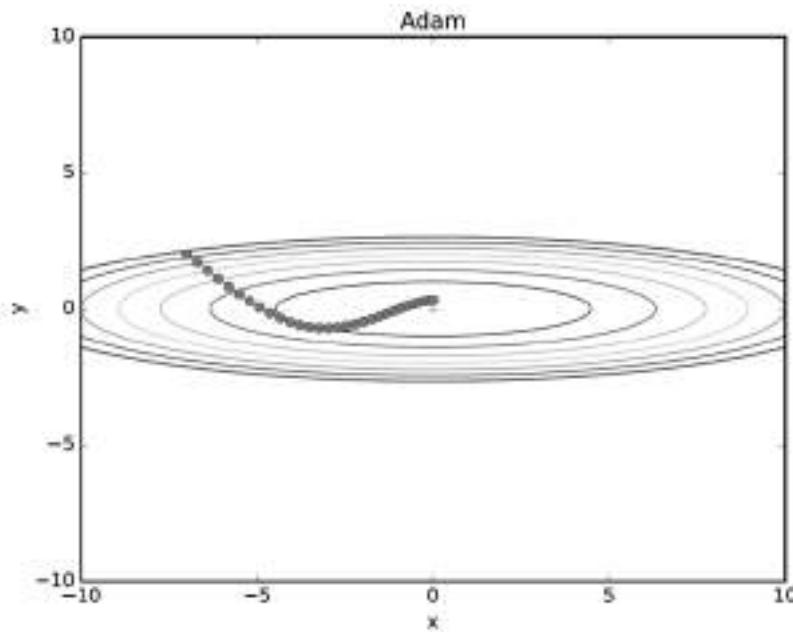
RMSProp-like

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning rate * dx / (np.sqrt(cache) + 1e-7)
```

# 매개변수 갱신

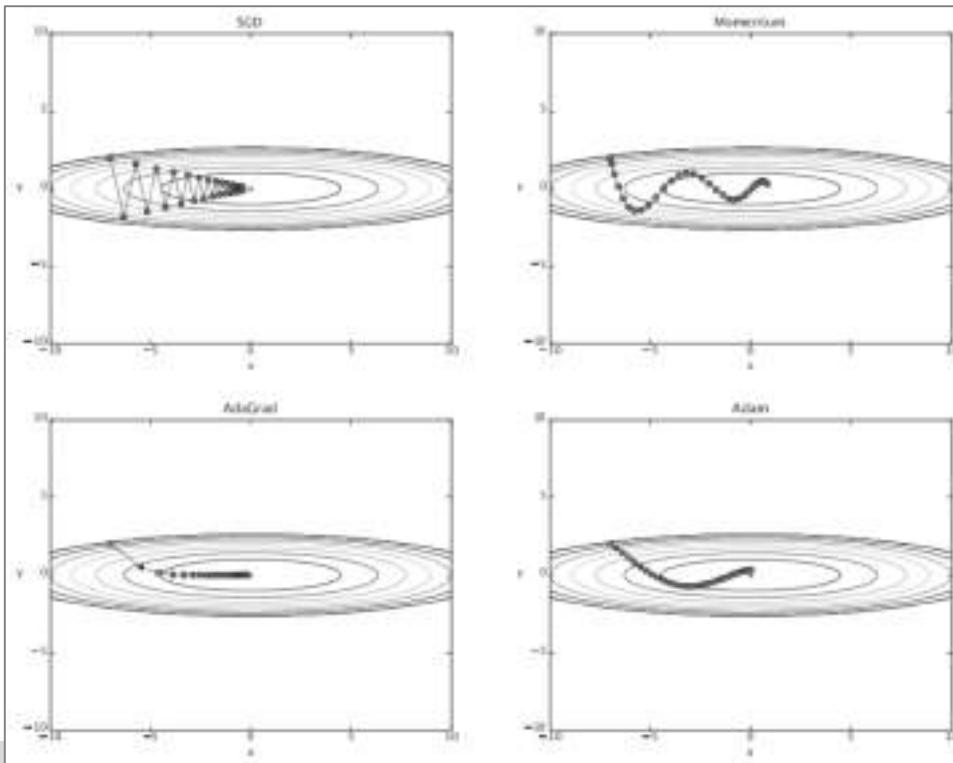
- Adam
  - RMSProp + 모멘텀



- 모멘텀처럼 그릇 바닥을 구르듯 움직임
- 모멘텀 보다 좌우 흔들림이 적음

# 매개변수 갱신

- 최적화 기법 비교

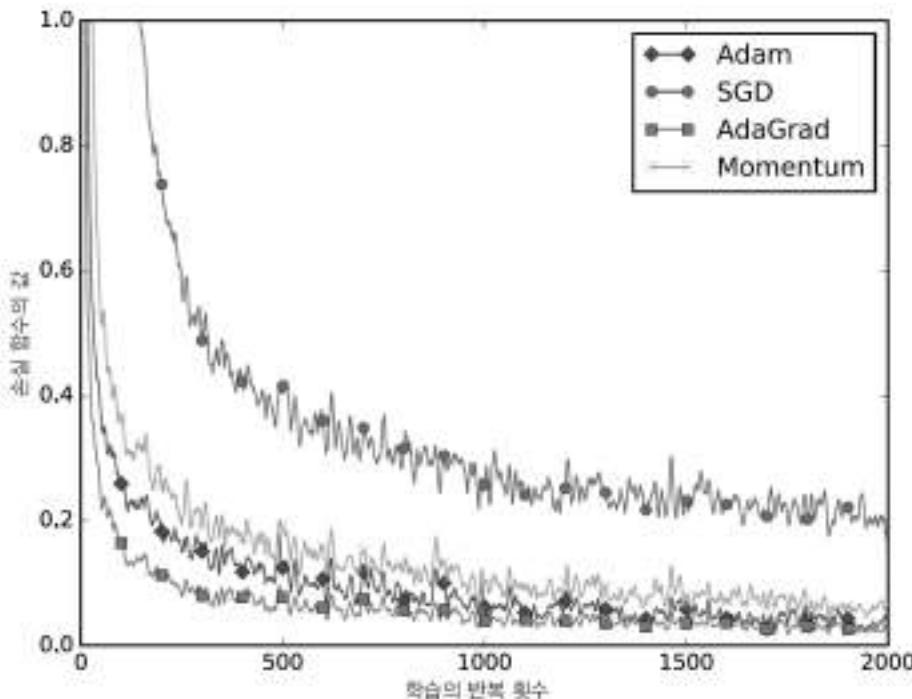


모든 문제에서 항상 뛰어난 기법은 아직 없습니다. 각자의 장단이 있습니다.

요즘 Adam을 많이 사용하는 편입니다.

# 매개변수 갱신

- 최적화 기법 비교 (mnist에서 비교)



- 2\_day/optimizer\_compare\_mnist.py

# 산내려오는 작은 오솔길 잘찾기(Optimizer)의 발달 계보

모든 자료를 다 검토해서  
내 위치의 산기울기를 계산해서  
갈 방향을 찾겠다.

**GD**

스텝방향

**Momentum**

스텝 계산해서 움직인 후,  
아까 내려오던 관성 방향 또 가자

**SGD**

전부 다봐야 한걸음은  
너무 오래 걸리니까  
조금만 보고 빨리 판단한다  
같은 시간에 더 많이 간다

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!

**RMSProp**

보폭을 줄이는 건 좋은데  
이전 맥락 상황봐가며 하자.

**Adagrad**

안가본곳은 성큼 빠르게 걸어 훑고  
많이 가본 곳은 잘아니까  
갈수록 보폭을 줄여 세밀히 탐색

**AdaDelta**

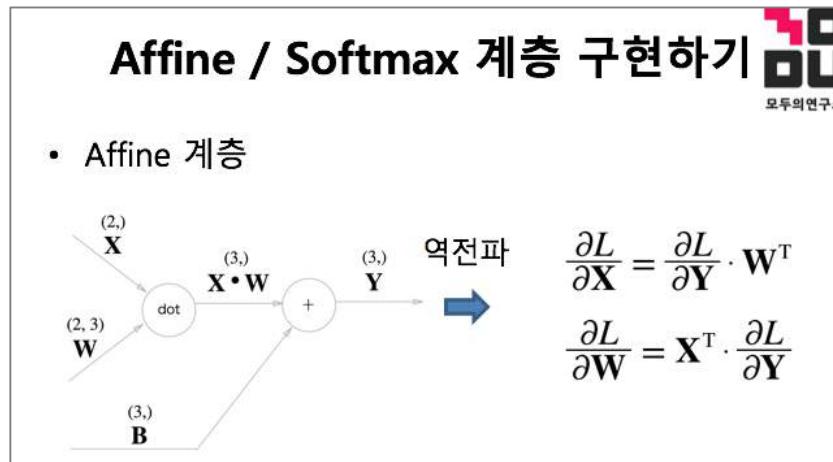
종종걸음 너무 작아져서  
정지하는 걸 막아보자.

# 가중치의 초기값

- 가중치 값을 작게하면 - 오버피팅이 덜 발생한다
- 가중치 값을 0으로 하면 ?

# 가중치의 초기값

- 가중치 값을 작게하면 - 오버피팅이 덜 발생한다
- 가중치 값을 0으로 하면 ?
  - 학습이 발생하지 않습니다
  - 초기값을 무작위로 설정해야 합니다



# 가중치의 초기값

- 은닉층의 활성화 값 분포
  - 초기값의 변화에 따라 은닉층의 활성화 값 분포가 어떻게 변하는지 확인해 보고자 합니다
  - 실험내용
    - 5개의 층, 각 층의 뉴런은 100개
    - 입력 데이터로 1000개의 데이터를 무작위로 생성
    - 활성화 함수로 시그모이드
    - 각 층의 활성화 함수 값을 activations 변수에 저장

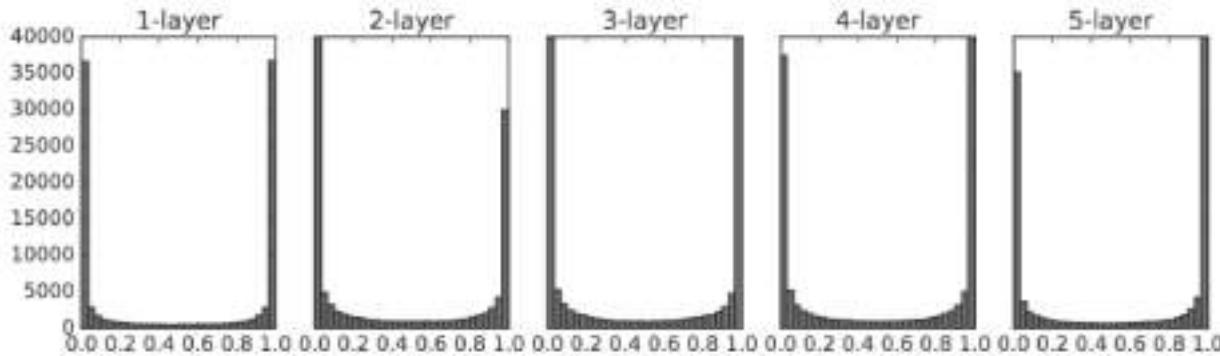
# 가중치의 초기값

- 은닉층의 활성화 값 분포
  - 2\_day/weight\_init\_activation\_histogram.py
  - 표준편차를 다르게 하여 실험

```
24 for l in range(hidden_layer_size):  
25     if l != 0:  
26         activations[l-1]  
27  
28     # 초기값을 다양하게 바꿔가며 실험해보자  
29     w = np.random.rand(node_num, node_num) * 1  
30     # w = np.random.rand(node_num, node_num) * 0.01  
31     # w = np.random.rand(node_num, node_num) * np.sqrt(1.0 / node_num)  
32     # w = np.random.rand(node_num, node_num) * np.sqrt(2.0 / node_num)  
33  
34  
35     a = np.dot(x, w)  
36  
37  
38     # 활성화 함수도 바꿔가며 실험해보자  
39     z = sigmoid(a)  
40     # z = ReLU(a)  
41     # z = tanh(a)  
42  
43     activations[l] = z  
44  
45     # 히스토그램 그리기  
46     for i, a in activations.items():  
47         plt.subplot(1, len(activations), i+1)  
48         plt.title(str(i+1) + "-layer")  
49         if i != 0: plt.yticks([])  
50         # plt.ylim(0, 1)  
51         # plt.ylim(0, 7000)  
52         plt.hist(a.flatten(), 30, range=[0,1])  
53 plt.show()
```

# 가중치의 초기값

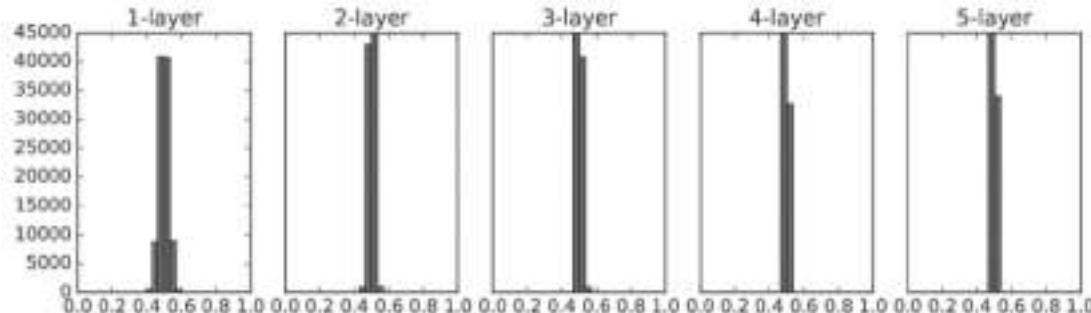
- 은닉층의 활성화 값 분포
  - 가중치를 표준편차가 1인 정규분포로 초기화 할때의 각 층의 활성화값 분포



- 값이 0과 1에 치우쳐 분포되어 있습니다
- 이 경우 시그모이드의 미분은 0에 가까워집니다
- 역전파 시 점점 그 값이 사라집니다 (**gradient vanishing**)
- 층을 깊게 하면 더 심각해 질 것임

# 가중치의 초기값

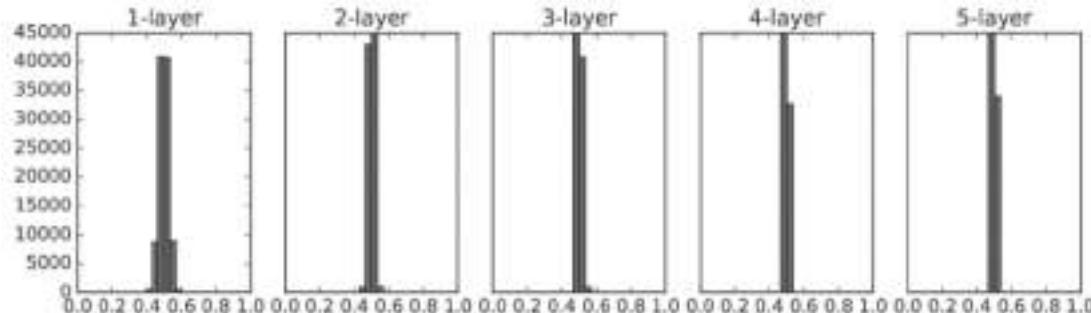
- 은닉층의 활성화 값 분포
  - 가중치를 표준편차가 0.01인 정규분포로 초기화 할 때의 각 층의 활성화값 분포



- 0.5 부근에 집중 됨. 기울기 소실이 발생하지 않음
- 활성화 값이 치우쳐 있다는 것은 표현력 관점에서 문제가 있는 것
  - 다수의 뉴런이 거의 같은 값을 출력하니 뉴런을 여러 개 둔 의미가 없어짐. 100개가 거의 같은 값을 출력하니 1개짜리 와 별반 다를 바 없음
  - 활성화 값이 치우치면 표현력이 제한되어 있는 것임

# 가중치의 초기값

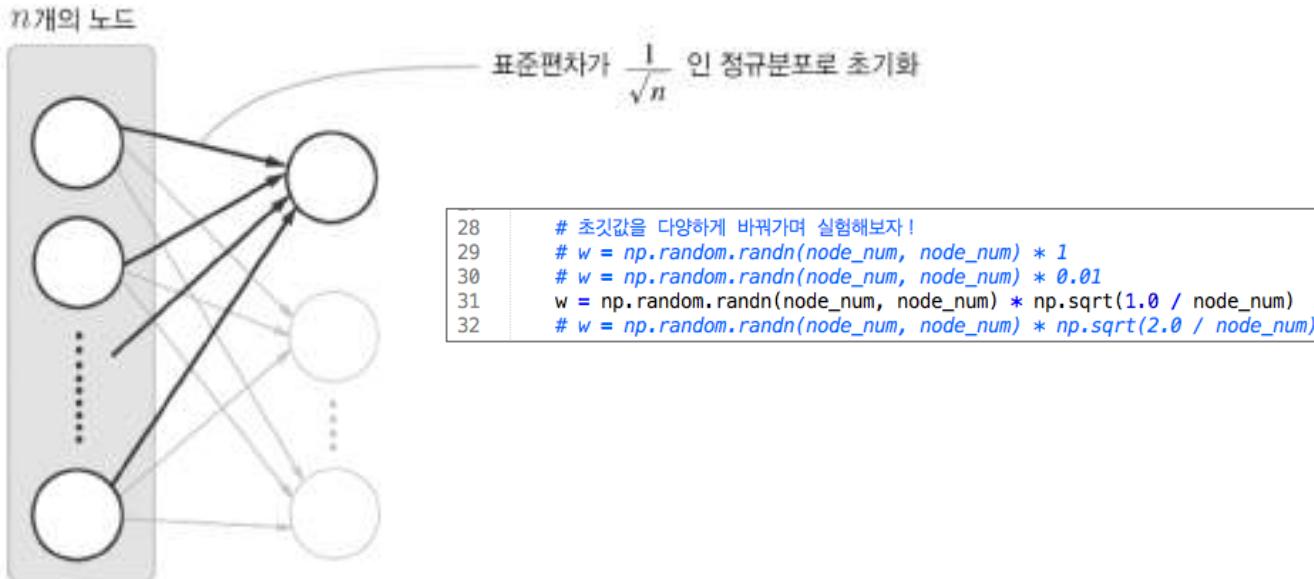
- 은닉층의 활성화 값 분포
  - 가중치를 표준편차가 0.01인 정규분포로 초기화 할 때의 각 층의 활성화값 분포



- **WARNING.** 각 층의 활성화 값은 적당히 고루 분포되어야 합니다. 층과 층 사이에 적당하게 다양한 데이터가 흐르게 해야 신경망 학습이 효율적으로 이뤄지기 때문입니다. 반대로 치우친 데이터가 흐르면 기울기 소실이나 표현력 제한 문제에 빠져 학습이 잘 이뤄지지 않는 경우가 생깁니다.

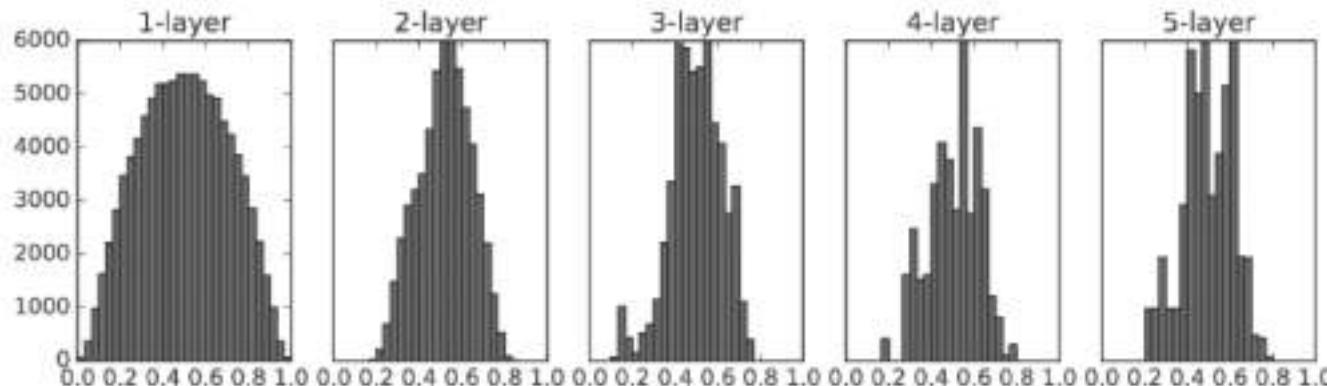
# 가중치의 초기값

- 은닉층의 활성화 값 분포
  - Xavier 초기값
    - 초기값의 표준편차가  $\frac{1}{\sqrt{n}}$ 인 분포를 사용



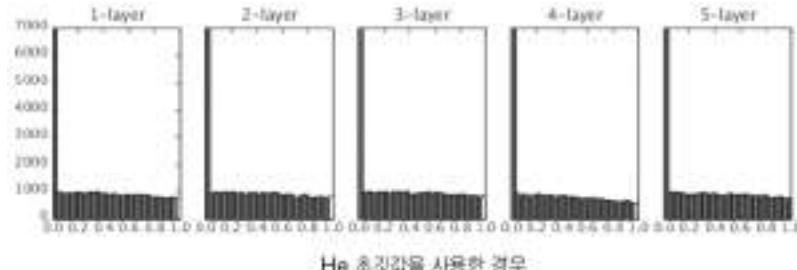
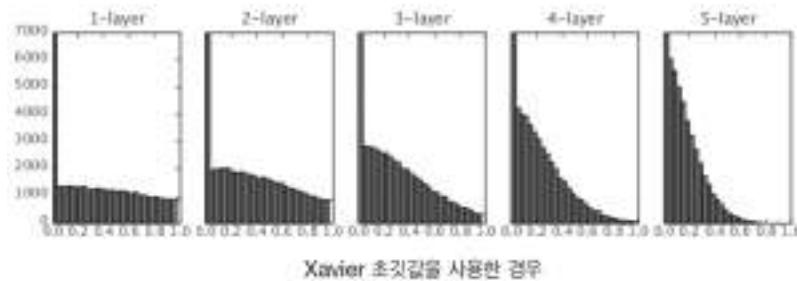
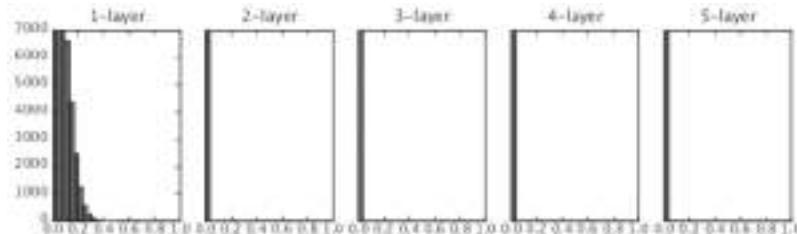
# 가중치의 초기값

- 은닉층의 활성화 값 분포
  - Xavier 초기값
    - 초기값의 표준편차가  $\frac{1}{\sqrt{n}}$ 인 분포를 사용



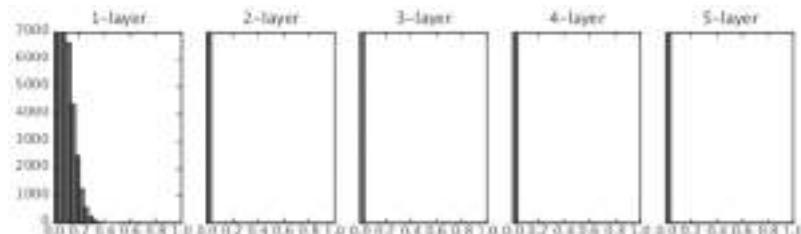
# 가중치의 초기값

- 시그모이드 대신 ReLU를 사용한다면
- He 초기값 (Kaming He) : 표준편차가  $\frac{2}{\sqrt{n}}$ 인 분포 사용

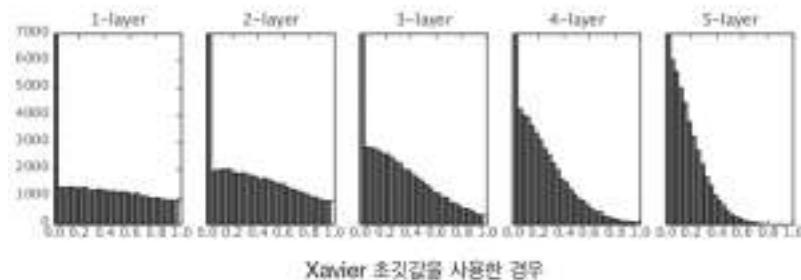


# 가중치의 초기값 (ReLU)

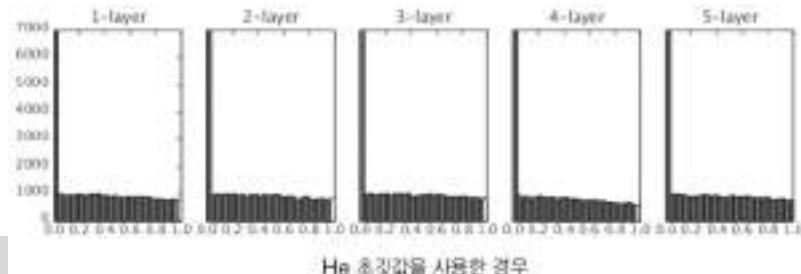
- 신경망에 작은 데이터가 흐른다는 것 -> 역전파 시 가중치의 기울기 역시 작아짐을 의미



- 깊어질 수록 0에 쓸림 증가

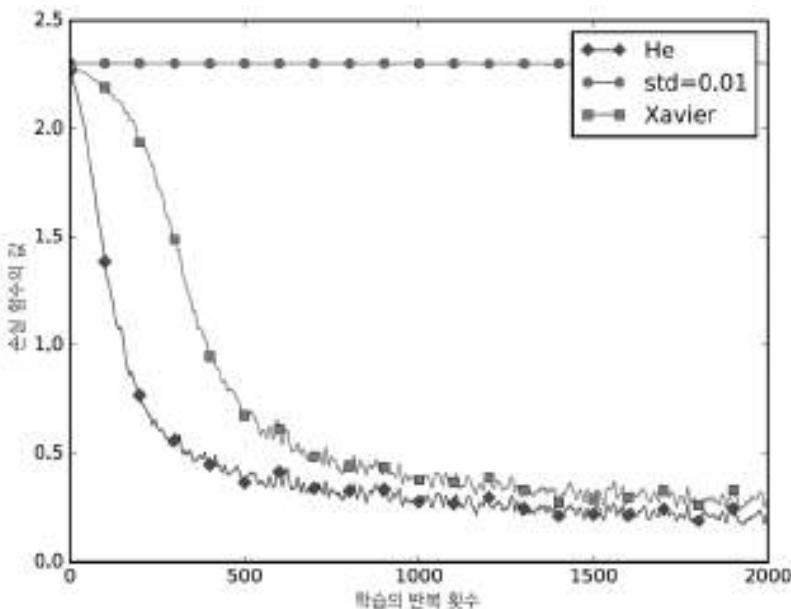


- 쓸만함



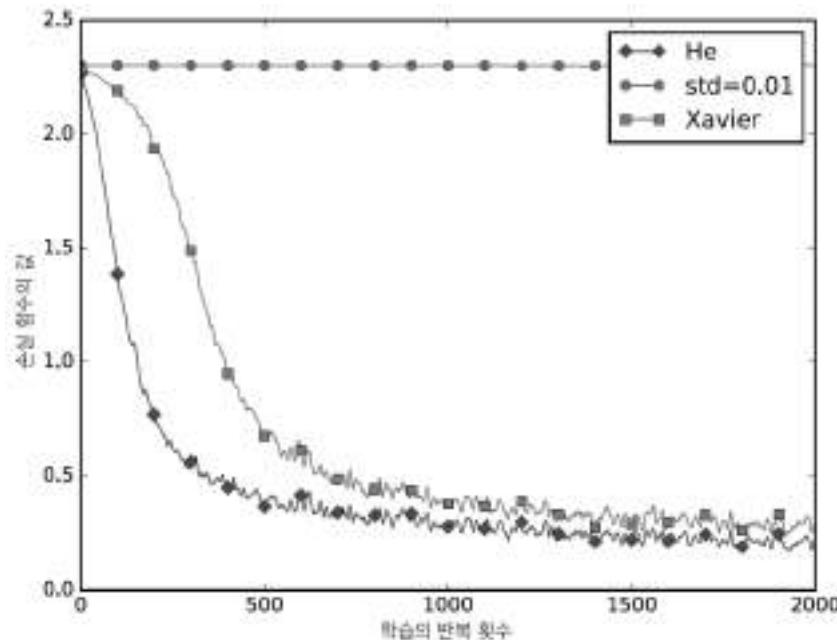
# 가중치의 초기값

- MNIST 데이터셋으로 본 가중치 초기값 비교
  - 5개층, 각 뉴런 100, 활성화 함수 ReLU
- 2\_day/weight\_init\_compare.py



# 가중치의 초기값

- MNIST 데이터셋으로 본 가중치 초기값 비교
  - 5개층, 각 뉴런 100, 활성화 함수 ReLU



초깃값이 매우 중요하군요.  
그렇지만 불편하네요~



배치 정규화

# 배치 정규화 (Batch Normalization)

- 2015년 등장
- 주목 받는 이유
  - 학습을 빨리 진행할 수 있다 (학습 속도 개선)
  - 초기값에 크게 의존하지 않는다 (아픈 초기값 선택 장애여 안녕)
  - 오버피팅을 억제한다 (드롭 아웃 등의 필요성 감소)
- 배치 정규화의 역할 : 각 층에서의 활성화값이 적당히 분포되도록 조정 (배치 정규화 계층을 삽입함)



# 배치 정규화 (Batch Normalization)



- 학습 시 미니배치를 단위로 데이터 분포가 평균이 0, 분산이 1이도록 정규화 수행

## 1) 배치 단위 정규화

미니배치 평균

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

미니배치 분산

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

평균0, 분산1로 정규화

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

학습 시 미니배치의 평균과 분산은 계속 저장되어 전체 데이터에 대한 평균과 분산으로 수렴하고 이는 테스트 시 사용됩니다

# 배치 정규화 (Batch Normalization)

- 학습 시 미니배치를 단위로 데이터 분포가 평균이 0, 분산이 1이도록 정규화 수행

## 2) 고유한 확대(scale) 이동(shift) 변환

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$

확대  이동

- 초기 값  $\gamma = 1, \beta = 0$
- 초기 값  $\gamma, \beta$ 는 학습되는 파라미터입니다

# 배치 정규화 (Batch Normalization)



## Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer.  
To make each dimension unit gaussian, apply:

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

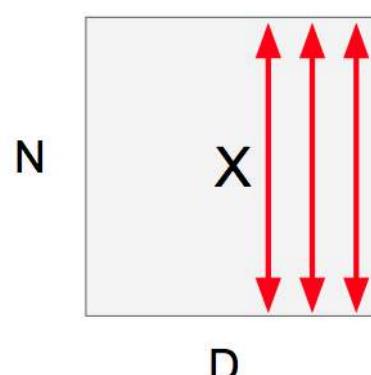
this is a vanilla  
differentiable function...

# 배치 정규화 (Batch Normalization)

## Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations?  
just make them so.”



1. compute the empirical mean and variance independently for each dimension.

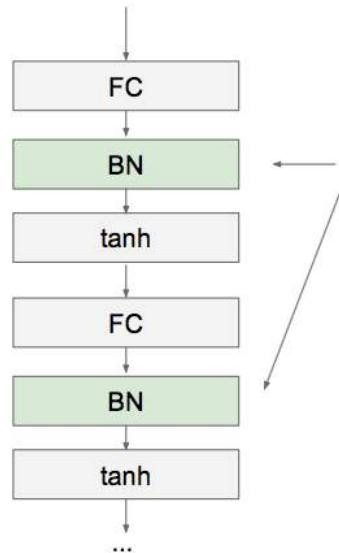
2. Normalize

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

# 배치 정규화 (Batch Normalization)

## Batch Normalization

[Ioffe and Szegedy, 2015]



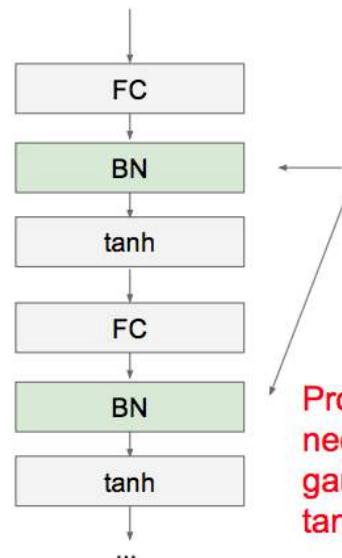
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

# 배치 정규화 (Batch Normalization)

## Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

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

# 배치 정규화 (Batch Normalization)



## Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \text{E}[x^{(k)}]$$

to recover the identity mapping.

# 배치 정규화 (Batch Normalization)



## Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;  
Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

# 배치 정규화 (Batch Normalization)



## Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;  
Parameters to be learned:  $\gamma, \beta$

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

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

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

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

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

[Ioffe and Szegedy, 2015]

**Note: at test time BatchNorm layer functions differently:**

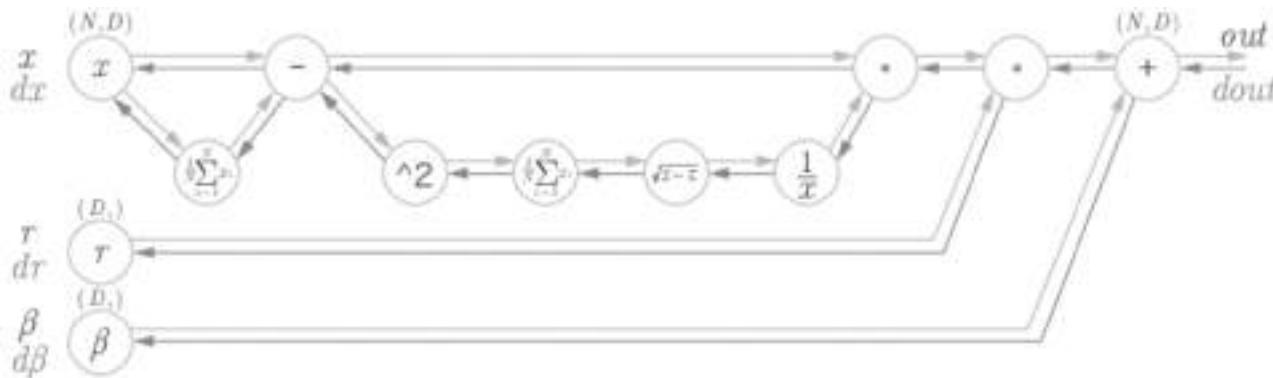
The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

# 배치 정규화 (Batch Normalization)

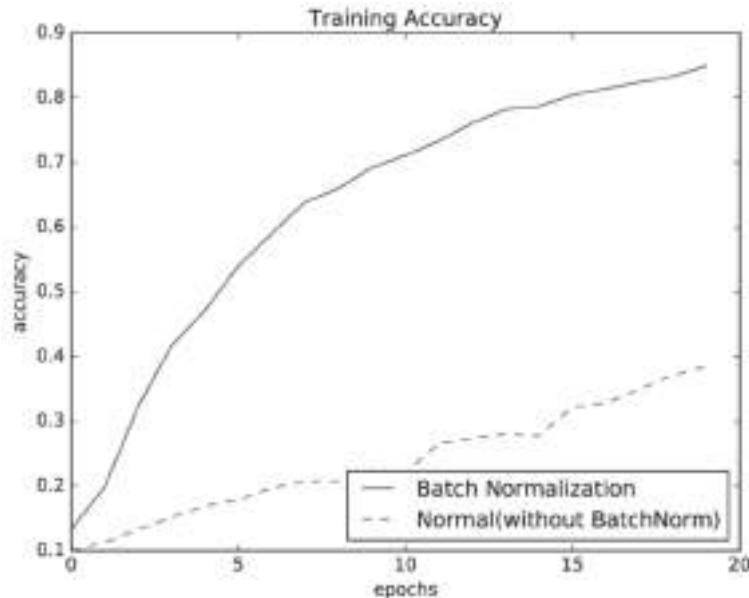
- 학습 시 미니배치를 단위로 데이터 분포가 평균이 0, 분산이 1이도록 정규화 수행

## 배치 정규화 계산 그래프



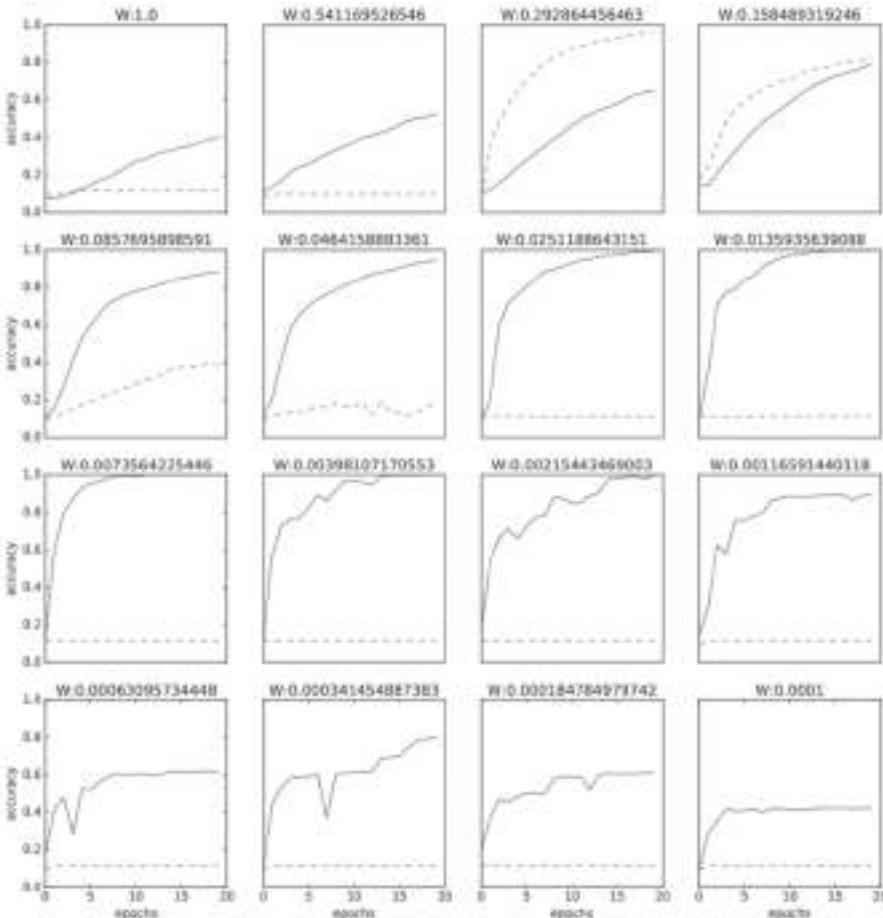
# 배치 정규화 (Batch Normalization)

- 학습 시 미니배치를 단위로 데이터 분포가 평균이 0, 분산이 1이도록 정규화 수행  
배치 정규화 효과 : 배치 정규화가 학습 속도를 높인다



- 2\_day/batch\_norm\_test.py

# 배치 정규화 (Batch Normalization)



- 실선 : Batch norm
- 점선 : 사용 안함

# 바른 학습을 위해

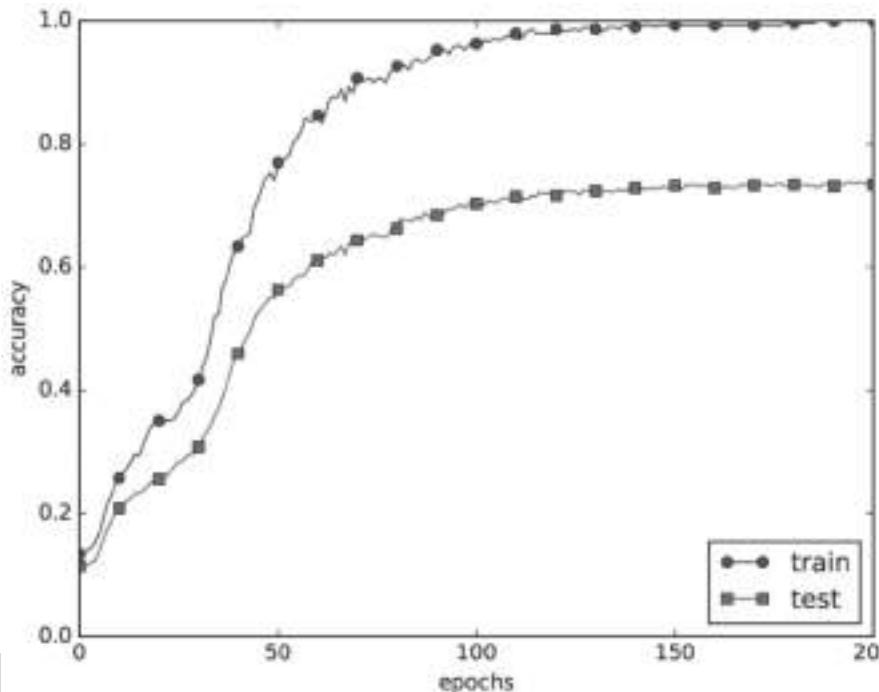
- 오버피팅 (Over fitting) : 훈련데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응 못함
- 주로 언제 발생하나요?

# 바른 학습을 위해

- 오버피팅 (Over fitting) : 훈련데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응 못함
- 주로 언제 발생하나요?
  - 매개변수가 많고 표현력이 높은 모델
  - 훈련 데이터가 적음
- 일부러 오버피팅을 발생 시키는 실험을 해봅니다
  - 데이터 : mnist데이터에서 300개만 사용
  - 각 층의 뉴런이 100개인 7층 네트워크 구성
  - ReLU적용

# 바른 학습을 위해

- 오버피팅 (Over fitting) : 훈련데이터에만 지나치게 적응되어 그 외의 데이터에는 제대로 대응 못함

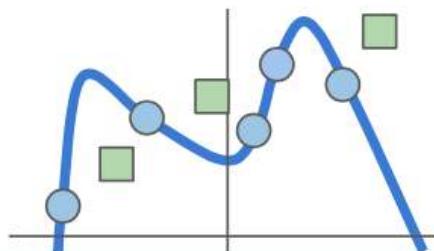


# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)$$

**Data loss:** Model predictions  
should match training data

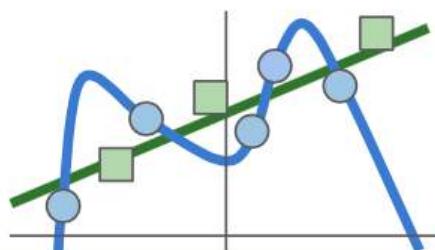


# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \underbrace{\lambda R(W)}_{\text{Regularization: Model should be "simple", so it works on test data}}$$

**Data loss:** Model predictions should match training data      **Regularization:** Model should be “simple”, so it works on test data



# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)

## Regularization

$\lambda$  = regularization strength  
(hyperparameter)

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

In common use:

**L2 regularization**       $R(W) = \sum_k \sum_l W_{k,l}^2$

**L1 regularization**       $R(W) = \sum_k \sum_l |W_{k,l}|$

**Elastic net (L1 + L2)**     $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

**Max norm regularization** (might see later)

**Dropout** (will see later)

**Fancier:** Batch normalization, stochastic depth

# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)

## L2 Regularization (Weight Decay)

$$x = [1, 1, 1, 1]$$

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

$$w_1^T x = w_2^T x = 1$$

# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)

## L2 Regularization (Weight Decay)

$$x = [1, 1, 1, 1]$$

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

(If you are a Bayesian: L2 regularization also corresponds MAP inference using a Gaussian prior on W)

$$w_1^T x = w_2^T x = 1$$

# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)
- Weight decay 구현 : common/multi\_layer\_net.py

```
75     def loss(self, x, t):  
76         """손실 함수를 구한다.  
77  
78         Parameters  
79         _____  
80         x : 입력 데이터  
81         t : 정답 레이블  
82  
83         Returns  
84         _____  
85         손실 함수의 값  
86         """  
87         y = self.predict(x)  
88  
89         weight_decay = 0  
90         for idx in range(1, self.hidden_layer_num + 2):  
91             W = self.params['W' + str(idx)]  
92             weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W ** 2)  
93  
94         return self.last_layer.forward(y, t) + weight_decay
```

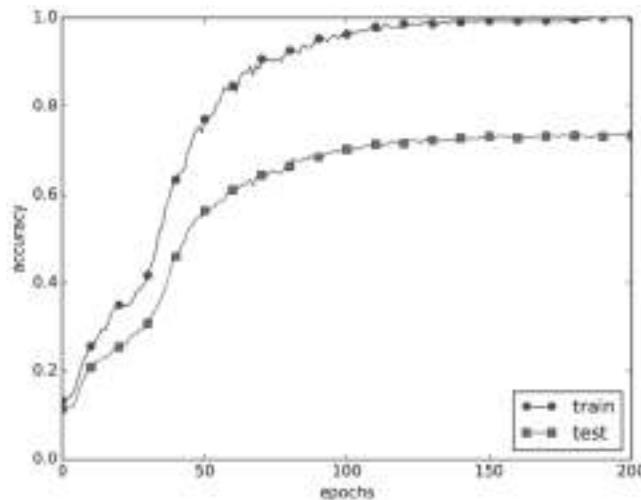
# 바른 학습을 위해

- Weight decay 구현 : common/multi\_layer\_net.py

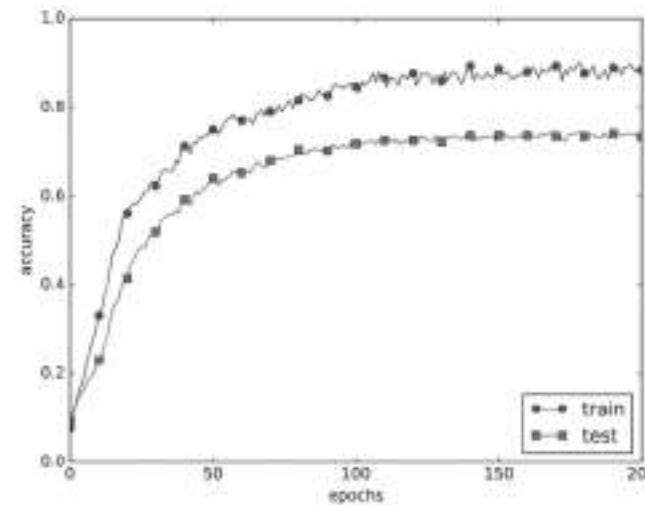
```
127     def gradient(self, x, t):
128         """기울기를 구한다(오차역전파법)."""
129
130         Parameters:
131         -----
132         x : 입력 데이터
133         t : 정답 레이블
134
135         Returns:
136         -----
137         각 층의 기울기를 담은 딕셔너리(dictionary) 변수
138         grads['W1'], grads['W2'], ... 각 층의 가중치
139         grads['b1'], grads['b2'], ... 각 층의 편향
140
141         # forward
142         self.loss(x, t)
143
144         # backward
145         dout = 1
146         dout = self.last_layer.backward(dout)
147
148         layers = list(self.layers.values())
149         layers.reverse()
150         for layer in layers:
151             dout = layer.backward(dout)
152
153         # 결과 저장
154         grads = {}
155         for idx in range(1, self.hidden_layer_num+2):
156             grads['W' + str(idx)] = self.layers['Affine' + str(idx)].dw + \
157                 self.weight_decay_lambda * self.layers['Affine' + str(idx)].W
158             grads['b' + str(idx)] = self.layers['Affine' + str(idx)].db
159
160         return grads
```

# 바른 학습을 위해

- 오버피팅 억제 : 가중치 감소 (weight decay)



Weight decay 미 적용



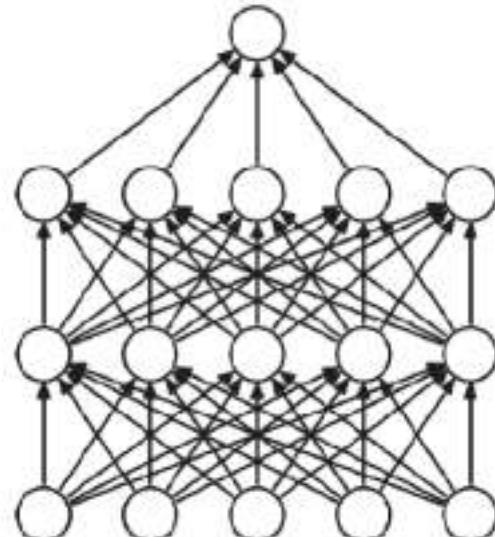
Weight decay 적용

# 바른 학습을 위해

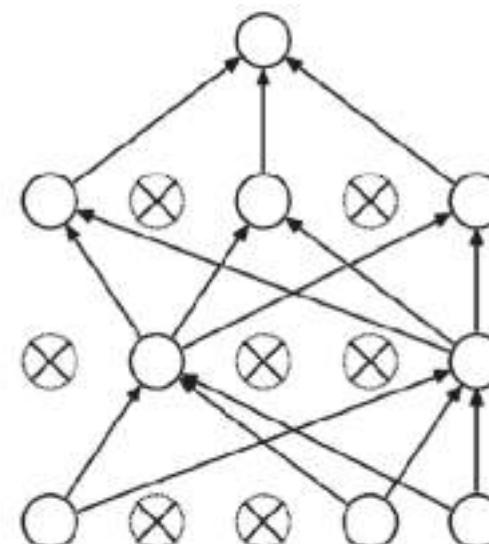
- 오버피팅 억제 : 드롭아웃 (Dropout)
  - 훈련때 은닉층의 뉴런을 무작위로 골라 삭제하면서 학습하는 방법
  - 방법론
    - 1) 훈련때 데이터를 흘릴 때마다 삭제할 뉴런을 무작위로 선택하고
    - 2) 시험때는 모든 뉴런에 신호를 전달
      - 단, 시험 때는 각 뉴런의 출력에 훈련 때 삭제한 비율을 곱하여 출력

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)



(a) 일반 신경망



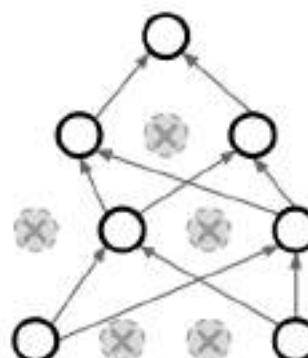
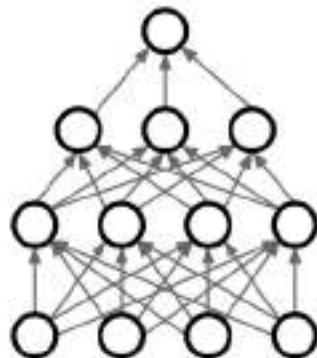
(b) 드롭아웃을 적용한 신경망

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Regularization: Dropout

In each forward pass, randomly set some neurons to zero  
 Probability of dropping is a hyperparameter; 0.5 is common



Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Regularization: Dropout

```

p = 0.5 # probability of keeping a unit active. higher = less dropout

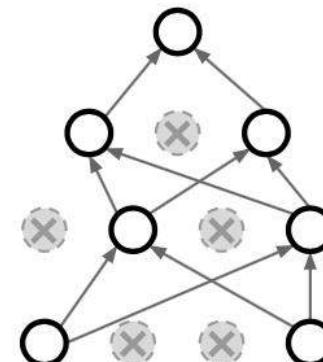
def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

```

Example forward pass with a 3-layer network using dropout

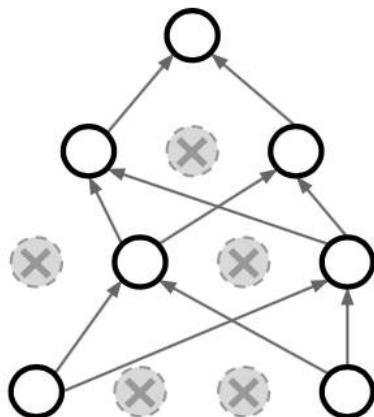


# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Regularization: Dropout

How can this possibly be a good idea?



Forces the network to have a redundant representation;  
Prevents co-adaptation of features

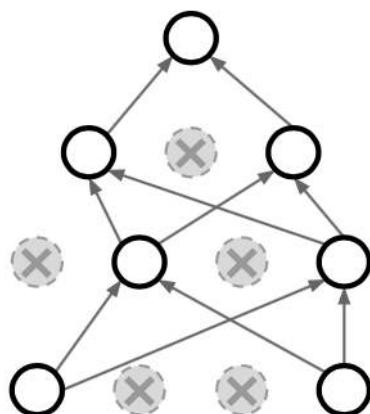


# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has  $2^{4096} \sim 10^{1233}$  possible masks!  
Only  $\sim 10^{82}$  atoms in the universe...

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Dropout: Test time

Dropout makes our output random!

$$y = f_W(x, z)$$

Output (label)	Input (image)	Random mask

Want to “average out” the randomness at test-time

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

But this integral seems hard ...

# 바른 학습을 위해

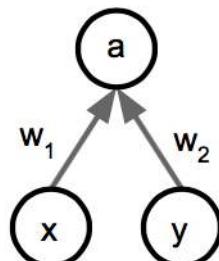
- 오버피팅 억제 : 드롭아웃 (Dropout)

## Dropout: Test time

Want to approximate  
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

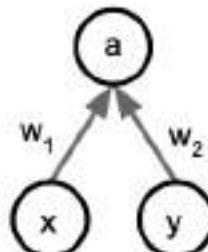
## Dropout: Test time

Want to approximate  
the integral

$$y = f(x) = E_z [f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.

At test time we have:  $E[a] = w_1x + w_2y$



# 바른 학습을 위해

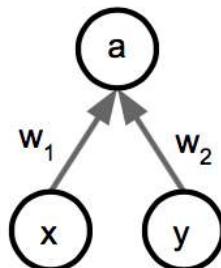
- 오버피팅 억제 : 드롭아웃 (Dropout)

## Dropout: Test time

Want to approximate  
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



At test time we have:  $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

# 바른 학습을 위해

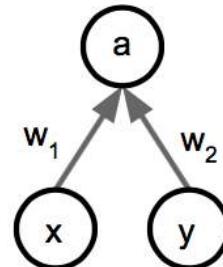
- 오버피팅 억제 : 드롭아웃 (Dropout)

## Dropout: Test time

Want to approximate  
the integral

$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Consider a single neuron.



At test time we have:  $E[a] = w_1x + w_2y$

During training we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

**At test time, multiply  
by dropout probability**

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

## Dropout: Test time

```
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:  
output at test time = expected output at training time

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

```


    """Vanilla Dropout: Not recommended implementation (see notes below)"""
p = 0.5 # probability of keeping a unit active: higher == less dropout

def train_step(X):
    """X contains the data"""

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    OUT = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # unrolled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
    out = np.dot(W3, H2) + b3


```

## Dropout Summary

drop in forward pass

scale at test time

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)

More common: “Inverted dropout”

```

p = 0.5 # probability of keeping a unit active. Higher = less dropout

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # unrolled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
  
```

test time is unchanged!

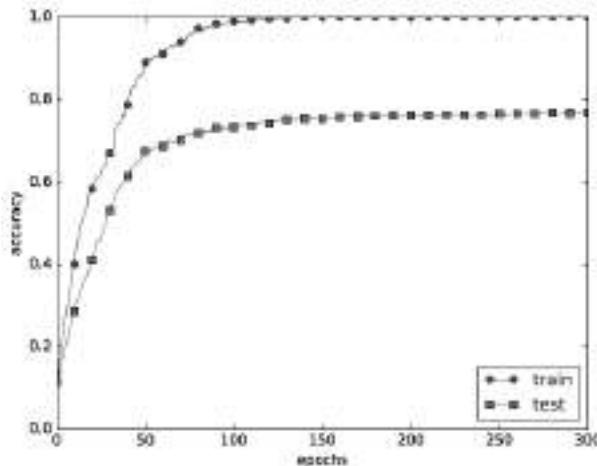
# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)
  - common/layers.py

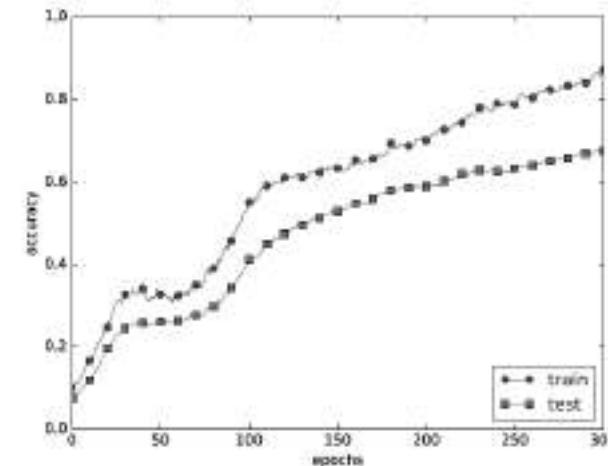
```
95  class Dropout:  
96      """  
97          http://arxiv.org/abs/1207.0580  
98      """  
99      def __init__(self, dropout_ratio=0.5):  
100         self.dropout_ratio = dropout_ratio  
101         self.mask = None  
102  
103     def forward(self, x, train_flg=True):  
104         if train_flg:  
105             self.mask = np.random.rand(*x.shape) > self.dropout_ratio  
106             return x * self.mask  
107         else:  
108             return x * (1.0 - self.dropout_ratio)  
109  
110     def backward(self, dout):  
111         return dout * self.mask
```

# 바른 학습을 위해

- 오버피팅 억제 : 드롭아웃 (Dropout)



드랍아웃 미적용



드랍아웃 적용

# 적절한 하이퍼파라미터 값 찾기



- 뉴런 수는 어떻게 하지? 배치 크기는? 학습율은? 가중치 감소는 ?

**하이퍼 파라미터의 값을 효율적으로 탐색  
하는 방법을 알아 봅시다**

# 적절한 하이퍼파라미터 값 찾기



- 검증 데이터 (validation data)
  - 하이퍼 파라미터를 찾을 때 시험 데이터를 사용하면 안됩니다
    - 하이퍼 파라미터 값이 시험 데이터에 오버피팅 함
  - 검증 데이터 : 하이퍼파라미터 전용 확인 데이터
- 데이터의 구분
  - 훈련 데이터 : 매개변수 학습
  - 검증 데이터 : 하이퍼파라미터 성능 평가
  - 시험 데이터 : 신경망의 범용 성능 평가 (마지막에 이용)

# 적절한 하이퍼파라미터 값 찾기

- 검증 데이터 (validation data)
  - MNIST 훈련 데이터 중 20%를 검증 데이터로 가져오기

```
11 (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True)
12
13 # 결과를 빠르게 얻기 위해 훈련 데이터를 줄임
14 x_train = x_train[:500]
15 t_train = t_train[:500]
16
17 # 20%를 검증 데이터로 분할
18 validation_rate = 0.20
19 validation_num = x_train.shape[0] * validation_rate
20 x_train, t_train = shuffle_dataset(x_train, t_train) # 데이터 섞기
21 x_val = x_train[:validation_num]
22 t_val = t_train[:validation_num]
23 x_train = x_train[validation_num:]
24 t_train = t_train[validation_num:]
```

# 적절한 하이퍼파라미터 값 찾기



- 하이퍼 파라미터 최적화
  - 방법론
    - 하이퍼 파라미터 대략적 범위를 설정한 후 이 값을 무작위로 선택하여 정확도 평가 후 범위를 재설정하고 다시 찾기를 반복
  - 범위 설정
    - '대략적으로' 지정
    - 실제로도 0.001에서 1000사이 ( $10^{-3} \sim 10^3$ )과 같이 '10의 거듭제곱' 단위로 범위를 지정 : 로그 스케일(log scale)로 지정

# 적절한 하이퍼파라미터 값 찾기



- **하이퍼 파라미터 최적화**
  - 하이퍼 파라미터 최적화는 몇일 혹은 몇 주 이상의 오랜 시간이 걸림
    - 나쁠듯한 값은 일찍 포기하는게 좋음
  - 시간이 오래 걸리기 때문에 에폭을 작게 하여, 1회 평가에 걸리는 시간을 단축하는게 효과적임
- **하이퍼 파라미터 최적화 정리**
  - 0 단계 : 하이퍼파라미터 값의 범위를 설정
  - 1 단계 : 설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출
  - 2 단계 : 1단계에서 샘플링한 하이퍼파라미터 값을 사용하여 학습하고, 검증 데이터로 정확도를 평가 (단, 에폭은 작게 설정)
  - 3단계 : 1단계와 2단계를 특정 횟수(100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힘

# 적절한 하이퍼파라미터 값 찾기



- **하이퍼 파라미터 최적화**
- **Note :** 여기에서 설명한 하이퍼파라미터 최적화 방법은 실용적인 방법입니다. 하지만 과학이라기 보다는 다분히 수행자의 ‘지혜’와 ‘직관’에 의존한다는 느낌이 들죠. 더 세련된 기법을 원한다면 베이즈 최적화(Bayesian optimization)를 소개할 수 있겠네요. 베이즈 최적화는 베이즈 정리(bayes's theorem)을 중심으로 한 수학 이론을 구사하여 더 엄밀하고 효율적으로 최적화를 수행합니다. 자세한 내용은 <Practical Bayesian Optimization of Machine Learning Algorithms> 논문 등을 참조하세요.

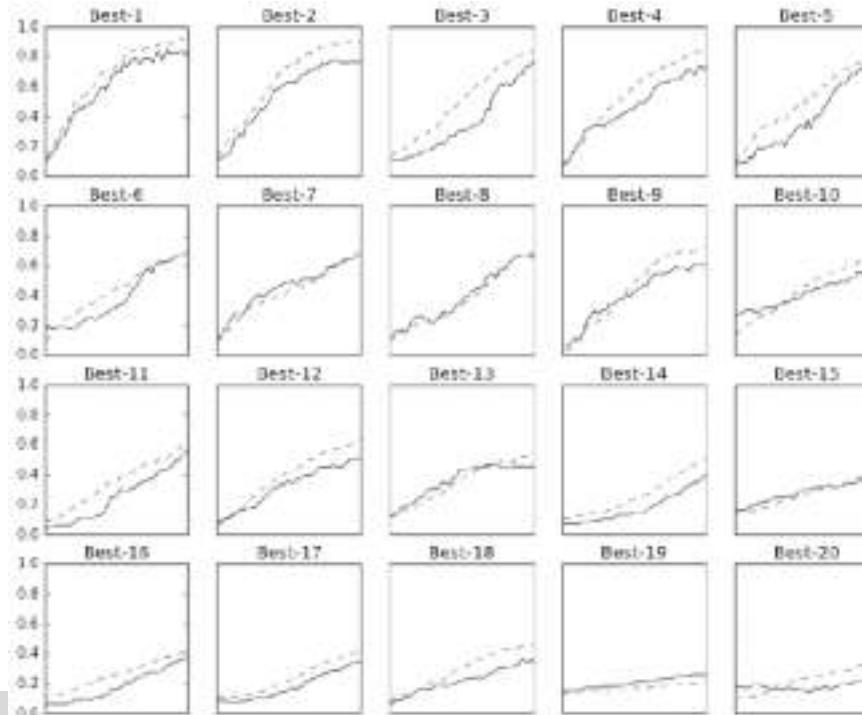
# 적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화 구현

```
38 # 하이퍼파라미터 무작위 탐색=====
39 optimization_trial = 100
40 results_val = {}
41 results_train = {}
42 for _ in range(optimization_trial):
43     # 탐색한 하이퍼파라미터의 범위 지정=====
44     weight_decay = 10 ** np.random.uniform(-8, -4)
45     lr = 10 ** np.random.uniform(-6, -2)
46     #
47
48     val_acc_list, train_acc_list = __train(lr, weight_decay)
49     print("val acc:" + str(val_acc_list[-1]) + " | lr:" + str(lr) + ", "
50           key = "lr:" + str(lr) + ", weight decay:" + str(weight_decay)
51           results_val[key] = val_acc_list
52           results_train[key] = train_acc_list
```

# 적절한 하이퍼파라미터 값 찾기

- 하이퍼 파라미터 최적화 구현



# 적절한 하이퍼파라미터 값 찾기



## Babysitting the Learning Process

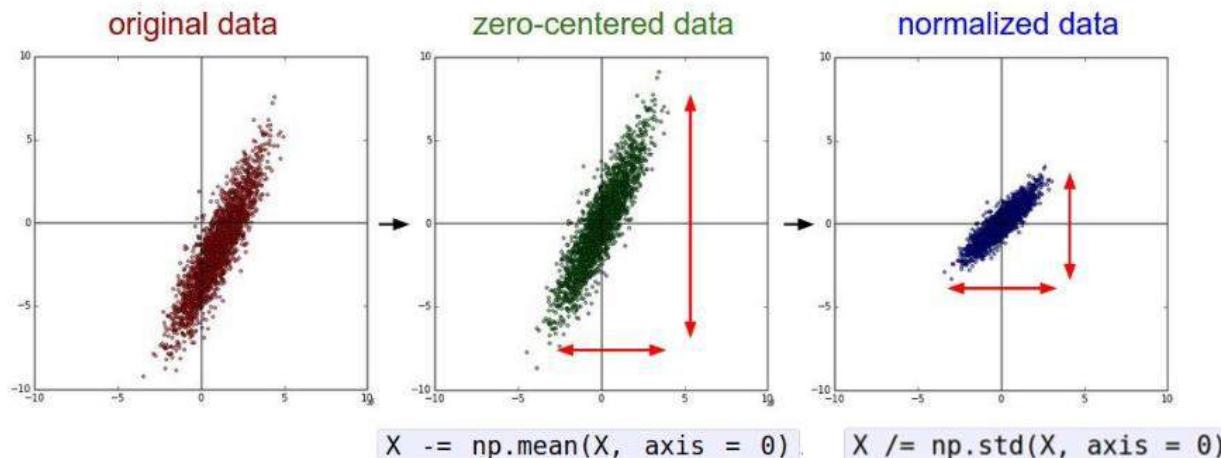
Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 6 - 61

April 20, 2017

# 적절한 하이퍼파라미터 값 찾기

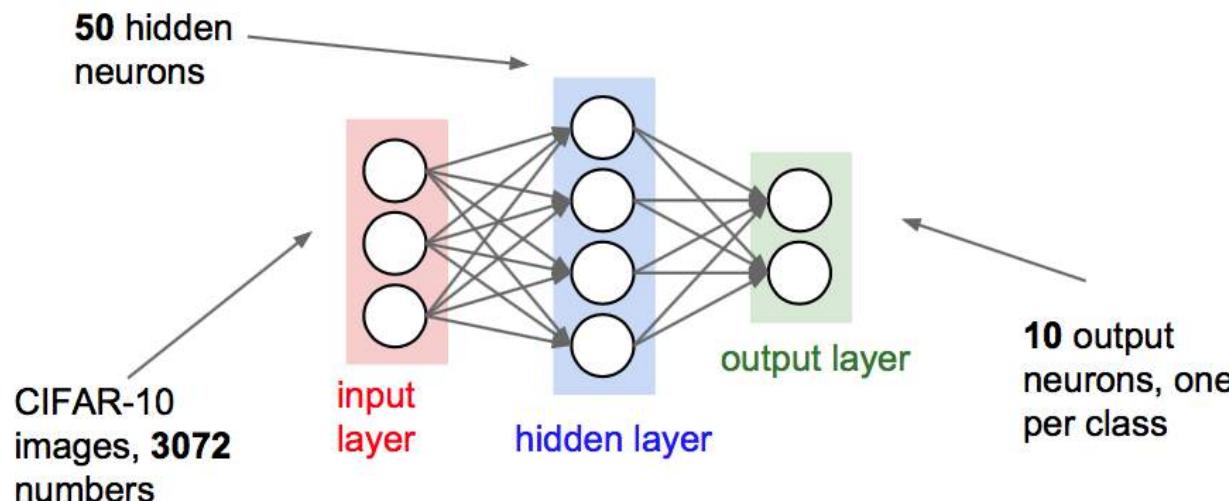
## Step 1: Preprocess the data



(Assume  $X$  [NxD] is data matrix,  
each example in a row)

# 적절한 하이퍼파라미터 값 찾기

**Step 2: Choose the architecture:**  
say we start with one hidden layer of 50 neurons:



# 적절한 하이퍼파라미터 값 찾기

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0) disable regularization
print loss
```

2.30261216167

loss ~2.3.

"correct" for  
10 classes

returns the loss and the  
gradient for all parameters

# 적절한 하이퍼파라미터 값 찾기

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3) crank up regularization
print loss
```

3.06859716482

loss went up, good. (sanity check)

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

**Tip:** Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization ( $\text{reg} = 0.0$ )
- use simple vanilla 'sgd'

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

**Tip:** Make sure that you can overfit very small portion of the training data

Very small loss,  
train accuracy 1.00,  
nice!

```
model = init two layer model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)

Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974098, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737438, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 20 / 200: cost 1.395760, train: 0.550000, val 0.550000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches = True,
                                    learning_rate=1e-6, verbose=True)
```

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches=True,
                                    learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10 cost 2.302420 train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches=True,
                                    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches=True,
                                    learning_rate=1e-6, verbose=True)

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                    model, two_layer_net,
                                    num_epochs=10, reg=0.000001,
                                    update='sgd', learning_rate_decay=1,
                                    sample_batches = True,
```

Now let's try learning rate 1e6.

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)

/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero encountered in log
    data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value encountered in subtract
    probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

# 적절한 하이퍼파라미터 값 찾기

Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)

Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes....

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

# 적절한 하이퍼파라미터 값 찾기



## Hyperparameter Optimization

Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 6 - 75

April 20, 2017

# 적절한 하이퍼파라미터 값 찾기

## Cross-validation strategy

**coarse -> fine cross-validation in stages**

**First stage:** only a few epochs to get rough idea of what params work

**Second stage:** longer running time, finer search

... (repeat as necessary)

Tip for detecting explosions in the solver:

If the cost is ever  $> 3 * \text{original cost}$ , break out early

# 적절한 하이퍼파라미터 값 찾기

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6) ←
        note it's best to optimize
        in log space!
    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning rate=lr, verbose=False)
```

```
val acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

# 적절한 하이퍼파라미터 값 찾기

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 6.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.018889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good  
for a 2-layer neural net  
with 50 hidden neurons.

# 적절한 하이퍼파라미터 값 찾기

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good  
for a 2-layer neural net  
with 50 hidden neurons.

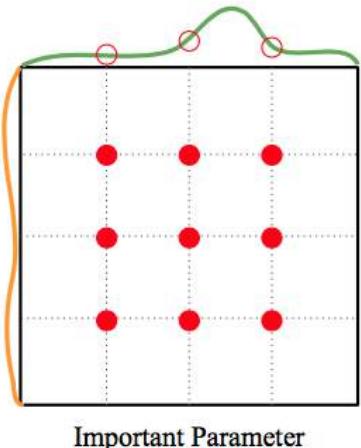
But this best  
cross-validation result is  
worrying. Why?

# 적절한 하이퍼파라미터 값 찾기

## Random Search vs. Grid Search

*Random Search for  
Hyper-Parameter Optimization  
Bergstra and Bengio, 2012*

Grid Layout



Random Layout

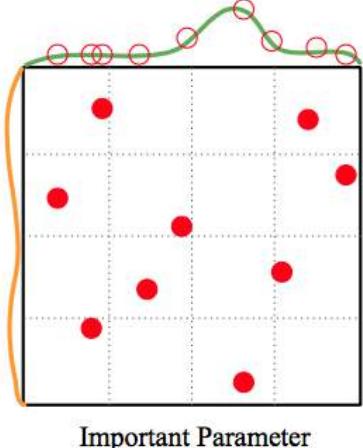


Illustration of Bergstra et al., 2012 by Shayne  
Longpre, copyright CS231n 2017

# 적절한 하이퍼파라미터 값 찾기

## Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner  
music = loss function

This image by Paolo Guereta is licensed under CC-BY 2.0



# 적절한 하이퍼파라미터 값 찾기

Cross-validation  
“command center”



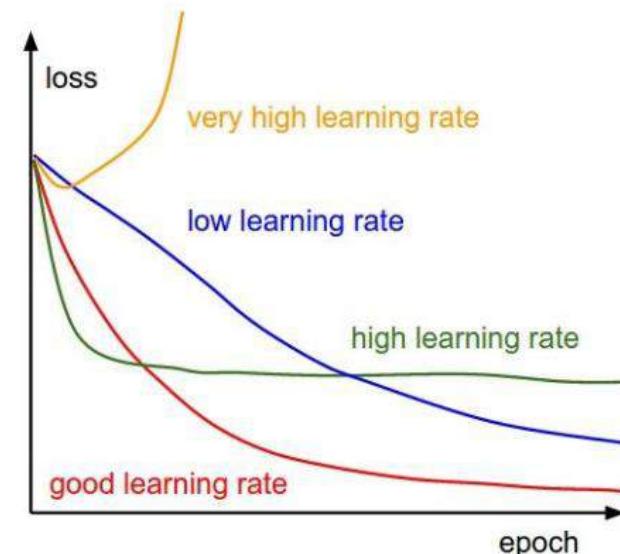
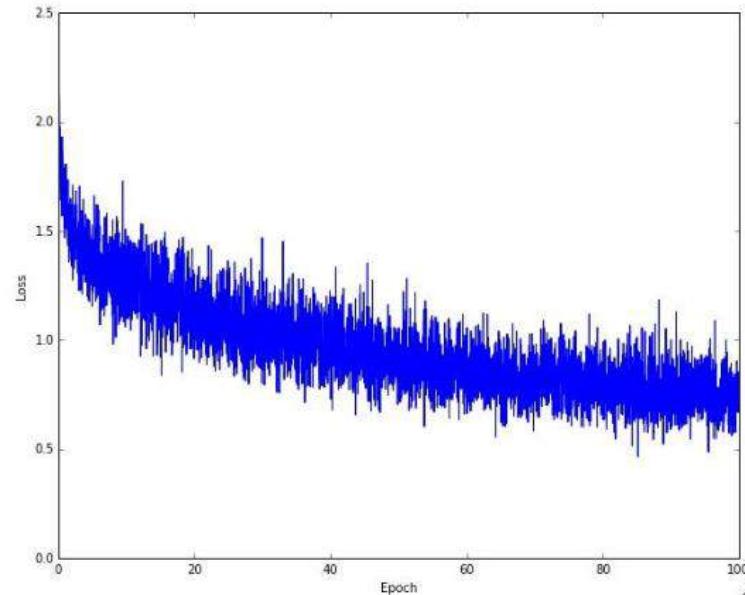
Fei-Fei Li & Justin Johnson & Serena Yeung

Lecture 6 - 82

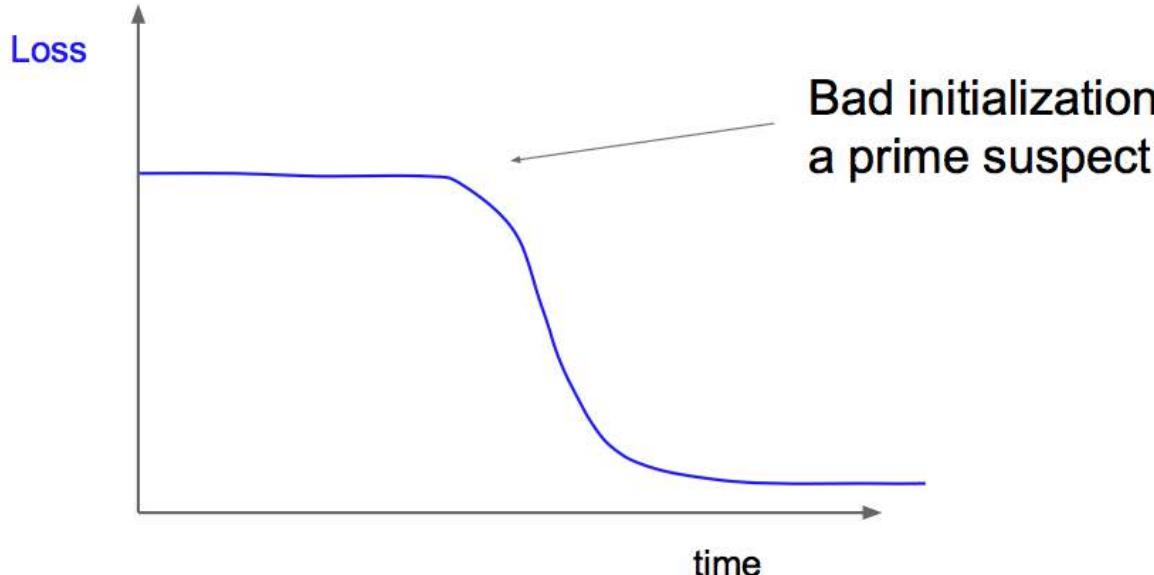
April 20, 2017

# 적절한 하이퍼파라미터 값 찾기

Monitor and visualize the loss curve

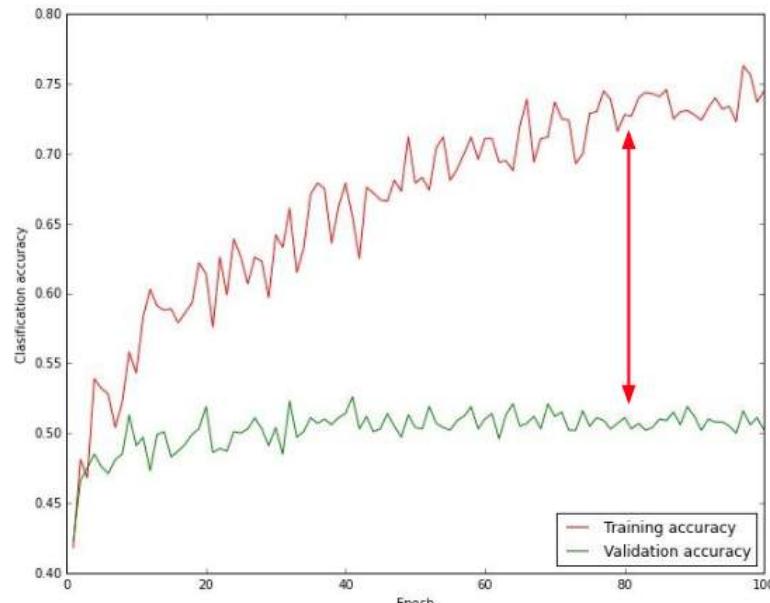


# 적절한 하이퍼파라미터 값 찾기



# 적절한 하이퍼파라미터 값 찾기

Monitor and visualize the accuracy:



big gap = overfitting  
=> increase regularization strength?

no gap  
=> increase model capacity?

# 적절한 하이퍼파라미터 값 찾기

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values:  $\sim 0.0002 / 0.02 = 0.01$  (about okay)  
**want this to be somewhere around 0.001 or so**

# 적절한 하이퍼파라미터 값 찾기



## Summary

We looked in detail at:

- Activation Functions ([use ReLU](#))
- Data Preprocessing ([images: subtract mean](#))
- Weight Initialization ([use Xavier init](#))
- Batch Normalization ([use](#))
- Babysitting the Learning process
- Hyperparameter Optimization  
([random sample hyperparams, in log space when appropriate](#))

## TLDRs

# 정리

- 매개변수 갱신 방법에는 확률적 경사 하강법(SGD) 외에도 모멘텀, AdaGrad, Adam 등이 있다
- 가중치 초기값을 정하는 방법은 올바른 학습을 하는데 매우 중요하다
- 가중치 초기값으로는 ‘Xavier 초기값’과 ‘He 초기값’이 효과적이다
- 배치 정규화를 이용하면 학습을 빠르게 진행할 수 있으며, 초기값에 영향을 덜 받게 된다
- 오버피팅을 억제하는 정규화 기술로는 가중치 감소와 드랍아웃이 있다
- 하이퍼파라미터 값 탐색은 최적 값이 존재할 법한 범위를 점차 좁히면서 하는 것이 효과적이다



박 은 수 Research Director

E-mail : es.park@modulabs.co.kr