M2177.003100
Deep Learning

[9: Optimization]

Electrical and Computer Engineering
Seoul National University

(last compiled at 14:18:00 on 2018/10/28)

Outline

References

- *Deep Learning* by Goodfellow, Bengio and Courville ▸ Link
  - Chapter 8: Optimization for Training Deep Models
- online resources:
  - *Deep Learning Specialization (coursera)* ▸ Link
  - *Stanford CS231n: CNN for Visual Recognition* ▸ Link
  - *Machine Learning Yearning* ▸ Link

Outline

Optimization in deep learning

- most difficult optimization task in DL: training
  - so important and so expensive $\Rightarrow$ need specialized techniques
- mainstream: stochastic gradient descent (sgd) and its variants
- more complicated methods: not popular
  - second-order methods
    - often too expensive to compute/store Hessian
    - memory-efficient techniques emerging
  - convex optimization
    - its importance greatly diminished
- for clarity: this lecture focuses on unregularized supervised case

Derivatives and optimization order

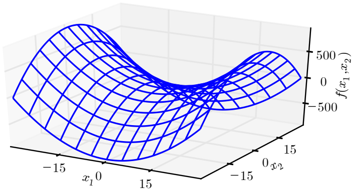
- derivatives
  - first derivative (= gradient) $\Rightarrow$ slope (Jacobian)
  - second derivative $\Rightarrow$ curvature(곡률) nC2 => n^2이므로 문제가 생김 (Hessian)
- optimization
  - first-order algorithms
    - use only gradient (*e.g.* gradient descent)
  - second-order algorithms
    - also use Hessian matrix (*e.g.* Newton's method)

Critical points (= stationary points) => 기울기=0인 지점

- points with zero slope: $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \mathbf{0}$
  - derivative gives no info about which direction to move
  - three types: maxima (− curvature), minima (+ curvature), saddle points

- a saddle point: contains both positive and negative curvature

$$f(\boldsymbol{x}) = x_1^2 - x_2^2$$

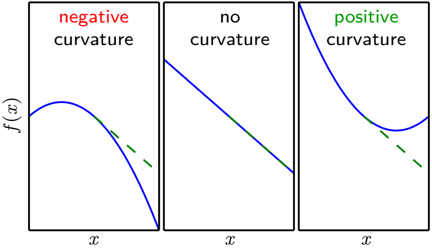


- along x_1 axis: f curves upward
  - direction of $\text{eigenvec}(\boldsymbol{H})$ with $\lambda > 0$
  - local minimum

- along x_2 axis: f curves downward
  - direction of $\text{eigenvec}(\boldsymbol{H})$ with $\lambda < 0$
  - local maximum

Use of second derivative

1. to characterize critical points
2. to measure curvature
3. to predict performance of an update in gradient -based optimization



positive curvature의 경우 때문에 gradient based에서 문제가 생기는 경우가 종종 있음

- negative curvature
  - f decreases faster than gradient predicts
- no curvature
  - gradient predicts the decrease correctly
- positive curvature
  - f decreases slower than gradient predicts (eventually increases)

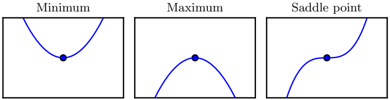
In deep learning

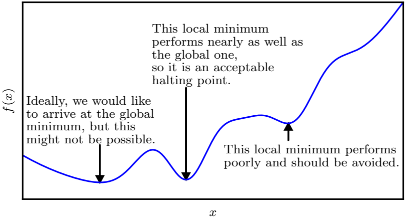
- our objective function has
  - ▶ many local minima + many saddle points surrounded by very flat regions
  - ⇒ makes optimization very difficult (especially in high-dim space)

- we therefore usually settle for finding a very low value of f
  - ▶ not necessarily minimal in any formal sense

local minima => 2번 미분했을 때 모든 dimension이 전부다 양수여야함
but 그러기는 몹시 어렵고 일부는 양수, 일부는 음수인 saddle point에 도달할 확률이 매우 높다 => 그래서 생각보다 sgd가 잘된다(local minima에 잘 안빠짐)

- recent research (Dauphin, 2014) reports
  - ▶ in high dim: saddle points are much more common than local minima





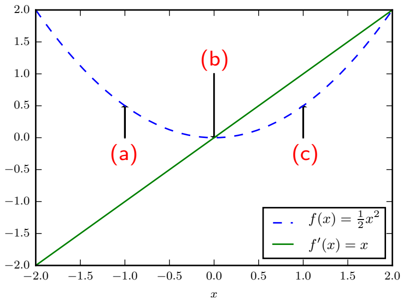
Outline

Method of gradient descent

- derivative: useful for minimizing a function
  - for small ϵ: $f\left(x - \epsilon \cdot \operatorname{sign}\left[f'(x)\right]\right) < f(x)$

- we can thus reduce $f(x)$ by
  - moving x in small steps with opposite sign of derivative

  $=$ method of gradient descent



$$\underbrace{\boldsymbol{x}'}_{\text{new}} = \underbrace{\boldsymbol{x}}_{\text{old}} - \underbrace{\epsilon}_{\text{learning rate}} \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

(a) $x < 0$: $f'(x) < 0$

$\Rightarrow$ can decrease f by moving rightward

(b) $x = 0$: $f'(x) = 0$

$\Rightarrow$ gd halts here (global min)

(c) $x > 0$: $f'(x) > 0$

$\Rightarrow$ can decrease f by moving leftward

Sgd and its variants

- probably the most used optimization algorithms for ML/DL
  - can obtain an $\underbrace{\text{unbiased estimate of gradient}}_{\uparrow}$

    by taking average gradient on a minibatch of m examples

Algorithm 1 gradient descent

---

initialize $\boldsymbol{\theta}$
while stopping criterion not met **do**
  sample m examples: $\mathbb{X}_m = \{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots (\boldsymbol{x}^{(m)}, y^{(m)})\}$
  compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)})$ ▷ m forward props
  apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$ ▷ ϵ: learning rate
end while

- three variants (N: total number of examples)
  - $m = 1$: stochastic gradient descent (sgd)
  - $1 < m < N$: minibatch sgd (typical m: 64, 128, 256, 512)
  - $m = N$: batch gradient descent

Properties of sgd: good ones

어차피 1개나 minibatch하므로 trainset이 커도
sgd시간에는 차이가 없음

- property #1:

computation time per update does not grow with # of training examples

  - most important property of sgd/minibatch/online gradient-based optimization

$\Rightarrow$ allows convergence even when # of training examples becomes large

- property #2 (see textbook):

sgd works better in practice than its theoretical analysis says

  - some benefits of sgd: obscured in asymptotic analysis

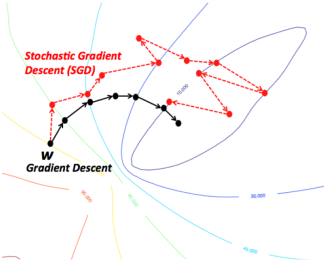
Properties of sgd: bad ones

- sgd may suffer in the following situations:
  - local minima/saddle points

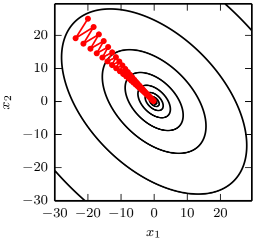
local minima,
saddle point에서
빠져나올 수 없음

zero gradient
$\Rightarrow$ gradient descent gets stuck

  - gradient noise



  - poor conditioning of $\boldsymbol{H}$



(source: wikidocs, cs231n)

Ravine

- Chloe Kim (2018 Olympic Champion, Women's Snowboard Halfpipe) ▸ Clip

Poor conditioning of $\boldsymbol{H}$

- consider a point $\boldsymbol{x}$ in multiple dimensions:
  - different second derivative for each direction

- condition number of Hessian $\boldsymbol{H}$ at $\boldsymbol{x}$ => 가장 급한방향과 가장 안급한 방향의 차이비율
  - measures how much the second derivatives differ from each other
  - recall: condition number of a matrix with eigenvalues $\{\lambda\}$:

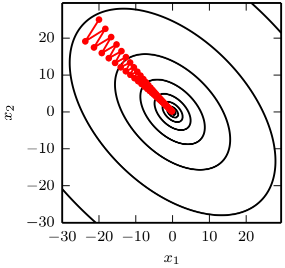
$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|$$

- when $\boldsymbol{H}$ has a large condition number ("poorly conditioned")
  1. gradient descent performs poorly

  $\because$ in a direction, derivative increases rapidly; in another, it increases slowly
  - gradient descent[1] is unaware of this change in the derivative
  2. it is difficult to choose a good step size ϵ

[1] it does not know it needs to explore preferentially in the direction where derivative remains negative for longer

example: == ravine, long canyon



- assume: Hessian $\boldsymbol{H}$ has condition number 5
  - ▶ most curvature: 5 times more curvature than least (a long canyon)
  - most curvature: direction $[1, 1]$ $\nearrow$
  - least curvature: direction $[1, -1]$ $\searrow$

- gradient descent (red lines): slow (zig-zag)
- by contrast: methods considering $\boldsymbol{H}$
  - ▶ can predict: the steepest direction is not promising

    (large $\lambda > 0 \Rightarrow$ large positive curvature $\Rightarrow$ bad; see page 8)
- how to handle poor conditioning without directly considering $\boldsymbol{H}$?

Outline

Exponentially weighted moving average (EWMA)

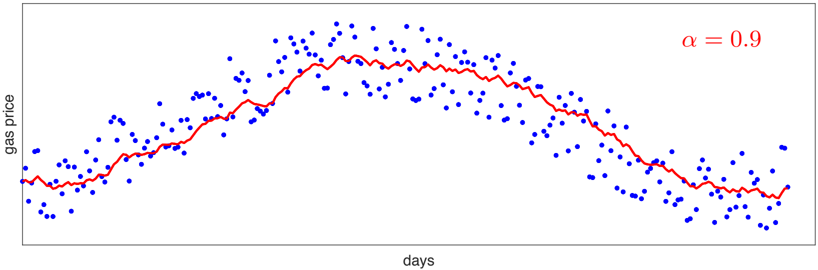
- given: time series $g_1, g_2, \ldots$
- EWMA defined as:

$$v_t = \begin{cases} g_1 & t = 1 \\ \alpha v_{t-1} + (1-\alpha) g_t & t > 1 \end{cases}$$

  - v_t: EWMA at time t
  - g_t: observation at t
  - $\alpha \in [0,1]$: degree of weighting decrease (constant smoothing factor)

- example: gas price over time
  - blue dot: gas price g
  - red curve: EWMA v

$$v_t = \alpha \cdot \underbrace{v_{t-1}}_{\text{previous EWMA}} + (1-\alpha) \cdot \underbrace{g_t}_{\text{current observation}}$$



Properties of EWMA

- effective weighting decreases exponentially over time:

$$
\begin{aligned}
v_t &= \alpha v_{t-1} + (1-\alpha) g_t \\
&= \alpha \left[\alpha v_{t-2} + (1-\alpha) g_{t-1} \right] + (1-\alpha) g_t \\
&\;\vdots \\
&= \alpha^k v_{t-k} + (1-\alpha) \underbrace{\left[g_t + \alpha g_{t-1} + \alpha^2 g_{t-2} + \cdots + \alpha^{k-1} g_{t-k+1} \right]}_{\text{weight exponentially decreases toward the past}}
\end{aligned}
$$

$\Rightarrow$ thus called "exponentially weighted"

- approximation[2]

$$
\begin{aligned}
v_t &= (1-\alpha)g_t + \alpha v_{t-1} \\
&= (1-\alpha)\left[g_t + \alpha g_{t-1} + \alpha^2 g_{t-2} + \alpha^3 g_{t-3} + \cdots\right] \\
&= \frac{g_t + \alpha g_{t-1} + \alpha^2 g_{t-2} + \cdots}{1 + \alpha + \alpha^2 + \cdots} \quad \Rightarrow \quad \text{weighted average formula}
\end{aligned}
$$

  - in such a formula, denominator $=$ effective number of observations:

$$
1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}
$$

  - bottom line:

$$
\boxed{v_t \approx \text{average over } \underline{\text{1 / (1-a)}} \text{ last time points}}
$$

e.g. $\alpha = 0.9 \Rightarrow$ average over $1/(1-0.9) = 10$ points
$\alpha = 0.98, 0.5 \Rightarrow$ average over $50, 2$ points, respectively

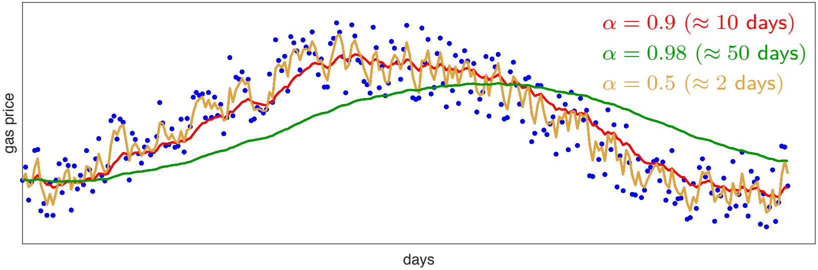
---

[2] recall: $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \cdots$

Effect of smoothing factor α

- higher α ($=$ more weight to past, less weight to present)
  - ▸ smoother curve $\leftarrow$ averaging over more days
  - ▸ shifted further $\leftarrow$ averaging over larger window

$\Rightarrow$ curve adapts more slowly to changes with more latency

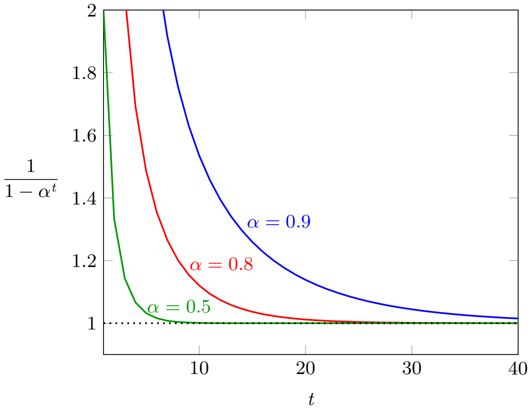


a = 0.98 => 원래 data보다 delay가 있음

Bias correction

- first few iterations: inaccurate average (have not seen enough samples)
  - instead of v_t, we thus use:

$$\frac{v_t}{1-\alpha^t}$$



Outline

Method of momentum

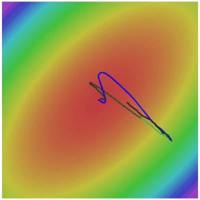
- sgd: very popular but sometimes slow
- method of momentum (Polyak, 1964):
  - designed to accelerate learning, especially in the face of
    - high curvature
    - small but consistent gradients
    - noisy gradients
  - can be combined to existing sgd variants
- common algorithm
  - accumulates exponentially decaying moving average of past gradients
  - then continues to move in their direction => gradient + velocity방향

Gradient descent with momentum

- idea: compute EWMA of gradients and use it to update weights
  - works almost always faster than standard gradient descent

- in physics
  - momentum = mass · velocity
  - for unit mass: momentum = velocity
  - smoothing factor α: friction

- sometimes
  - smoothing factor α
  - $\Rightarrow$ called momentum (misnomer)



— sgd
— sgd + momentum
— Nesterov

(source: cs231n)

Three (equivalent) forms of sgd + momentum

- let $\boldsymbol{g} \triangleq \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ ($\boldsymbol{\theta}$ represents $\boldsymbol{W}$ and $\boldsymbol{b}$ altogether)

form 1	$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$	$\Longrightarrow$	$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \left(\boldsymbol{g} + \frac{\alpha}{-\epsilon} \boldsymbol{v} \right)$
form 2	$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + (1-\alpha) \boldsymbol{g}$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{v}$	$\Longrightarrow$[3]	$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \tilde{\epsilon} \left(\boldsymbol{g} + \frac{\alpha}{1-\alpha} \boldsymbol{v} \right)$
form 3	$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \boldsymbol{g}$ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{v}$	$\Longrightarrow$	$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \left(\boldsymbol{g} + \alpha \boldsymbol{v} \right)$

- bottom line: $\boldsymbol{\theta}$ is updated by linear combination of gradient and velocity

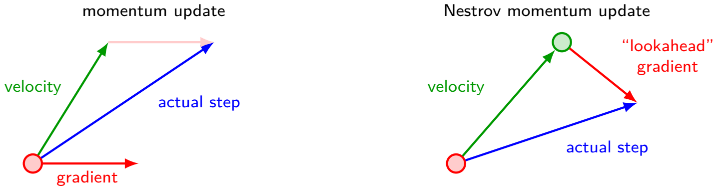
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \left(\underbrace{\boldsymbol{g}}_{\text{gradient}} + \text{constant} \cdot \underbrace{\boldsymbol{v}}_{\text{velocity}} \right)$$

[3] $\tilde{\epsilon} \triangleq \epsilon(1-\alpha)$

Nesterov momentum

- difference from standard momentum:
  - where gradient $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J$ is evaluated

일단 velocity만큼 이동한 후 그
다음에 gradient를 구한다

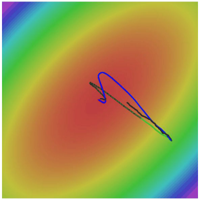


standard momentum	Nesterov momentum
$\boldsymbol{g}$ evaluated at current position $\boldsymbol{\theta}$ (red circle)	$\boldsymbol{g}$ evaluated at "lookahead" position $\boldsymbol{\theta} + \alpha \boldsymbol{v}$ (green circle)

- rationale: momentum is about to carry us to a new position
  - make sense to evaluate $\boldsymbol{g}$ at new position instead of "old/stale" position

- Nesterov momentum update rule:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \alpha \boldsymbol{v})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$



— sgd
— sgd + momentum
— Nesterov

(source: cs231n)

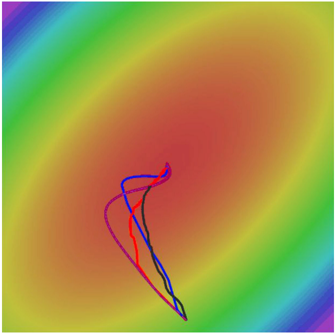
- Nesterov momentum
  - ▶ gradient is evaluated after the current velocity is applied
  - ⇒ interpreted as adding a correction factor to standard momentum

- advantages
  - ▶ stronger theoretical converge guarantees for convex functions
  - ▶ consistently works slightly better than standard momentum

Outline

Per-parameter adaptive learning rates

- optimization methods explained so far
  - set learning rate ϵ globally and equally for all parameters
- methods presented now: AdaGrad, RMSProp, Adam
  - adaptively tune ϵ for each parameter



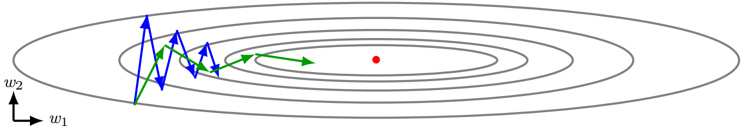
– sgd
– sgd + momentum
– RMSProp
– Adam

(source: cs231n)

Motivation

=> learning rate를 빠르게 해야하는 direction도 있고
느리게 해야하는 direction도 있음!

- recall: limitation of gradient descent



  - goal: move horizontally
  - problem: huge vertical oscillations

- solution: we want to
  - slow down learning vertically
  - speed up (or at least not slow down) learning horizontally

- how to implement this idea (without relying on H explicitly)?

AdaGrad

=> 각 차원별로 다른 learning rate 적용

gradient가 크면 e를 줄이고
gradient가 작으면 e를 키우자

- individually adapts learning rate of each direction (*i.e.* each parameter)
  - steep direction (large $\frac{\partial J}{\partial \theta_j}$): slow down learning
  - gently sloped direction (small $\frac{\partial J}{\partial \theta_j}$): speed up learning
- adjusts learning rates per parameter:

$$\epsilon_j = \frac{\epsilon}{\sqrt{\sum_{\text{all previous iterations}} (g_j \cdot g_j)}}$$

  - ϵ: global learning rate
  - ϵ_j: learning rate of dimension j (parameter θ_j)
  - $g_j = \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j}$: gradient wrt dimension j
- net effect:
  - greater progress in more gently sloped directions

- downside (esp in deep learning)
  - monotonically decreasing ϵ: too aggressive

  $\Rightarrow$ stops learning too early

- TF: `AdagradOptimizer`
  - but do not use it for neural nets

- Adadelta: an extension of Adagrad => fixed window를 보고 판단
  - restricts the window of accumulated past gradients to some fixed size

  $\Rightarrow$ reduces aggressive, monotonically decreasing learning rate

RMSProp (root-mean-square prop)

=> 과거를 모두 더하는 대신에 exponentially weighted한 합을 이용 (최근 과거에 더 높은 weight을 줌)

- modifies AdaGrad to perform better in non-convex setting
  - changes gradient accumulation to EWMA

- use of exponentially decaying average allows RMSprop to
  - discard history from extreme past
  - ⇒ converge rapidly after finding a convex bowl

- comparison ($\boldsymbol{r}$: accumulation variable)

e: element wise learning rate

1 / (1-r) = 과거의 보는 갯수

AdaGrad	RMSprop
$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$	$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
$\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + r}} \odot \boldsymbol{g}$	$\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + r}} \odot \boldsymbol{g}$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$	$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

  - decay rate ρ: hyperparameter (typically 0.9, 0.99, 0.999)

Adam (adaptive moment estimation)

=> first order momentum(momentum) + second order momentum(RMSProp)의 결합
+ bias correction

- idea: RMSProp + momentum
  - with bias correction
- for each iteration:
  1. compute gradient $\boldsymbol{g}$
  2. update first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1)\boldsymbol{g}$ $\leftarrow$ "momentum"
  3. update second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2)\boldsymbol{g} \odot \boldsymbol{g}$ $\leftarrow$ "RMSProp"
  4. bias correction:

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}, \qquad \hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$$

  5. update parameter:

momentum => gradient 보정
RMSProp => adaptive learning rate 이용

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}} + \delta}}$$

Algorithm 2 Adam optimizer

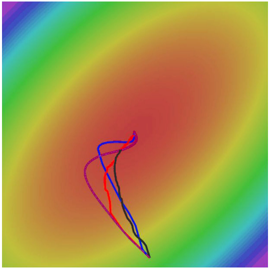
Require:

- step size ϵ
- exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$
- small constant δ used for numerical stabilization
- initial parameters $\boldsymbol{\theta}$

initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
initialize time step $t = 0$
while stopping criterion not met **do**
  sample a minibatch $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  $t \leftarrow t + 1$
  update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
  update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
  correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{s}{1 - \rho_1^t}$
  correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{r}{1 - \rho_2^t}$
  compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ (operations applied element-wise)
  apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

- recommended values in the paper
  - learning rate ϵ: needs tuning (suggested default: 0.001)
  - for momentum ρ_1: 0.9
  - for RMSProp ρ_2: 0.999
  - for stability δ: 10^{-8}

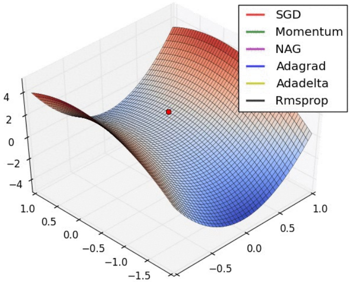
- Adam: often works better than RMSProp
  - recommended as the default algorithm to use
  - alternative to Adam worth trying: sgd + Nestrov momentum => manual tunning

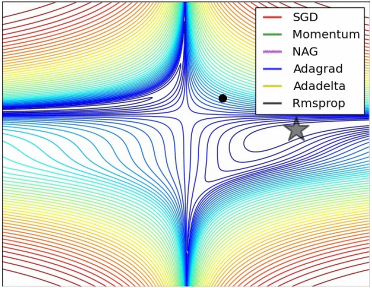


– sgd
– sgd + momentum
– RMSProp
– Adam

(source: cs231n)

Comparison





(source: Radford)

- more information: ▸ Link

Outline

Learning rate

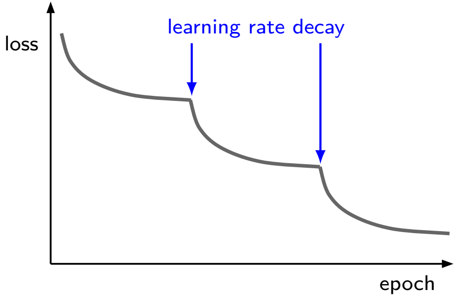
SF to 분당 =>
비행기 + 버스 + 보도 =>
decaying learning rate

- hyperparameter for many gradient-based algorithms
  - sgd, sgd + momentum, AdaGrad, RMSProp, Adam

- need to gradually decrease learning rate over time

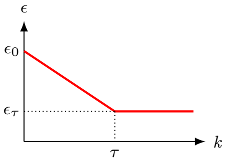
  $\Rightarrow$ now denote ϵ_k : learning rate at iteration k (ϵ_0: initial)

  - more critical with sgd + momentum (less common with Adam)



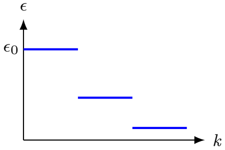
(source: cs231n)

How to decay learning rate

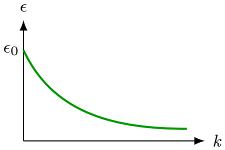


- linear decay (until τ, and then constant)

$$\epsilon_k = \left(1 - \frac{k}{\tau}\right)\epsilon_0 + \frac{k}{\tau}\epsilon_\tau$$



- step decay   => Alex NET에서 사용한 방법
  - discrete staircase



- exponential decay: *e.g.* $\epsilon = \epsilon_0 (0.95)^k$
- $1/k$ or $1/\sqrt{k}$ decay:

- also popular: __manual__ decay (by trial-and-error or monitoring learning curve)

How to set initial learning rate

ϵ_0	**if too large:**	**if too low:**
	• violently oscillating learning curve • cost function often increases significantly	• learning proceeds slowly • learning may stuck with a high cost value

- typically:
  - optimal $\epsilon_0 > \underbrace{\epsilon^*_{\sim 100}}_{\uparrow}$

    learning rate that yields best performance after first 100 iterations or so

- advice: monitor the first several iterations and
  - use a learning rate that is
    - higher than best-performing ϵ at this time
    - but not so high that it causes severe instability

Outline

Idea behind Newton's method

- consider a second-order Talyor series approximation
  - to function $f(\boldsymbol{x})$ around the current point $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)})$$

    - $\boldsymbol{g} \triangleq \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)})$: gradient of f at $\boldsymbol{x}^{(0)}$
    - $\boldsymbol{H} \triangleq \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})$: Hessian of f at $\boldsymbol{x}^{(0)}$

- solving for the critical point of f gives Newton's update rule:

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \boldsymbol{H}^{-1}\boldsymbol{g}$$

  - pros: (in theory) no hyperparemter (*i.e.* learning rate)
  - cons: efficiency ~~=> 특히 DL에서는~~ n이 몹시큼; ($\boldsymbol{H}$ has $O(n^2)$ elements and takes $O(n^3)$ for inverting)

※ Levenberg-Marquardt algorithm

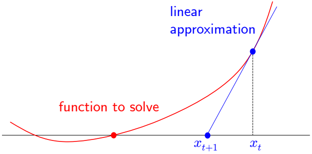
=> n이 작으면 매우 잘됨
(resource 제약이 없는 경우)

  - switches between Newton's and gradient descent

Comparison (1D)

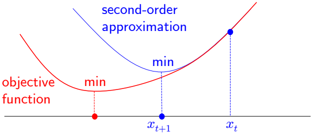
- Newton's method: second-order
  - zero-finding

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$$



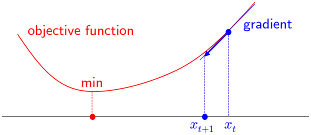
  - minimization/maximization

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$$

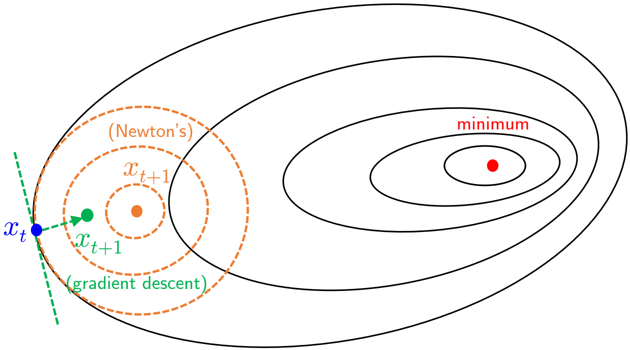


- gradient descent: first-order
  - minimization/maximization

$$x_{t+1} = x_t - \epsilon f'(x_t)$$



Comparison (2D)



- Newton's method for optimization
  - idea: get a <u>second</u>-order approximation and minimize it

$\Rightarrow$ faster than gradient descent

Quasi-Newton methods

=> Newton method의 approximation
(H와 H^-1을 구하는 것과 저장 하는 것이 불가능)

(learning rate대신 H^-1을 사용하므로)

- idea: avoid directly inverting $\boldsymbol{H}$
  - approximate $\boldsymbol{H}^{-1}$ with matrix $\boldsymbol{M}_t$
    - $\boldsymbol{M}_t$: iteratively refined by low-rank updates
  - determine direction of descent by $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$ and update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \boldsymbol{\rho}_t$$

- Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm
  - most popular quasi-Newton method
  - still requires $O(n^2)$ memory to store $\boldsymbol{H}^{-1}$
- L-BFGS (limited memory BFGS): does not form/store full $\boldsymbol{H}^{-1}$
  - usually works very well in full batch/deterministic mode
  - but performs poorly in minibatch/stochastic setting (research topic)

Practical advice on choosing optimizer in DL

- Adam
  - a good default choice in many cases
- sgd + momentum + learning rate decay
  - often outperforms Adam
  - but requires more tuning
- L-BFGS
  - try it if you can afford to do full batch updates
  - but should disable all sources of noise

(source: cs231n)

Outline

Summary

- optimization in deep learning
  - mostly sgd and its variants
- gradient estimate

$$\hat{\boldsymbol{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$$

- stochastic gradient descent (sgd)

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$$

- method of momentum ($\alpha \in [0, 1)$)
  => momentum과 gradient의 combination

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \hat{\boldsymbol{g}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

- Nesterov momentum (corrected momentum)

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right]$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

- AdaGrad ($\boldsymbol{r}$: for gradient accumulation)

$$\boldsymbol{r} \leftarrow \boldsymbol{r} + \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$$
$$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \hat{\boldsymbol{g}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$

- RMSProp (gradient accumulation by EWMA)

$$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$$
$$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \hat{\boldsymbol{g}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$

- Adam (a reasonable default choice)

$$\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \hat{\boldsymbol{g}} \qquad \text{(momentum)}$$
$$\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}} \qquad \text{(RMSProp)}$$
$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}, \quad \hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t} \qquad \text{(bias correction)}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}} + \delta}}$$