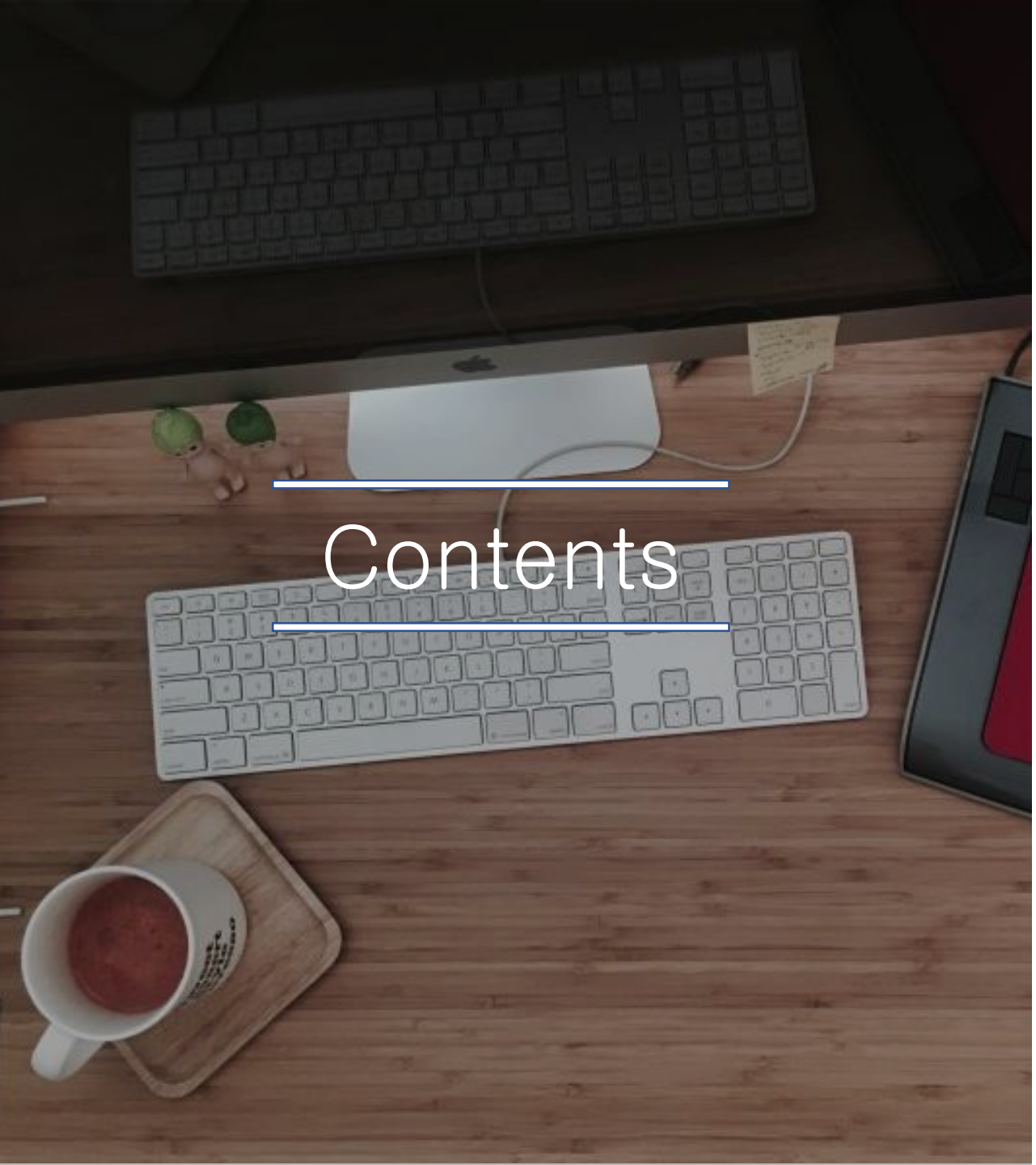
인공지능(Artificial Intelligence)

인하대학교



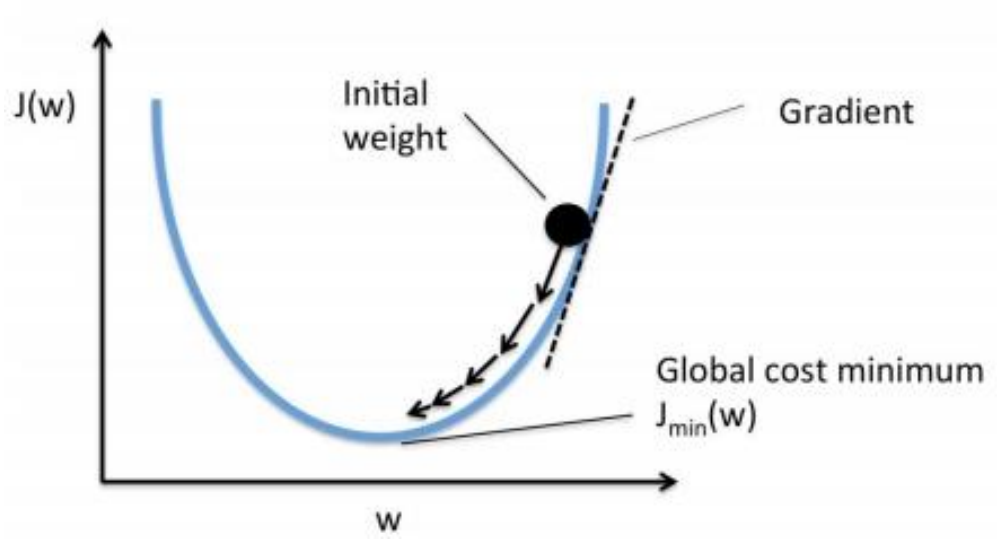
Contents

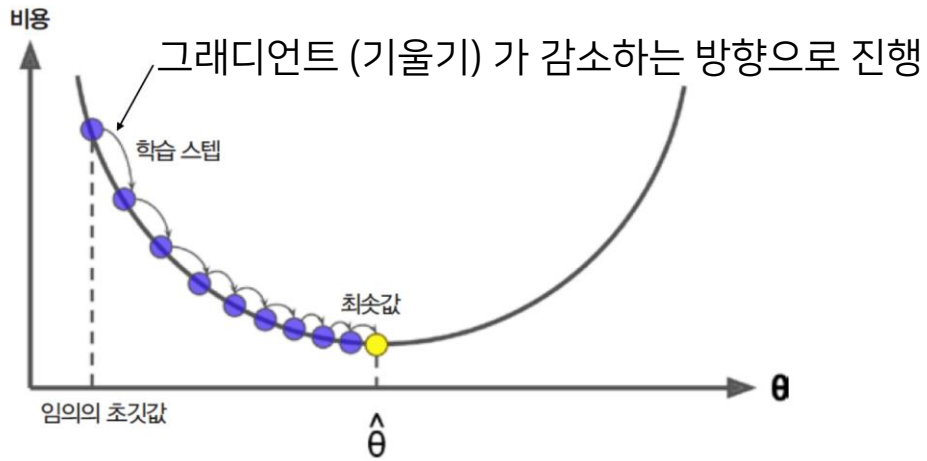
Let's Recap

경사 하강법(Remind)

- Linear Regression(4주차 복습!)

  경사하강법(Gradient Descent!)





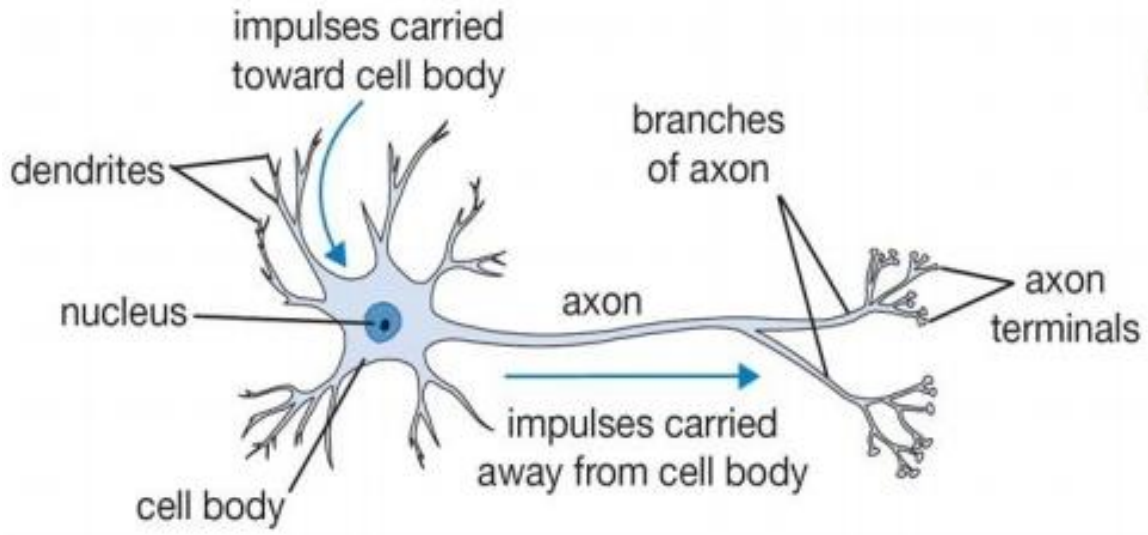
논리회로

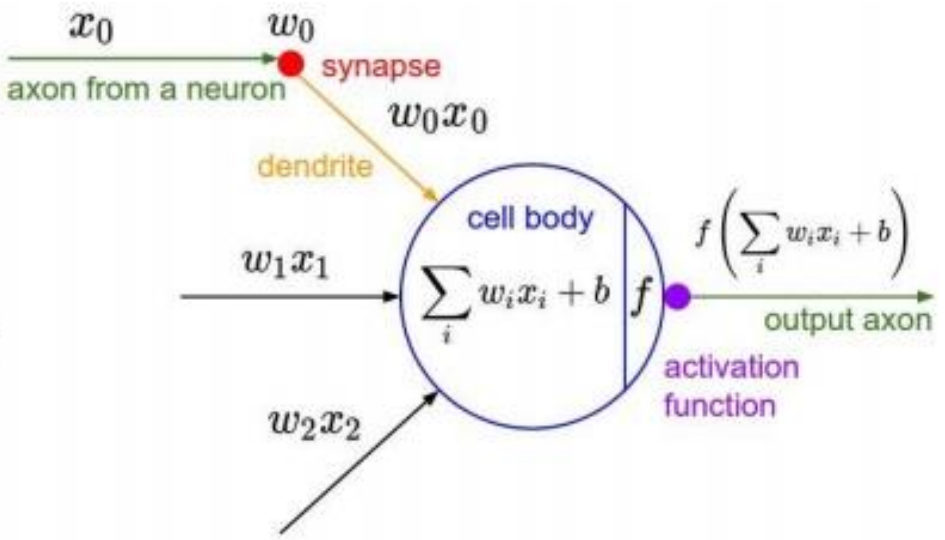
- **학습과정**

1. 임의의 w와 b를 설정
2. 주어진 훈련 데이터를 이용하여, 결과값(y)를 도출
3. 결과값(y)와 실제 결과값($\hat{y}$) 사이의 오차 계산
4. 오차(Loss)를 줄이는 방향으로 w와 b를 재설정(학습)
5. 오차를 최소한으로 줄이도록 2~4의 과정을 계속반복진행

퍼셉트론이란?

- 퍼셉트론(Perceptron, 1957): 뇌의 신경(Neuron)을 모델화
- 다수의 신호를 입력으로 받아 하나의 신호를 출력한다
- 퍼셉트론은 신호 1과 0을 가질 수 있고 1은 신호 흐름, 0 은 신호 흐르지 않음을 의미





여러 자극이 들어오고 일정 기준을 넘으면 이를 다른 뉴런에 전달하는 구조

(출처 : http://cs231n.github.io/neural-networks-1/)

Neural Network

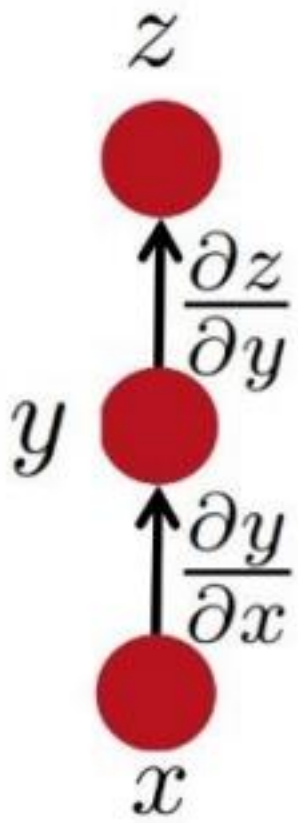
만약 activation function 이 없다면 아래의 식은 결국 linear function.

$$y = w4(act\left(w3\left(act(w2(act(w1 * input + b1)) + b2)\right) + b3)\right) + b4$$

퍼셉트론이 끝난뒤 activation function 으로 non-linearity 를 추가해야 함

Forward & Back Prop.

Simple Chain Rule



$$\Delta z = \frac{\partial z}{\partial y}\Delta y$$

$$\Delta y = \frac{\partial y}{\partial x}\Delta x$$

$$\Delta z = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x$$

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

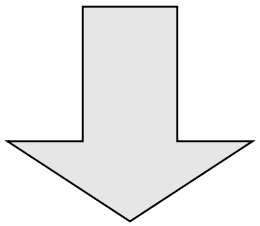
(출처 :https://www.slideshare.net/roelofp/041114-dl-nlpwordembeddings)

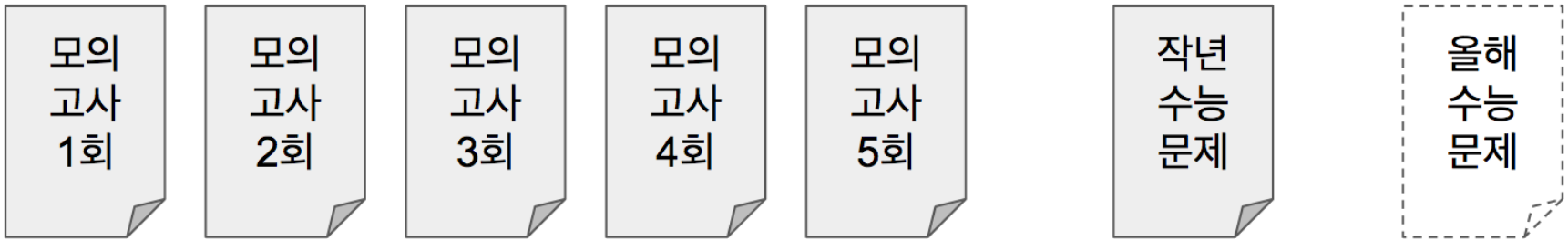
1 데이터 주도 학습

훈련데이터와 시험데이터

훈련 데이터(training data), 시험 데이터(test data)로 나누어 학습과 실험을 수행
=> Why?

- ≫ 범용능력을 기르기 위해!
- ≫ 오버피팅(Overfitting)!



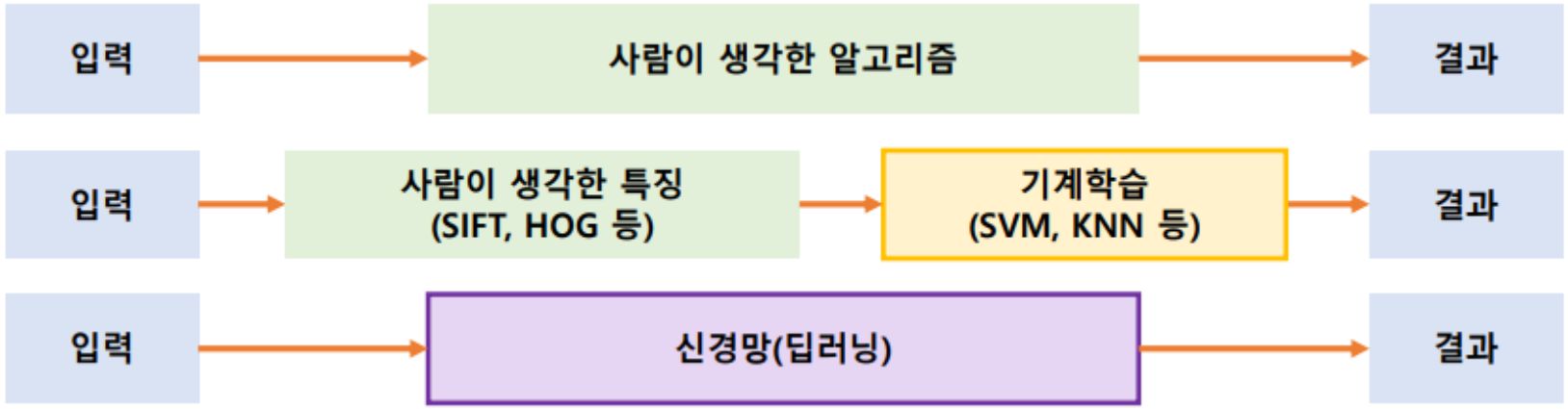


기계학습과 딥러닝의 차이점

- 기울기 소실에 나가기 앞서...
- 기계학습은 **데이터**로부터 모델을 만들어 나간다
- 만약 손글씨 이미지 데이터를 인식하기 위한 알고리즘을 기계학습을 이용해서 구현한다면?

✓**기계학습의 두 가지 접근법**

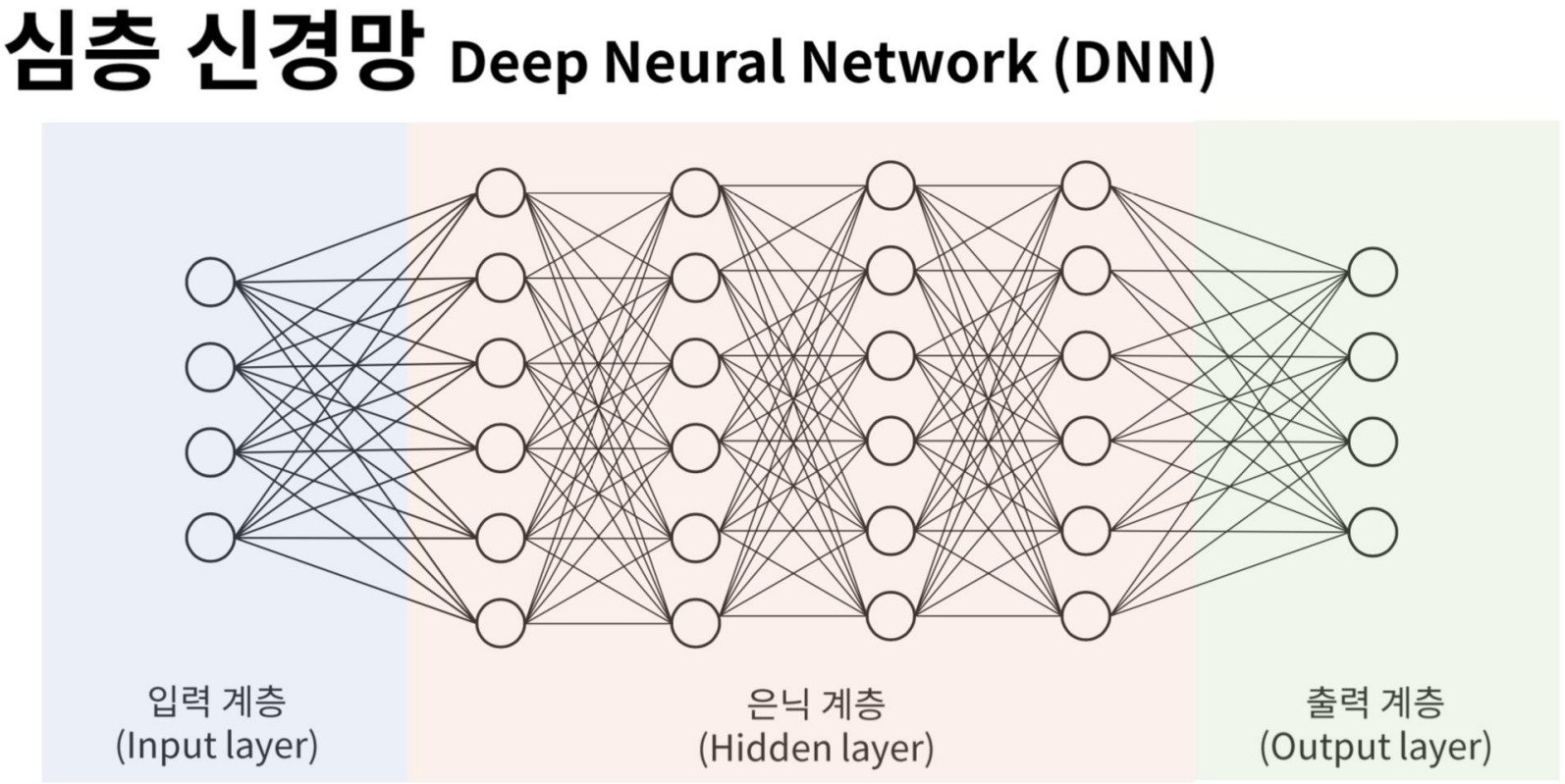
1. 전통적 방법: 이미지에서 특징 추출(사람) → 특징의 패턴을 기계학습(기계)
2. 딥러닝: 데이터(입력)에서 결과(출력)를 사람의 개입 없이 얻는다는 뜻으로, 종단간 기계학습 (End-to-end learning)이라고도 함



2 심층신경망(딥러닝:Deep Learning))

심층신경망(딥러닝, Deep Learning)

- 은닉층을 많이 쌓으면 성능이 좋아지는가?



- 얕은 신경망보다 은닉 계층이 많은 신경망을 DNN이라고 부른다.
- 보통 5개 이상의 계층이 있는 경우 '깊다' (Deep 하다)라고 표현

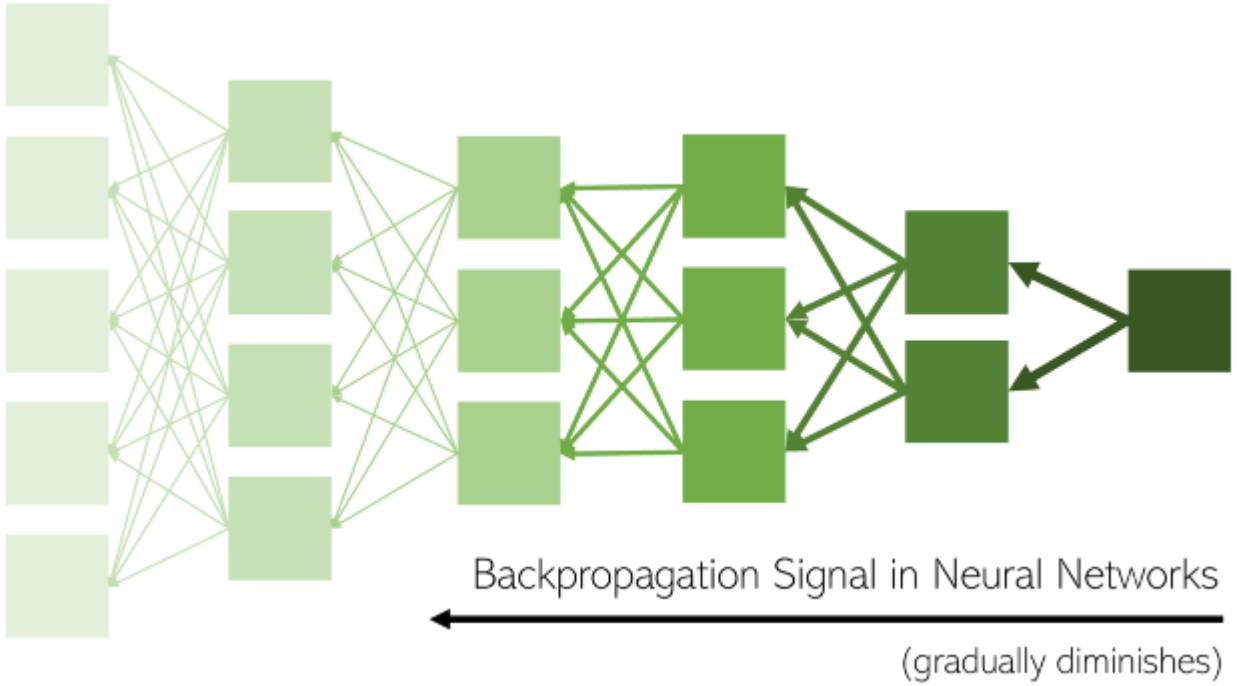
심층신경망(딥러닝, Deep Learning)

- 학습이 잘 안 된다.

1. 잘못된 활성화 함수
2. 불안정한 학습과정
3. 초기값 설정

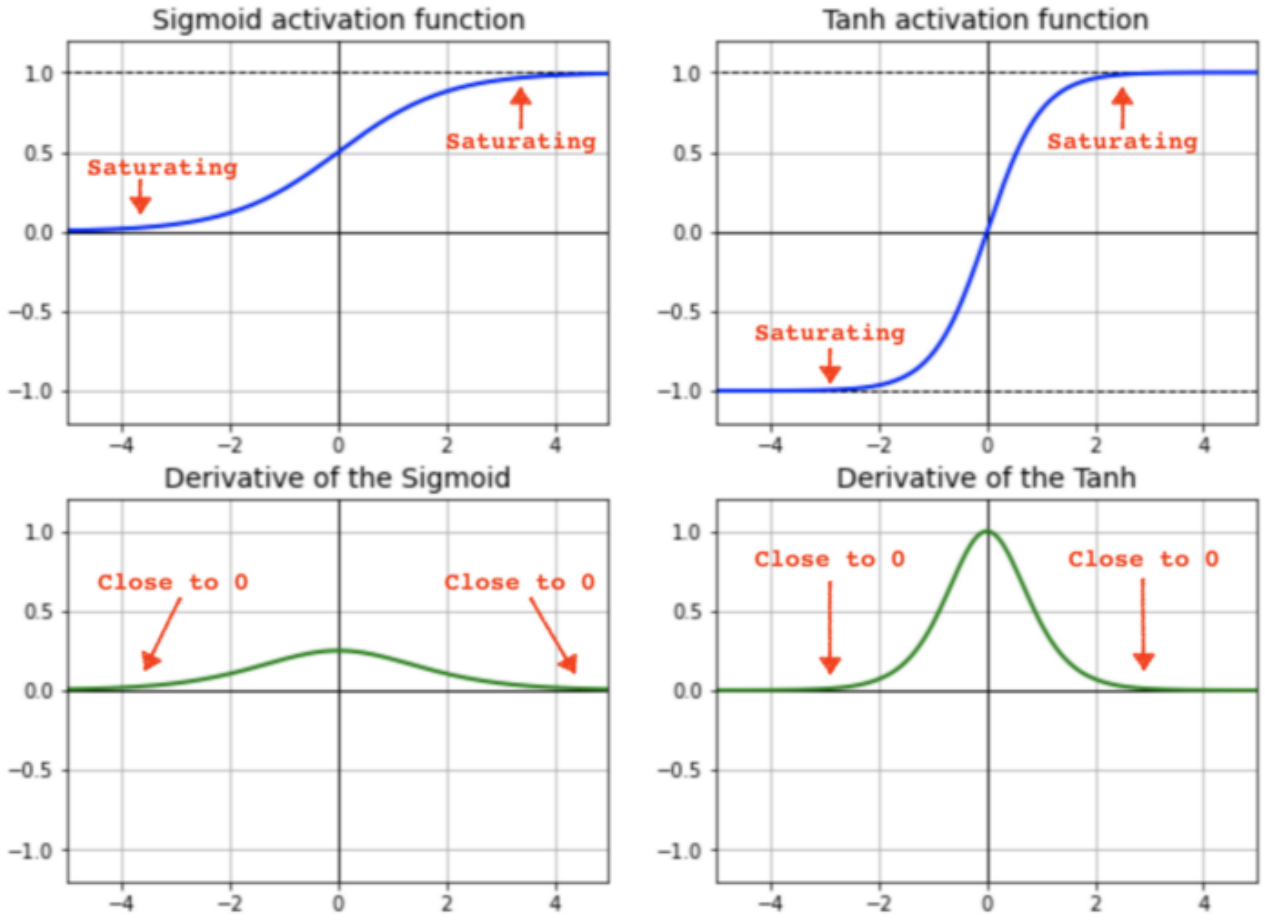
기울기 소실

- 문제의 발생
  : 은닉층이 깊어질수록 출력 신호의 미분의 값이 **입력 신호까지 전달이 안되는 현상**이 발생



잘못된 활성화 함수

- 문제의 원인: 활성화 함수는 **미분**할수록 **값이 작아지기 때문**에



$$*W_x = W_x - a\left(\frac{\partial Error}{\partial W_x}\right)$$

Old weight

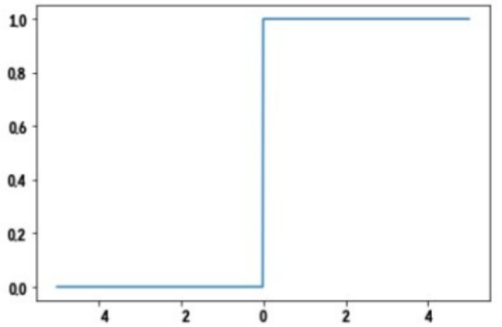
Derivative of Error with respect to weight

New weight

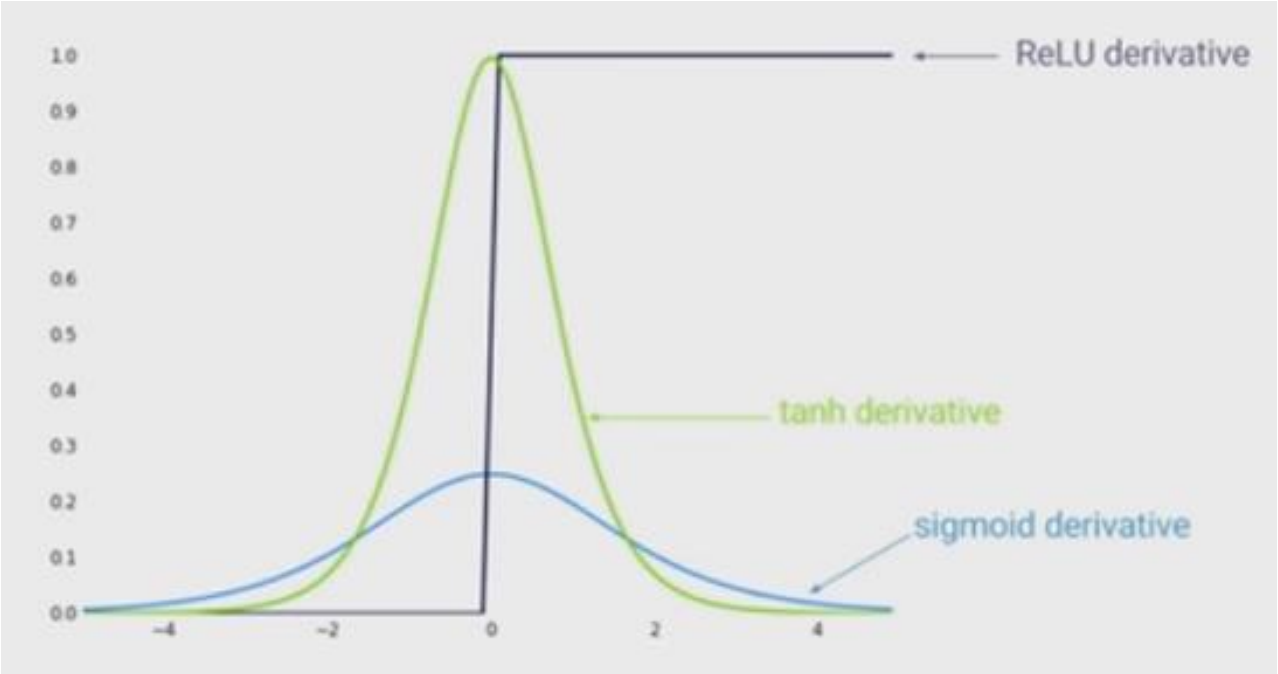
Learning rate

잘못된 활성화 함수

- **활성화 함수 ReLU사용**
  - ✓입력 > 0 일 때 기울기 1, 그렇지 않으면 0
  - ✓학습이 빠르고 컴퓨팅 자원도 덜 소모함(미분값이 0, 1 두 개 중 하나)

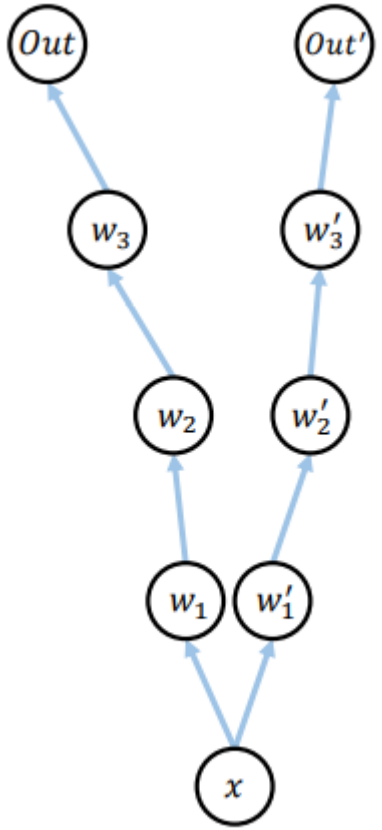


$$R'(y) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$



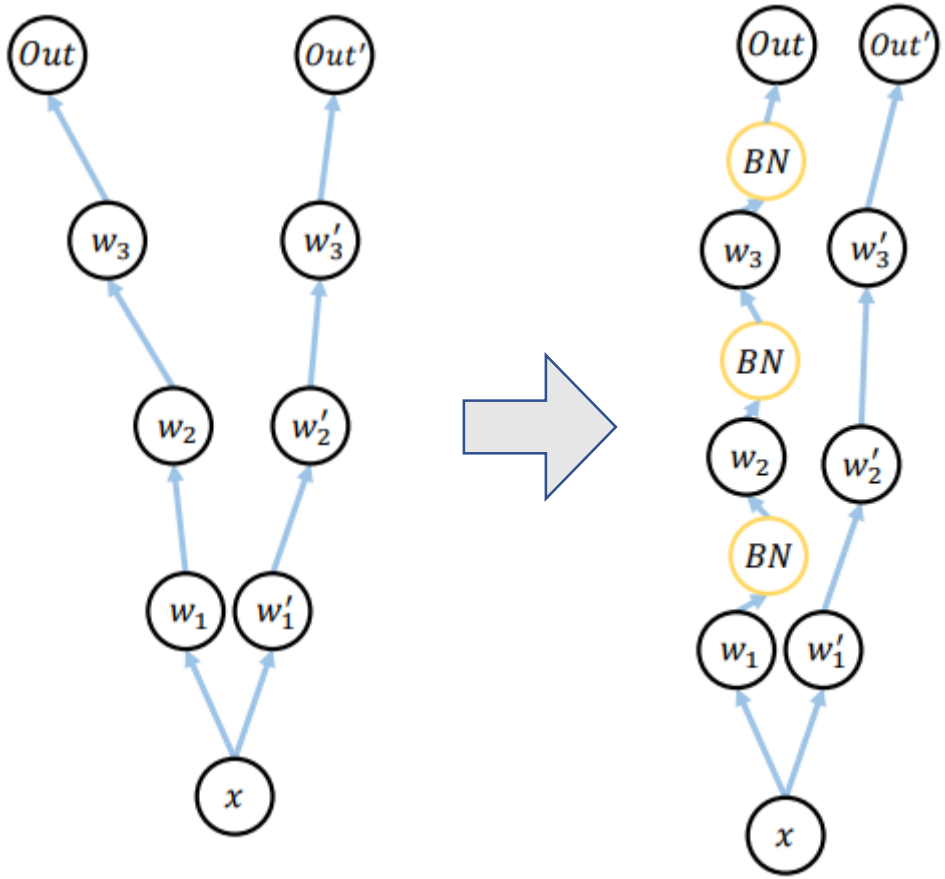
불안정한 학습과정

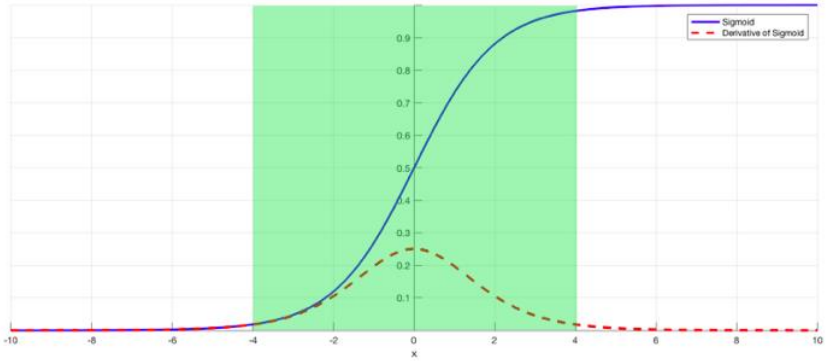
- 입력이 같아도 가중치가 조금씩 달라지면 **출력이 완전히 달라지는 경우**가 생길 수 있음



불안정한 학습과정

- 각 확설함수의 출력값을 정규화 함으로써 학습 시 모델을 더 안정화할 수 있음



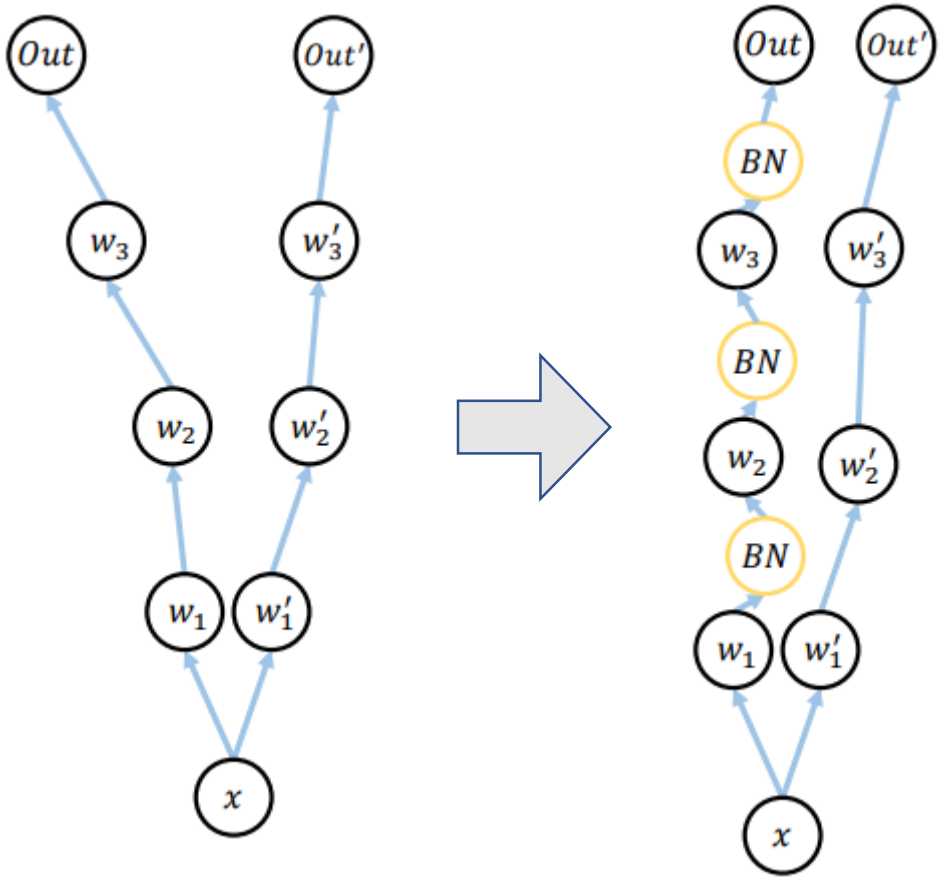


BN: What do you get?

- Very deep nets are much easier to train → more stable gradients
- A much larger range of hyperparameters works similarly when using BN

불안정한 학습과정

- 학습 시 옮긴 노드 간 이동 정도를 저장해놓고 테스트 시 transfer하여 사용
- 학습과정의 안정화를 통해 기울기 소실을 해결하는 근본적인 방법



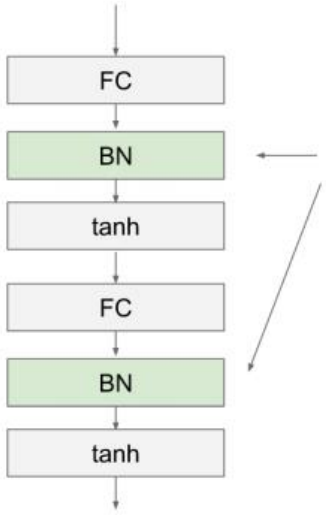
BN: What do you get?

- Very deep nets are much easier to train → more stable gradients
- A much larger range of hyperparameters works similarly when using BN

HCI Lab
Inha University
Human-Computer Interaction Lab

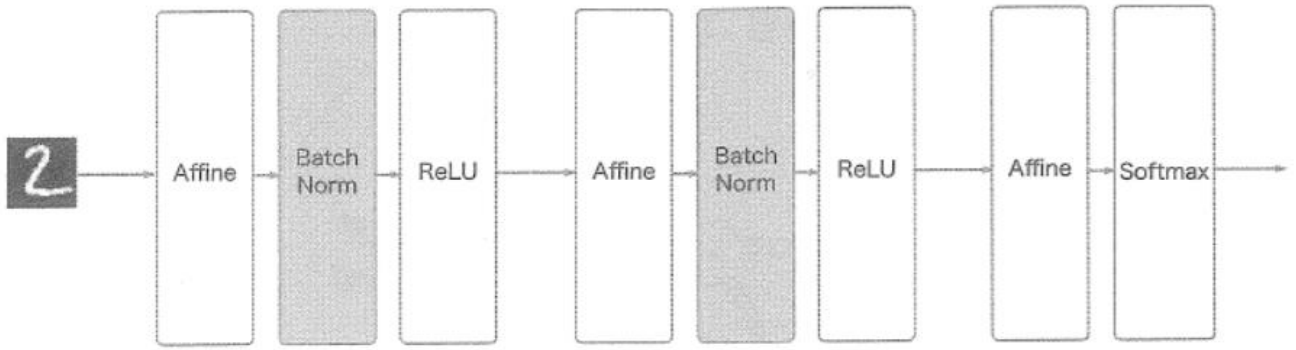
불안정한 학습과정

- 배치 정규화 사용 예



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

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

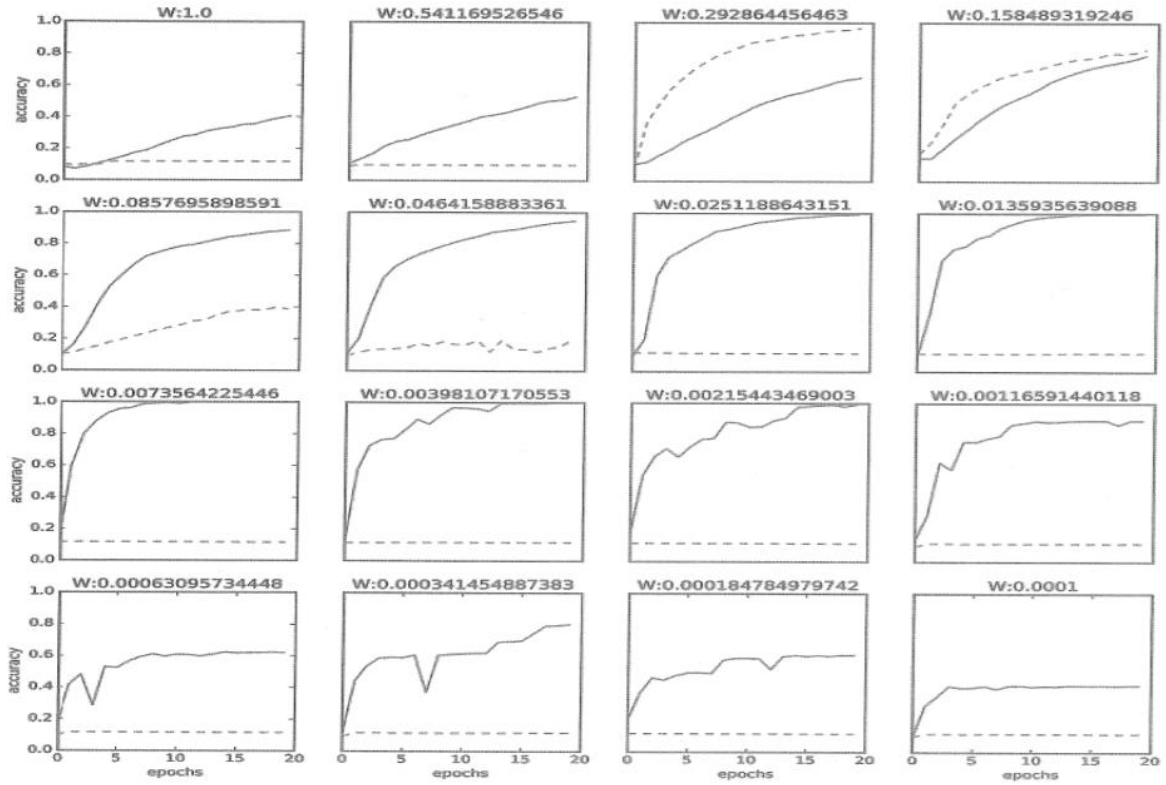


-> 활성화 함수 전달 이전에 미니배치단위로 조정

불안정한 학습과정

- **배치 정규화 적용 효과**

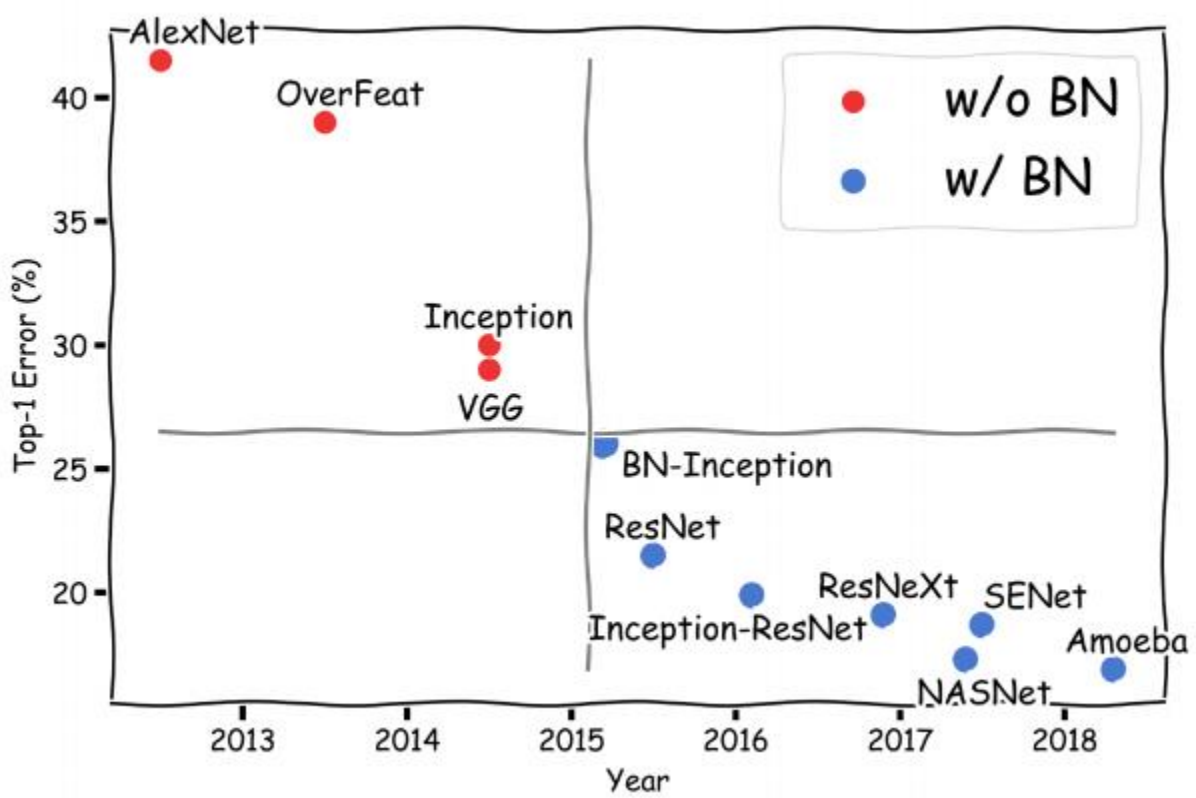
  - ✓ 기울기가 안정적임(학습이 쉬워짐)
  - ✓ 더 높은 학습속도 가능
  - ✓ 큰 범위 하이퍼 파라미터 가능
  - ✓ 초기화에 대한 강한 의존도 감소
  - ✓ 드랍아웃의 필요성 줄어듦



HCI Lab
Inha University
Human-Computer Interaction Lab

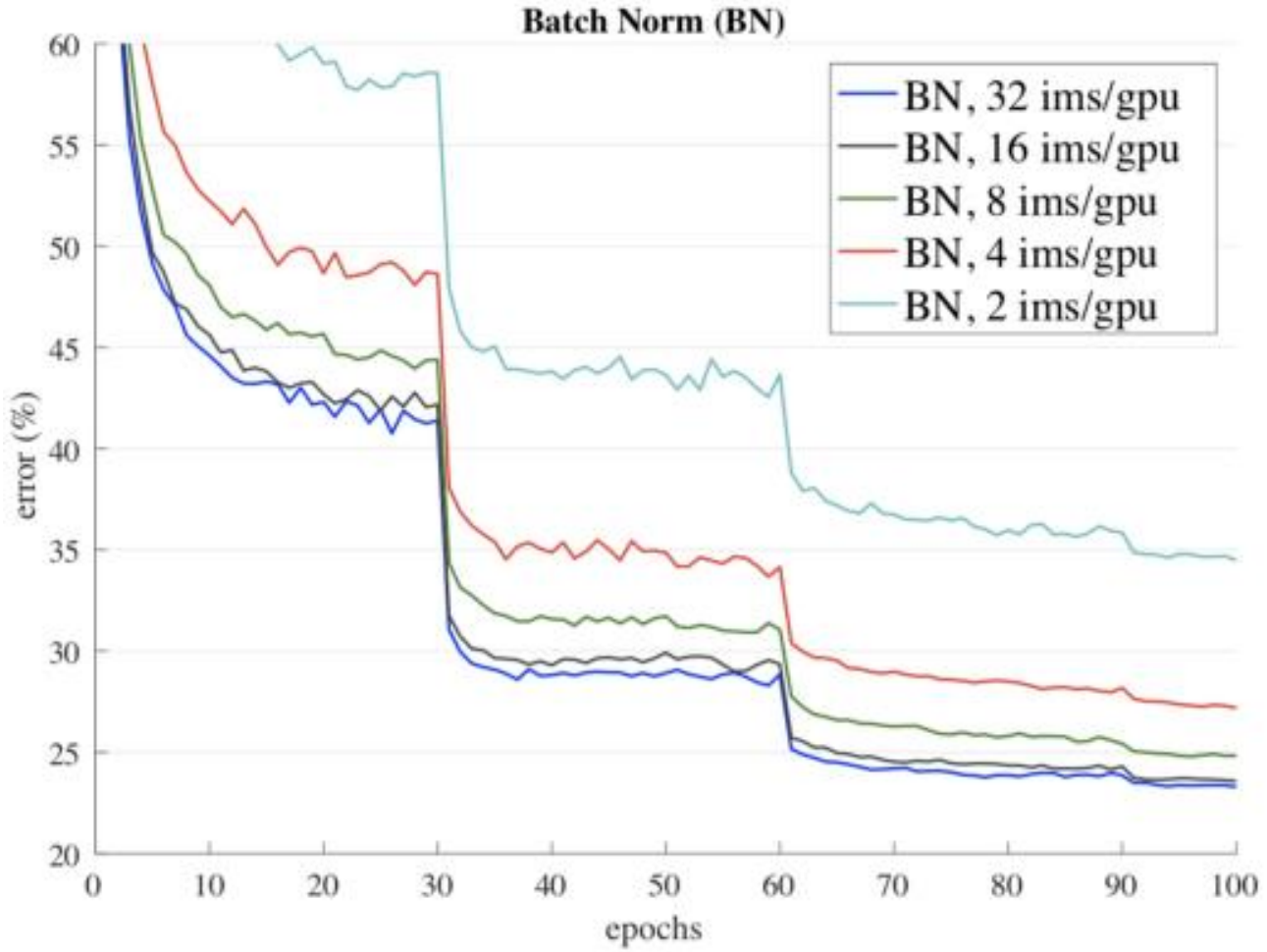
불안정한 학습과정

- 배치 정규화 적용 효과



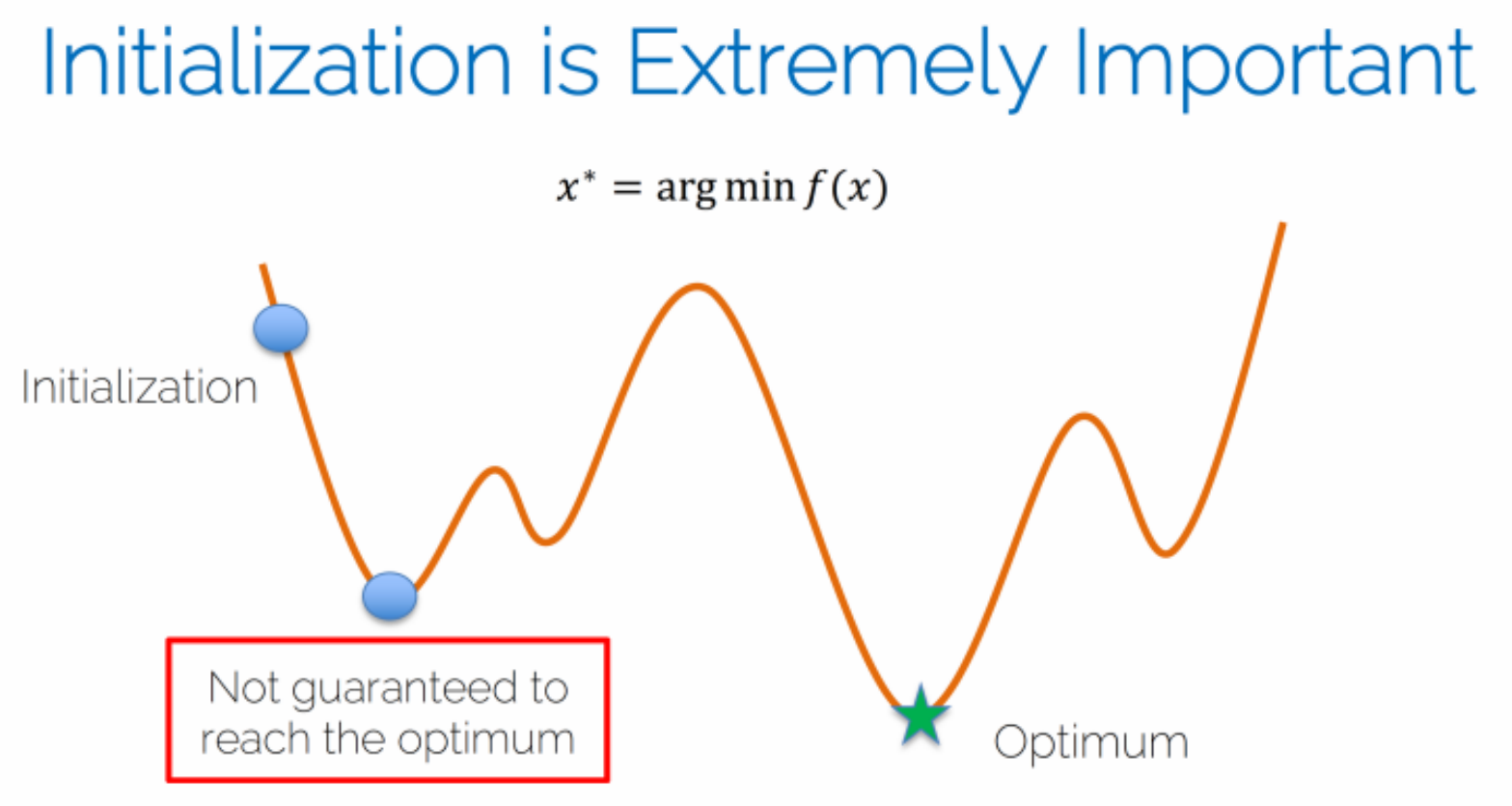
불안정한 학습과정

- 배치 정규화 적용 효과(2의 단위로 적용)



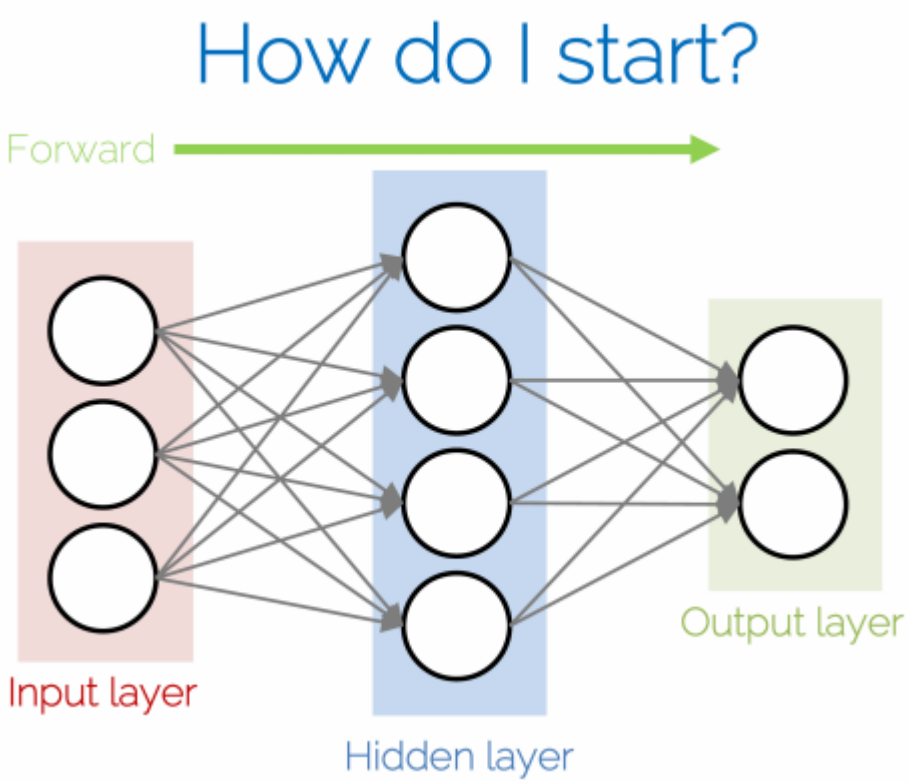
초기값 설정

- 가중치 초기값을 잘못했을 때 예시



초기값 설정

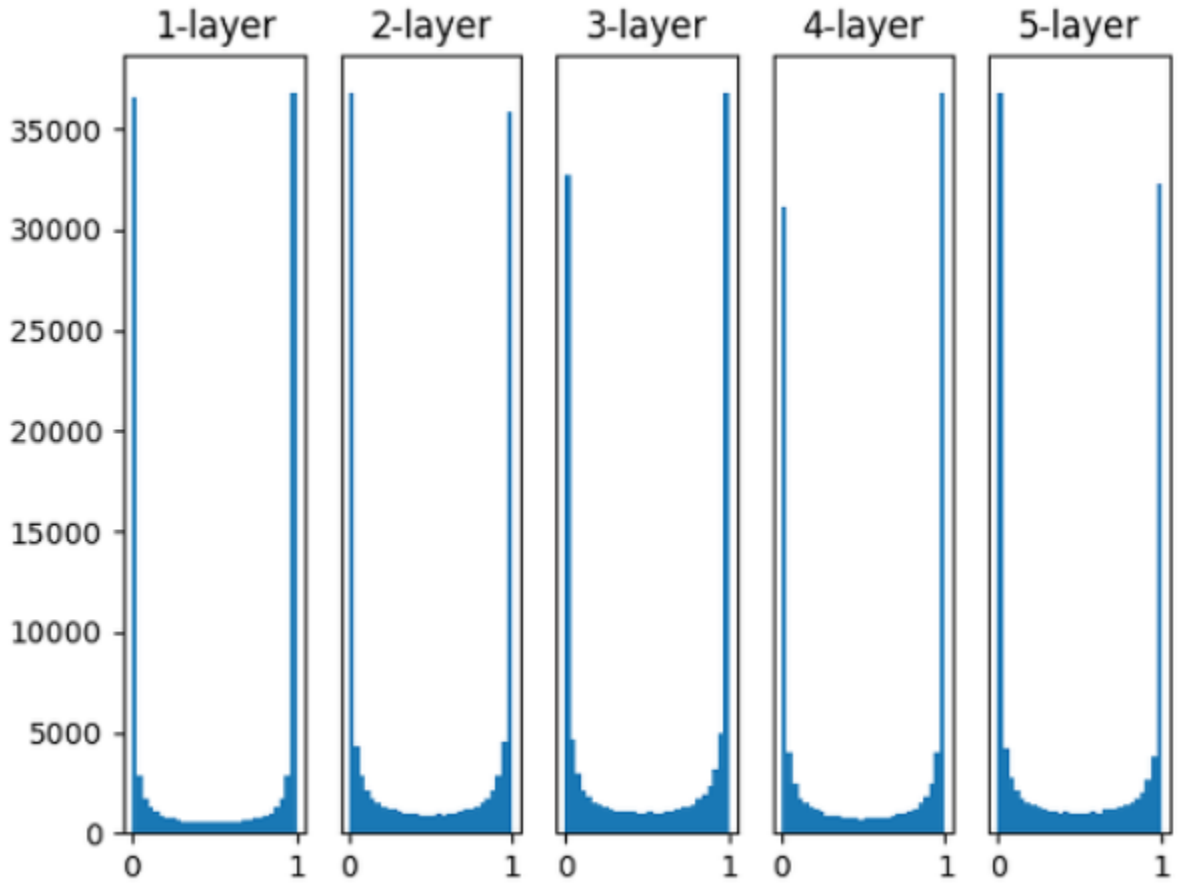
- 초기값을 그럼 어떻게 설정할까요?



- **가중치의 초깃값**을 무엇으로 설정하느냐에 따라서 신경망 학습의 성공 요인 중 하나

초기값 설정

- 초기값 평균 0, 표준편차가 1인 정규분포로 초기화한 경우
  - 활성화 함수의 값이 0, 1 양 끝단에 몰리게 되면서 기울기가 점점 작아지는 **기울기 소실**(Gradient Vanish 현상이 일어남)



초기값 설정

- 원인: 초기의 활성화 함수로 사용된 Sigmoid, TanH함수가 입력을 작은 출력값으로 비선형적으로 만들었기 때문에 발생하는 문제

  ex) TanH [1, 3, 5, 7, 9] = [0.76...,0.995.., 0.999..., 0.999..., 0.999...]로 큰 출력값이 발생

✓이를 통해 나온 아이디어 : 초기화시 **가중치를 작게** 만들어보자!

초기값 설정

- 그렇기 때문에 초기에 딥러닝 연구에는 작은 임의의 변수로 설정하여 학습!

```
W = 0.01 * np.random.randn(Din, Dout)
```

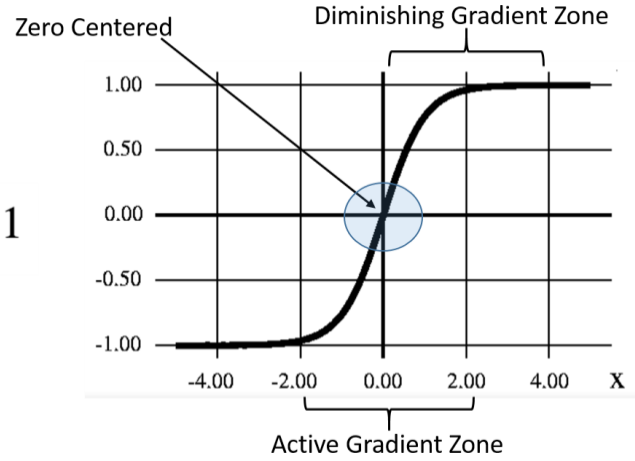
문제) 다음의 입력 16, tanH활성화 함수인 7개의 은닉층을 가진 MLP의 마지막 층은 어떻게 되었을 까요?

단, 입력은 Gaussian Distribution을 따른다.

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$$



초기값 설정

- 초기에 딥러닝 연구에는 작은 임의의 변수로 설정하여 학습!

```
W = 0.01 * np.random.randn(Din, Dout)
```

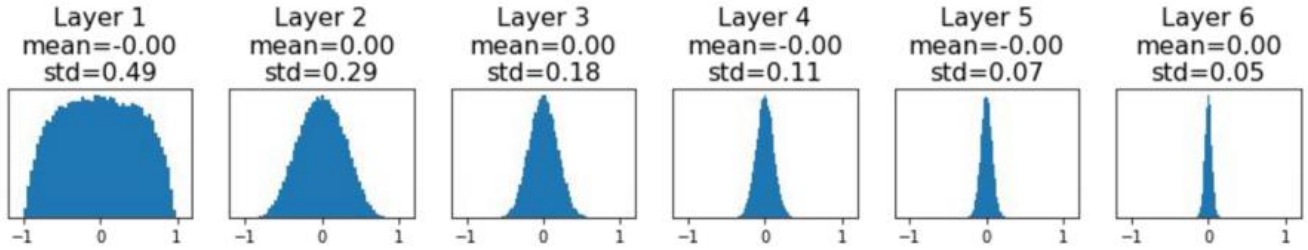
문제) 다음의 입력 16, tanH활성화 함수인 7개의 은닉층을 가진 MLP의 마지막 층은 어떻게 되었을 까요?

단, 입력은 Gaussian Distribution을 따른다.

정답) 표준편차가 0.01로 작기 때문에 기울기 소실은 일어나지 않는다.
하지만, 초기화값이 0주변에 많이 몰려있어 표현력이 제한되어 뉴런(노드)들이 다양한 값을 표현하지 못하기 때문에 여러 개 두는 의미가 없어져버린다. 즉, 활성화값이 특정 부근에 몰려있어 표현력이 제한됨!

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096



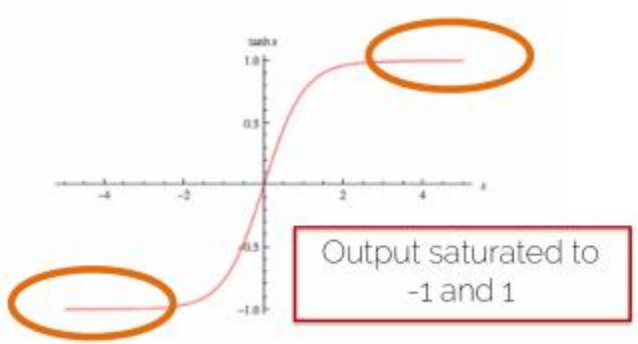
HCI Lab
Inha University
Human-Computer Interaction Lab

초기값 설정

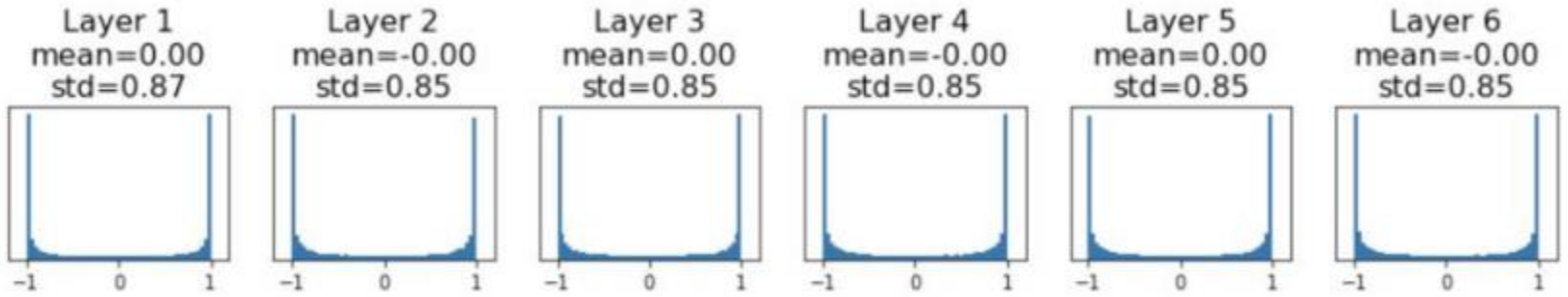
- 가중치를 0.05로 약간 증가시킨다면?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05



답: 기울기가 0으로 감소되어 학습이 안된다.



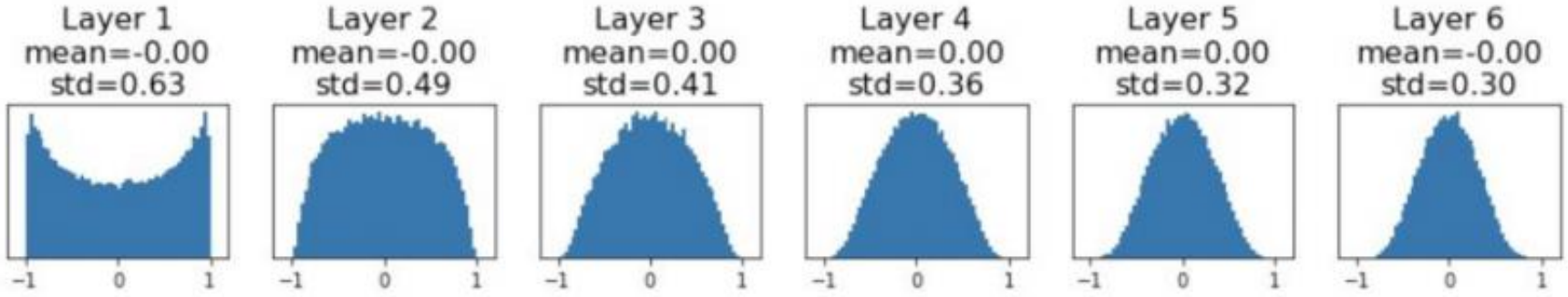
초기값 설정(Xavier Init)

- 가중치 표준편차 = **Xavier 초깃값**
  → n개의 노드(뉴런)개일때, 표준편차가 '$1/\sqrt{n}$'인 정규분포 사용
  (합성곱 층이라면, 필터사이즈의 제곱*채널수)

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

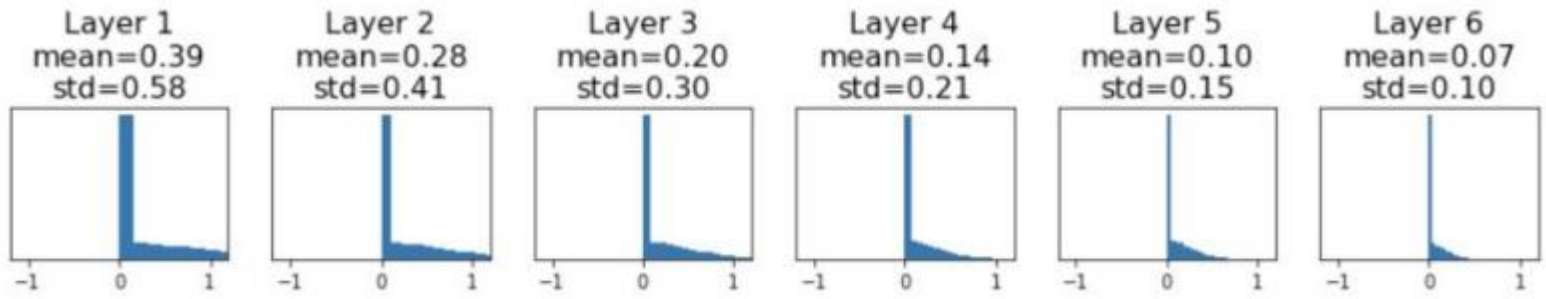
초기값 설정

- 활성화 함수가 ReLU 라면?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

- Xavier 초기화 방법은, 값들이 0에 중심되어 있지만, ReLU의 경우는 0으로 몰려있기 때문에 학습이 안되는 현상이 발생



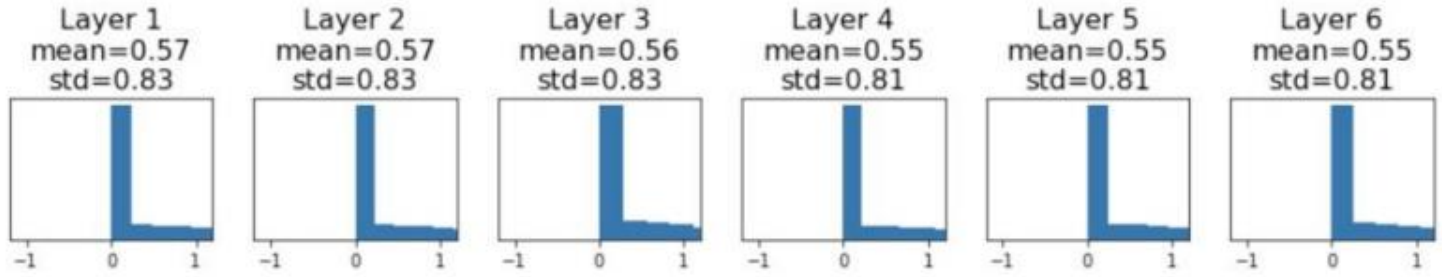
초기값 설정(He Init)

- 활성화 함수가 ReLU 라면?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!

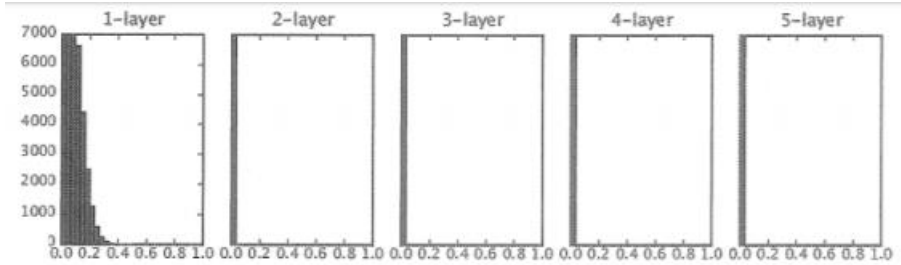


He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

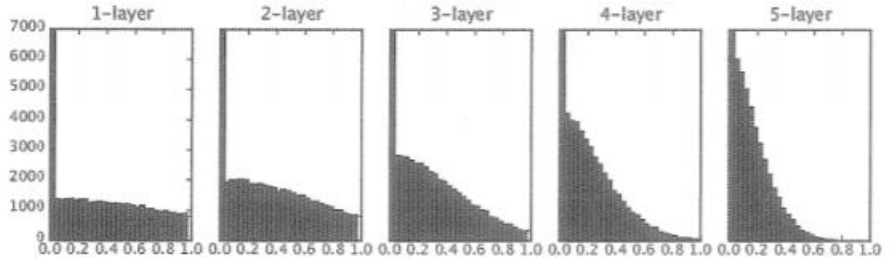
- He 초깃값
  ->n개의 노드(뉴런)라면 '$\sqrt{2}/\sqrt{n}$'의 표준편차 사용
  ->ReLU의 불필요한 영역(음의 영역) 처리

초기값 설정

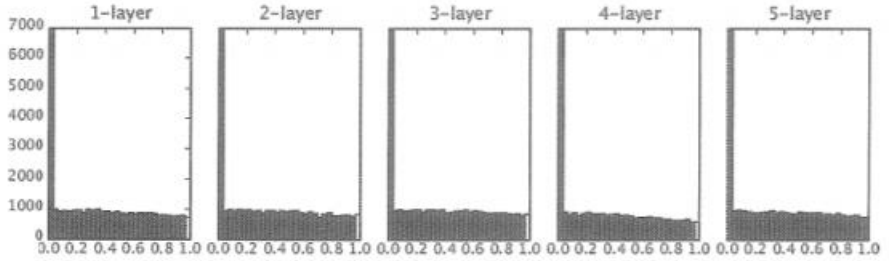
- ReLU함수 사용시 각 가중치 초깃값 별 활성화값 분포



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

초기값 설정

- 요약
- 활성화 함수 Sigmoid, TanH보다는 ReLU(한줄..)
- 배치정규화 사용 (한줄..)
- 초기화할때 Xavier혹은 He (한줄..)

```
>>> m = nn.ReLU()
>>> input = torch.randn(2)
>>> output = m(input)
```

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = torch.randn(20, 100, 35, 45)
>>> output = m(input)
```

```
def initialize_weights(m):
  if isinstance(m, nn.Conv2d):
      nn.init.kaiming_uniform_(m.weight.data,nonlinearity='relu')
      if m.bias is not None:
          nn.init.constant_(m.bias.data, 0)
  elif isinstance(m, nn.BatchNorm2d):
      nn.init.constant_(m.weight.data, 1)
      nn.init.constant_(m.bias.data, 0)
  elif isinstance(m, nn.Linear):
      nn.init.kaiming_uniform_(m.weight.data)
      nn.init.constant_(m.bias.data, 0)
```

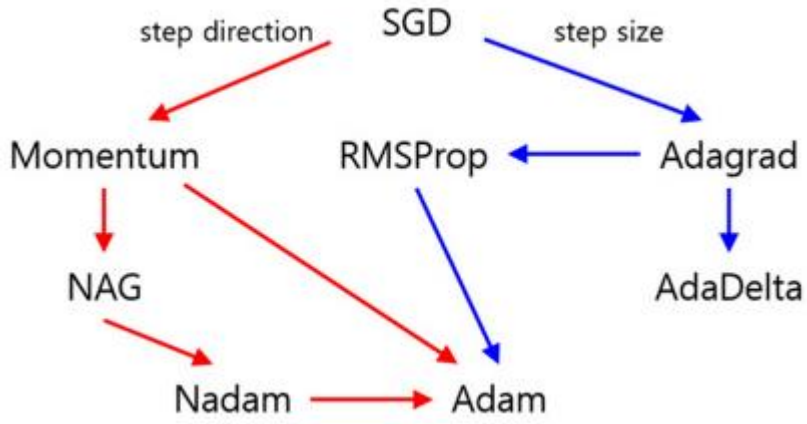
```
model=CNN()
model.apply(initialize_weights)
```

```
if isinstance(m, nn.Conv2d):
    '''
    # 작은 숫자로 초기화하는 방법
    # 가중치를 평균 0, 편차 0.02로 초기화합니다.
    # 편차를 0으로 초기화합니다.
    m.weight.data.normal_(0.0, 0.02)
    m.bias.data.fill_(0)

    # Xavier Initialization
    # 모듈의 가중치를 xavier normal로 초기화합니다.
    # 편차를 0으로 초기화합니다.
    init.xavier_normal(m.weight.data)
    m.bias.data.fill_(0)
    '''

    # Kaming Initialization
    # 모듈의 가중치를 kaming he normal로 초기화합니다.
    # 편차를 0으로 초기화합니다.
    init.kaiming_normal_(m.weight.data)
    m.bias.data.fill_(0)
```

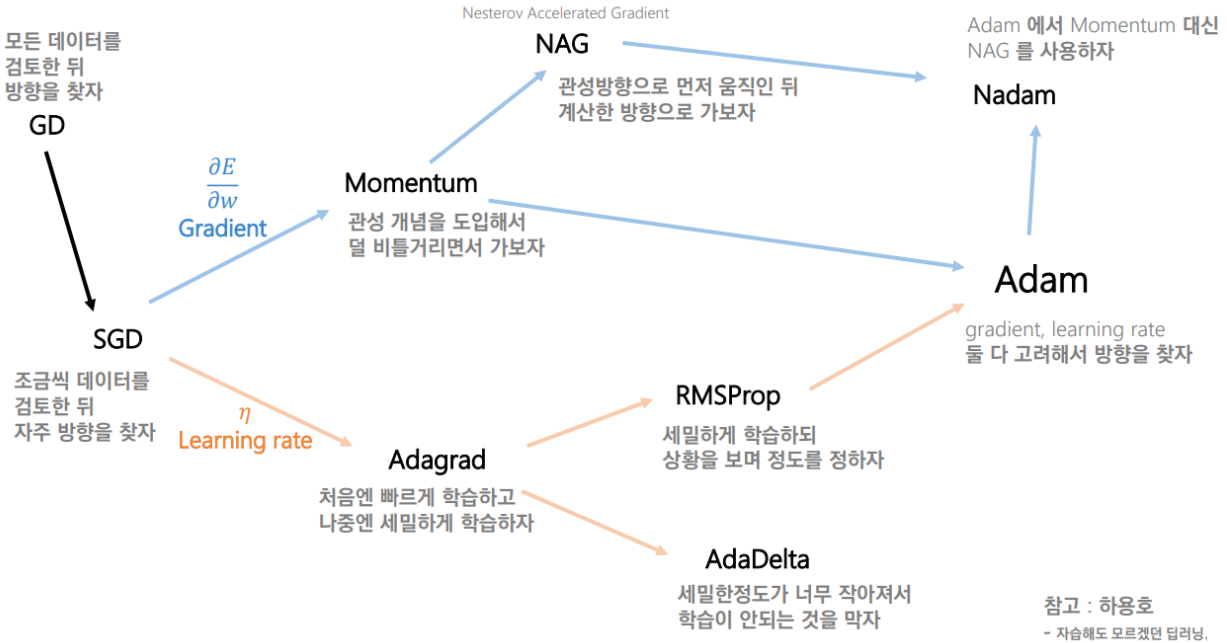
https://androidkt.com/initialize-weight-bias-pytorch/

3 최적화 함수

가중치 매개변수의 최적화 함수



- 가중치 학습 파라미터 갱신
  - 신경망 학습의 목적
    →학습 파라미터 최적값 찾기 = **최적화(Optimization)**
- **아담(Adam)**이 상수의 학습률에서는 많은 경우에서 우수함
- SGD+Momentum은 아담보다 우수할 수 있으나 많은 학습 스케줄링을 요구함



참고 : 하용호
- 자습해도 모르겠던 딥러닝,
머리속에 인스톨 시켜드립니다.

가중치 매개변수의 최적화 함수

- SGD(Stochastic Gradient Descent)
  - 확률적 경사 하강법
  - 매개변수의 기울기(미분)이용하여 점점 최적값에 근접

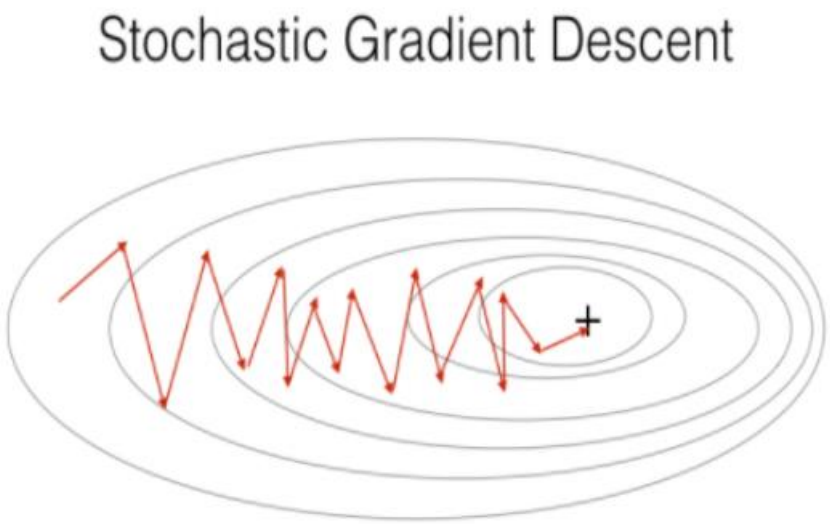
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\frac{\partial L}{\partial \boldsymbol{W}}$$

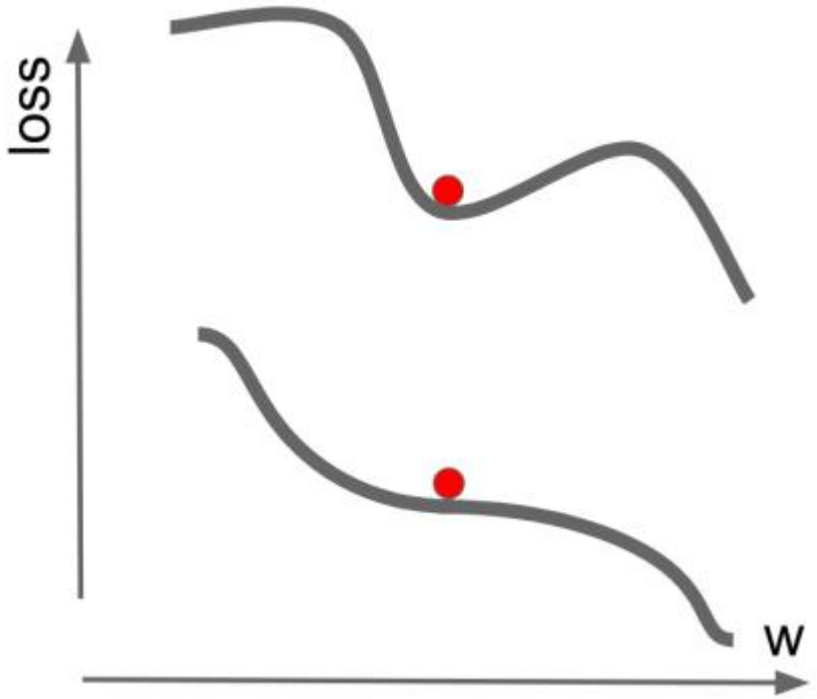
가중치 매개변수의 최적화 함수

SGD의 단점

→ 비효율적인 움직임, 지그재그

→ 안장점의 형태의 경우, 기울기가 0에 가까워져 학습 속도가 매우 느림





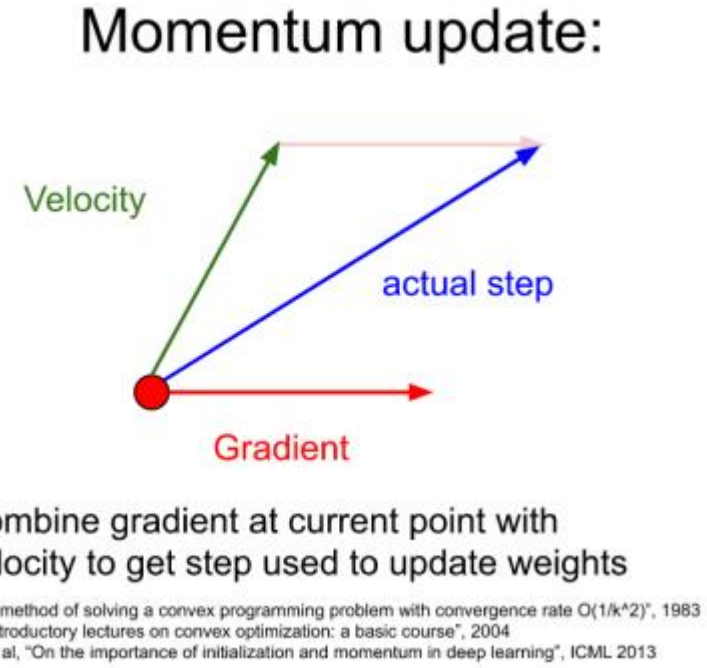
가중치 매개변수의 최적화 함수

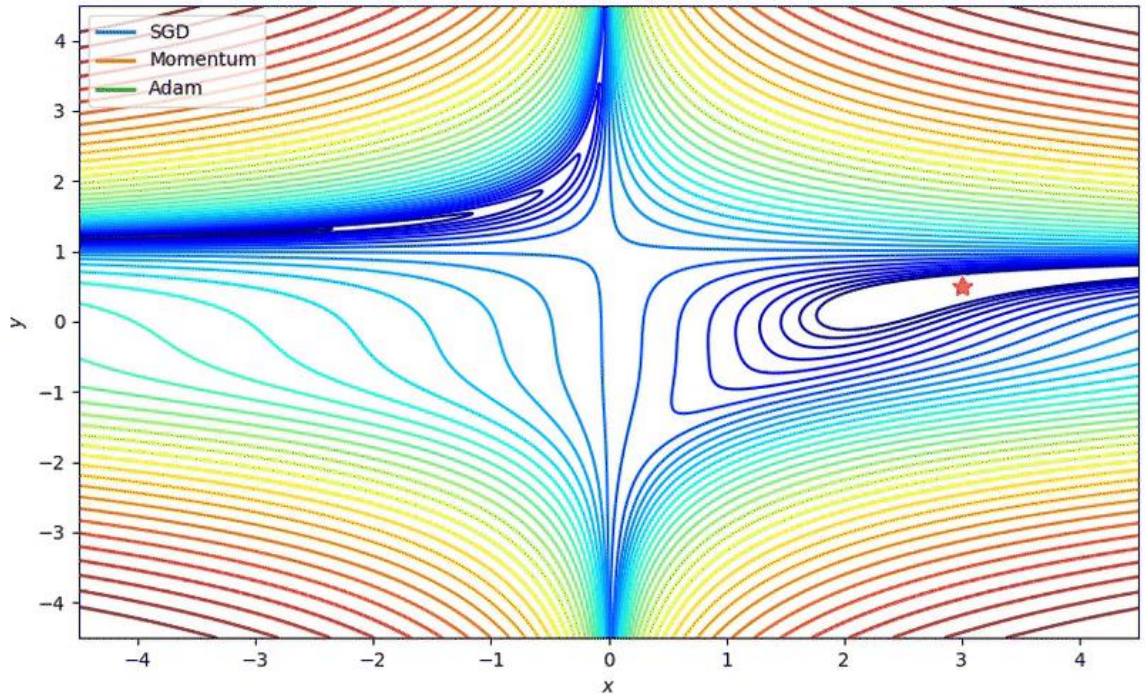
- 좀 더 나은 방법이 없을까?
  → 대표적인 3가지 방법

1. Momentum
2. Nesterov Momentum Update
3. AdaGrad
4. Adam

가중치 매개변수의 최적화 함수

공 굴러가듯, x축으로의 이동 점점 증가





가중치 매개변수의 최적화 함수

- Momentum
  - '운동량', 가속도의 개념 도입

$$v_{t+1} = \rho v_t - \alpha f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

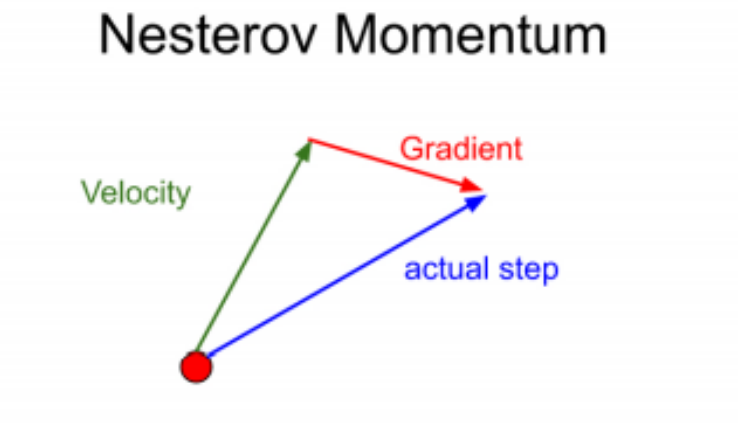
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

가중치 매개변수의 최적화 함수

- Nesterov Momentum
  - 모멘텀이 이동시킬 방향으로 이동해서 기울기 계산

$$v_{t+1} = \rho v_t - \alpha f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$



$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Nesterov Momentum
(NAG, Nesterov Accelerated Gradient)

가중치 매개변수의 최적화 함수

- **AdaGrad: 변수의 업데이트가 잦으면 학습속도를 줄여 이동 보폭을 조절**
  - 각 Element마다 다르게 학습률 감소
  - 학습을 진행함에 따라 학습률 점차 감소 : Adaptive

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\delta L}{\delta \boldsymbol{W}} \odot \frac{\delta L}{\delta \boldsymbol{W}}$$

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

- $\boldsymbol{h}$: 학습을 더해 갈수록 증가
- $\odot$: 행렬 각 원소별 곱

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

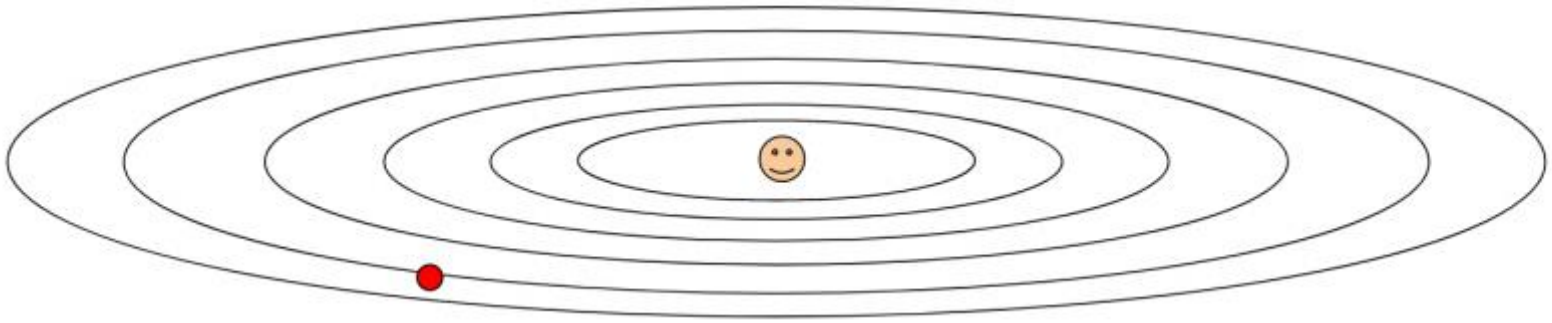
"Per-parameter learning rates" or "adaptive learning rates"

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

가중치 매개변수의 최적화 함수

- 단점

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



학습이 오래됨에 따라 0으로 줄어듬(Decay to zero)

가중치 매개변수의 최적화 함수

- **RMSProp**: decayrate를 도입하여 민감도 보완

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

가중치 매개변수의 최적화 함수

- **Adam**: 모멘텀과 RmsProp을 합침

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

- AdaGrad + RMSProp+ Momentum
- 하이퍼파라미터의 '편향성' 보장

가중치 매개변수의 최적화 함수

- Adam

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

가중치 매개변수의 최적화 함수

- Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

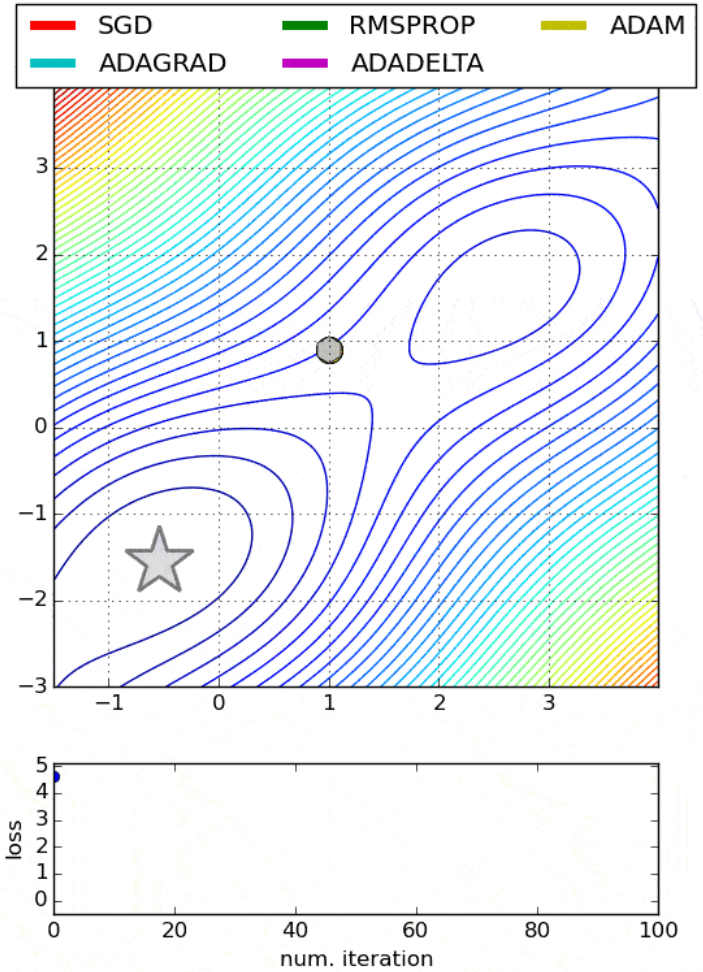
AdaGrad / RMSProp

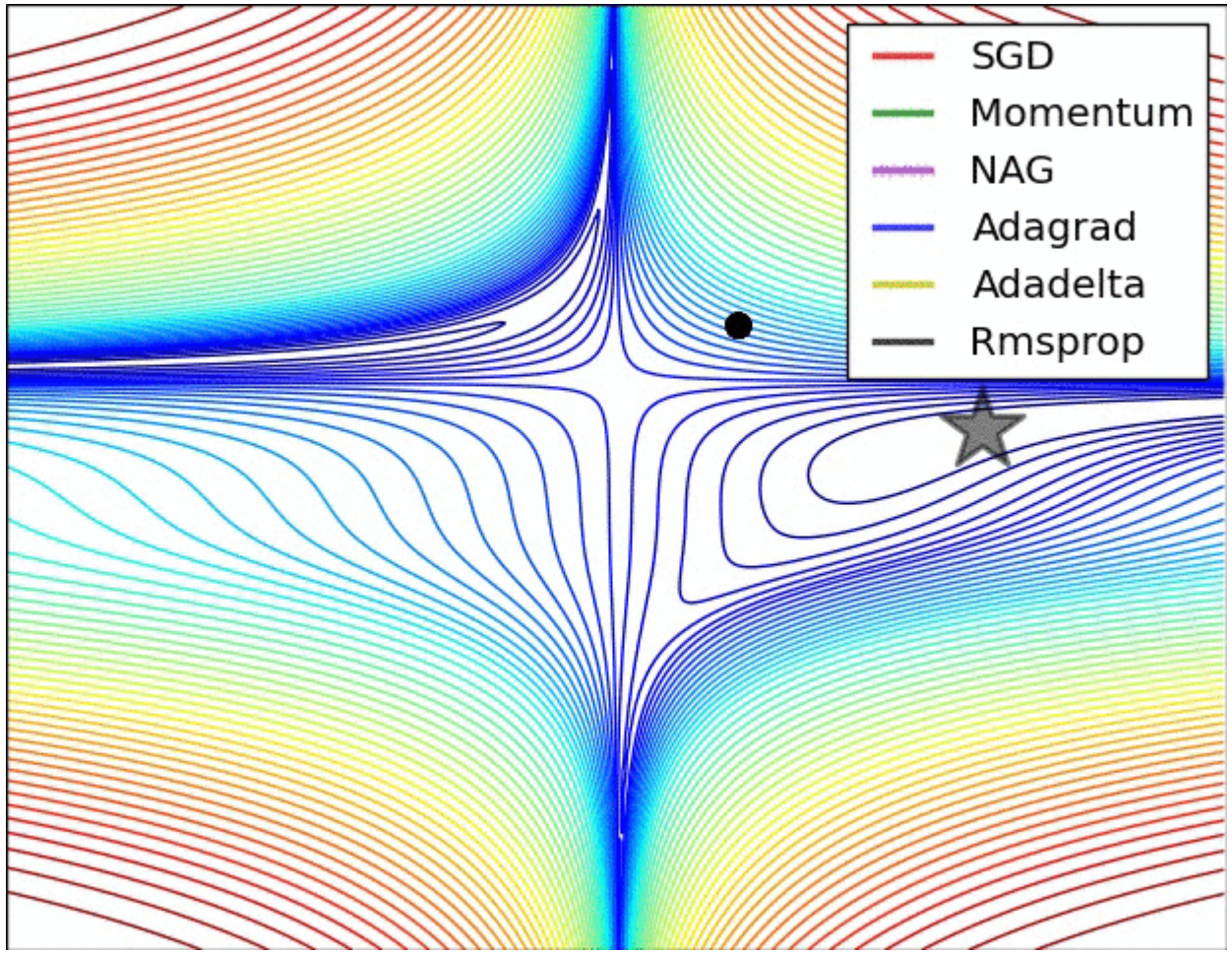
Bias correction for the fact that
first and second moment
estimates start at zero

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

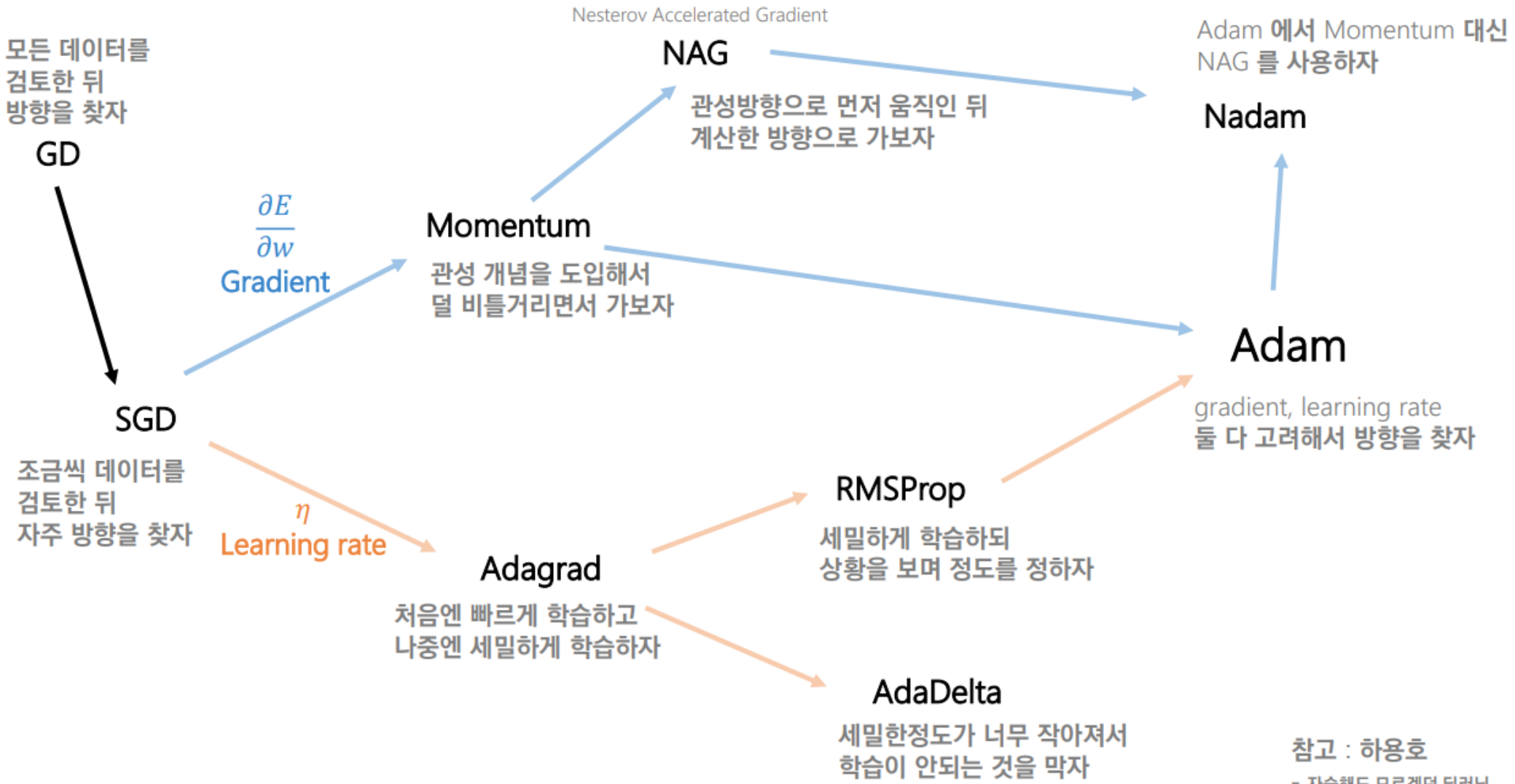
가중치 매개변수의 최적화 함수

- 비교





요약



참고 : 하용호

- 자습해도 모르겠던 딥러닝,
머리속에 인스톨 시켜드립니다.