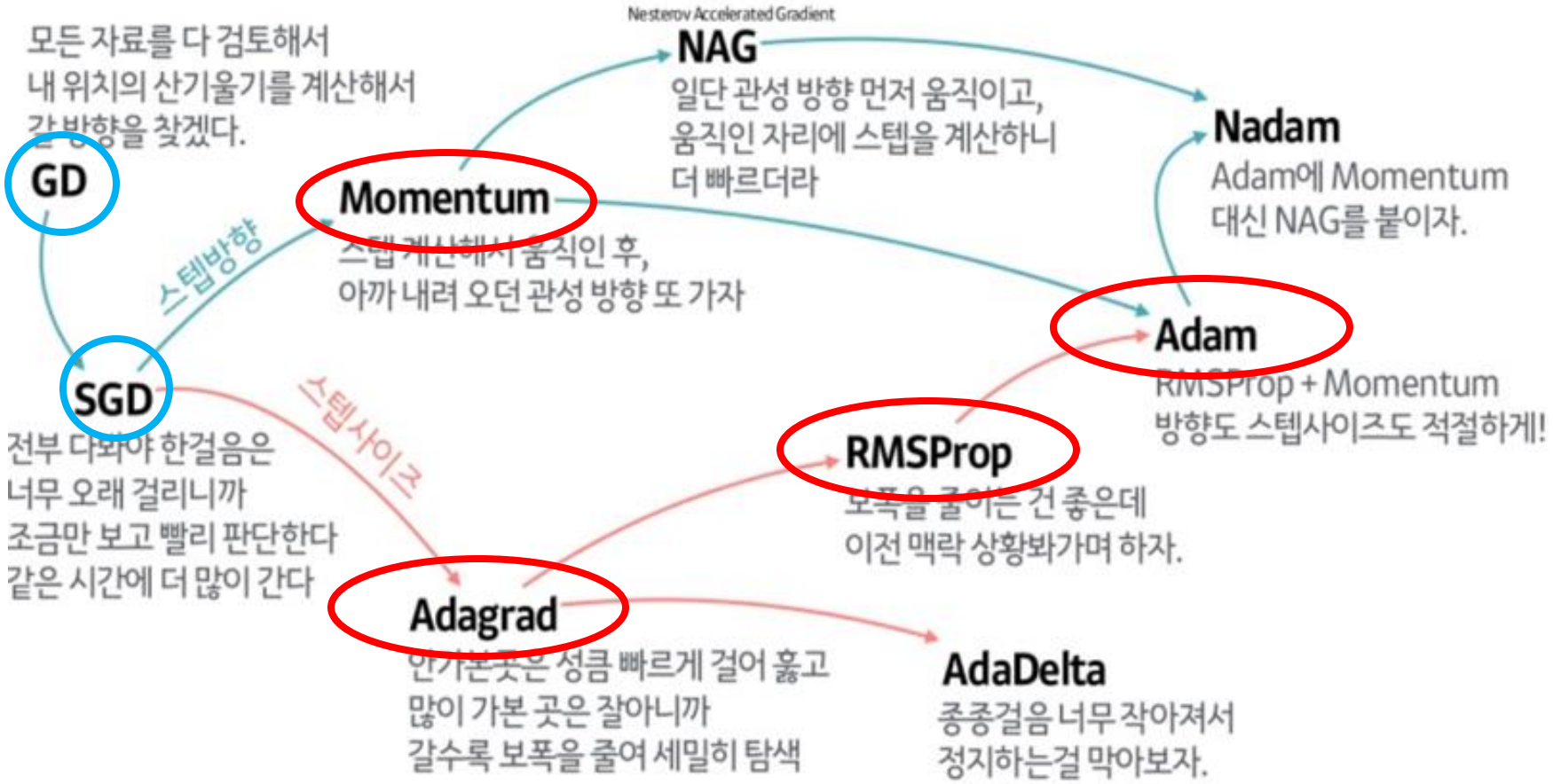
딥러닝을 활용한 **디지털 영상처리**

Digital Image Processing via Deep Learning

Lecture 7 – Learning Techniques

Contents

Optimizers

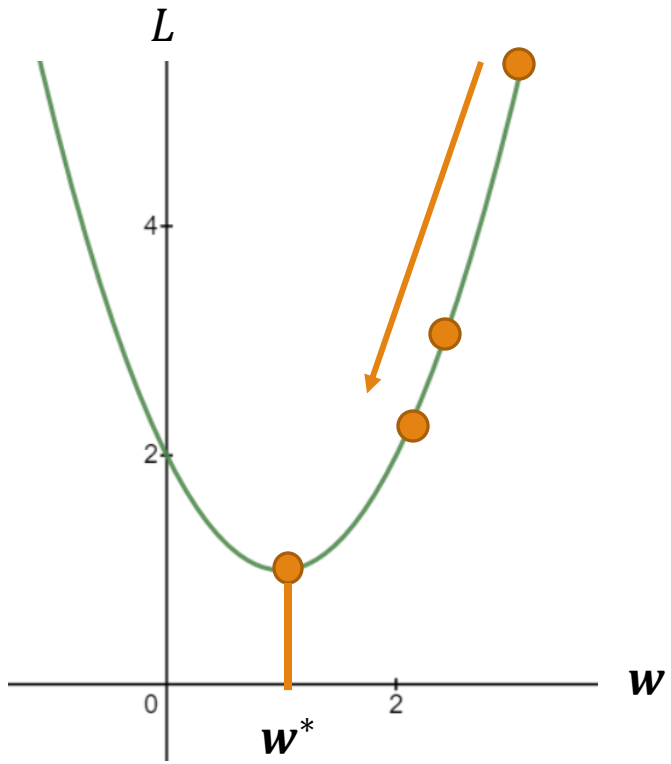

모든 자료를 다 검토해서
내 위치의 산기울기를 계산해서
GD
스텝방향
SGD
전부 다봐야 한걸음은
너무 오래 걸리니까
조금만 보고 빨리 판단한다
같은 시간에 더 많이 간다
Momentum
아까 내려 오던 관성 방향 또 가자
Nesterov Accelerated Gradient
NAG
일단 관성 방향 먼저 움직이고,
움직인 자리에 스텝을 계산하니
더 빠르더라
스텝사이즈
Adagrad
많이 가본 곳은 잘아니까
갈수록 보폭을 줄여 세밀히 탐색
RMSProp
이전 맥락 상황봐가며 하자.
AdaDelta
종종걸음 너무 작아져서
정지하는걸 막아보자.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

Stochastic Gradient Descent

Stochastic Gradient Descent: Applying mean loss of randomly sampled minibatch data and performing gradient descent algorithm.

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

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \eta\frac{\partial L}{\partial \boldsymbol{w}}$$



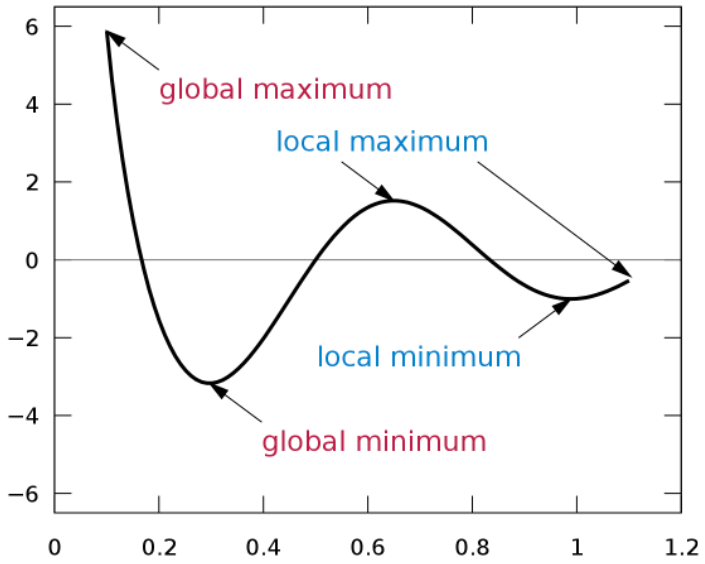
SGD Algorithm

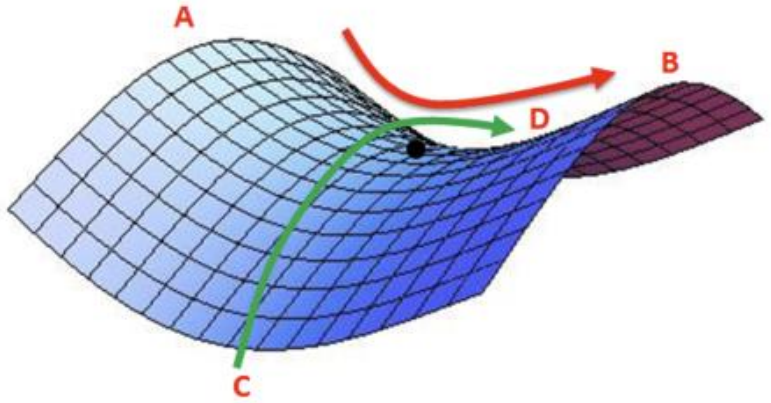
Algorithm: Stochastic Gradient Descent
Input: Training data D ={X,Y}, epoch e, learning rate η, stop threshold τ
Output: Optimum weight matric, $\boldsymbol{w}^*$

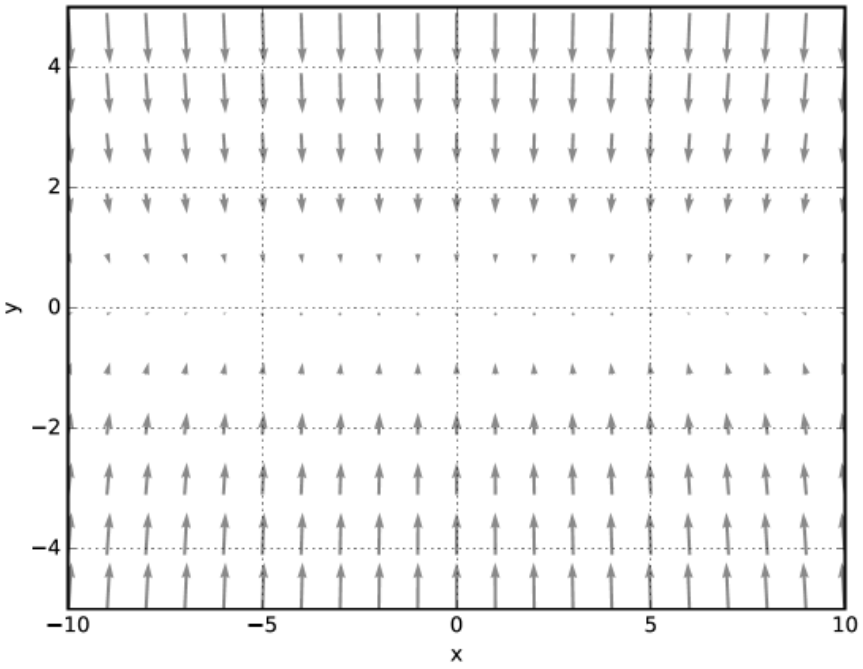
1. Initialize $\boldsymbol{w_0}$ with random numbers
2. For i=1,2,3,…,e repeat
    3. Sample minibatch data D_M from D
    4. Compute y via forward propagation
    5. Compute loss, L
        6. If L is below τ break
    7. Compute $\frac{\partial L}{\partial \boldsymbol{w}}$
    8. $\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \eta \frac{\partial L}{\partial \boldsymbol{w}}$
9. Return $\boldsymbol{w_e}$

Problems with SGD

1. Easy to fall into the **Local Minimum**
2. Difficult to escape the **Saddle Point**
3. **Inefficient** when gradients in certain axis are small compared to other





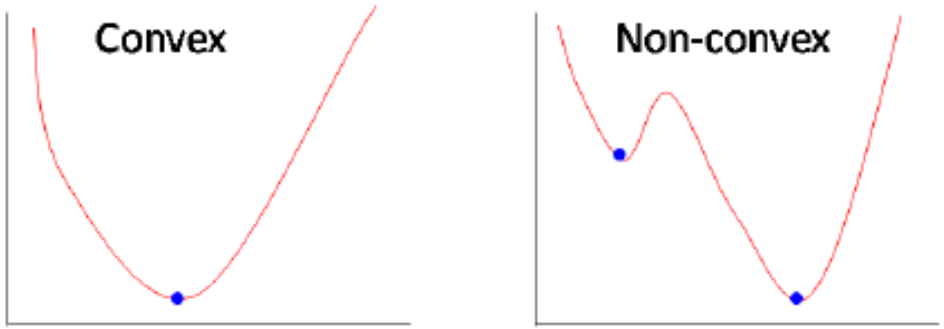


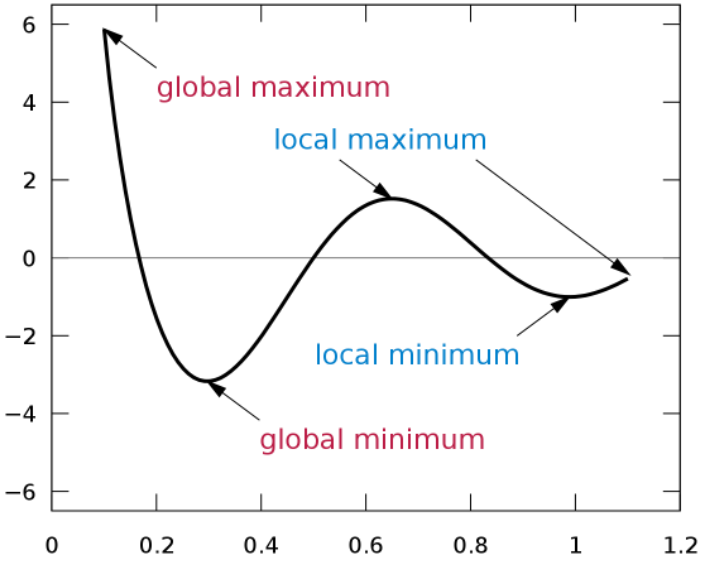
Local Minimum

Convex Functions: Easy to find Global Minimum

Non-Convex Functions: Hard to find the Global Minimum

The Loss function in a complex Neural Network is almost always **Non-Convex**!!

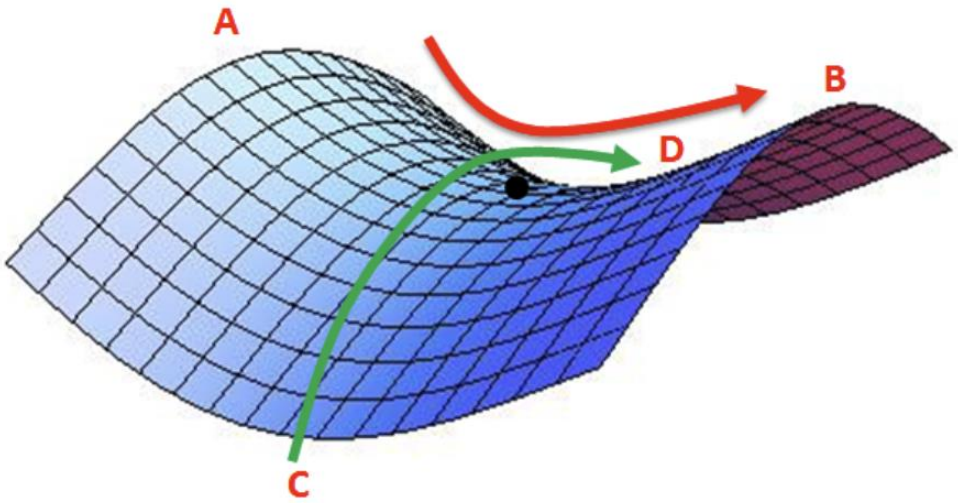




Saddle Point

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \eta \frac{\partial L}{\partial \boldsymbol{w}}$$

If the gradient is 0, there are no more updates!



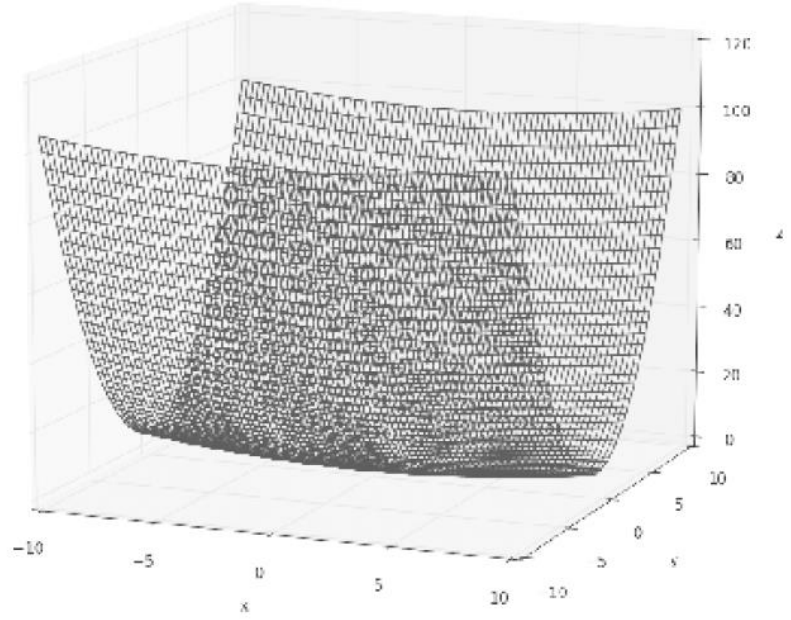
What if the gradient is 0 for the non-minimum point and the parameters point at this point?

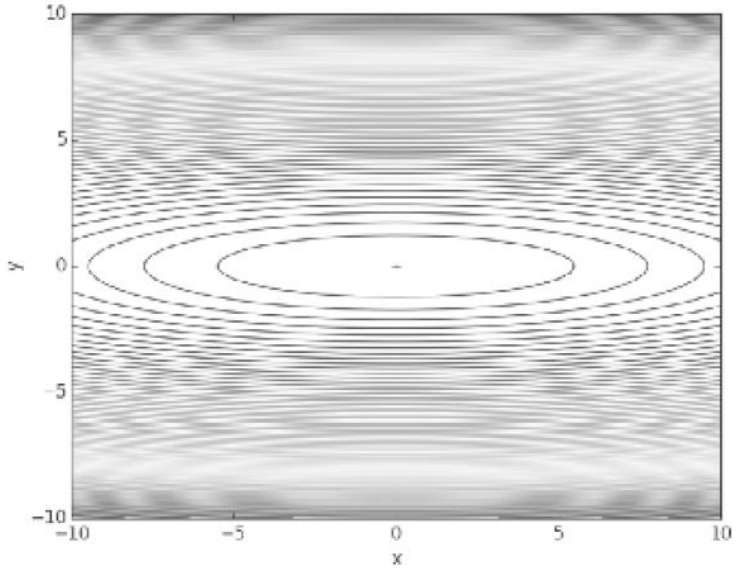
SGD algorithm can hardly escape from the **Saddle Point**

Inefficiency

This function has Large Gradient in “y” direction but Small Gradient in “x” direction

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

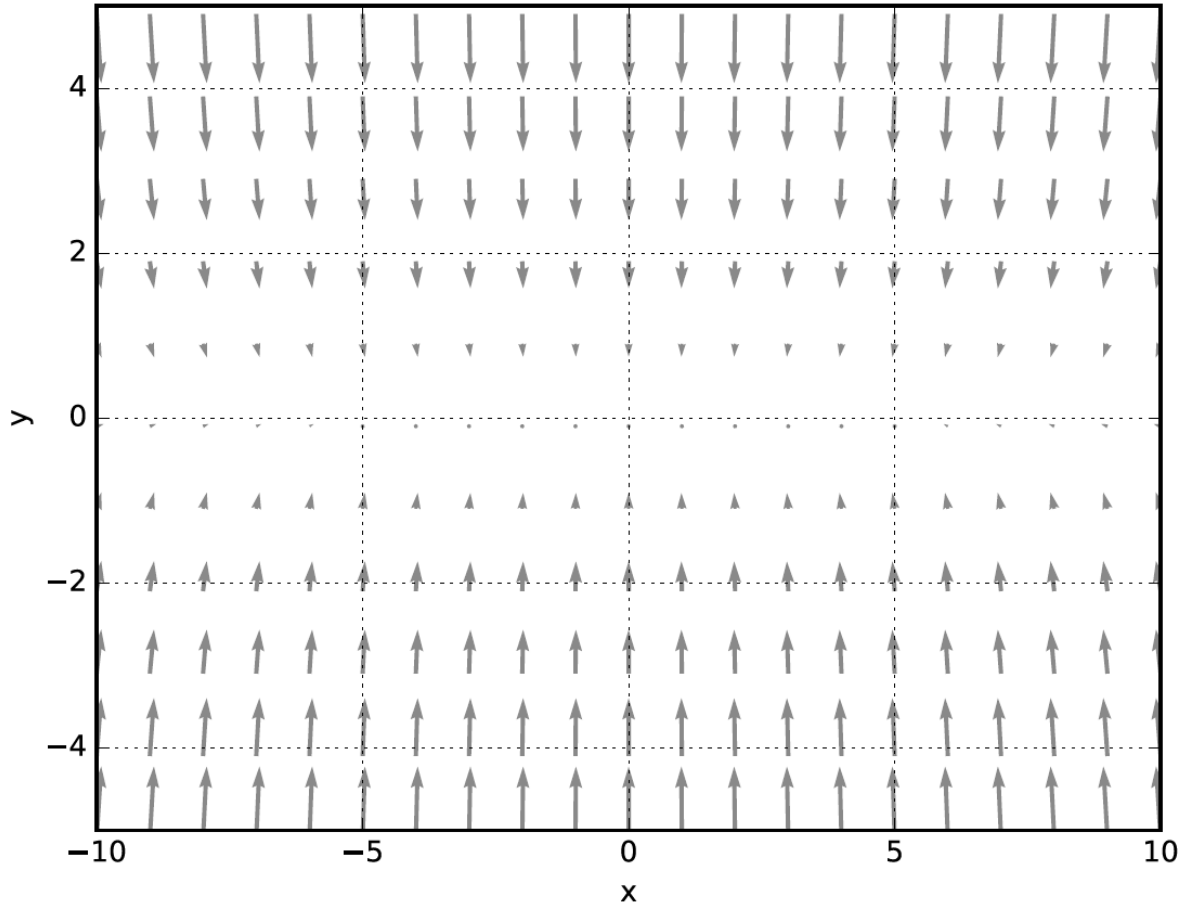




Inefficiency

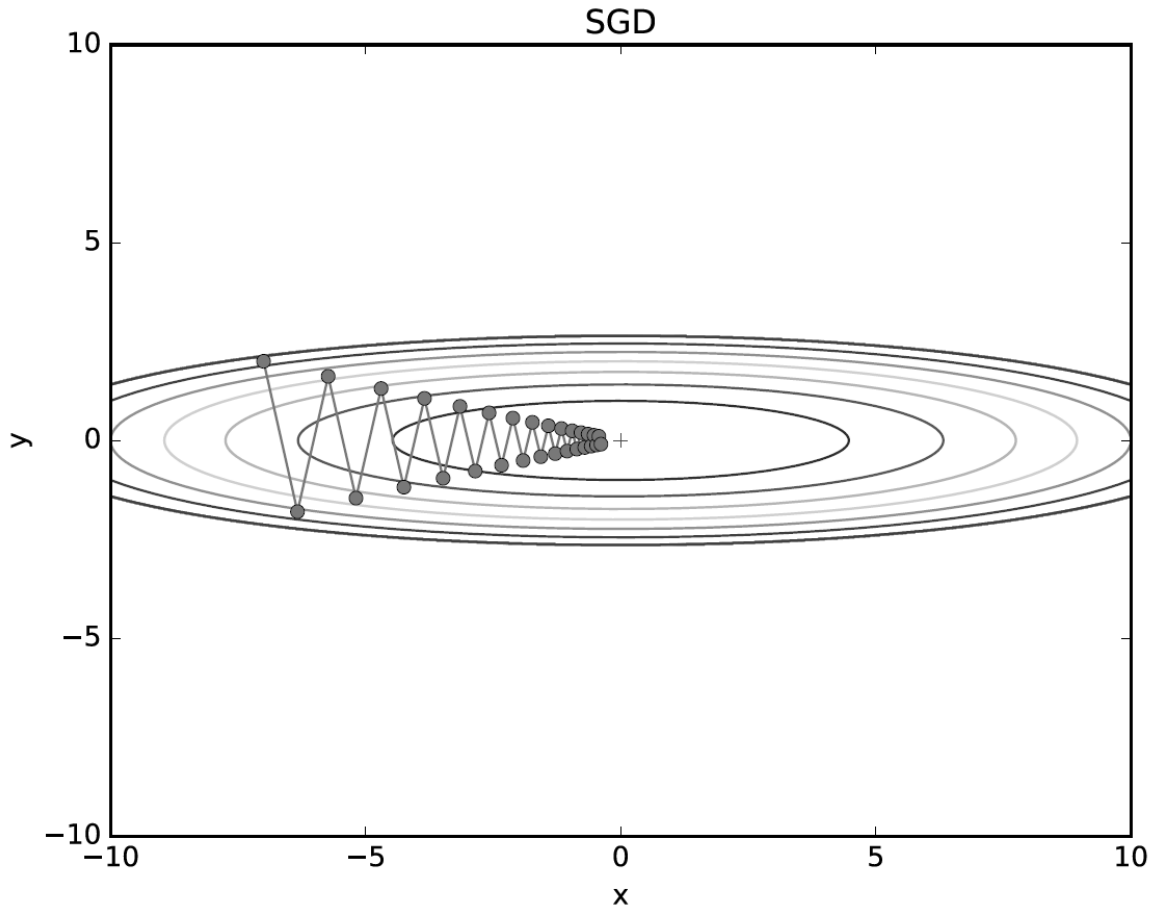
The Minimum point is the origin for this function.

However at most of the points, the gradient does not directly point to the minimum.



Inefficiency

The update gradually leads the weight values to their optimum. But seems somewhat very inefficient!



Momentum

The momentum method employs a velocity variable, “v”.

It updates the velocity directly with the gradient and then the velocity is used to update the weights.

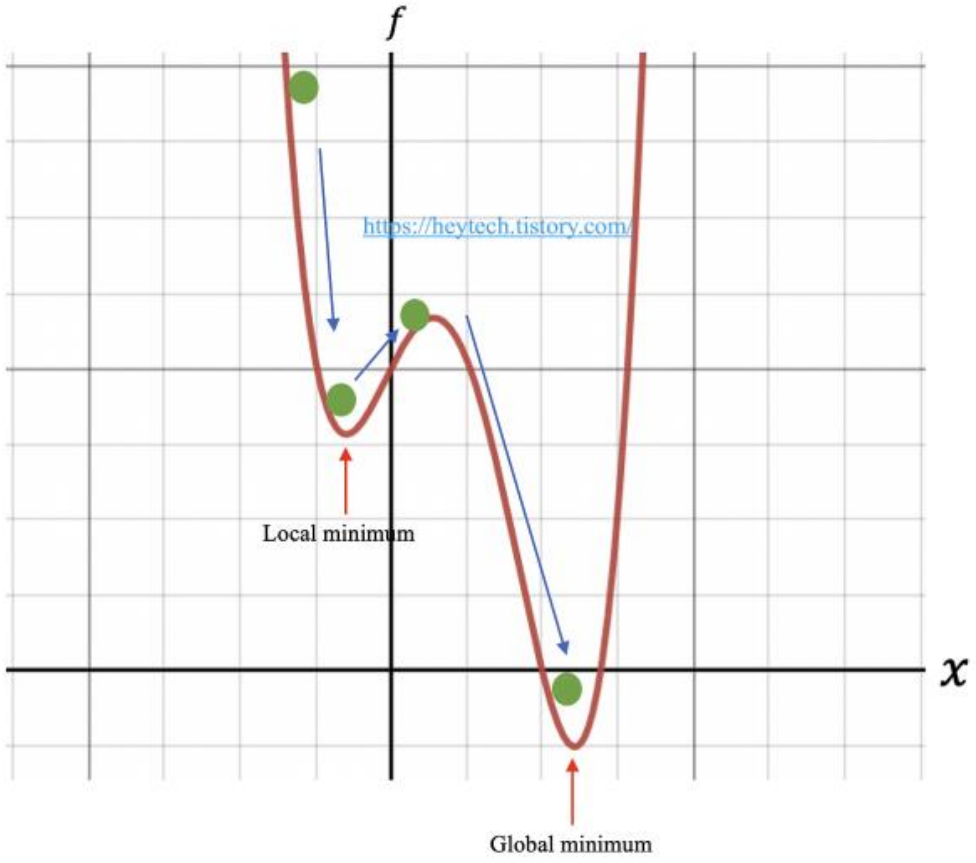
This prevents the algorithm to fall into the local minimum.

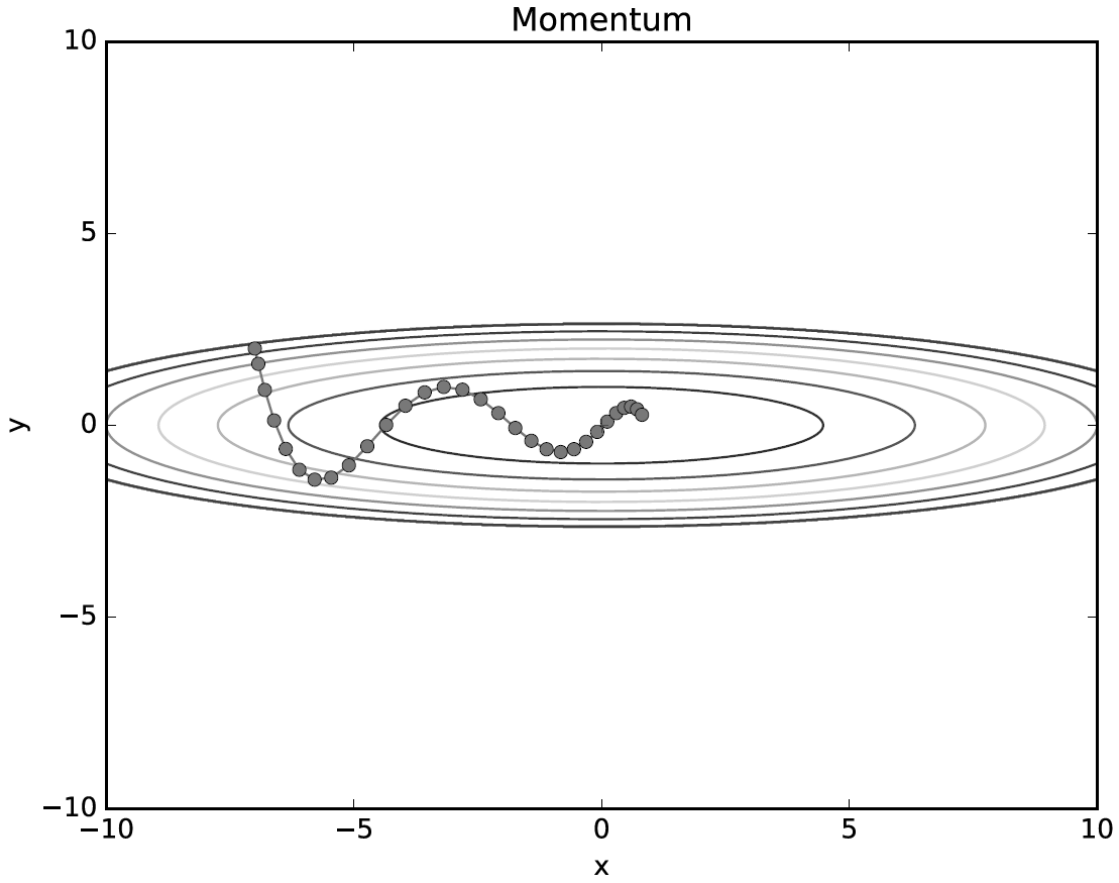
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

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

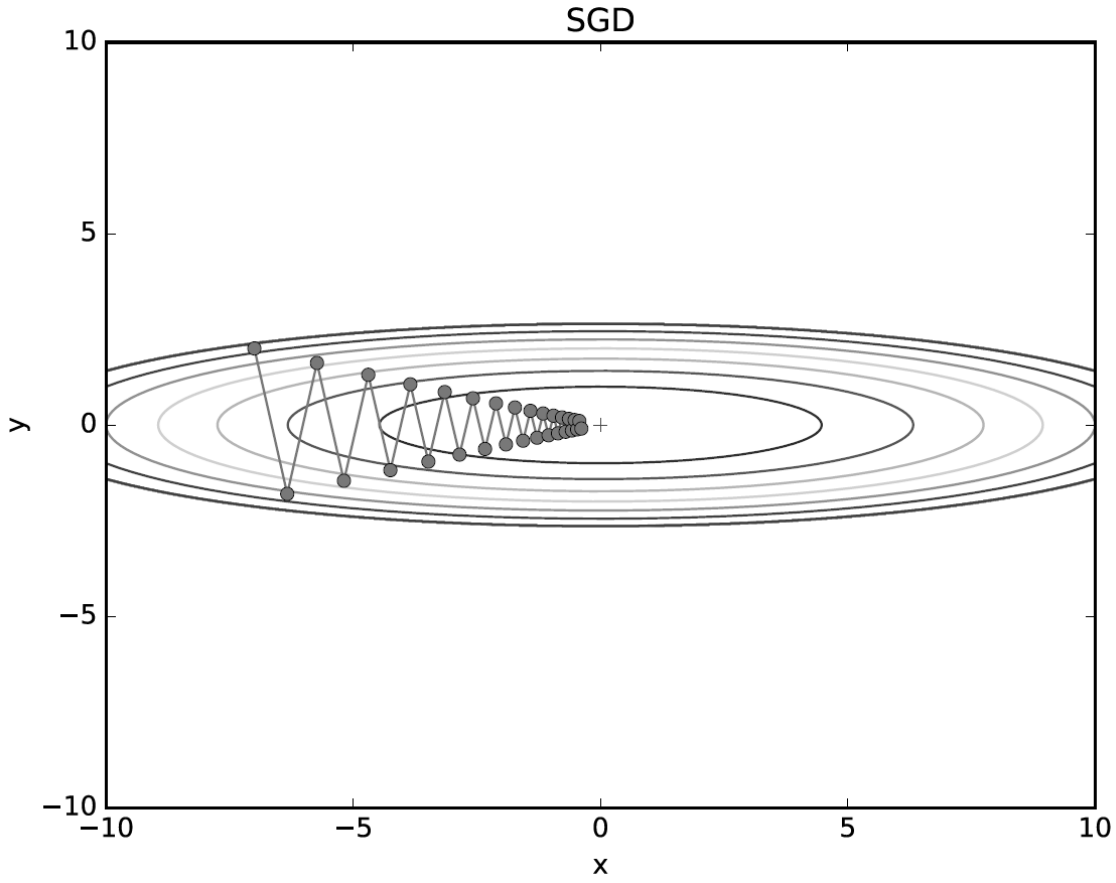
α is a constant value between 0 and 1. Usually set to 0.9

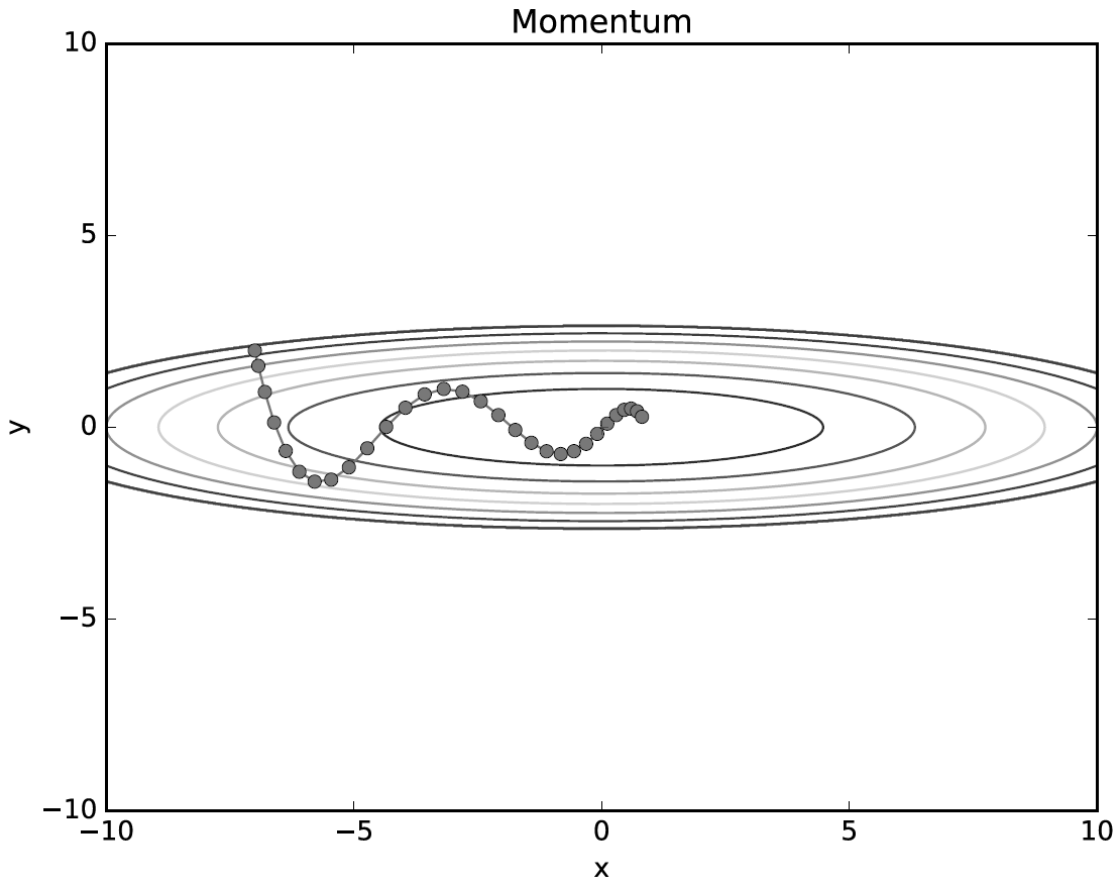
Momentum


f
https://heytech.tistory.com/
Local minimum
Global minimum
x

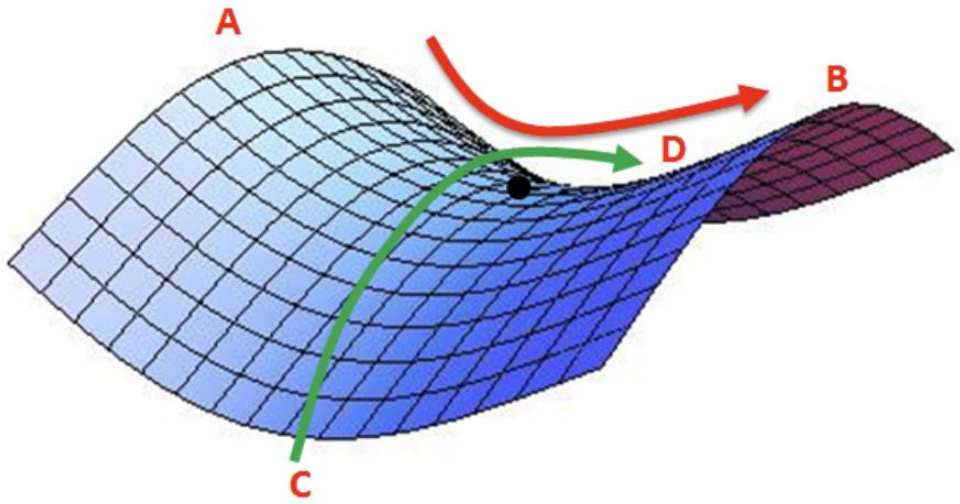

Momentum
10
5
0
−5
−10
y
−10
−5
0
5
10
x

SGD VS Momentum





Problem with Momentum



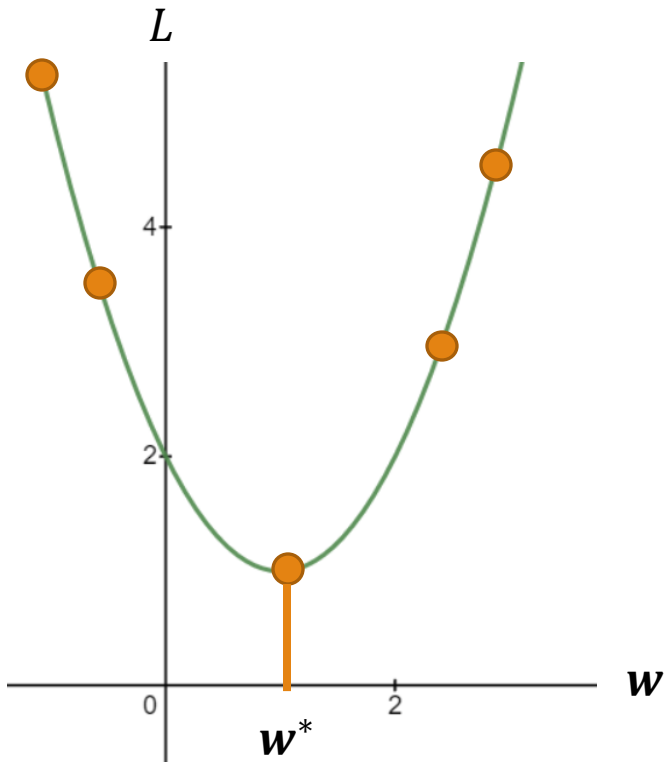
$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

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

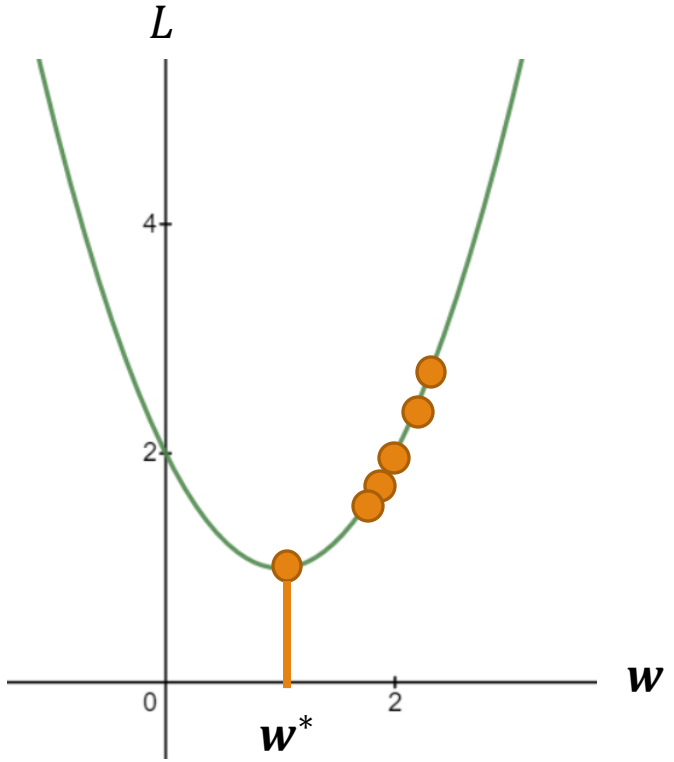
The Momentum algorithm still cannot escape from the **Saddle Point**

Learning rate Calibration

When learning rate is too big, say, 10.



When learning rate is too small, say 0.000001.



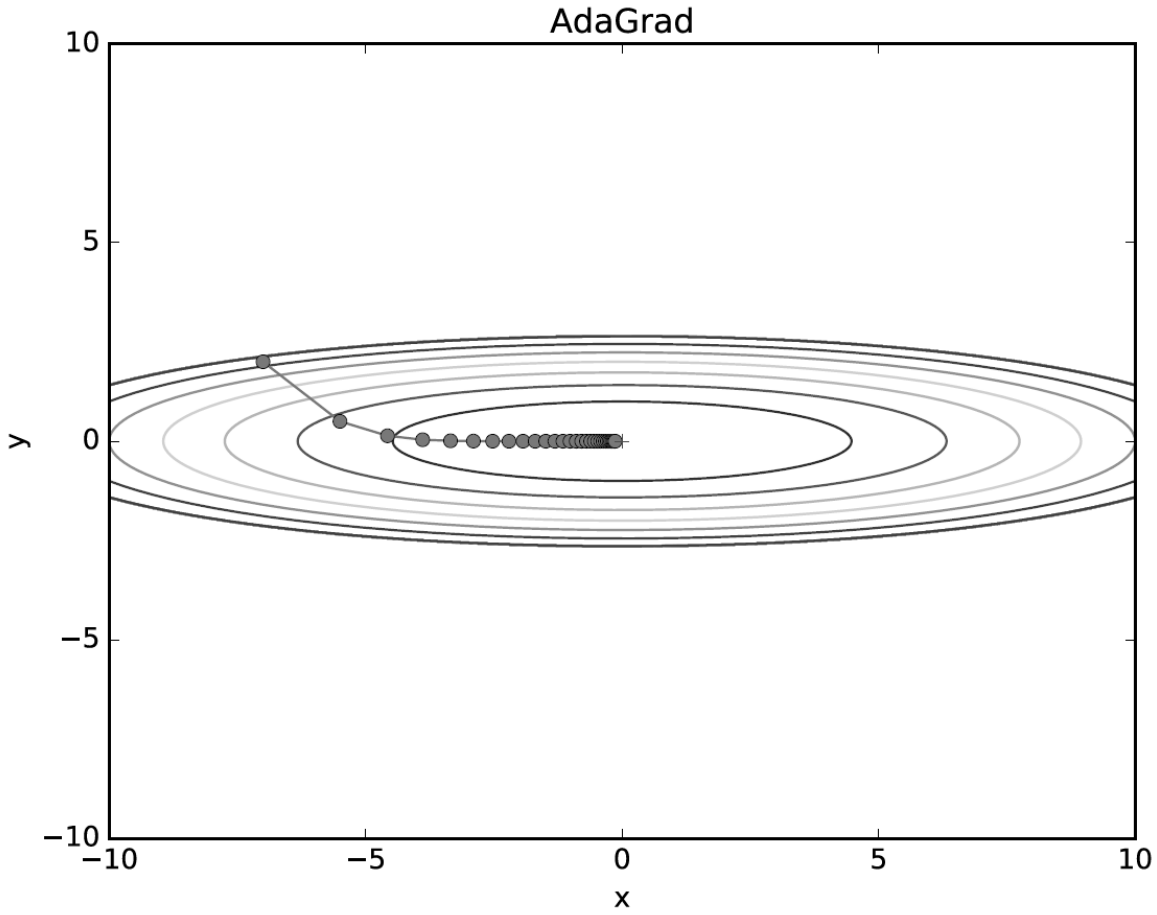
Adaptive Gradient (AdaGrad)

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

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

AdaGrad algorithm alters the learning rate depending on the gradient values, hence the name adaptive gradient!

For each weight parameter, the learning rates are separately set.



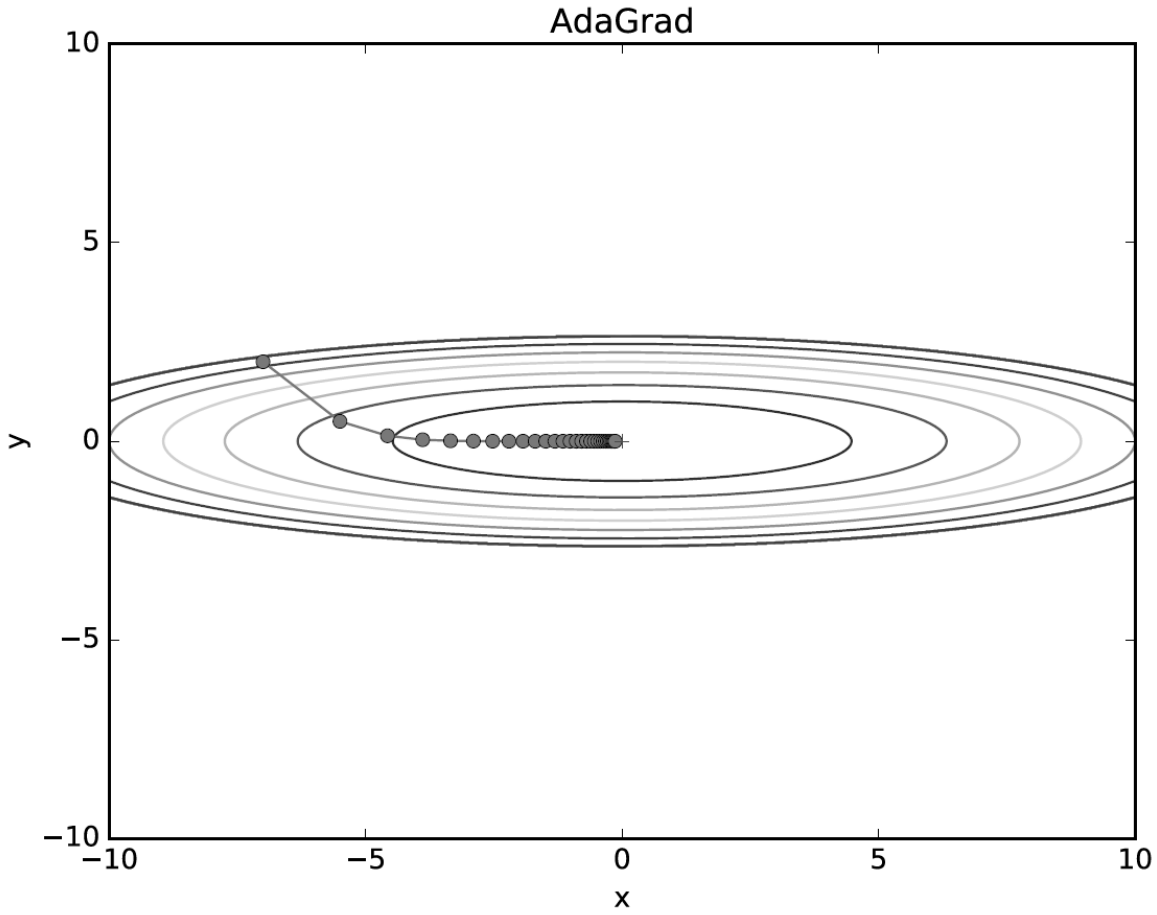
Adaptive Gradient (AdaGrad)

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

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

Since the variable h, is constantly being updated, it **accumulates the gradient information of each weight**, giving a smaller learning rate to the weights that have experienced a large gradient and a larger learning rate to the weights that have experienced a small gradient.

This mechanism helps the algorithm avoid falling into a saddle point and escaping from it.



Problem with AdaGrad

The variable, h is constantly being added by a positive numbers, hence when h approaches infinity, the learing rates will converge to 0, and the weights are then not being updated anymore!

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

$$\boldsymbol{h} \uparrow \qquad \frac{\eta}{\sqrt{\boldsymbol{h}}} \downarrow$$

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

Root Mean Squared Propagation (RMSProp)

$$\boldsymbol{h} \leftarrow \gamma \boldsymbol{h} + (1-\gamma)\left(\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}} \odot \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}}\right)$$

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

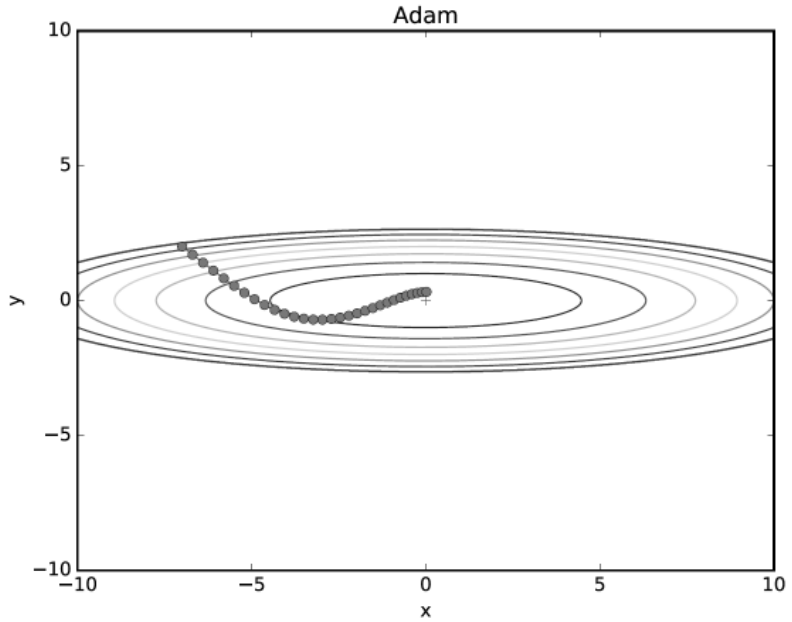
In the RMSProp algorithm, the variable h is updated in an average format of its old value and the square of gradients.

This provides an effect of forgetting old gradients, and giving more importance to new gradients.

Hence, the RMSProp algorithm takes much longer for the learning rate to be 0 than the AdaGrad algorithm.

Adaptive Moment Estimation (ADAM)

ADAM = Momentum + RMSProp



$$\boldsymbol{m} \leftarrow \delta \boldsymbol{m} + (1-\delta)\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}}$$

$$\boldsymbol{h} \leftarrow \gamma \boldsymbol{h} + (1-\gamma)\left(\frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}} \odot \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}}\right)$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \frac{1}{\sqrt{\widehat{\boldsymbol{h}}}} \widehat{\boldsymbol{m}}$$

ADAM takes both momentum and adaptive gradient considerations.

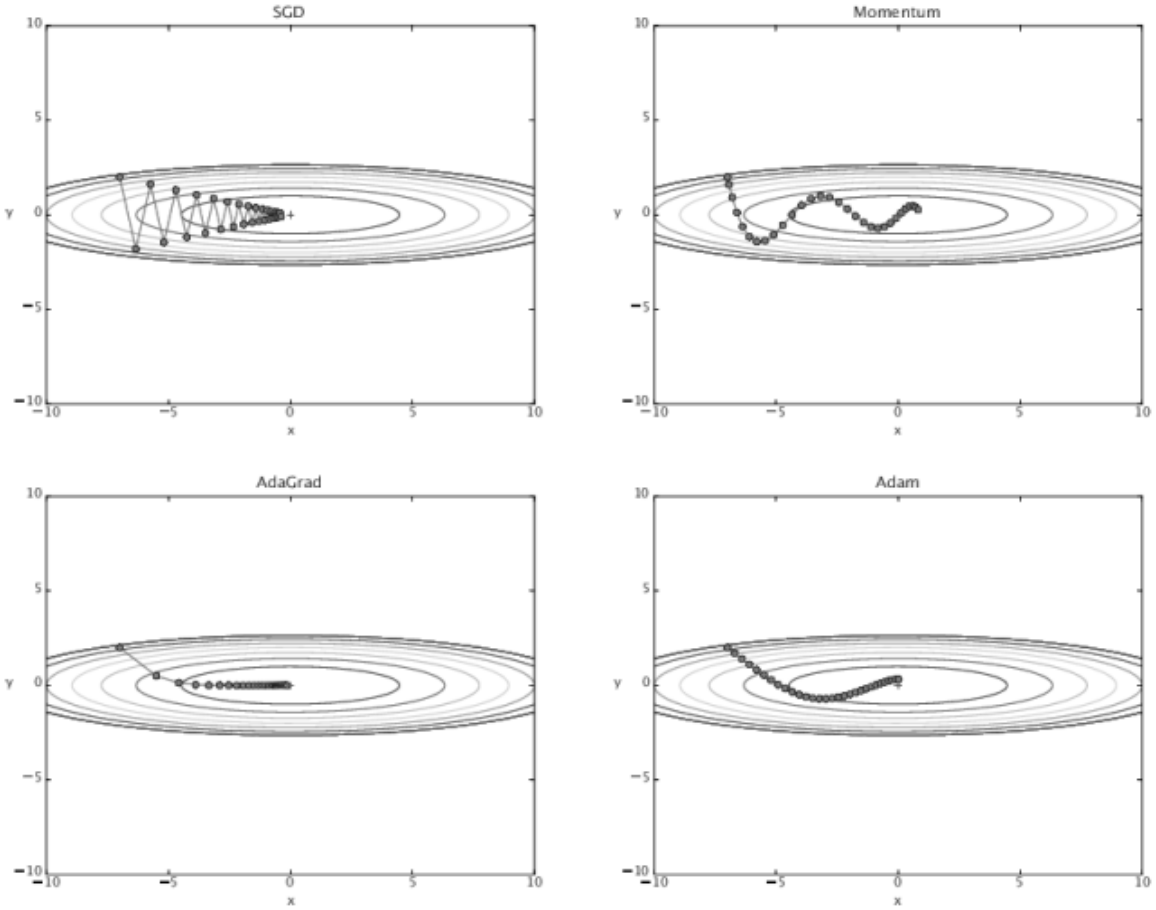
The equation on right is a brief version of the original. Please refer to other resources for the exact update equation.

Which one to use?

The best optimization algorithm depends on the problem to be solved.

To find the best one, it is best to try all of them and compare the results.

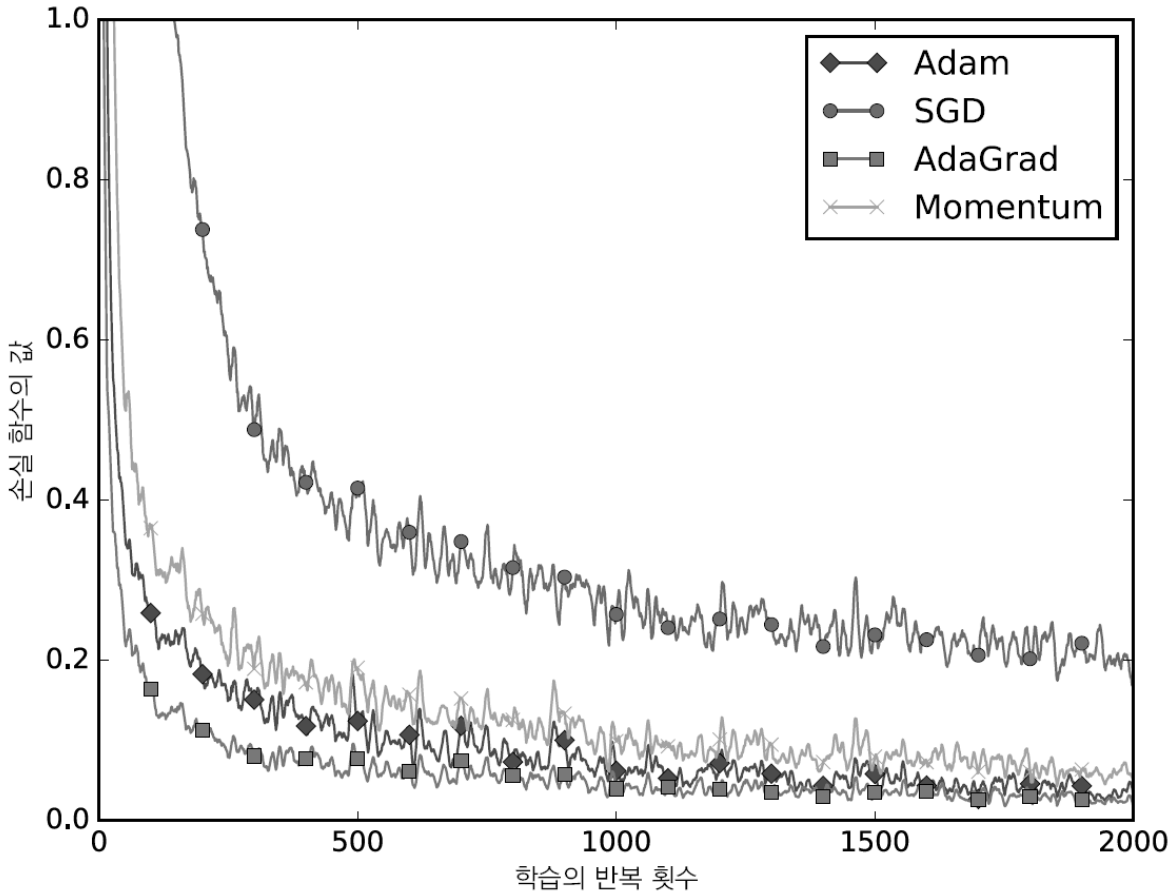
However, since it is computationally inefficient to do this, in most cases, just use ADAM, which is known to work well in most of the problems.



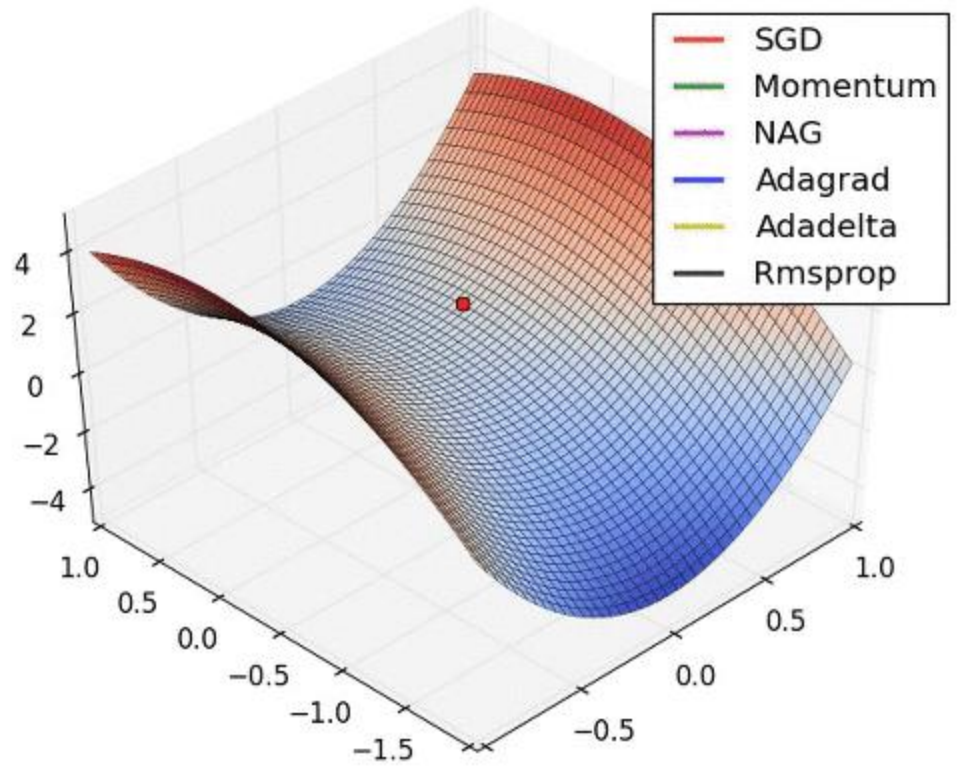
Which one to use?

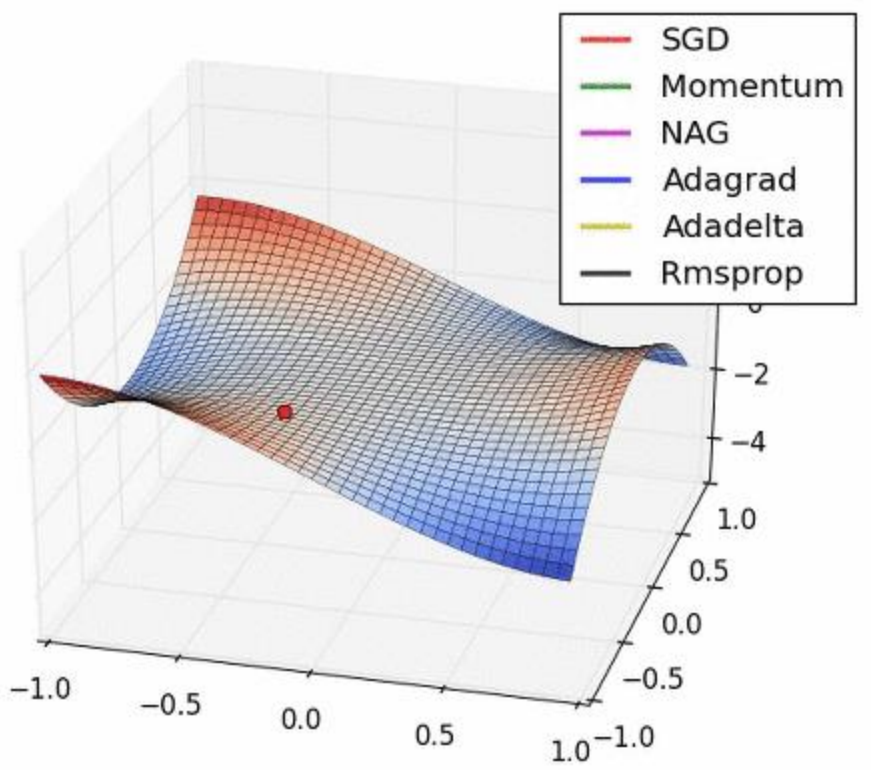
For this specific case, AdaGrad seems to be the best one!

Note that this may be due to the simplicity of this problem. For more complicated tasks, ADAM is usually the best.

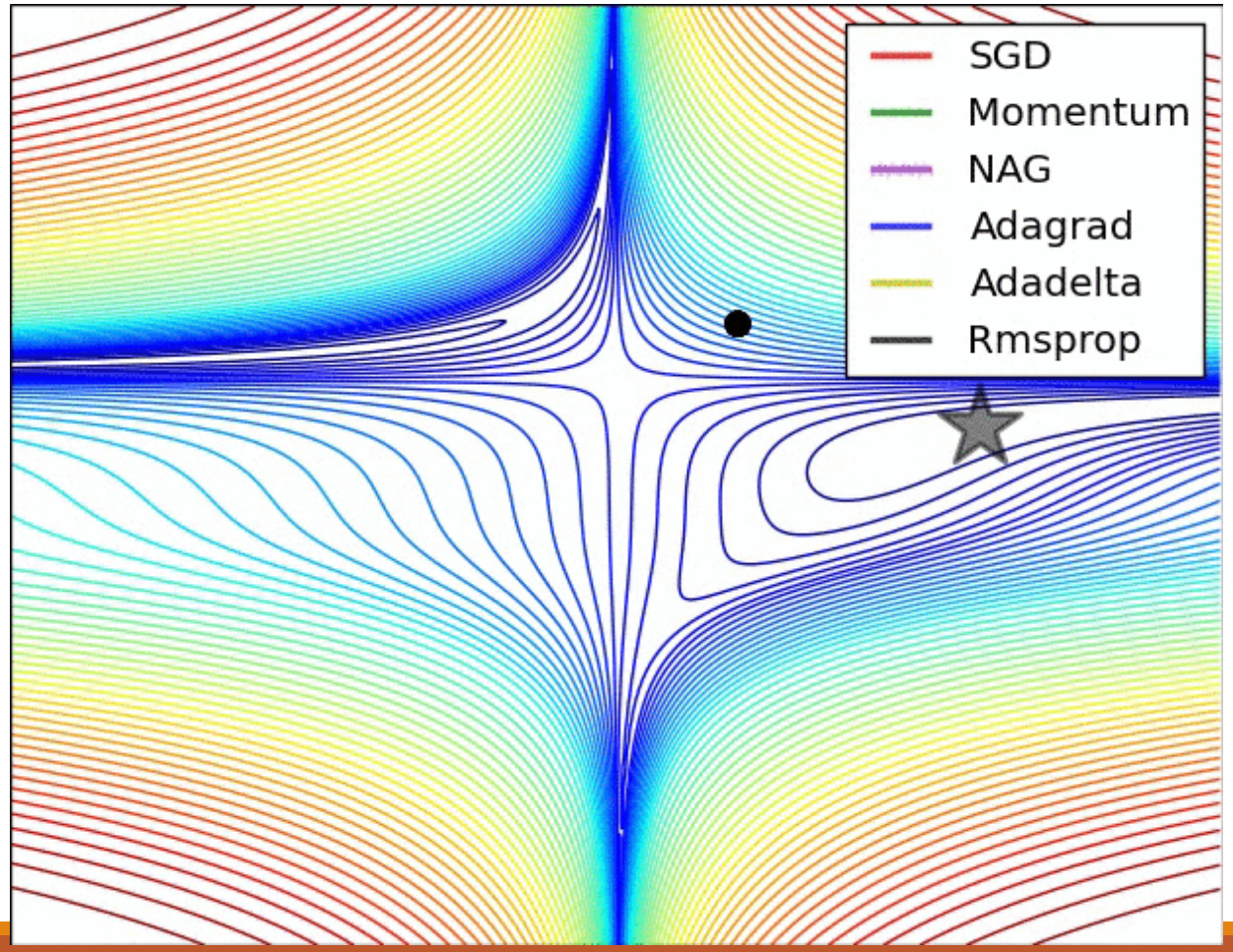


More Comparisons





More Comparisons

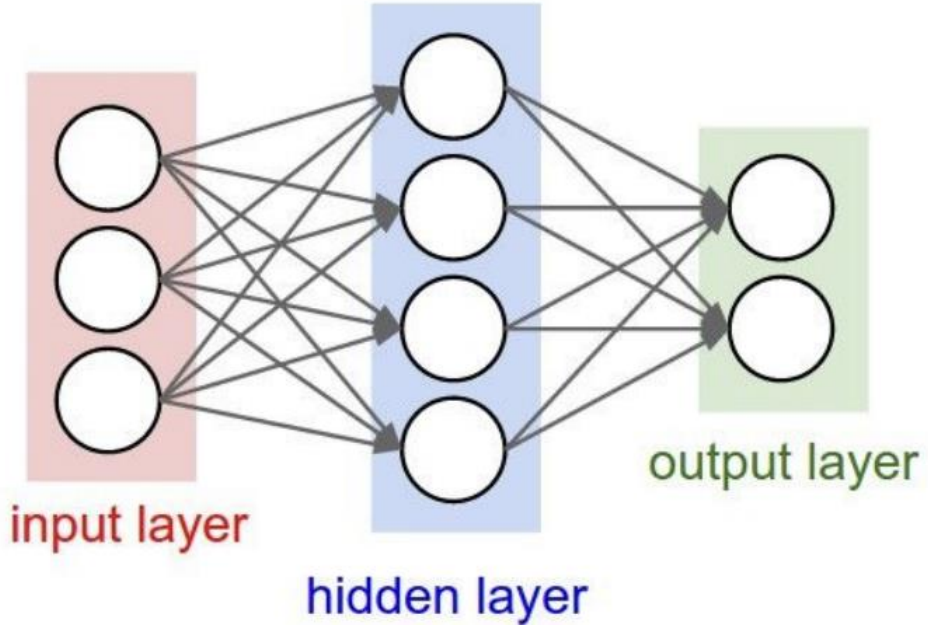


Contents

Initial Weight Values: 0

It is important that we select suitable initial weights!

Q: What happens if all initial weights of the network below are set to 0?



Initial Weight Values: 0

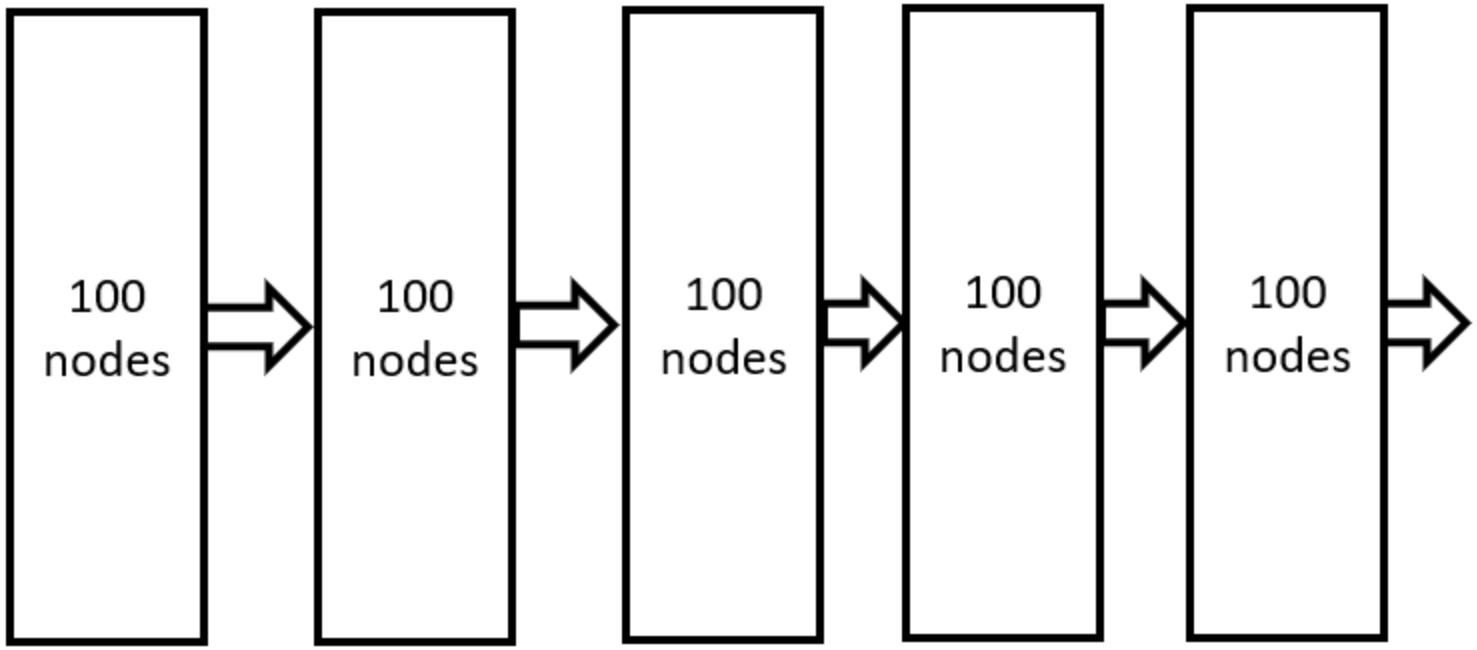
It is important that we select suitable initial weights!

Q: What happens if all initial weights of the network below are set to 0?

ANS:
Symmetry: All neurons and layers will produce **identical outputs**. Hence, during backpropagation, **all the weights will receive the same gradient update**. The network cannot produce complex representations.

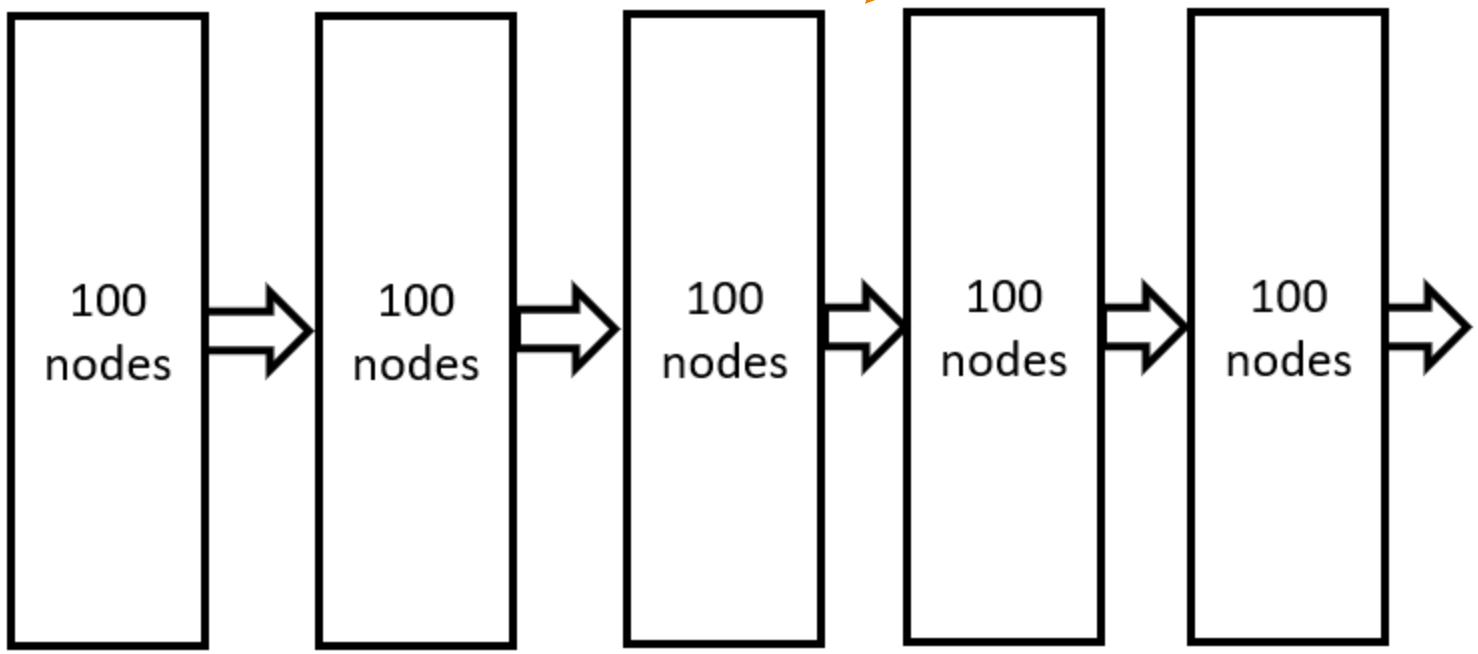
Initial Weight Values: Normal, sd=1

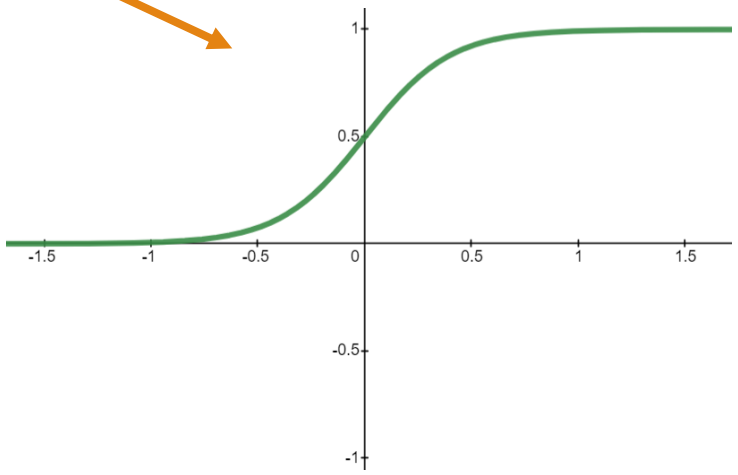
Let’s initialize the weights randomly, with the normal distribution of sd of 1
Consider the following Network:



Initial Weight Values: Normal, sd=1

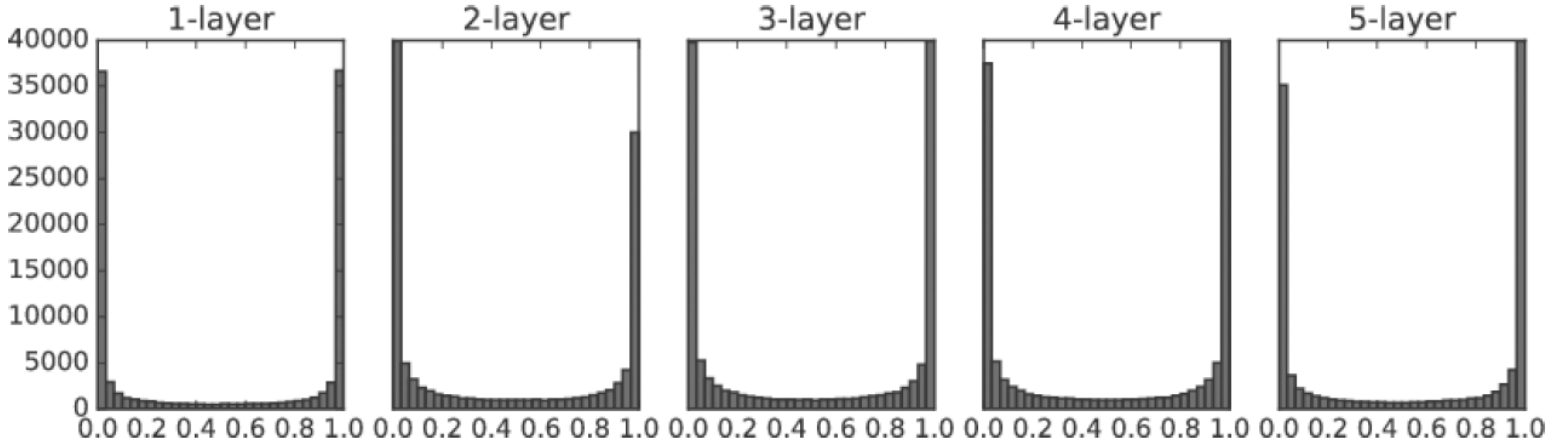
Let's initialize the weights randomly, with the normal distribution of sd of 1
Consider the following Network structure and sigmoid is used as an activation f:

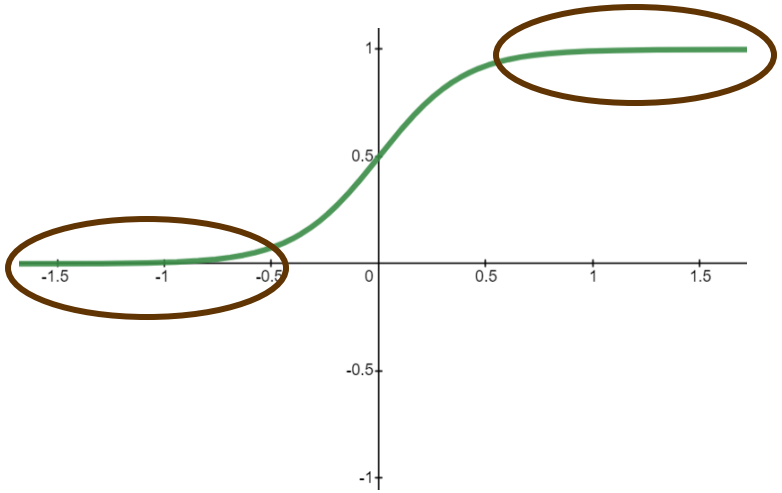




Initial Weight Values: Normal, sd=1

Let's initialize the weights randomly, with the normal distribution of sd of 1
Consider the following Network structure and sigmoid is used as an activation f:

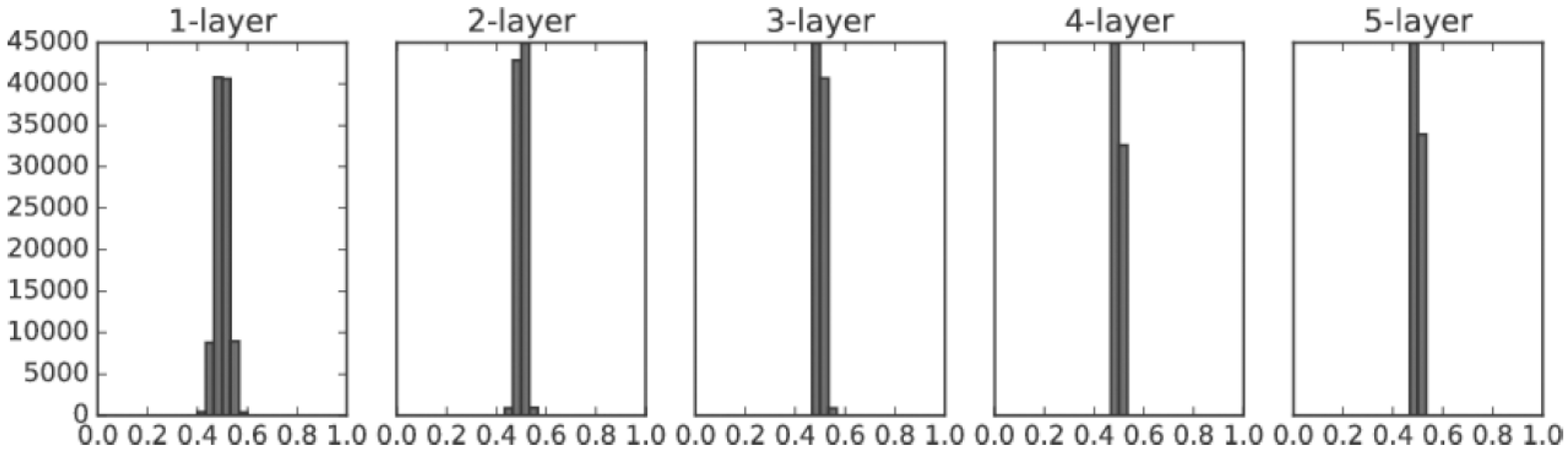


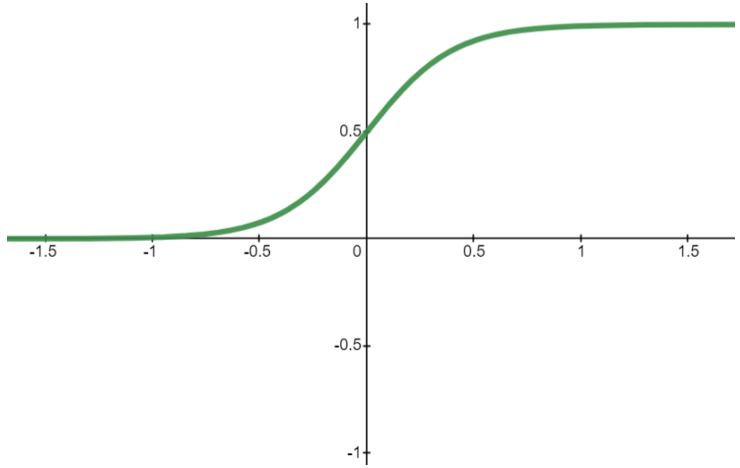


Node output values are distributed mainly to values of 0 and 1.
When the input to activation f is too large or too small → Gradient becomes near 0!

Initial Weight Values: Normal, sd=0.01

We now reduce the sd to 0.01,





Now, the network outputs are distributed around 0.5

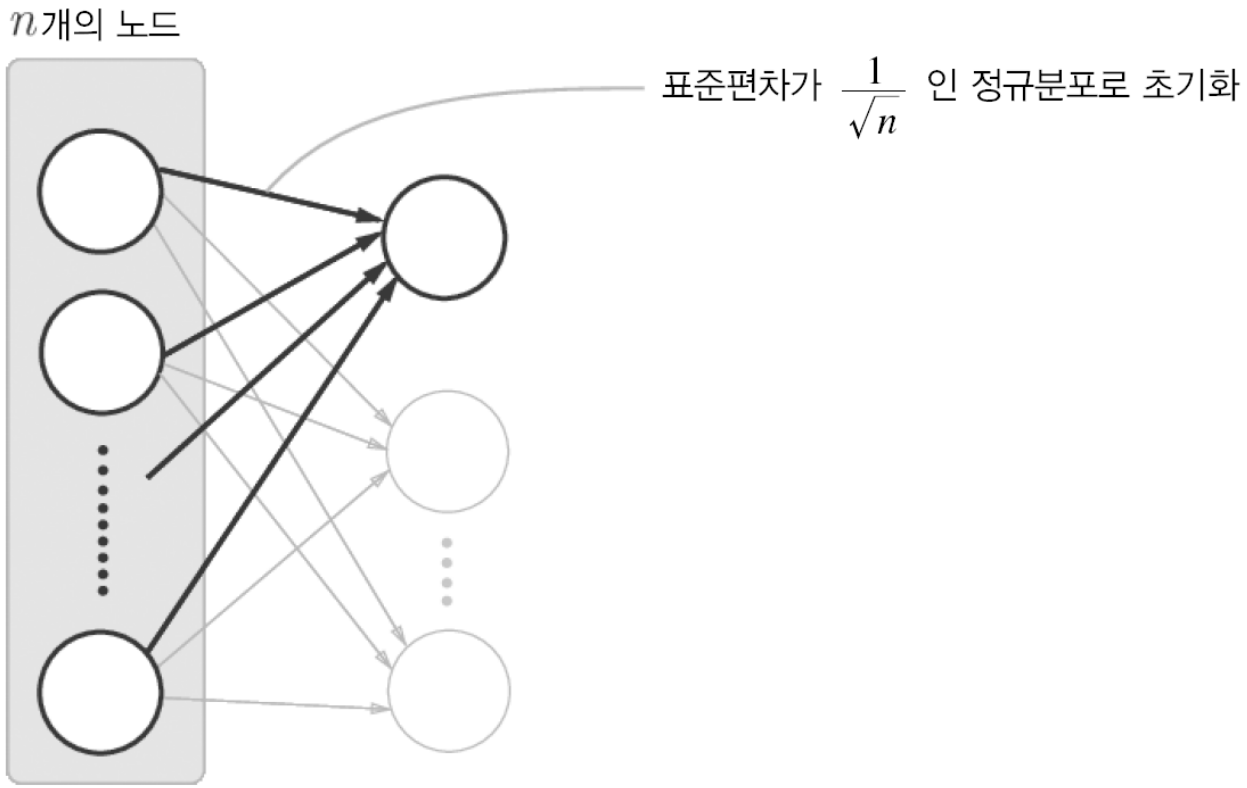
→ Problem! Meaningless neuron numbers!

Initial Weight Values

Initial values must be distributed evenly so that a variety of values must be passed through the layers!

We must use a stochastic distribution with reasonable values of parameters!

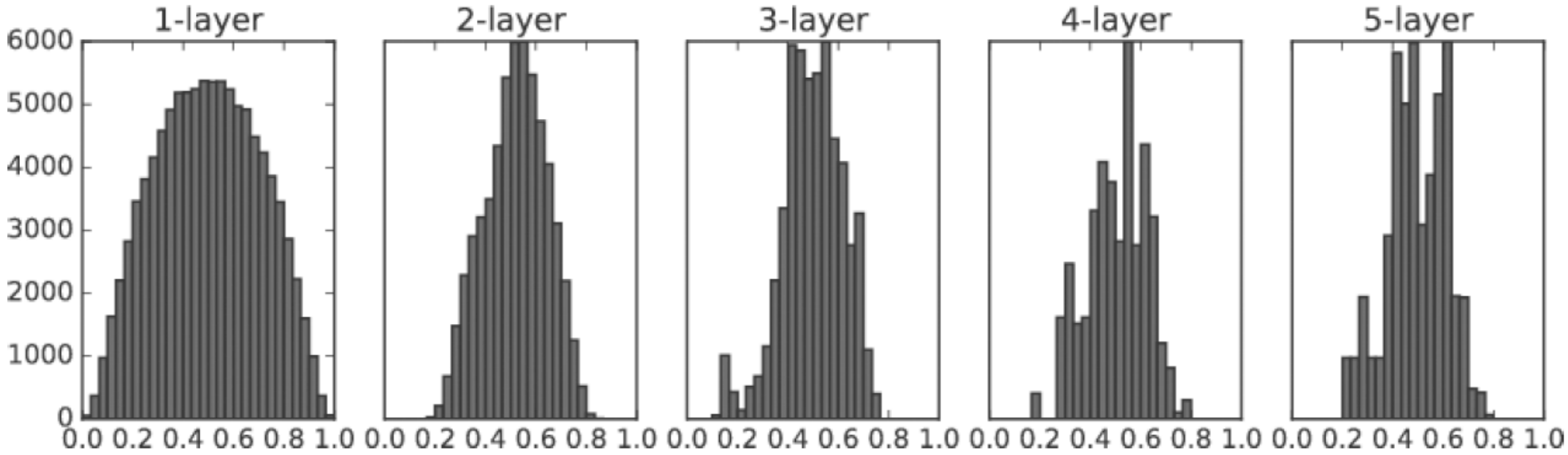
Initial Weight Values: Xavier



For the purpose of distributing values widely across nodes, the equivalent value of sd is found to be one over the root of the number of nodes.
→ known as Xavier distribution

Initial Weight Values: Xavier

The initial values now seem to be evenly distributed!

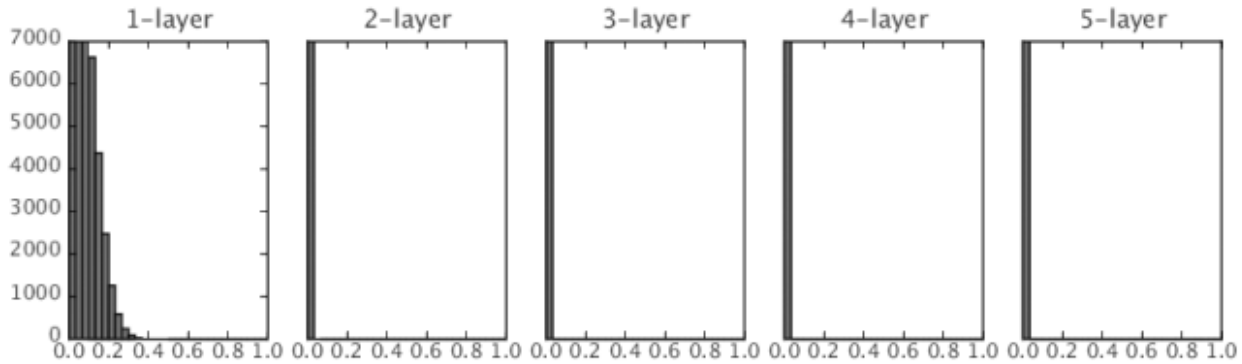


Note that Xavier initialization works well for the cases with activation functions with both linearity and symmetry properties.

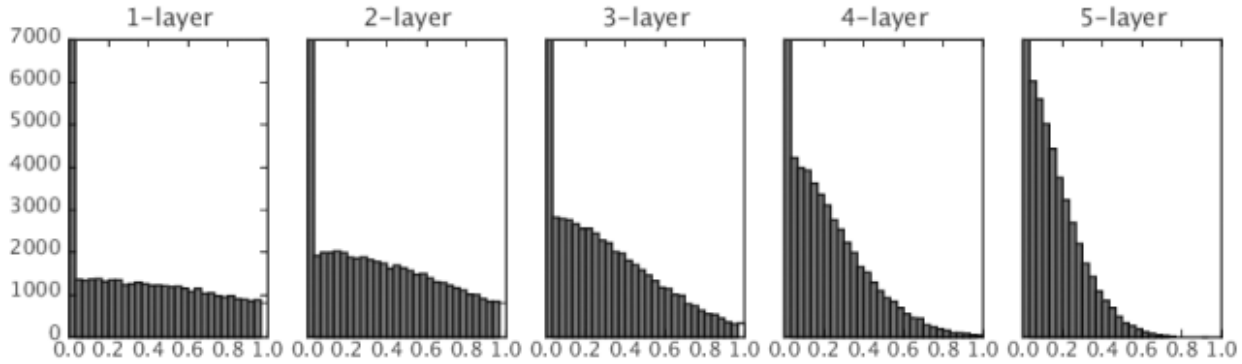
Initial Weight Values: He

When using ReLU, since it is not symmetrical (values below 0 are ignored), we distribute the data twice as much as the case for Xavier initialization.

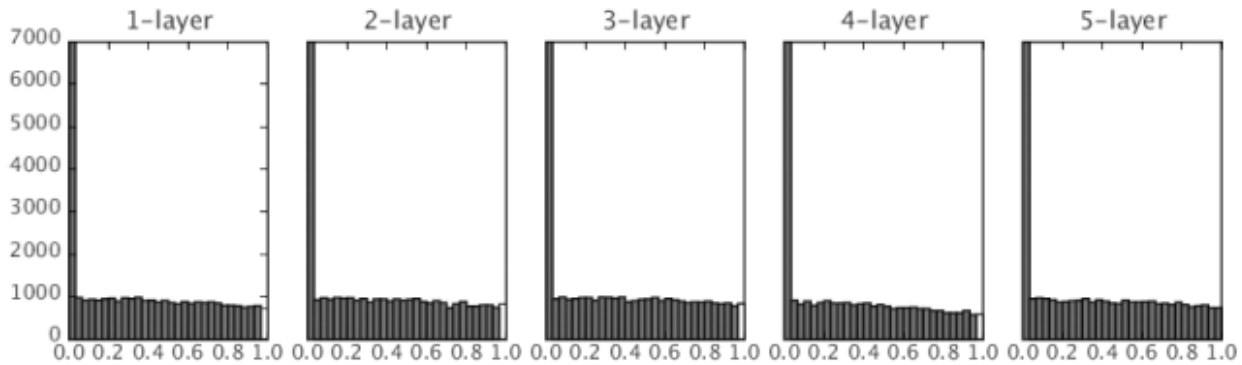
He initialization: sd = $\sqrt{\frac{2}{n}}$



표준편차가 0.01인 정규분포를 가중치 초깃값으로 사용한 경우



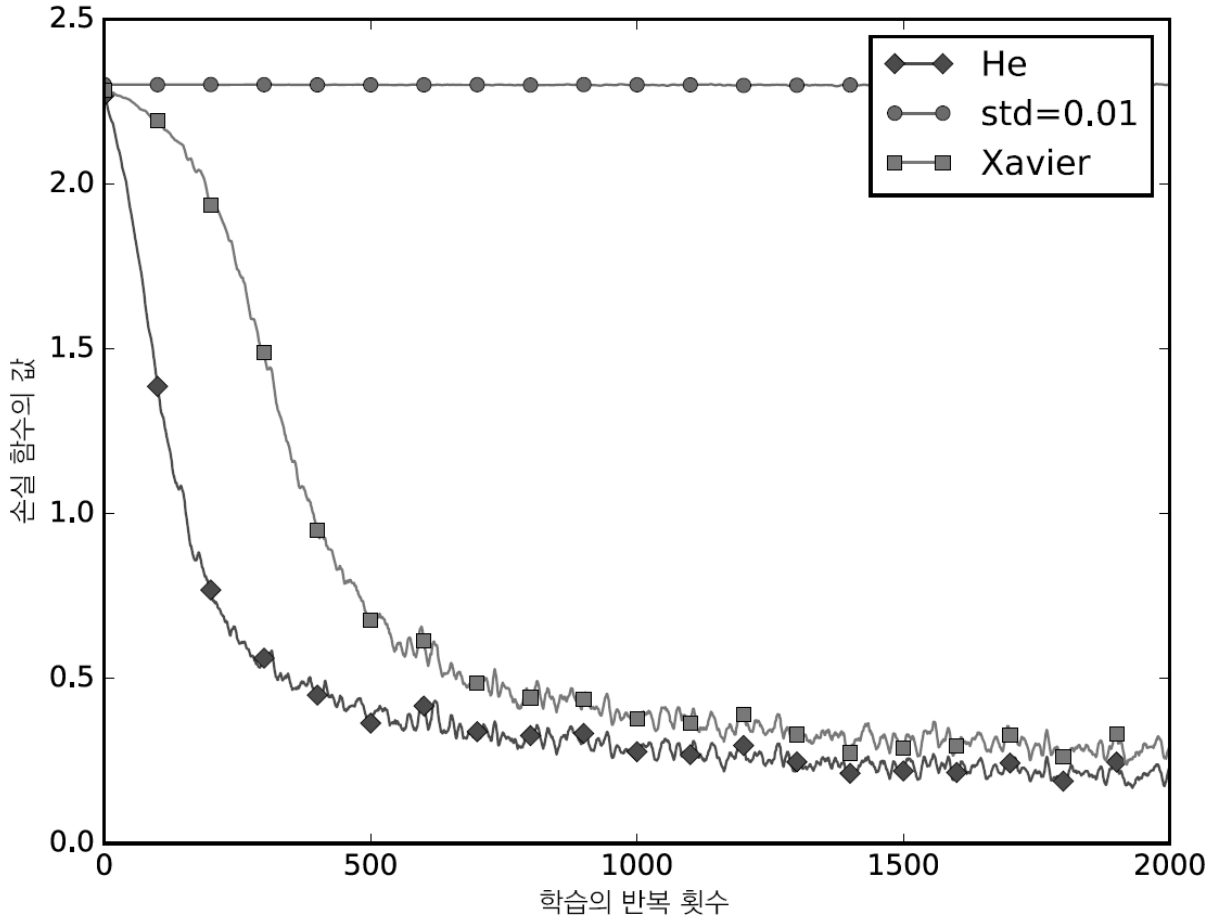
Xavier 초깃값을 사용한 경우



He 초깃값을 사용한 경우

Gradient
Vanishing!

Initial Weight Values: Comparison



Next Lecture

1. Optimizers
2. Weight Initialization
3. Batch Normalization
4. Overfitting and Dropout
5. Hyperparameter Tuning