Lecture 7

Training Neural Networks, Part2

휴먼기계바이오공학부
2070022 김현서

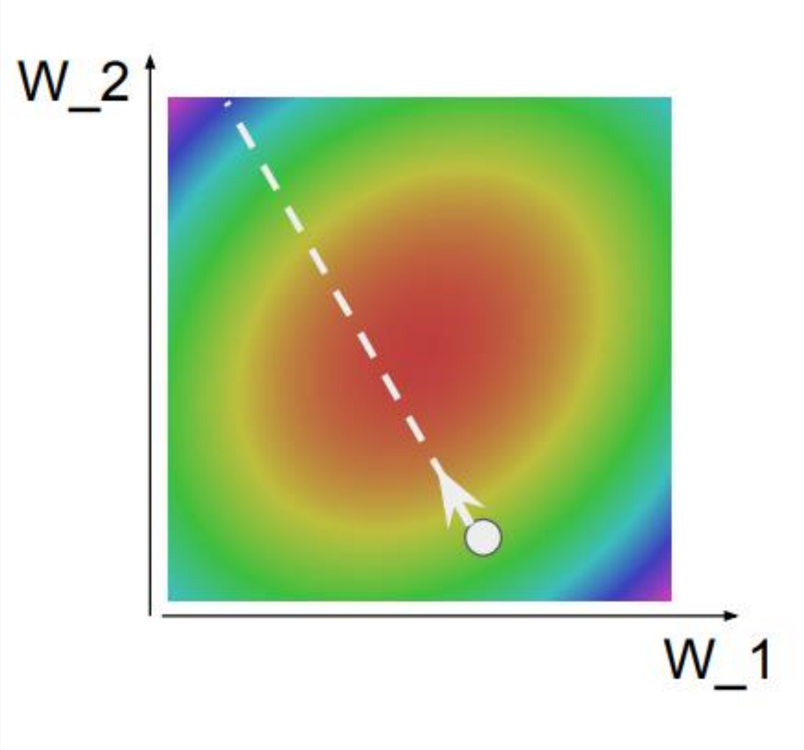
CONTENTS

Contents 1
Optimization

Loss Function 손실 함수

Network의 가중치 Weight으로 정의되며 Loss function 은 그 가중치가 얼마나 좋은지, 나쁜지를 나타냄.

가중치에 대한 landscape으로 표현하면 다음과 같음.
색상은 loss 값을 나타내며, 빨간색이 가장 loss가 낮은 영역. → 목표



Contents 1
Optimization

Stochastic Gradient Descent

Loss function 계산할 때, 전체 데이터 대신 일부 데이터 (Mini batch) 사용하여 loss function 구함.

1. 일부 mini-batch data의 loss in the gradient 얻기
2. Gradient 반대 방향으로 parameter vector 업데이트
3. Repeat
4. 빨간 영역(loss 가장 낮은 영역)으로 converge

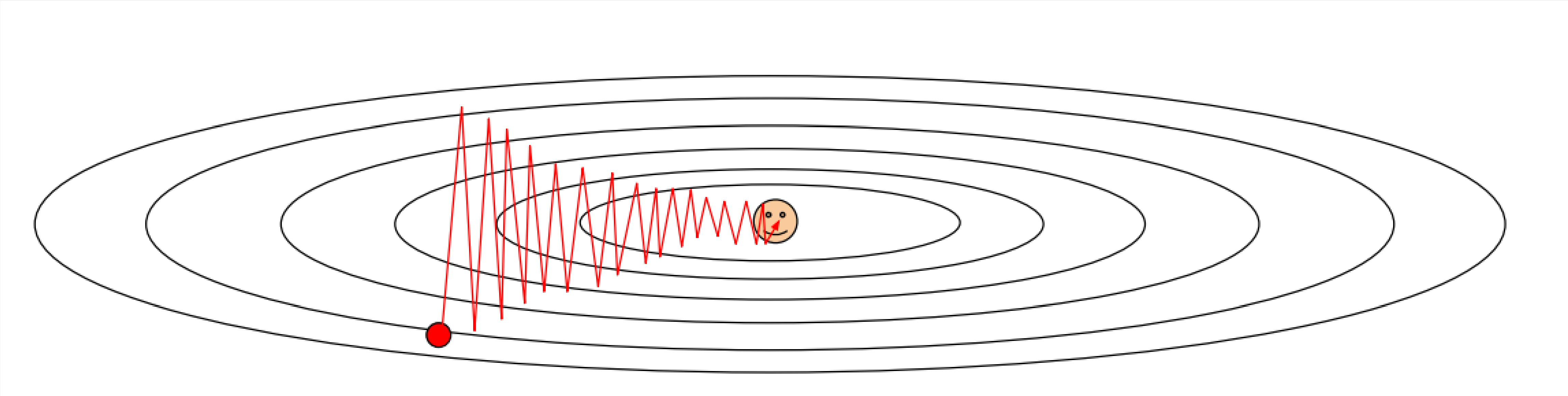
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Contents 1
Optimization

Stochastic Gradient Descent-Problem 1

다음과 같이 loss가 수평 방향보다 수직 방향 가중치 변화에 더 민감하게 반응할 때.

→ Gradient 방향이 고르지 않아 지그재그 형태.
Loss에 영향 적은 수평 방향으로는 느리게 업데이트.



Contents 1
Optimization

Condition number ▼

Ratio of largest to smallest singular value of the Hessian matrix

헤시안 행렬의 가장 큰 단수값 대 가장 작은 단수값의 비율.

Hessian matrix ▼

어떤 함수의 이계도함수를 행렬로 표현한 것.

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Contents 1
Optimization

Stochastic Gradient Descent-Problem 2

•Local minima

Gradient=0 →Opposite Gradient=0→학습 stop

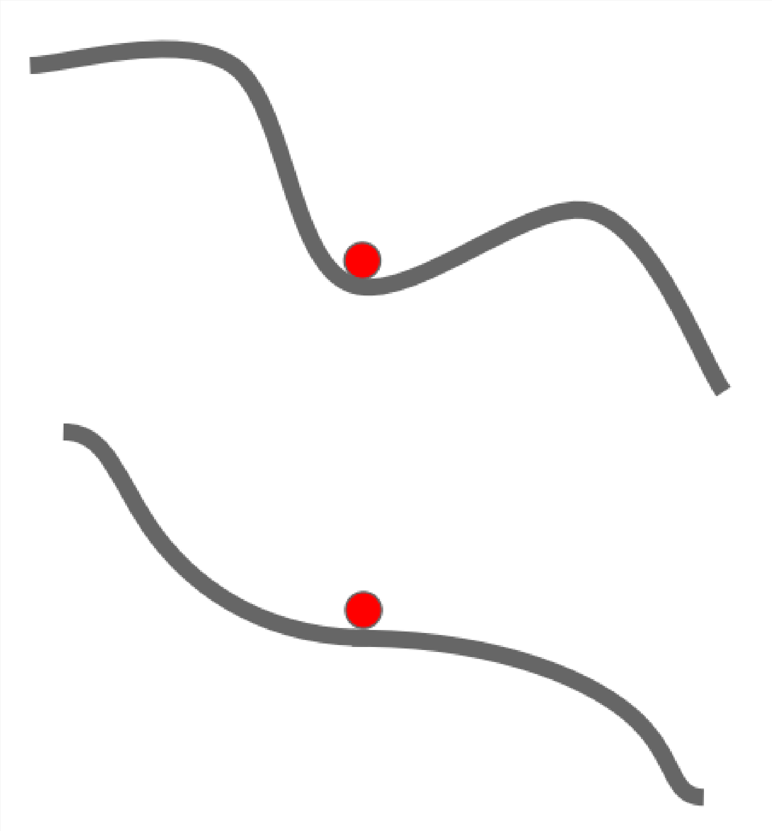
•Saddle Point

한쪽 방향은 기울기 증가, 한쪽 방향은 감소하는 점.

→ Gradient=0

→ Saddle point 주변의 gradient도 0에 가깝기 때문에 업데이트가 굉장히 느림.

→ Neural Network는 saddle point에 더 취약



Contents 1
Optimization

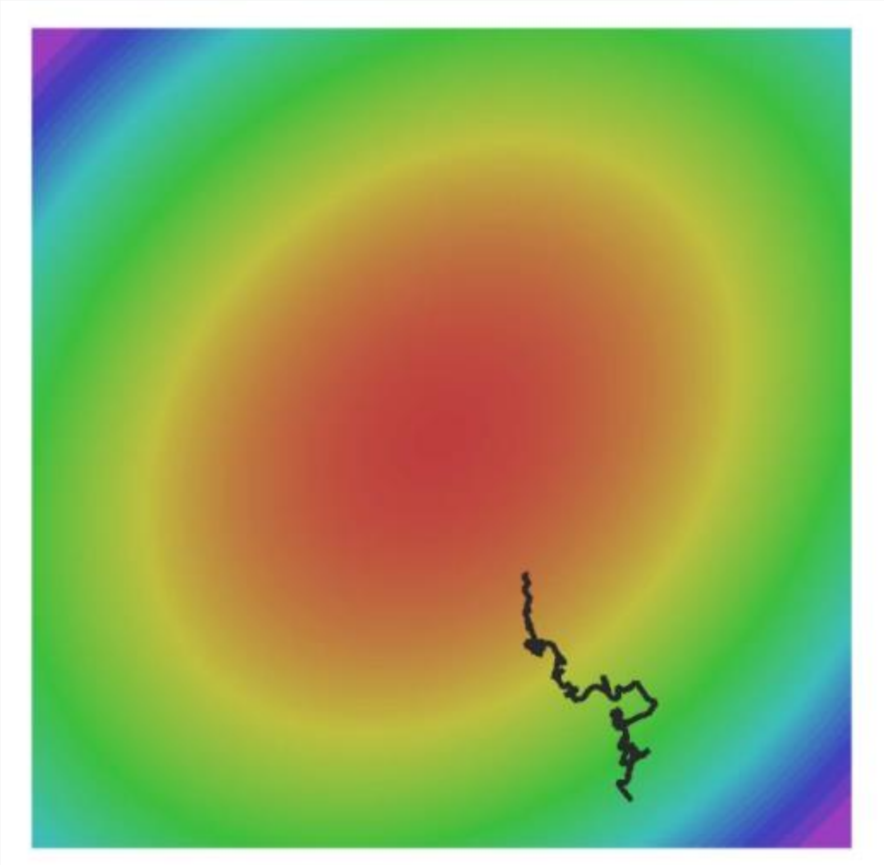
Stochastic Gradient Descent–Problem 3

SGD는 전체 데이터가 아닌 mini batch로 부터 실제 loss를 추정한 것.

→ 진짜 gradient 정보라고 할 수 없음.

→ 부정확한 추정 Noisy estimate

Gradient에 noise가 추가되면 다음과 같이 minima에 달하는 시간이 오래 걸림.



Momentum

Velocity 와 gradient값을 더함.

ρ 는 hyperparameter. 주로 0.9 0.99 같은 값을 사용.

SGD

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

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

SGD+Momentum

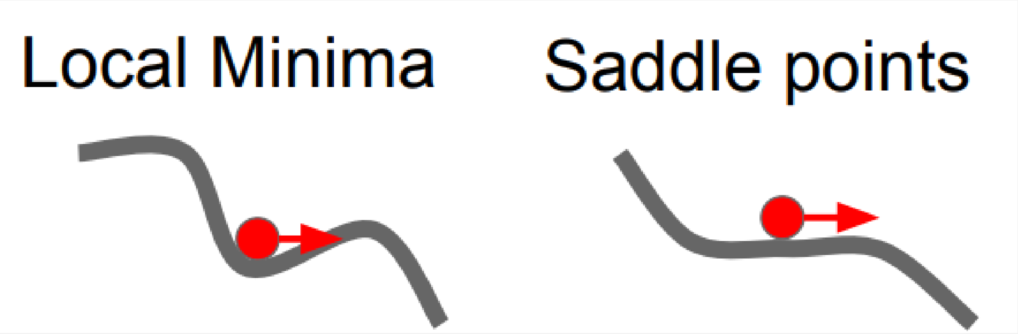
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

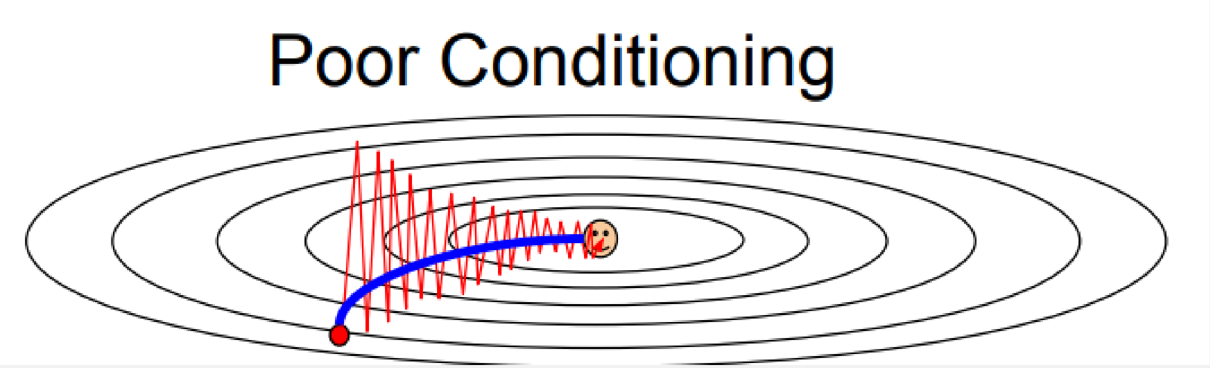
- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

Contents 1
Optimization

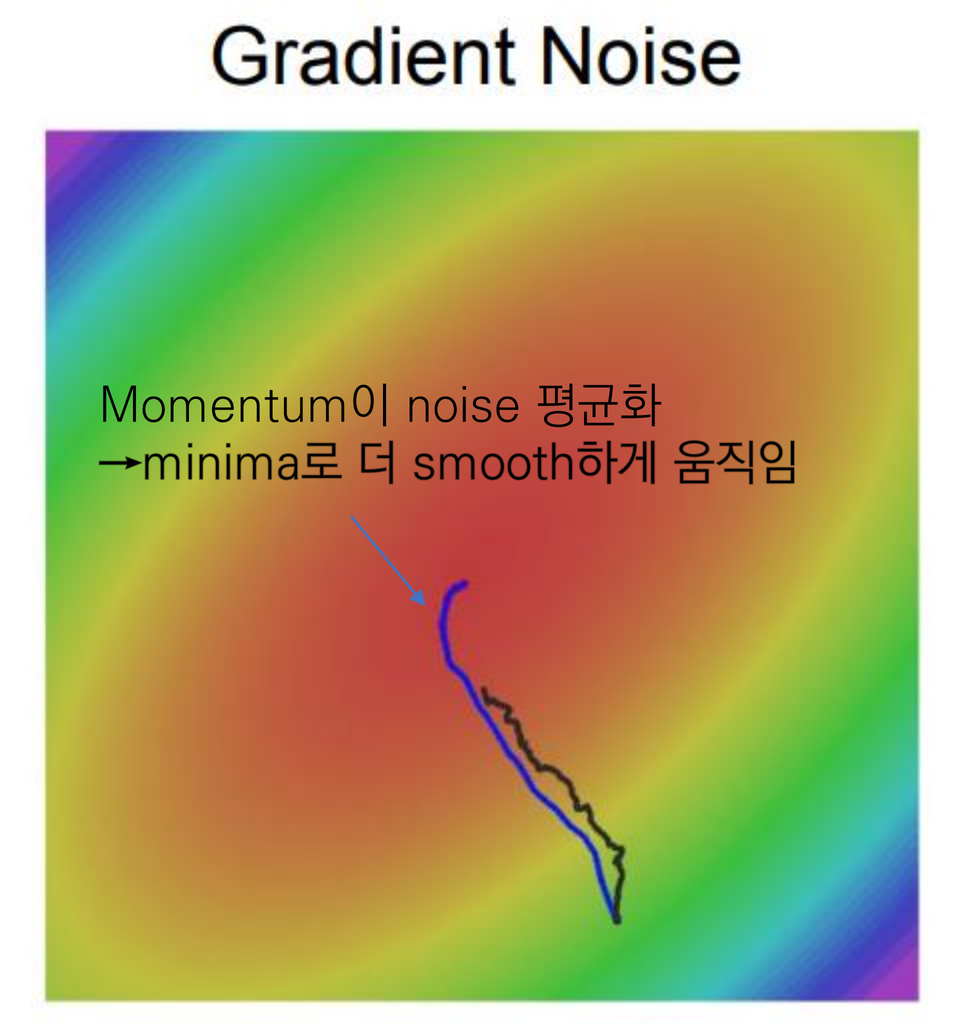
Momentum



Gradient=0 이어도 velocity값이 있으므로 움직임.

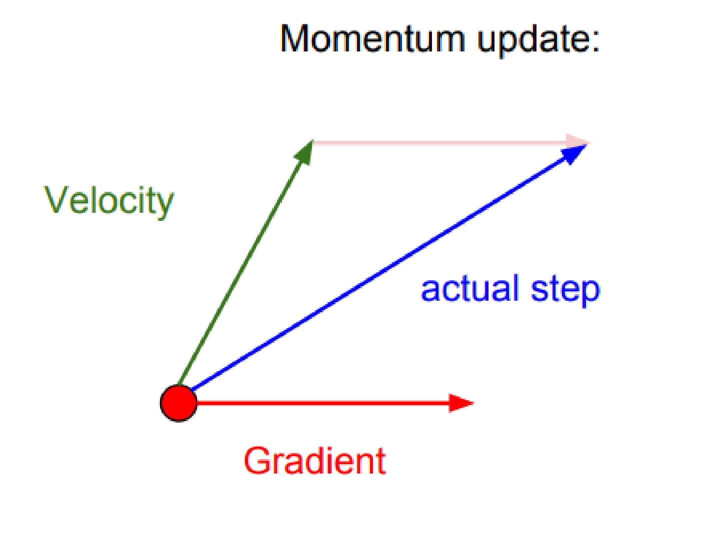


loss에 민감한 수직 방향의 변동은 reduce
수평방향의 움직임은 accelerate



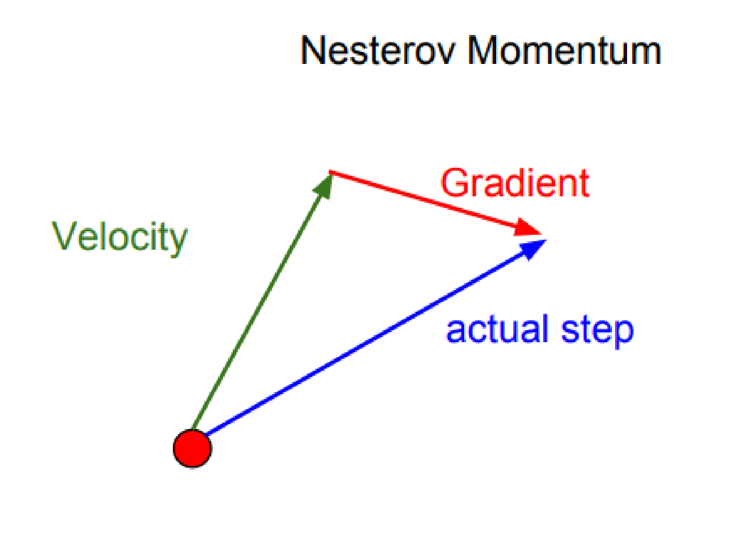
Contents 1
Optimization

Nesterov Momentum



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

현재 지점 gradient + velocity



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

velocity 방향으로 이동 + 그 지점의 gradient
Convex optimization에 뛰어남

Contents 1
Optimization

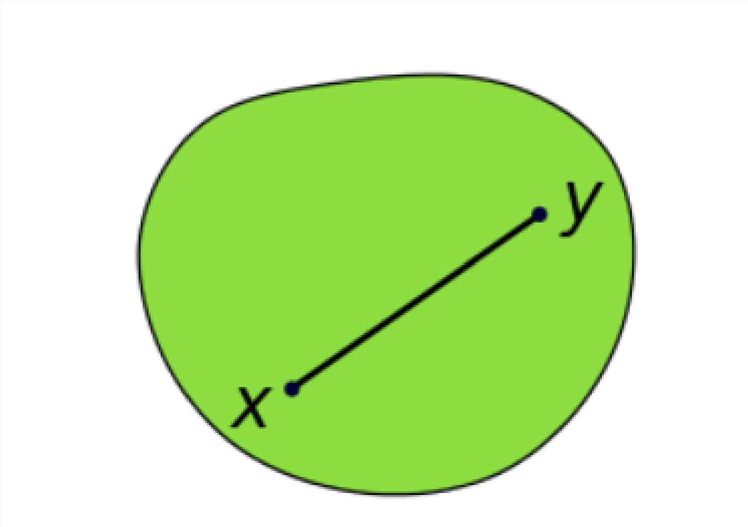
Convex ▼

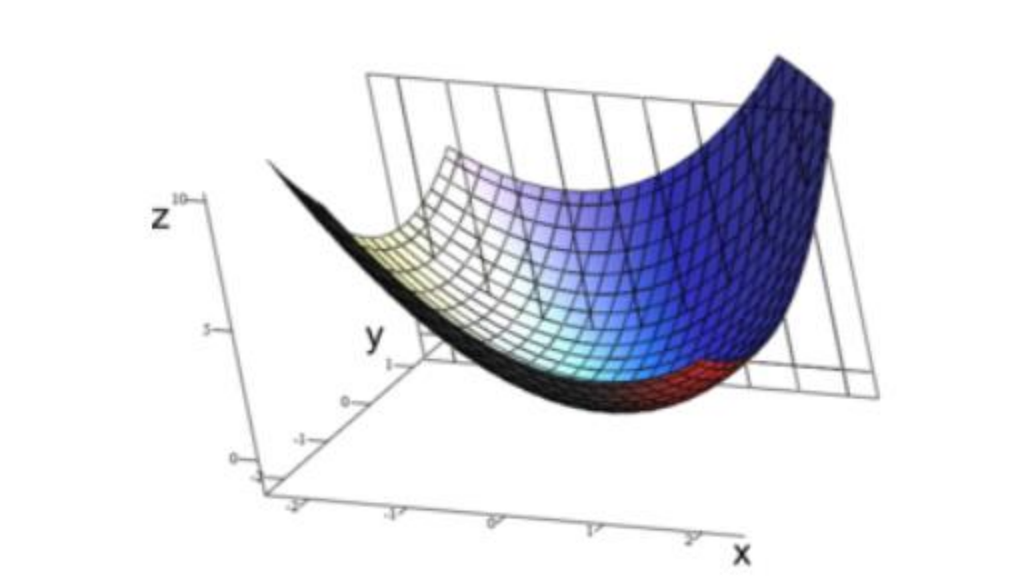
함수 위의 임의의 두 점을 연결하는 선을 그래프에 그었을 때,
그 선이 아래 그림과 같이 함수 그래프의 위쪽만을 지나가는 경우

Ex. 아래로 볼록한 이차함수

집합이 이루는 공간 내의 두 점을 연결한 선분이 그 집합 안에 포함되는 경우

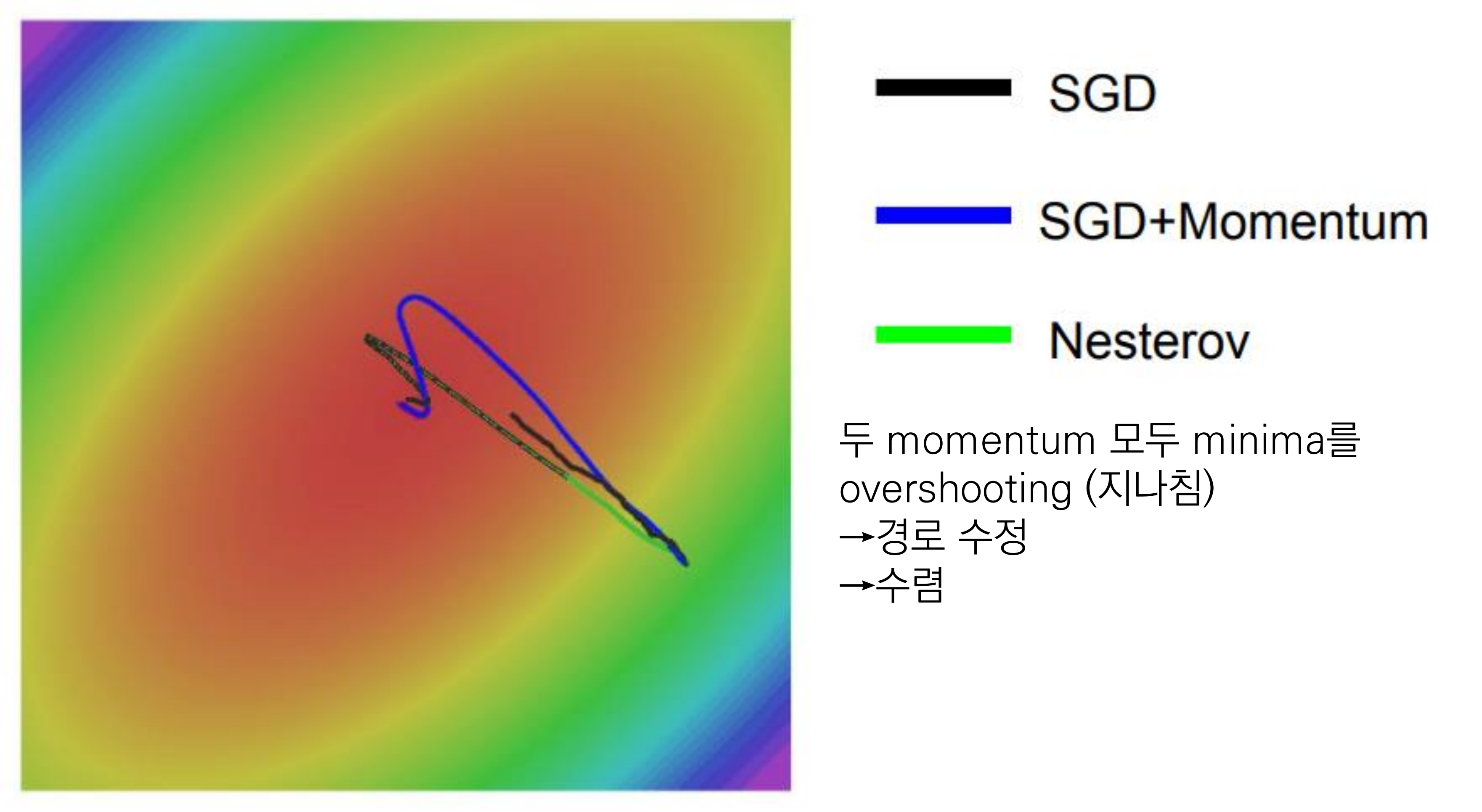
Local minima=Global minima





출처: 위키피디아

Nesterov Momentum



두 momentum 모두 minima를 overshooting (지나침)
→경로 수정
→수렴

AdaGrad

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

→ 훈련 중 계산된 gradient 사용

→ 제곱

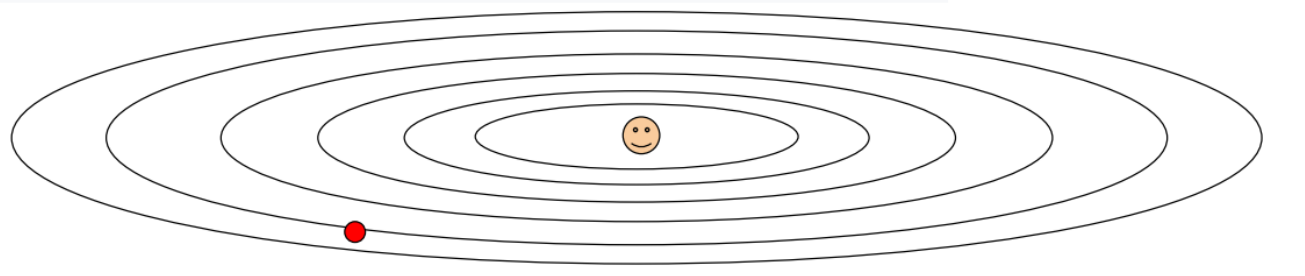
→ 제곱한 값을 나눠 업데이트

→ 0으로 나누지 않도록 작은 값(1e-7) 더함

Contents 1
Optimization

AdaGrad

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



수평: 작은 gradient → 제곱값 합 작음 → 작은 것으로 나눔 → 큰 값 → 빨라짐

수직: 큰 gradient → 제곱값 합 큼 → 큰 것으로 나눔 → 작은 값 → 느려짐

AdaGrad-Problem

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

Gradient 제곱이 계속 더해짐 → 제곱값 합 증가 → Step size 작아짐

Convex 경우: minimum에 가까워질수록 작은 step size로 수렴
Non-convex 경우: saddle point에 걸려 멈출 수 있음

Contents 1
Optimization

RMSProp

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

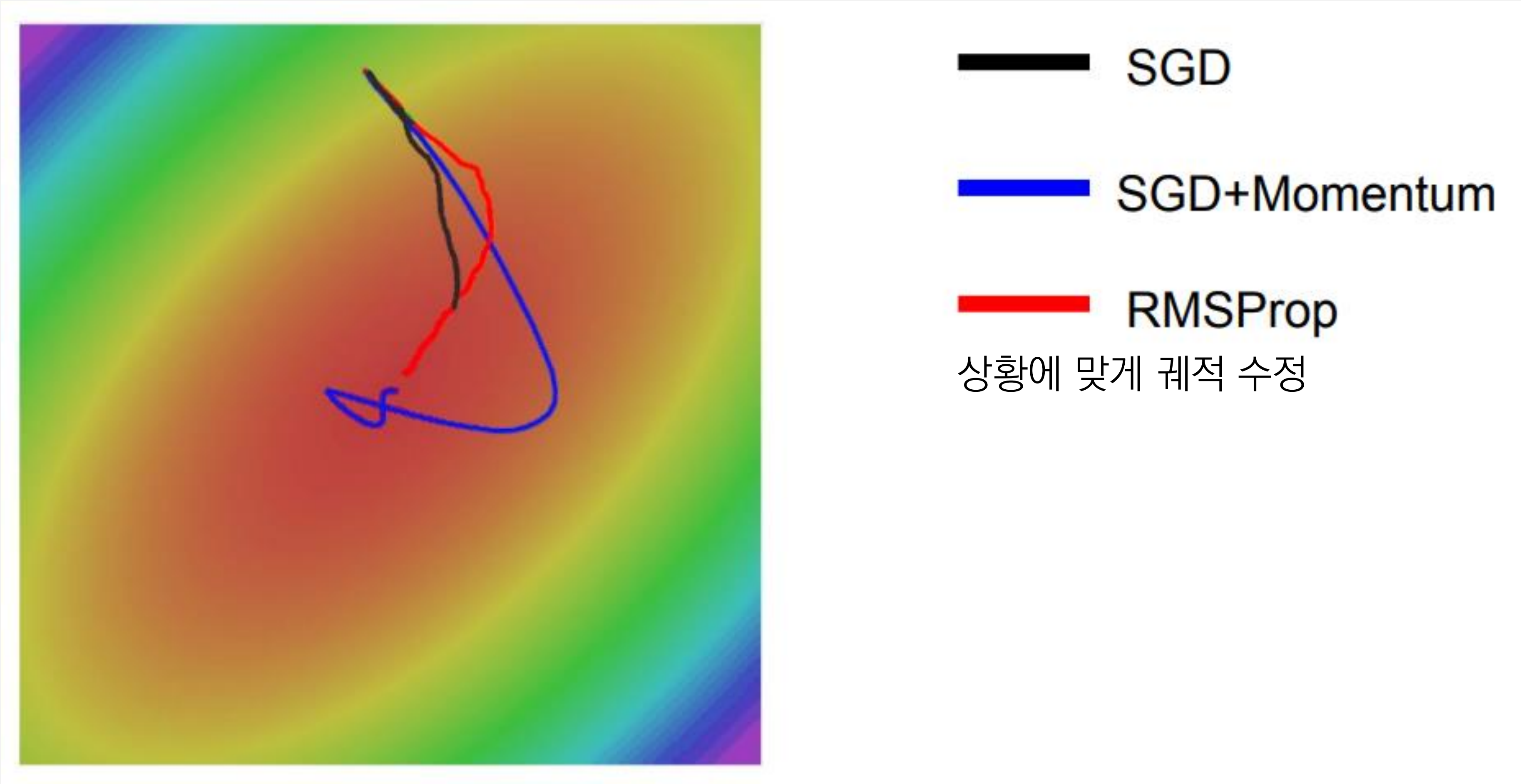
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

gradient의 제곱을 계속 누적함

Decay rate 이용해 step 속도 감속/가속 조절

Contents 1
Optimization

RMSProp



상황에 맞게 궤적 수정

Contents 1
Optimization

Adam (almost)

```
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

Momentum과 RMSProp 합친 느낌.

At first timestep,

Second momentum 0으로 초기화 → 1회 Update → beta2(=decay_rate) = 0.9 / 0.99
→ 여전히 0에 가까움 → 작은 값으로 나눔 → 큰 값 → 초기 step이 인공적으로 커짐

Contents 1
Optimization

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(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

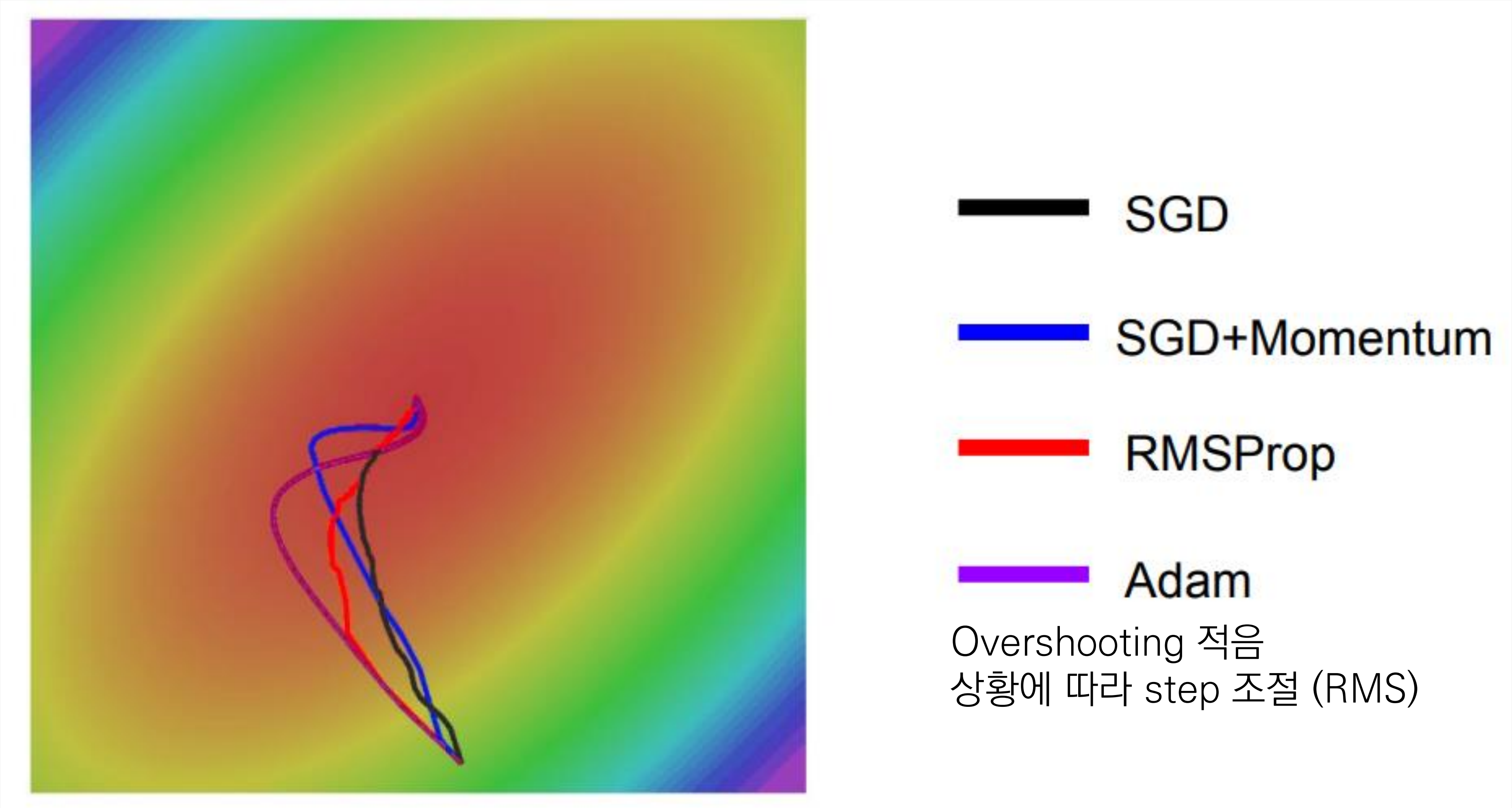
AdaGrad / RMSProp

초기 step이 너무 커지지 않도록 방지함.

beta1=0.9 beta2=0.99 learning_rate=1e−3 / 5e−4 일 때 거의 모든 모델에서 적절함

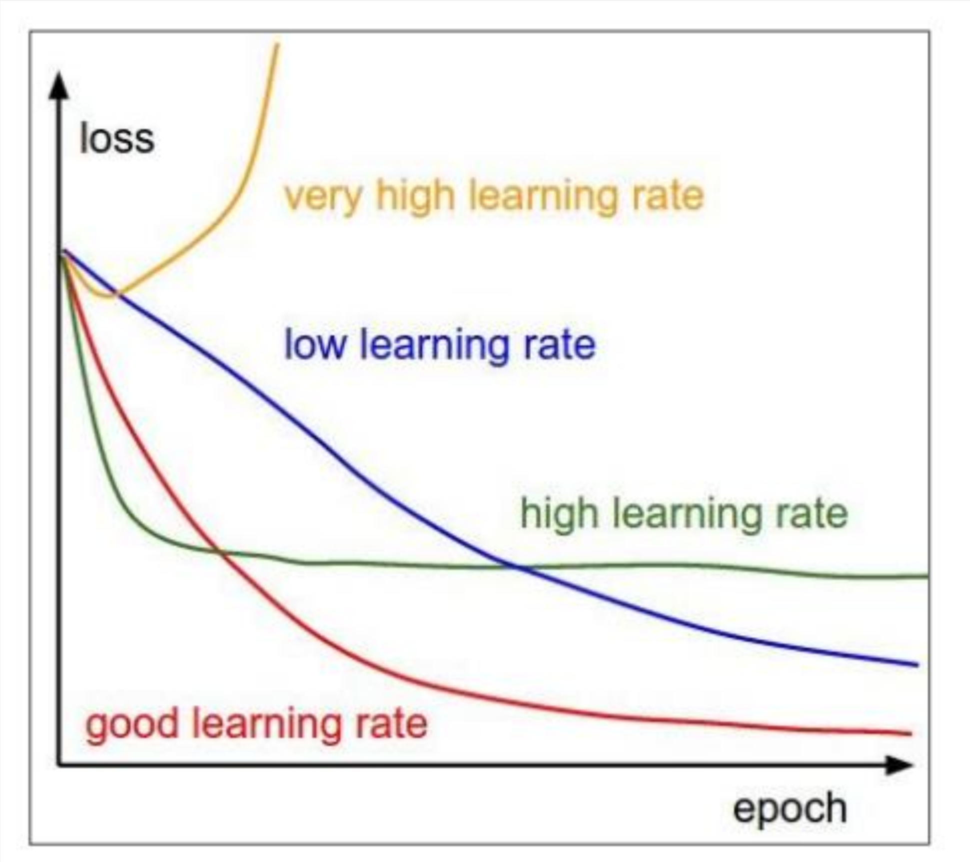
Contents 1
Optimization

Adam (full form)



Overshooting 적음
상황에 따라 step 조절 (RMS)

Learning rate



exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

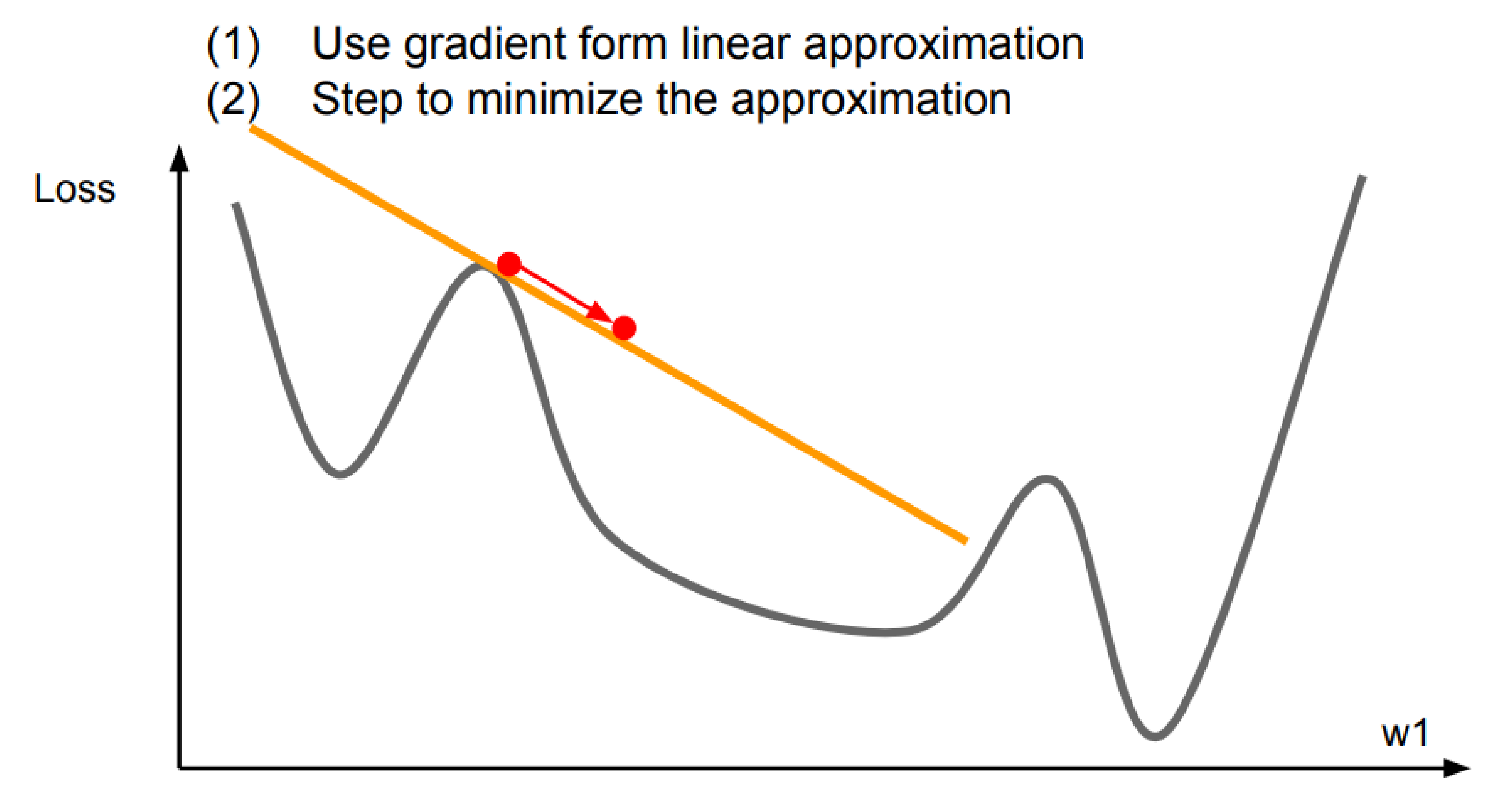
$$\alpha = \alpha_0 / (1 + kt)$$

Learning rate decay: 학습 진행될수록 점점 낮춤

참고: https://velog.io/@good159897/Learning-rate-Decay%EC%9D%98-%EC%A2%85%EB%A5%98

Contents 1
Optimization

First-Order Optimization

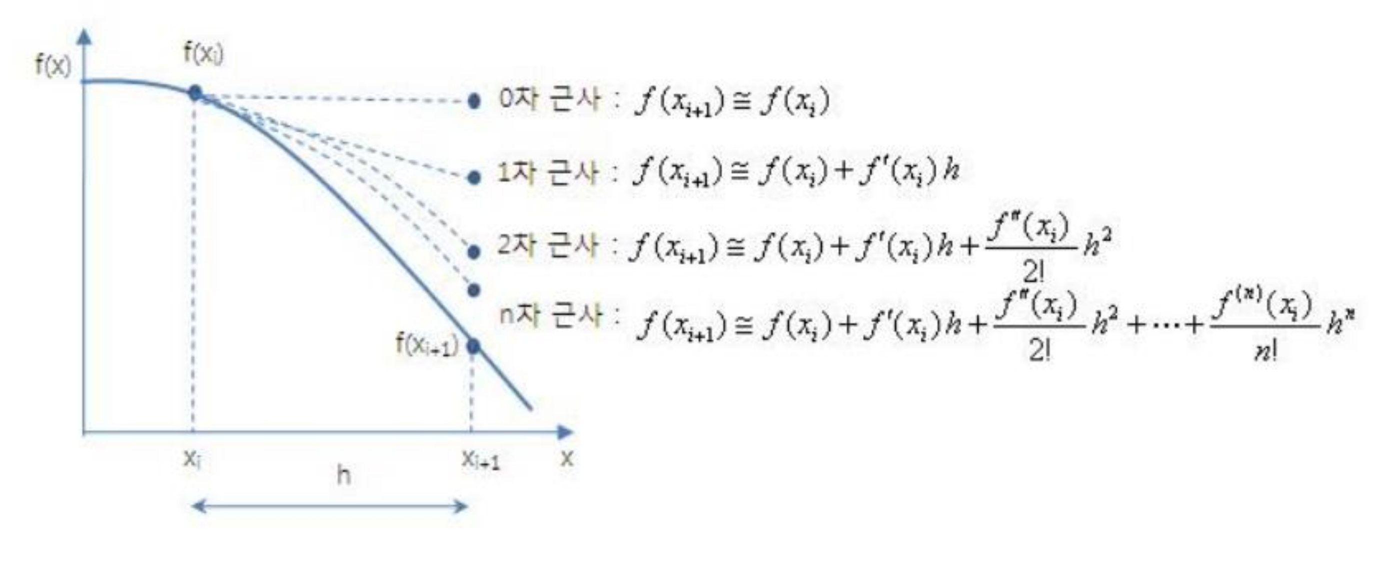


Contents 1
Optimization

테일러 급수 ▼

어떤 미지의 함수 f(x)를 아래 식과 같이 근사 다항함수로 표현하는 것

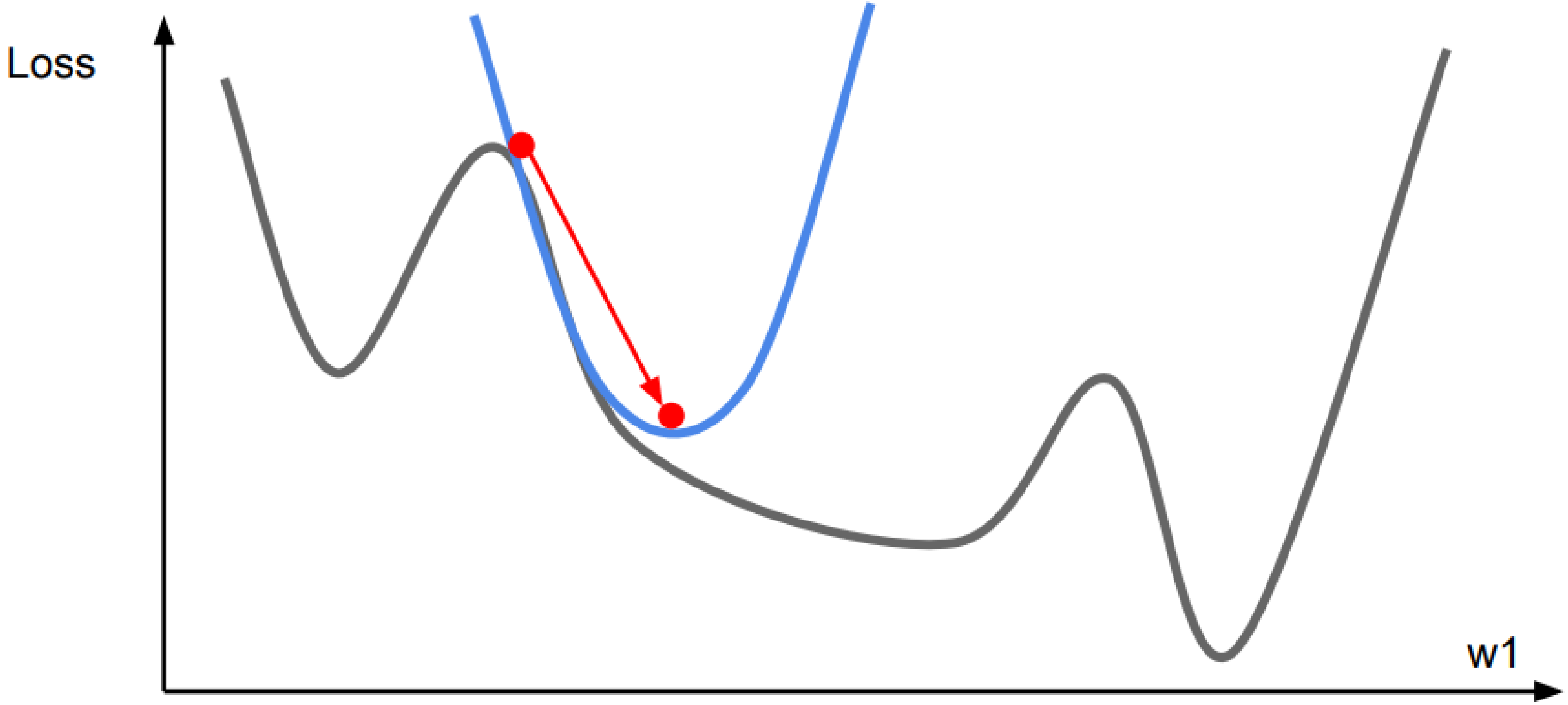
n차 테일러 근사 ▼



Contents 1
Optimization

Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Contents 1
Optimization

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

No hyperparameters!
No learning rate!

매 step마다 minima로 이동하기 때문에 hyperparameter 불필요

H (Hessian matrix)가 N x N 행렬이기 때문에 크기가 너무 커져 딥러닝에서는 사용 불가

Second-Order Optimization

- Quasi-Newton methods
  Hessian을 근사해서 사용

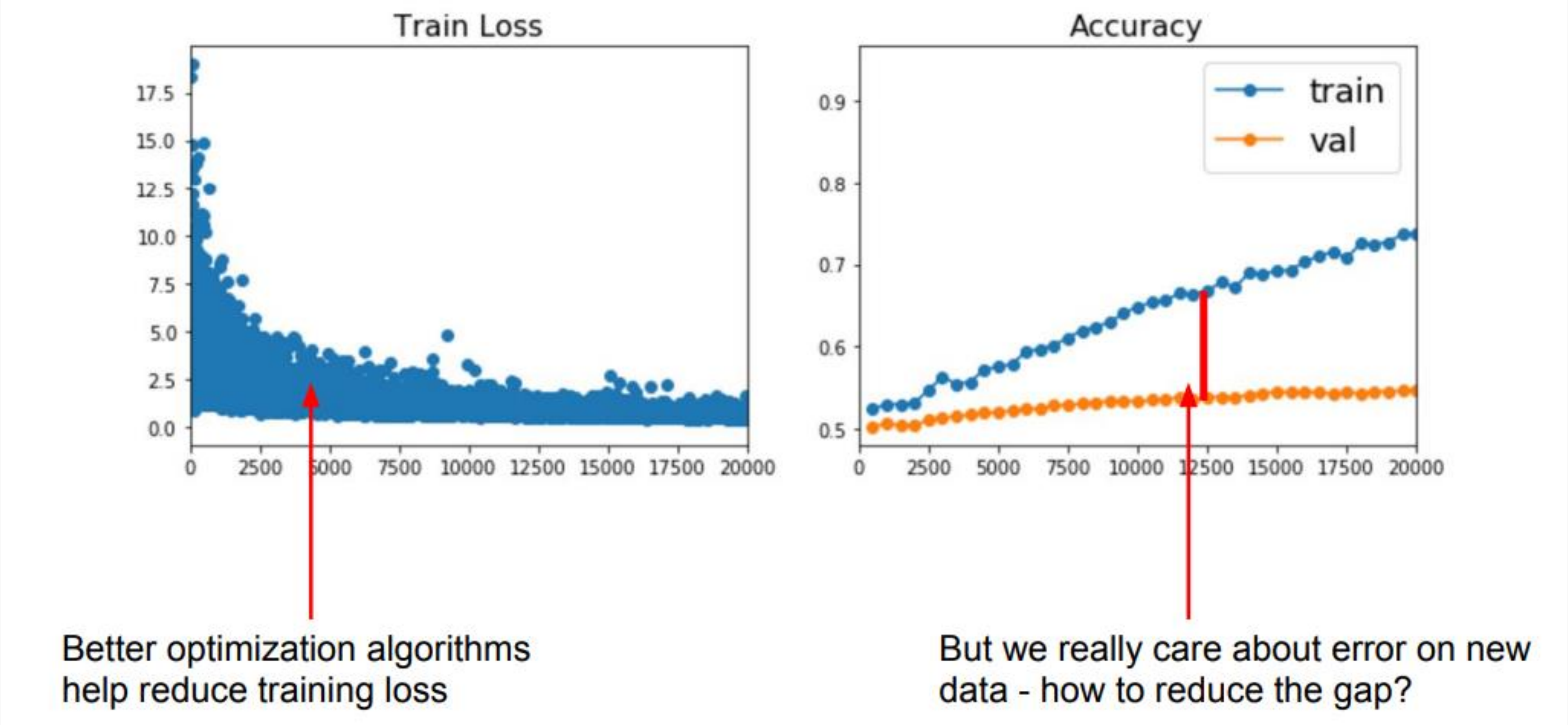
- L-BFGS
  Hessian을 근사해서 사용
  2차 근사가 stochastic case에서 잘 작동하지 않아 사용하지 않음

Contents 2
Regularization

Good model?

- Optimization
  Training error를 줄이고 Loss function 최소화

But, 중요한 건 train/test 사이 error 줄이기. Validation set 성능 최대화.



Model Ensemble

여러 모델을 개별적으로 학습시키고 그 평균을 구하는 방법

① 보팅(Voting)

여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정하는 방식. 서로 다른 알고리즘을 여러 개 결합.

-하드 보팅: 다수의 분류기가 예측한 결과를 최종 예측 결과로 결정

-소프트 보팅: 모든 분류기가 예측한 레이블 값의 결정 확률 평균을 구한 뒤 가장 확률이 높은 레이블 값을 최종 결과로 선정

② 배깅(Bagging)

데이터 샘플링(Bootstrap)을 통해 모델을 학습시키고 결과를 집계(Aggregating) 하는 방법. 같은 유형의 알고리즘을 사용.

Overfitting 방지에 효과적. Ex.랜덤 포레스트

③ 부스팅(Boosting)

여러 개의 분류기가 순차적으로 학습을 수행하고, 이전 분류기가 예측이 틀린 데이터에 대해서 올바르게 예측할 수 있도록 다음 분류기에게 가중치(weight)를 부여하면서 학습과 예측을 진행.

예측 성능이 뛰어나지만, 속도가 느리고 Overfitting이 발생할 가능성이 존재.

Dropout-Advantage

• Co-adaptation
특성이 집단에 유리할 경우: 더 잘 발현되게 진화
특성이 집단에 불리할 경우: 퇴화되고 발현되지 않음

→ 근접한 feature들이 서로 영향을 받음
→ Overfitting 발생

→ Dropout 통해 방지

→ Dropout은 단일 모델로 ensemble 효과를 낼 수 있음

Batch Normalization

Train 과정: Random mini batch의 평균, 분산을 이용해 정규화
→ Stochasticity. Randomness 존재

Test 과정: Global 평균, 분산을 이용해 정규화
→ Regularization

Training: Add some kind of randomness

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

Testing: Average out randomness (sometimes approximate)

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

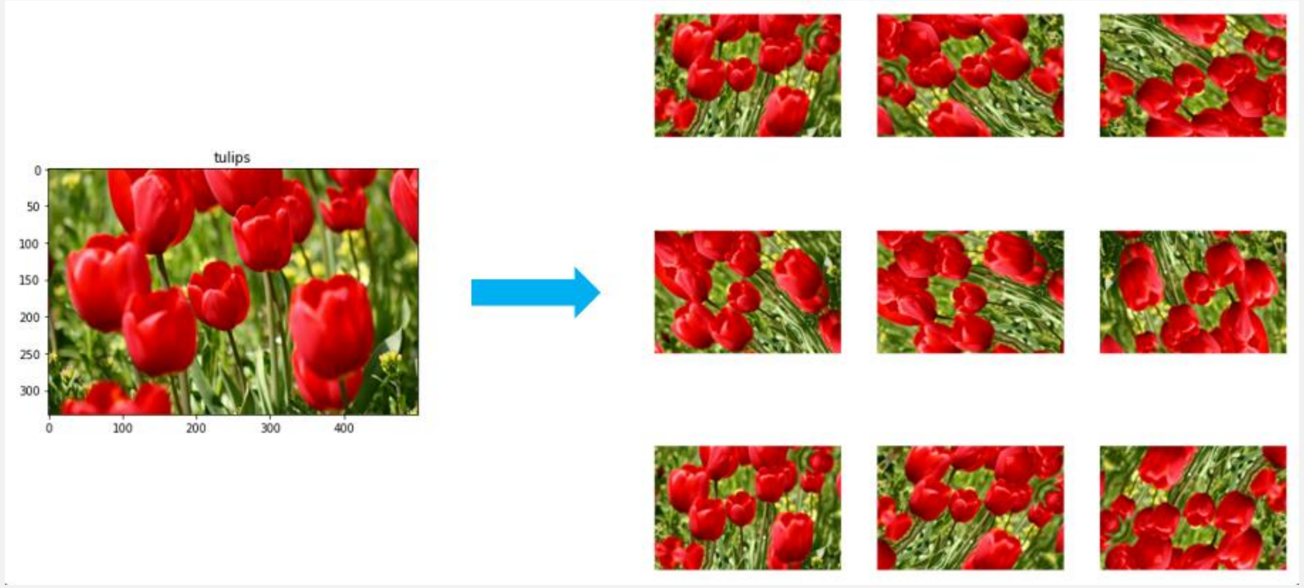
Contents 2
Regularization

Data Aumgentation

하나의 데이터로 여러 개의 데이터를 만들어내는 등의 기법

데이터를 증강시켜 overfitting 해결

Ex. Translation, rotation, stretching, color transformation

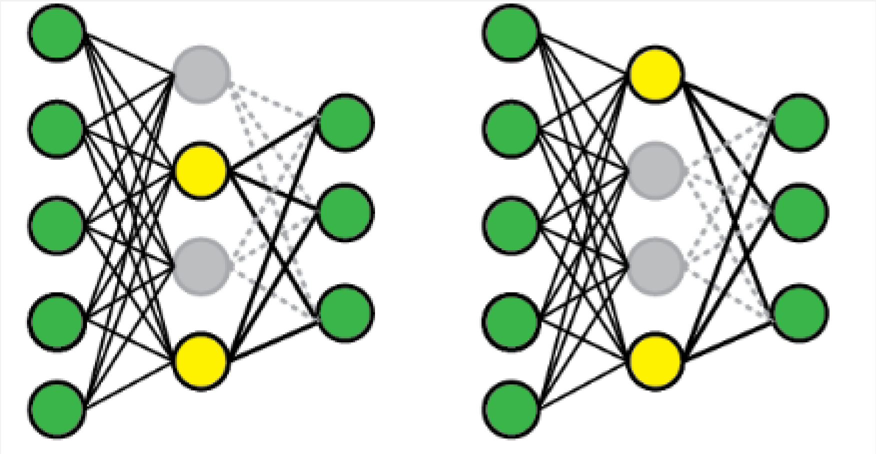


참고: https://foreverhappiness.tistory.com/112

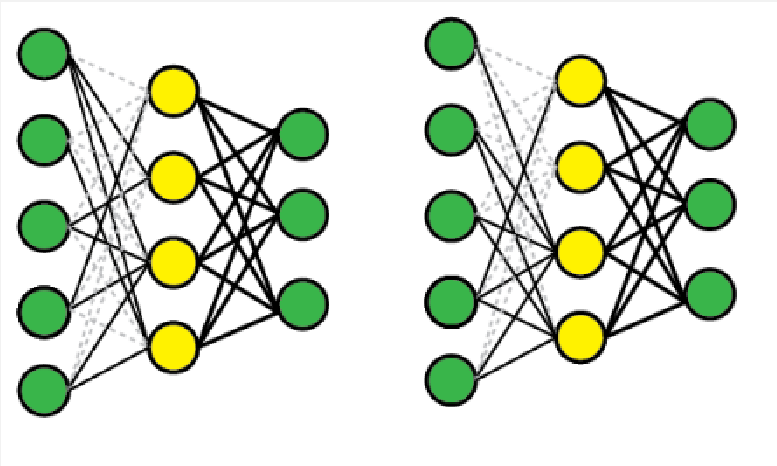
Contents 2
Regularization

Drop Connect

Weights를 임의로 비활성화 시키고, node는 그대로 활성화 되어 있는 방법



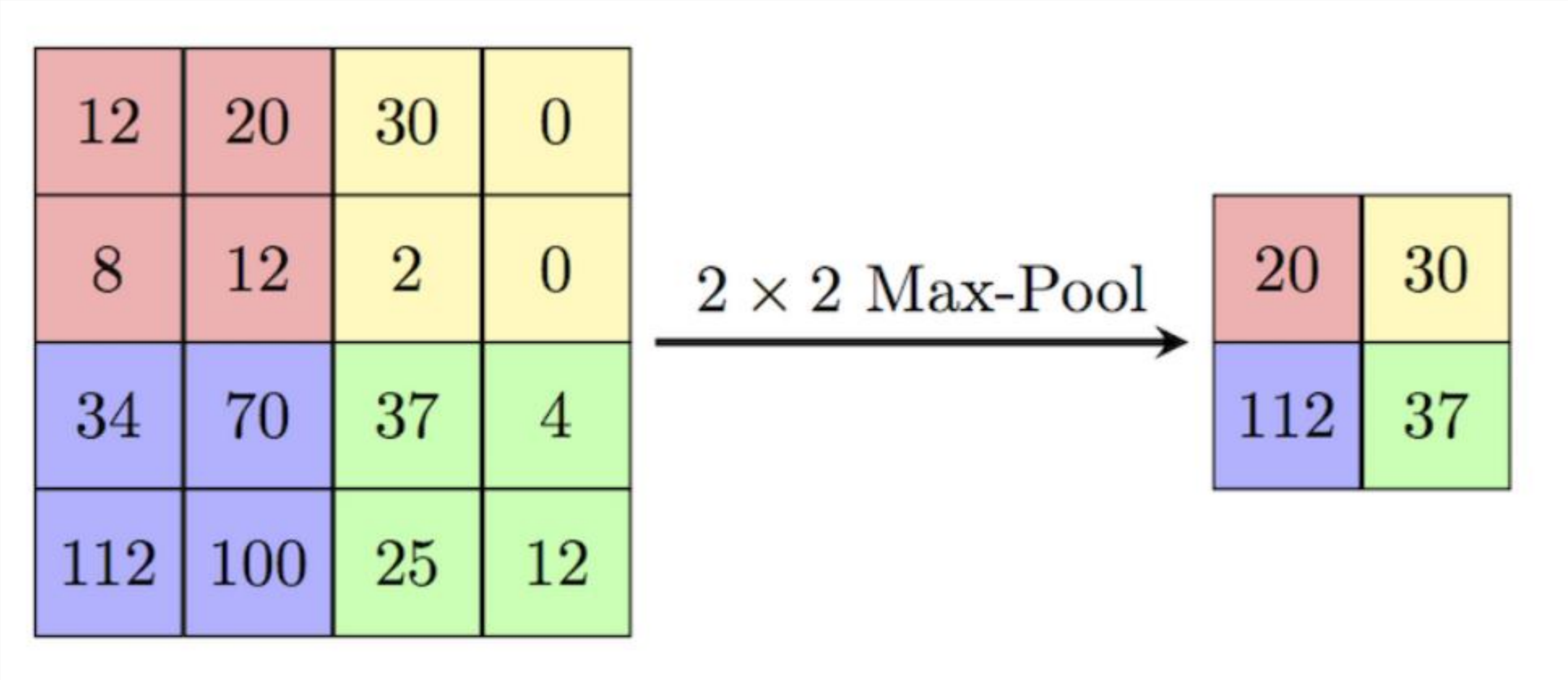
Dropout



DropConnect

Fractional max pooling

Pooling연산을 수행 할 지역이 임의로 선정

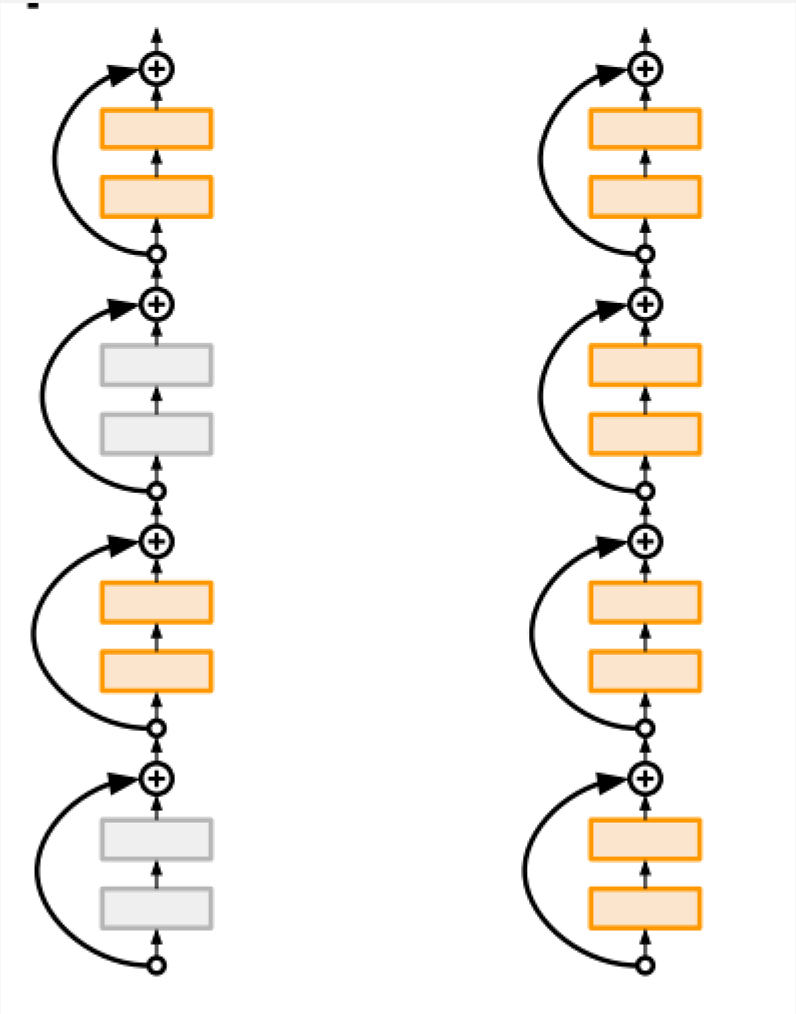


Contents 2
Regularization

Stochastic depth

Train 과정: Layer 중 일부를 제거하고 일부만 사용해서 학습 (왼쪽)

Test 과정: 모든 layer 사용 (오른쪽)



Depth감소 →training 중의 forward propagation 및
gradient computation의 chain 감소

일종의 ensemble 효과 (network에 대해 다른 depth를 사용)

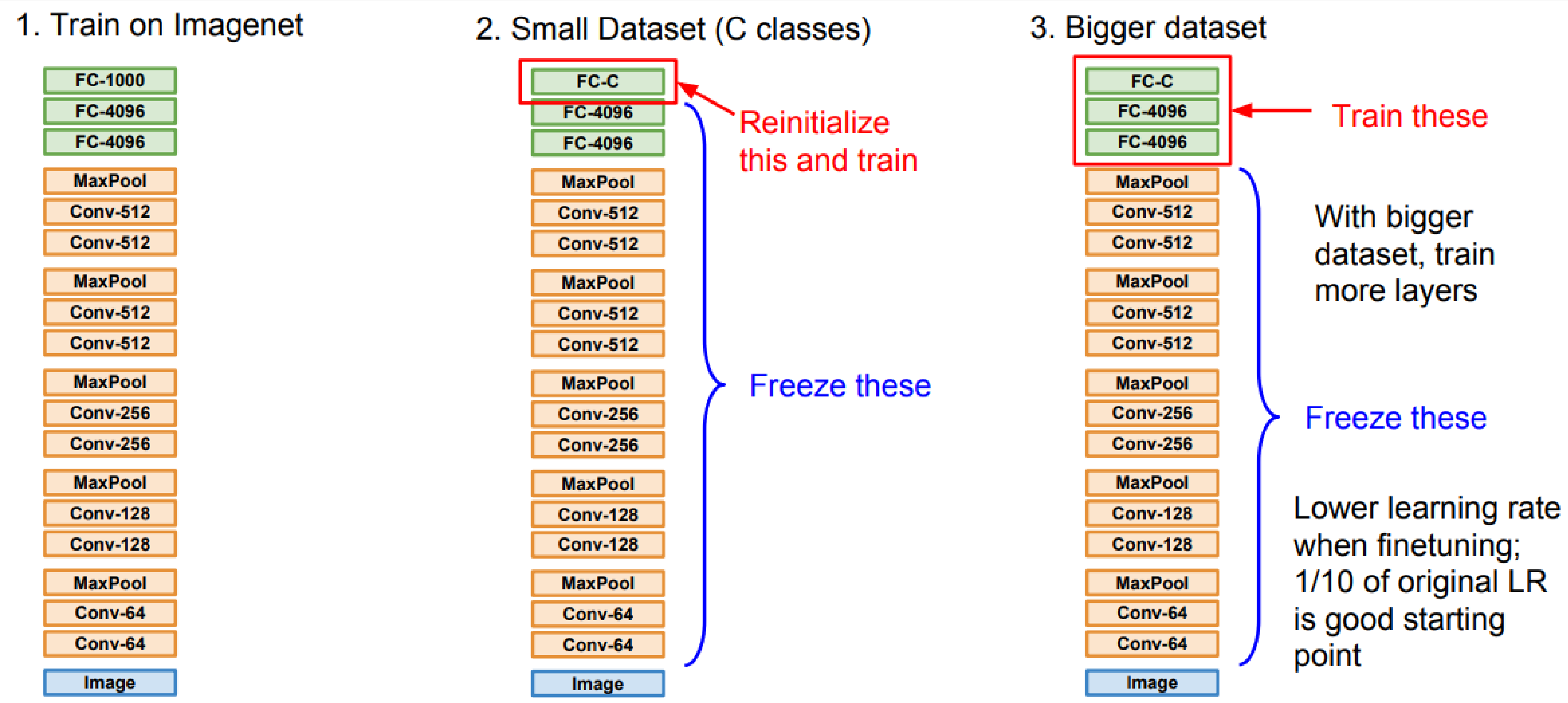
참고: https://deepseow.tistory.com/26

Contents 3
Transfer Learning

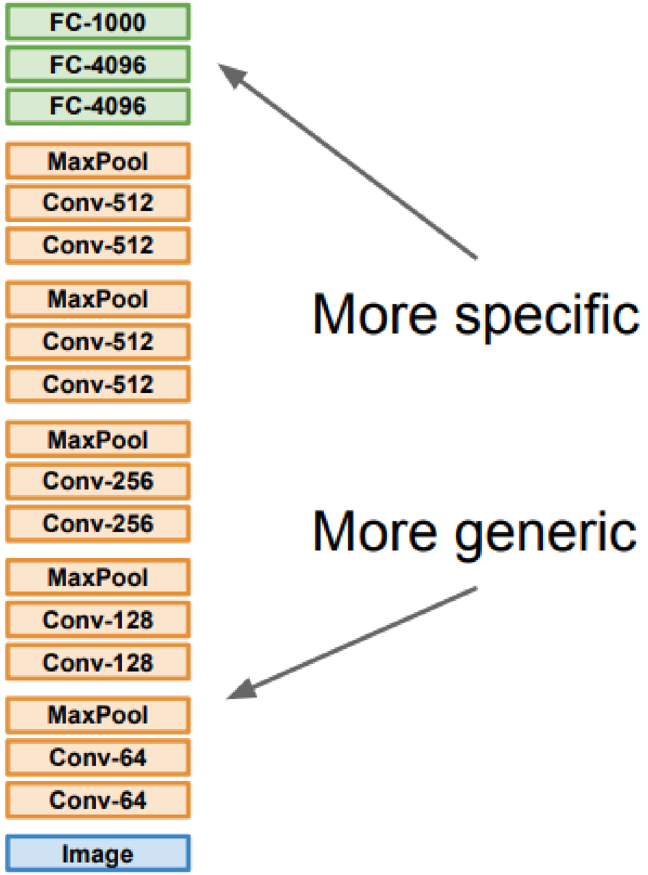
Transfer Learning

모델을 ImageNet 과 같은 방대한 dataset을 학습시킴

→ 우리가 가진 작은 dataset에 적용



Transfer Learning



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Thank you