

# Training Neural Networks: Part 2

## Optimization

### SGD의 문제점

#### 1. 학습 과정 비효율 문제

만약 loss가 한 방향으로는 sensitive하고 다른 방향으로는 느리게 움직일 때 GD는,

- 가파른 방향은 jitter(진동함)
- 느리게 움직이는 방향으로 천천히 진행

→ minima 방향으로 optimize 되지 않음(zigzag) ⇒ 바람직하지 않음, 비효율적

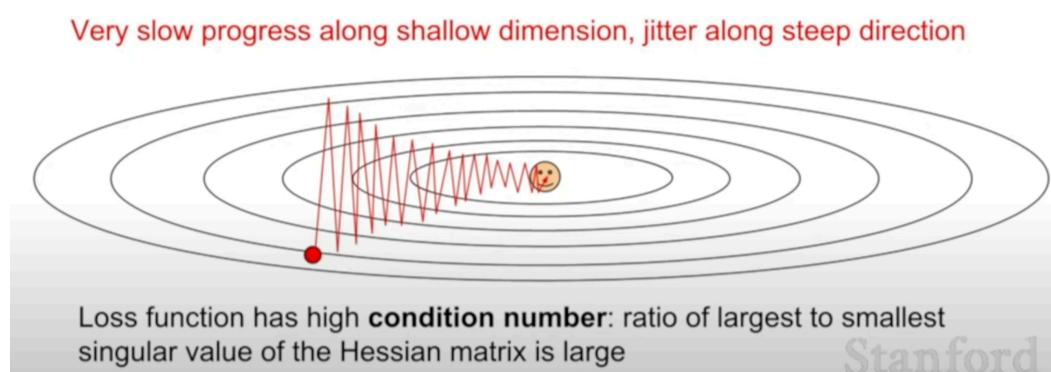


그림 자료에서는 단지 2차원의 예시만 보여주지만,  
다차원의 공간에서는 차원 간의 progress 차이가 더 확연할 것

Loss function은 높은 condition number를 가지고 있을 때 위와 같은 문제 발생

condition number: 입력의 작은 변화가 출력에 얼마나 큰 영향을 주는지 측정하는 수치

- 값이 크면 sensitive ⇒ 수치 계산이 불안정

#### 2. Local minima와 saddle point에 취약한 문제

Local minima

: 어떤 지점 주변에서, 그 지점이 모든 이웃보다 함수값이 작은 지점

Saddle point

: 극값이 아님에도 gradient가 0인 지점

#### 3. minibatch의 gradient가 noisy할 가능성이 있는 문제

## Momentum

### SGD

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

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

### SGD + Momentum

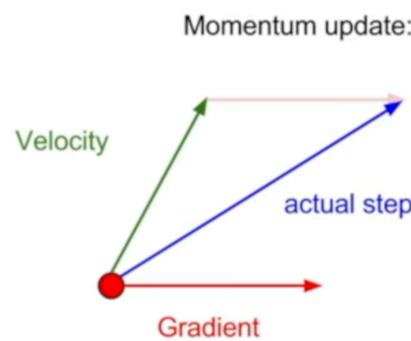
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```

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

```



Raw gradient에 velocity  $v_t$ 를 반영해 stepping

Gradients의 지수 이동 평균(running mean)인 velocity  $v_t$ 를 도입

⇒ 이전 gradients과의 평균 방향을 기억하면서 업데이트

- Momentum을 적용하면, 단순히 현재 gradient의 방향이 아닌 velocity  $v_t$ 를 반영한 방향으로 업데이트
- $\rho(0\sim1)$ 는 마찰(friction)처럼 작용해서 속도가 갑자기 커지거나 방향이 튀지 않도록 방지
- $\rho$ 가 낮을수록 과거 정보(gradient)를 빠르게 잊고 급격하게 감속  
→ 통상적으로 0.9 or 0.99 사용

## Momentum: SGD의 문제해결

### 1. 학습 과정 비효율

"High condition number 문제(Zigzagging)" 해결

Sensitive한 방향의 영향을 효과적으로 줄이고, less sensitive한 방향은 계속 build up해서 loss 감소를 촉진시킴

### 2. Local minima와 saddle point에 취약

"Gradient가 0이 되는 상황(local minima, saddle point)" 해결

: Momentum을 통해 (raw gradient보다) velocity를 통해 업데이트하므로 강건함

## Nesterov Momentum

: 먼저 momentum 방향으로 살짝 가본 후, 그 위치에서 gradient를 측정해 더 정확히 step

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

$$x_{t+1} = x_t + v_{t+1}$$

위 수식에서  $\tilde{x}_t = x_t + \rho v_t$ 로 변수를 바꿔서 다시 정리해보면,

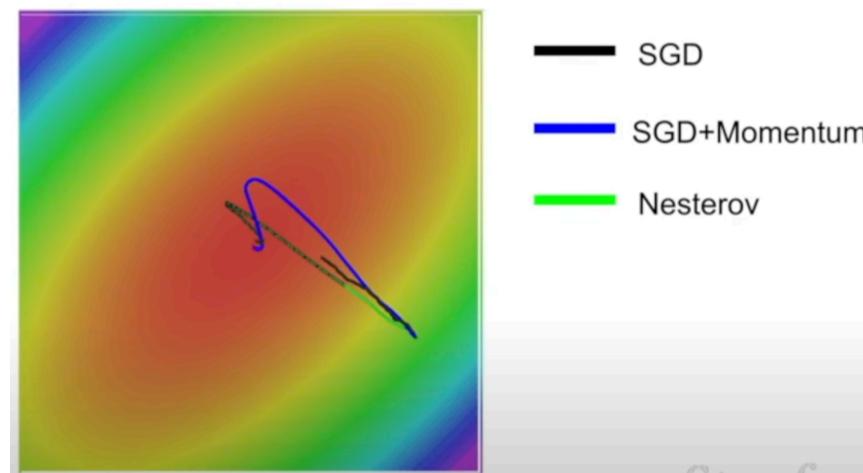
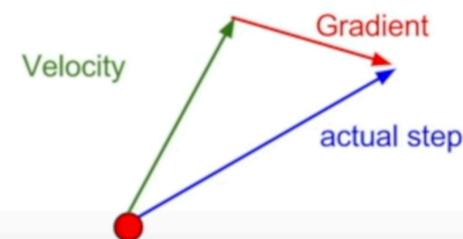
$$\begin{aligned}
 v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\
 \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1} \\
 &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)
 \end{aligned}$$

```

dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v (1 + rho) * v

```

Nesterov Momentum



Vanila momentum에 비해 trajectory가 안정적이고 빠르게 수렴  
⇒ less overshoot

## AdaGrad

```

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

```

Momentum의 velocity term 대신 grad squared term 사용

- Training 동안, gradient를 제공한 값을 grad squared term에 계속 더함
- 이 후, weight를 업데이트할 때, gradient squared term으로 나눠 줌

그렇다면, very high condition number 상황에서 Adagrad의 양상?

- less sensitive한 축은 업데이트 폭이 작고, 그 작은 step size를 작은 숫자로 나눠줌
- sensitive한 축은 업데이트 폭이 크지만, 그 큰 step size를 큰 숫자로 나눠줌

결과적으로 progress 수준이 적절하게 조정돼 안정적으로 학습할 수 있음

하지만 grad squared term은 갈수록 증가하므로 훈련을 거듭할수록 step size가 매우 작아짐  
⇒ 학습이 매우 더딤, non-convex case에서 큰 문제 (saddle point에 취약)

## 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)
```

Adagrad의 취약점을 보완하기 위해 하이퍼파라미터 decay\_rate 추가

- gradient squared term이 훈련이 거듭할수록 decay  
→ gradient squared term에 대한 momentum과 같음
- decay\_rate는 통상적으로 .9 or .99를 사용

Adagrad의 취약점은 어느정도 해결했지만, 훈련 중 원치않게 느려지는 현상이 자주 발생

## Adam

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

- first\_moment : Momentum과 같은 기능
  - second\_moment : AdaGrad / RMSProp과 같은 기능
- second\_moment의 초기값을 0으로 했을 때, 첫 step에서 매우 큰 step size로 시작하게 됨

위 취약점을 보완하기 위해 Bias correction(unbias estimates) 추가

```
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
    # add bias correction
    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)
```

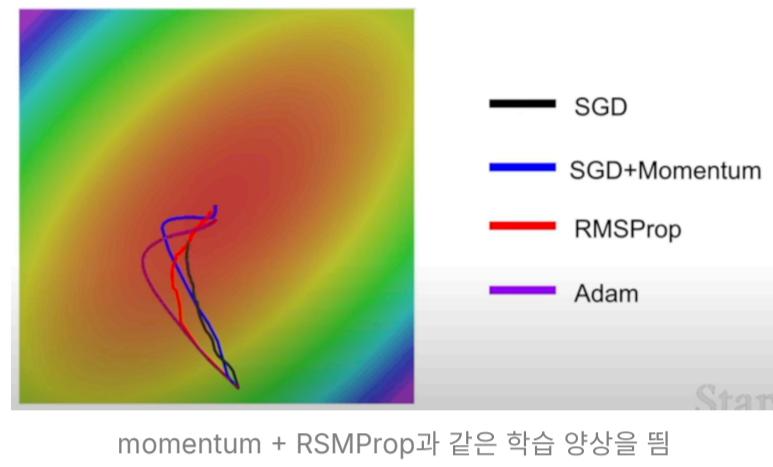
## Adam hyperparameter 추천 초기값

beta1 = 0.9

beta2 = 0.999

learning\_rate = 1e-3 or 5e-4

## Adam 학습 양상



1) momentum보다 적당히 overshoot, 2) RMSProp처럼 모든 축을 따라 똑같이 progress

## Learning rate decay

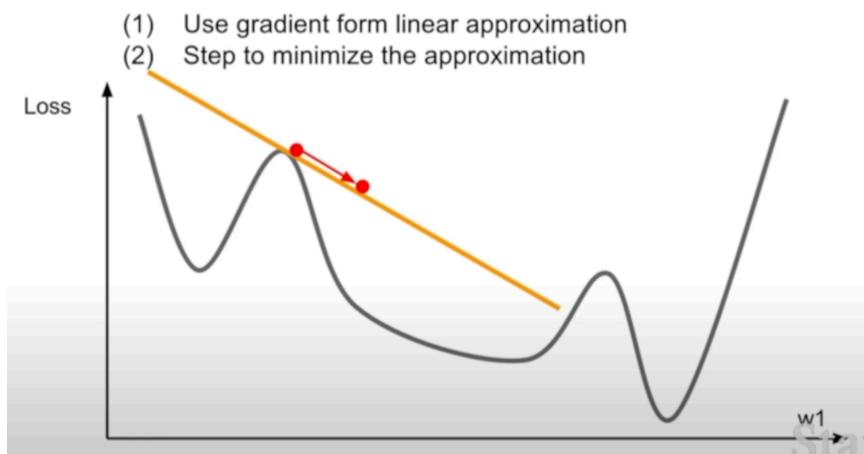
처음에는 큰 learning rate로 시작해서 점점 학습률을 줄여나가는 기법

SGD + momentum 같은 회적화 알고리즘에는 간간히 사용

Adam같은 알고리즘에서는 잘 사용되지 않음

대체로 처음 학습률을 정할 때는 사용하지 않고, 적절한 학습률은 찾은 후 학습 양상을 보며 decay를 적용할 구간을 선택하는 느낌으로 사용

## First-Order Optimization



1. Loss function의 선형적 근사치를 계산하기 위해 gradient 사용

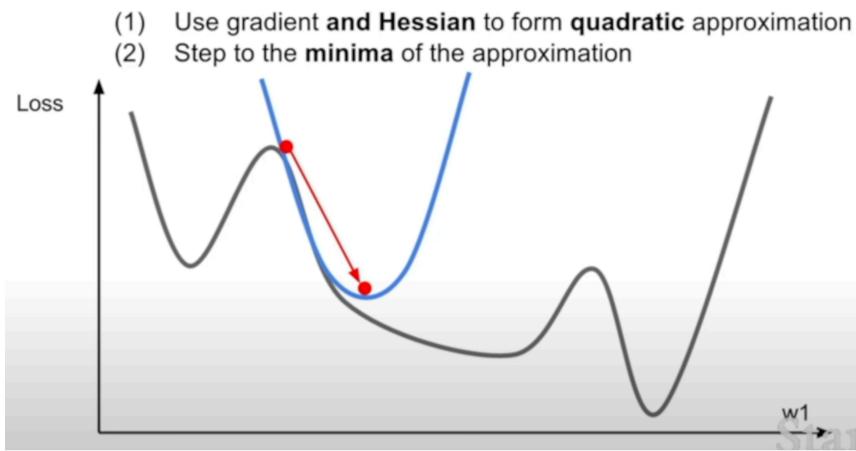
= first-order Taylor approximation

2. 그 근사치(loss)를 최소화하는 방향으로 step

→ 근사치를 넓은 구역을 다루지 않기에 큰 step을 할 수 없음

⇒ 위 과정은 함수의 first derivative만 포함

## Second-Order Optimization



first, second derivative를 함께 사용 = second-order Taylor approximation

quadratic으로 지역적 극소치를 계산 → minima로 곧바로 step 할 수 있음

second-order Taylor expansion :

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^\top H(\theta - \theta_0)$$

**Newton parameter update :**

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0)$$

Learning rate 없이 minima를 찾을 수 있지만, 저장할 메모리 양이 많아서 실용적이지 않음

→ Hessian은  $O(N^2)$  개의 element를 가지기에 비용이 비싸고, 당연히 invert 할 수 없음

Quasi-Newton methods (ex. BFGS)

- Hessian( $O(n^3)$ ) 역함수를 구하는 것 대신 Hessian 역함수의 극소치 구함  
→ 계산 비용 감소, (High condition num 조건일 때) 빠르게 수렴, 초기 추측에 덜 민감

L-BFGS (Limited memory BFGS)

- Hessian 전체를 메모리에 저장하지 않음
- 하지만, minibatch, non-convex 조건일 때 성능이 좋지 않음 → 딥러닝에서 사용 적음

## Improving performance on Prediction

### Model Ensembles

1. 다양한 모델을 각자 훈련시킨 후,
2. 예측 시 그 모델들의 결과를 평균

+ ) 많은 모델을 학습시키는 것 대신, 모델 학습 중 snapshot을 모아 ensemble할 수 있음