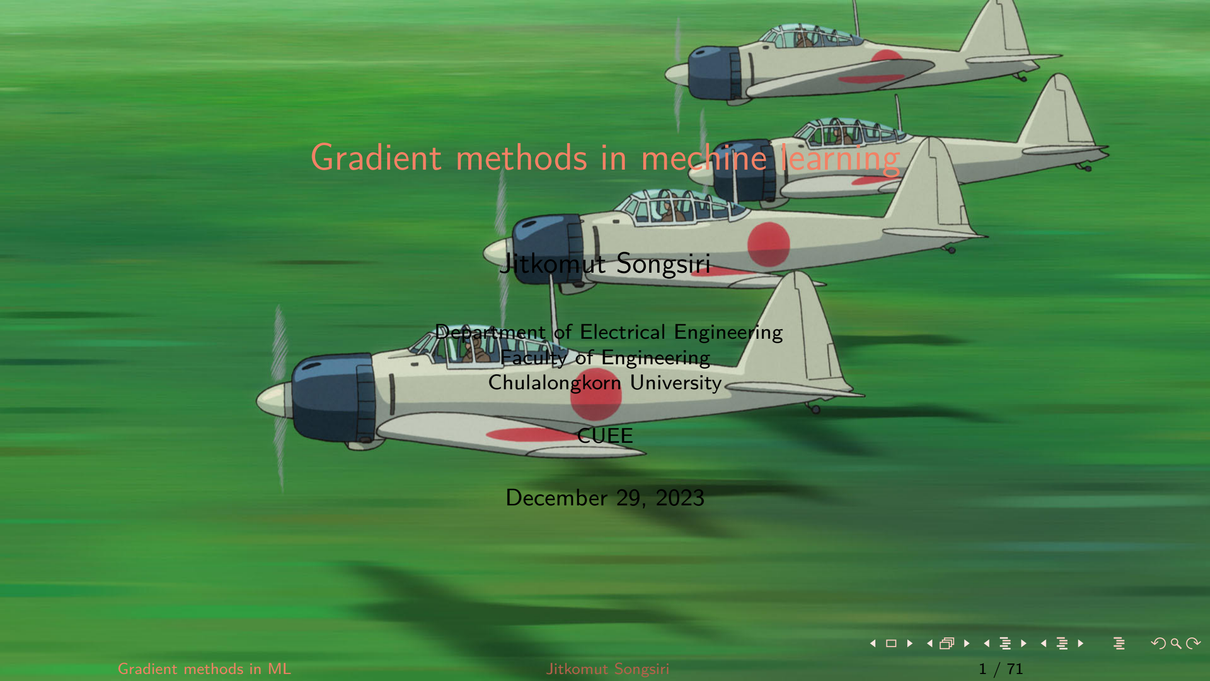


Gradient methods in mechine learning

Jitkomut Songsiri

Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University

CUEE

December 29, 2023

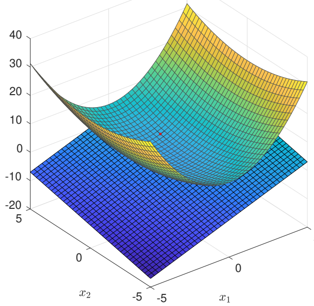
Outline

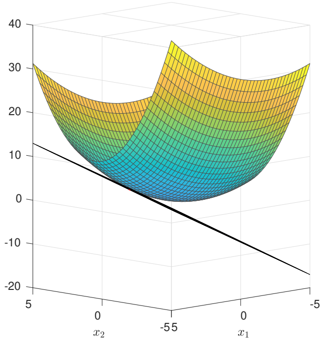
Derivative basics

Tangent plane

a tangent plane of $f(x)$ at x_0 is obtained by the first-order Taylor approximation

$$f(x) \approx f(x_0) + \nabla f(x_0)^T (x - x_0)$$



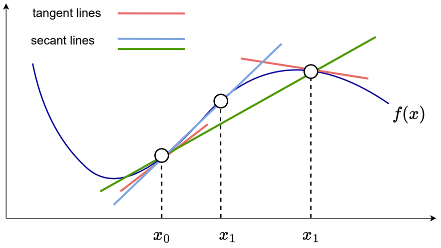


$$f(x) = x_1^2 + (1/4)x_2^2$$
$$x_0 = (1,2),\ \nabla f(x_0) = (2,1)$$
plane:$2 + 2(x_1 - 1) + (x_2 - 2) = 0$

the gradient of f is the normal vector of the tangent plane

Tangent and secant lines

for a scalar-input function



- first-order Taylor approximation at x_0

$$f(x) \approx \tilde{f}(x) = f(x_0) + f'(x_0)(x - x_0)$$

- **secant** line that passes through the two points $(x_1, f(x_1))$ and $(x_0, f(x_0))$

$$\tilde{f}(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot (x - x_0)$$

- let $x_1 = x_0 + \epsilon$, as $\epsilon \to 0$ the secant line gets closer to the tangent line
- the slope of secant line approaches $f'(x_0)$ (if exists)

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f(x_0 + \epsilon) - f(x_0)}{\epsilon} \to f'(x_0) \quad \text{as } \epsilon \to 0 \quad \text{if the limit exists}$$

Numerical differentiation

some common derivative rules

function	**derivative**	**function**	**derivative**
$\sigma(x) = \frac{1}{1+e^{-x}}$	$\frac{e^{-x}}{(1+e^{-x})^2}$	$f(x)g(x)$	$f'(x)g(x) + f(x)g'(x)$
$\tanh(x)$	$\operatorname{sech}^2(x)$	$f(g(x))$	$f'(g(x))g'(x)$
$\max(0, x)$	$\begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$		

when f is too complicated to calculate analytically, we can approximate f' by setting ϵ to some small number

- a more robust way is to use the average slope of the right and the left secant line

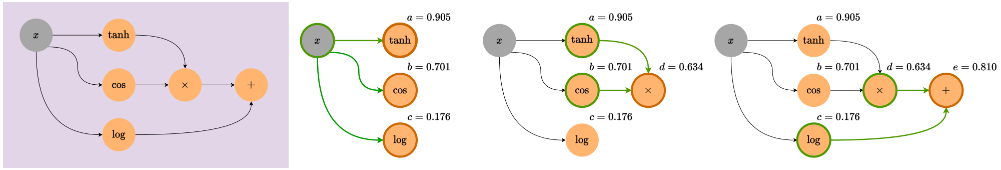
$$f'(x_0) \approx \frac{f(x_0 + \epsilon) - f(x_0 - \epsilon)}{2\epsilon}$$

- however, setting ϵ too small can create *round-off errors* due to numerical computations

Computation graph

Example: scalar-input function

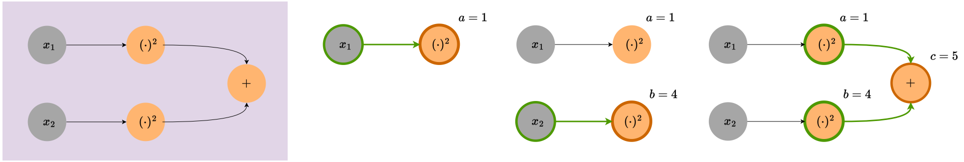
consider $f(x) = \tanh(x)\cos(x) + \log(x)$



- yellow nodes represents an elementary function or operation
- the input node x is parent to nodes a, b, c, while a, b are parents to node d
- the computation flows *forward* through the graph in sets of parent-child nodes
- to compute $d(1.5) = a(1.5) \times b(1.5)$, we only need access to its evaluated *parents*, which already computed

Vector-input function

consider $f(x) = x_1^2 + x_2^2$



- two inputs x_1 and x_2 are each represented by a distinct node
- first, substitute $x_1 = 1$ and evaluate its child $a(1) = 1^2$
- next, substitute $x_2 = 2$ and evaluate its child $b(2) = 2^2$
- finally, the last child node is $c = a(1) + b(2) = 5$

Exercises

draw the computation graph of the following functions

1. $f(x) = \cos^2(2x+3) + \log(|x|)$
2. $f(x) = \max(0, (2,-3,1)^T x + 4)$
3. $f(x) = f(x_1, x_2, x_3) = \tanh(w_1^T x + b_1) + \sigma(w_2^T x + b_2)$
4. $f(x) = f(x_1, x_2, x_3) = \dfrac{c_1 e^{\tanh(w_1^T x) + b_1}}{\sum_{k=1}^3 c_k e^{\tanh(w_k^T x) + b_k}}$

Automatic differentiation

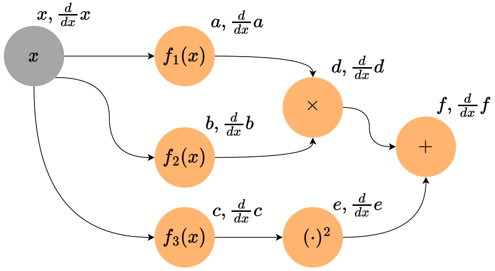
Forward mode of automatic differentiation

the computation graph of a function can be used to form its **gradient** by

- sweeping forward through the function's computation graph from left to right
- evaluating the gradient of each node w.r.t. the function's **original input**

this is called **forward mode of automatic differentiation**

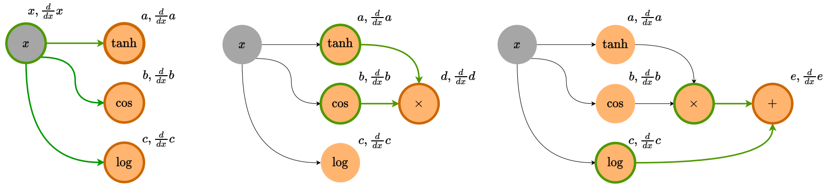
- performing each child node requires only the values computed at its **parents** along with derivative rules for elementary functions and operations
- faster and more reliable than performing $f'(x)$ manually and then hard-coded



$$f(x) = f_1(x)f_2(x) + f_3^2(x)$$

Example: scalar-input function

example: $f(x) = \tanh(x)\cos(x) + \log(x)$



1. set the input node x, and $\frac{d}{dx}x = 1$, and move to the childen node
2. for each child node, form both the node and its derivative w.r.t. to the input
3. form the next child node according to $\times$ operation and its derivative

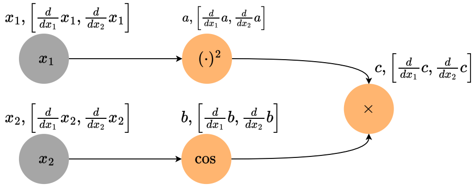
$$d = a \times b, \quad \frac{d}{dx}d(a,b) = \frac{\partial}{\partial a}d(a,b) \times \frac{d}{dx}a(x) + \frac{\partial}{\partial b}d(a,b) \times \frac{d}{dx}b(x) = ba'(x) + ab'(x)$$

4. form the next child node according to $+$ operation and its derivative

$$e = d + c, \quad \frac{d}{dx}e(d,c) = \frac{\partial}{\partial d}e(d,c) \times \frac{d}{dx}d(x) + \frac{\partial}{\partial c}e(d,c) \times \frac{d}{dx}c(x) = d'(x) + c'(x)$$

Example: vector-input function

$$f(x) = x_1^2 \cos(x_2)$$



$x_1, \left[\frac{d}{dx_1}x_1, \frac{d}{dx_2}x_1\right]$
x_1
$a, \left[\frac{d}{dx_1}a, \frac{d}{dx_2}a\right]$
$(\cdot)^2$
$c, \left[\frac{d}{dx_1}c, \frac{d}{dx_2}c\right]$
$\times$
$x_2, \left[\frac{d}{dx_1}x_2, \frac{d}{dx_2}x_2\right]$
x_2
$b, \left[\frac{d}{dx_1}b, \frac{d}{dx_2}b\right]$
cos

1. compute the gradient at each input node: $\nabla x_1 = (1, 0)$ and $\nabla x_2 = (0, 1)$
2. at the next child nodes, a and b, compute the **gradient** of a w.r.t. both x_1 and x_2

$$\frac{\partial}{\partial x_1} a = 2x_1, \quad \frac{\partial}{\partial x_2} a = 0, \quad \frac{\partial}{\partial x_1} b = 0, \quad \frac{\partial}{\partial x_2} b = -\sin(x_2)$$

3. compute the gradient at node c, using the chain rule

$$c = a + b, \quad \begin{bmatrix} \frac{\partial}{\partial x_1} c \\ \frac{\partial}{\partial x_2} c \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial a} c \frac{\partial}{\partial x_1} a + \frac{\partial}{\partial b} c \frac{\partial}{\partial x_1} b \\ \frac{\partial}{\partial a} c \frac{\partial}{\partial x_2} a + \frac{\partial}{\partial b} c \frac{\partial}{\partial x_2} b \end{bmatrix} = \begin{bmatrix} b \times 2x_1 + a \times 0 \\ b \times 0 + a \times (-\sin(x_2)) \end{bmatrix}$$

Exercises and notes

draw the computation graph to compute forward-mode diffferentiation

1. $f(x) = \tanh^2(2x_1 + 3x_2)$
2. $f(x) = f(x_1, x_2) = \tanh(x_1^2 + \cos(x_2))$
3. $f(x) = f(x_1, x_2, x_3) = \max(0, \tanh(2x_1 + 3x_2 + 4x_3))$

after doing the exercises, note that

- we compute the **complete gradient** w.r.t **all inputs** at every node, but most of the nodes may only take just a few inputs, *e.g.*, no 3. define child nodes: $a = 2x_1, b = 3x_2, c = 4x_3$
- the partial derivatives of a w.r.t. x_2, x_3 are zero, same conclusion for partial derivatives of b w.r.t. x_1, x_3
- this leads to considerable computation waste since the partial derivative of any node w.r.t. an original input x_i that it does not take in will always be zero

Reverse mode of automatic differentiation

to avoid computation waste due to many zero partial derivatives when computing the full gradient in the forward mode of autodiff,

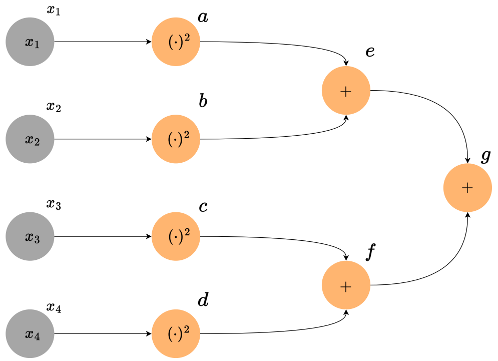
the reverse mode of autodiff or **backpropagation** algorithm was proposed

1. traverse along the computation graph in forward mode but compute *only* the partial derivatives needed at each node (ignore the ones that will always be zero) – until the forward sweep is complete
2. starting with the final nodes, and move **backward** through the graph, combining the previously computed partial derivatives to appropriately construct the gradient

- this requires constructing the computation graph *explicitly* and store it
- the reverse mode is more popular choice (as compared to forward mode), and implemented in `autograd` (python-based automatic differentiation)

Example: $f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2$

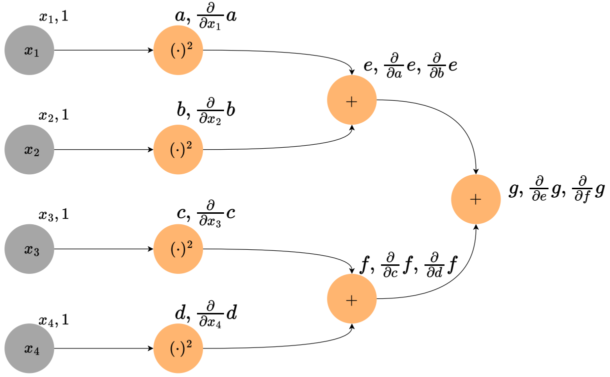
at each node, find the complete gradients w.r.t. $x_1, \ldots, x_4$



you can see that over half of partial derivatives are zeros

Example: forward sweeping

compute only the partial derivatives w.r.t. their parents that are non-zero



$x_1, 1$
x_1
$a, \frac{\partial}{\partial x_1} a$
$(\cdot)^2$
$x_2, 1$
x_2
$b, \frac{\partial}{\partial x_2} b$
$(\cdot)^2$
$x_3, 1$
x_3
$c, \frac{\partial}{\partial x_3} c$
$(\cdot)^2$
$x_4, 1$
x_4
$d, \frac{\partial}{\partial x_4} d$
$(\cdot)^2$
$e, \frac{\partial}{\partial a} e, \frac{\partial}{\partial b} e$
$+$
$f, \frac{\partial}{\partial c} f, \frac{\partial}{\partial d} f$
$+$
$g, \frac{\partial}{\partial e} g, \frac{\partial}{\partial f} g$
$+$

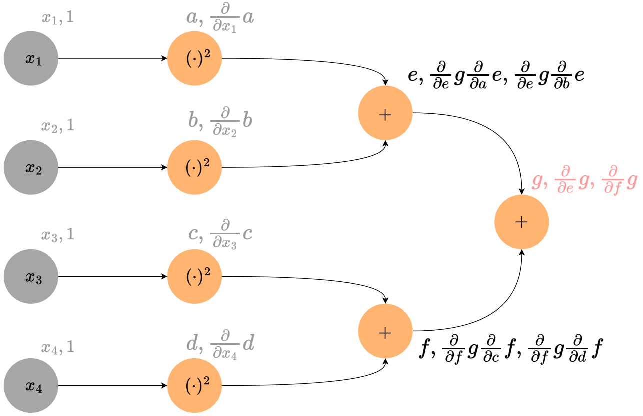
✎ write down

$$\frac{\partial}{\partial x_1} g =$$
$$\frac{\partial}{\partial x_2} g =$$
$$\frac{\partial}{\partial x_3} g =$$
$$\frac{\partial}{\partial x_4} g =$$

(more efficient than computing completing gradients)

Example: backpropagate 1

update the partial derivative of the parents (e, f) by multiplying the partial derivative of its children w.r.t. e and f



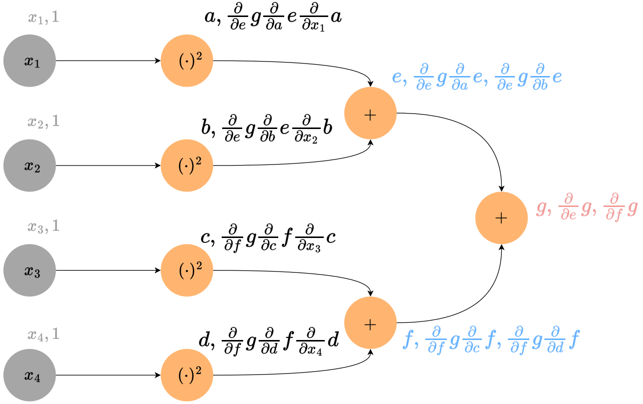
$x_1, 1$
$x_2, 1$
$x_3, 1$
$x_4, 1$
$a, \frac{\partial}{\partial x_1} a$
$b, \frac{\partial}{\partial x_2} b$
$c, \frac{\partial}{\partial x_3} c$
$d, \frac{\partial}{\partial x_4} d$
x_1
x_2
x_3
x_4
$(\cdot)^2$
$+$
$e, \frac{\partial}{\partial e} g \frac{\partial}{\partial a} e, \frac{\partial}{\partial e} g \frac{\partial}{\partial b} e$
$f, \frac{\partial}{\partial f} g \frac{\partial}{\partial c} f, \frac{\partial}{\partial f} g \frac{\partial}{\partial d} f$
$g, \frac{\partial}{\partial e} g, \frac{\partial}{\partial f} g$

go backward from

$$\frac{\partial}{\partial x_1} g = \frac{\partial}{\partial e} g \frac{\partial}{\partial a} e \frac{\partial}{\partial x_1} a \frac{\partial}{\partial x_1} x_1$$
$$\frac{\partial}{\partial x_2} g = \frac{\partial}{\partial e} g \frac{\partial}{\partial b} e \frac{\partial}{\partial x_2} b \frac{\partial}{\partial x_2} x_2$$
$$\frac{\partial}{\partial x_3} g = \frac{\partial}{\partial f} g \frac{\partial}{\partial c} f \frac{\partial}{\partial x_3} c \frac{\partial}{\partial x_3} x_3$$
$$\frac{\partial}{\partial x_4} g = \frac{\partial}{\partial f} g \frac{\partial}{\partial d} f \frac{\partial}{\partial x_4} d \frac{\partial}{\partial x_4} x_4$$

Example: backpropagate 2

update the partial derivative of the parent (a) by multiplying the partial derivative of its children (e) w.r.t. a



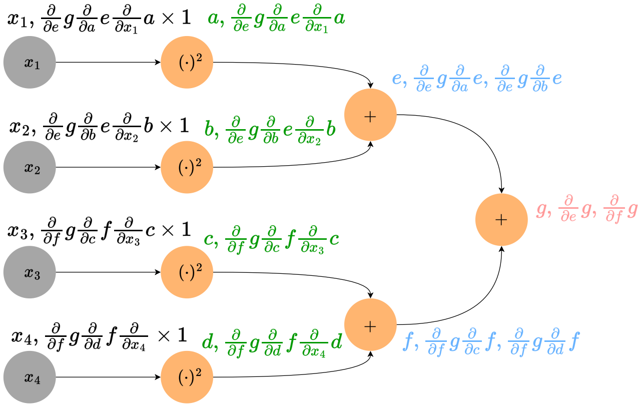
$x_1, 1$
$x_2, 1$
$x_3, 1$
$x_4, 1$
x_1
x_2
x_3
x_4
$(\cdot)^2$
$+$
$a, \frac{\partial}{\partial e} g \frac{\partial}{\partial a} e \frac{\partial}{\partial x_1} a$
$b, \frac{\partial}{\partial e} g \frac{\partial}{\partial b} e \frac{\partial}{\partial x_2} b$
$c, \frac{\partial}{\partial f} g \frac{\partial}{\partial c} f \frac{\partial}{\partial x_3} c$
$d, \frac{\partial}{\partial f} g \frac{\partial}{\partial d} f \frac{\partial}{\partial x_4} d$
$e, \frac{\partial}{\partial e} g \frac{\partial}{\partial a} e, \frac{\partial}{\partial e} g \frac{\partial}{\partial b} e$
$f, \frac{\partial}{\partial f} g \frac{\partial}{\partial c} f, \frac{\partial}{\partial f} g \frac{\partial}{\partial d} f$
$g, \frac{\partial}{\partial e} g, \frac{\partial}{\partial f} g$

go backward from

$$\frac{\partial}{\partial x_1} g = \frac{\partial}{\partial e} g \frac{\partial}{\partial a} e \frac{\partial}{\partial x_1} a \frac{\partial}{\partial x_1} x_1$$
$$\frac{\partial}{\partial x_2} g = \frac{\partial}{\partial e} g \frac{\partial}{\partial b} e \frac{\partial}{\partial x_2} b \frac{\partial}{\partial x_2} x_2$$
$$\frac{\partial}{\partial x_3} g = \frac{\partial}{\partial f} g \frac{\partial}{\partial c} f \frac{\partial}{\partial x_3} c \frac{\partial}{\partial x_3} x_3$$
$$\frac{\partial}{\partial x_4} g = \frac{\partial}{\partial f} g \frac{\partial}{\partial d} f \frac{\partial}{\partial x_4} d \frac{\partial}{\partial x_4} x_4$$

Example: backpropagate 3

update the partial derivative of the parent (x_1) by multiplying the partial derivative of its child (a) w.r.t. x_1



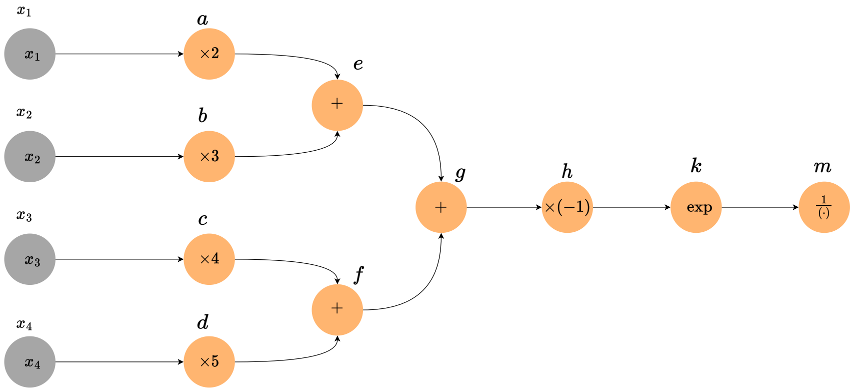
$x_1, \frac{\partial}{\partial e}g\frac{\partial}{\partial a}e\frac{\partial}{\partial x_1}a \times 1$
$a, \frac{\partial}{\partial e}g\frac{\partial}{\partial a}e\frac{\partial}{\partial x_1}a$
$x_2, \frac{\partial}{\partial e}g\frac{\partial}{\partial b}e\frac{\partial}{\partial x_2}b \times 1$
$b, \frac{\partial}{\partial e}g\frac{\partial}{\partial b}e\frac{\partial}{\partial x_2}b$
$x_3, \frac{\partial}{\partial f}g\frac{\partial}{\partial c}f\frac{\partial}{\partial x_3}c \times 1$
$c, \frac{\partial}{\partial f}g\frac{\partial}{\partial c}f\frac{\partial}{\partial x_3}c$
$x_4, \frac{\partial}{\partial f}g\frac{\partial}{\partial d}f\frac{\partial}{\partial x_4} \times 1$
$d, \frac{\partial}{\partial f}g\frac{\partial}{\partial d}f\frac{\partial}{\partial x_4}d$
$e, \frac{\partial}{\partial e}g\frac{\partial}{\partial a}e, \frac{\partial}{\partial e}g\frac{\partial}{\partial b}e$
$f, \frac{\partial}{\partial f}g\frac{\partial}{\partial c}f, \frac{\partial}{\partial f}g\frac{\partial}{\partial d}f$
$g, \frac{\partial}{\partial e}g, \frac{\partial}{\partial f}g$
x_1
x_2
x_3
x_4
$(\cdot)^2$
$(\cdot)^2$
$(\cdot)^2$
$(\cdot)^2$
$+$
$+$
$+$

go backward from

$$\frac{\partial}{\partial x_1}g = \frac{\partial}{\partial e}g\frac{\partial}{\partial a}e\frac{\partial}{\partial x_1}a\frac{\partial}{\partial x_1}x_1$$
$$\frac{\partial}{\partial x_2}g = \frac{\partial}{\partial e}g\frac{\partial}{\partial b}e\frac{\partial}{\partial x_2}b\frac{\partial}{\partial x_2}x_2$$
$$\frac{\partial}{\partial x_3}g = \frac{\partial}{\partial f}g\frac{\partial}{\partial c}f\frac{\partial}{\partial x_3}c\frac{\partial}{\partial x_3}x_3$$
$$\frac{\partial}{\partial x_4}g = \frac{\partial}{\partial f}g\frac{\partial}{\partial d}f\frac{\partial}{\partial x_4}d\frac{\partial}{\partial x_4}x_4$$

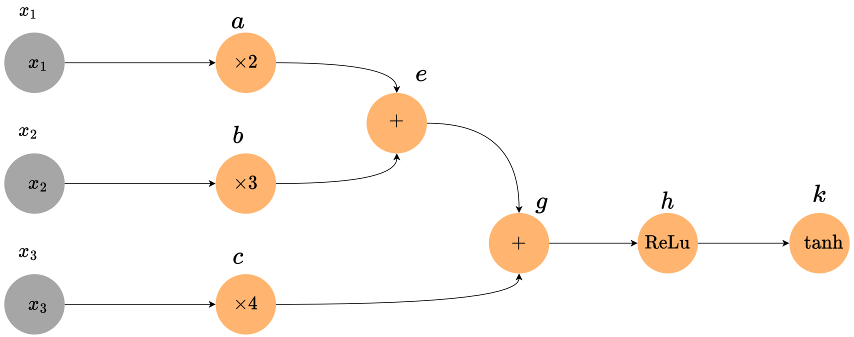
Exercise

$f(x) = \sigma(2x_1 + 3x_2 + 4x_3 + 5x_4)$ ✎ compute backprop gradient



Exercise

$f(x) = \tanh(\max(0, 2x_1 + 3x_2 + 4x_3))$ ✎ compute backprop gradient



References

1. Appendix B in J. Watt, R. Borhani, and A.K. Katsaggelos, *Machine Learning Refined: Foundations, Algorithms, and Applications*, 2nd edition, Cambridge University Press, 2020
2. `https://github.com/jermwatt/machine_learning_refined`, Google colab: `autograd` library and usage

Mini batch optimization

Mini-batch optimization

minimizing f that is a sum of N functions of the same form

$$\underset{x}{\text{minimize}} \quad f(x) := \sum_{i=1}^{N} f_i(x)$$

such as $\sum_{i=1}^N \|y_i - \hat{y}_i(x)\|_2^2$ where x is the model parameter (N is typically large)

- quadratic loss example: the conventional gradient step is $-\sum_{i=1}^N (y_i - \hat{y}_i(x)) \frac{d\hat{y}_i}{dx}$
- mini-batch optimization involves taking a search direction (often gradient step) in $f_1, f_2, \ldots, f_N$ *sequentially*
- the gradient step can be taken w.r.t. a subset of $f_1, f_2, \ldots, f_N$; the size of the subsets defines a **batch size** of mini-batch optimization
- a batch size of 1 is referred to as **stochastic optimization**

Illustration of mini-batch optimization

we define $x^{(0,0)}$ an initial point of optimization variable

- take a descent direction $s^{(0,1)}$ in f_1 alone and form $x^{(0,1)} = x^{(0,0)} + ts^{(0,1)}$
- determine a descent direction $s^{(0,2)}$ in f_2 and form $x^{(0,2)} = x^{(0,1)} + ts^{(0,2)}$

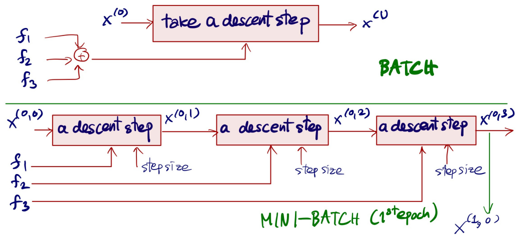
continue this pattern until we sweep through $f_1, \ldots, f_N$

$$\begin{aligned}
x^{(0,1)} &= x^{(0,0)} + ts^{(0,1)} \\
x^{(0,2)} &= x^{(0,1)} + ts^{(0,2)} \\
&\vdots \\
x^{(0,N)} &= x^{(0,N-1)} + ts^{(0,N)}
\end{aligned}$$

and one process of these N updates is referred to an **epoch**

Example of mini-batch scheme

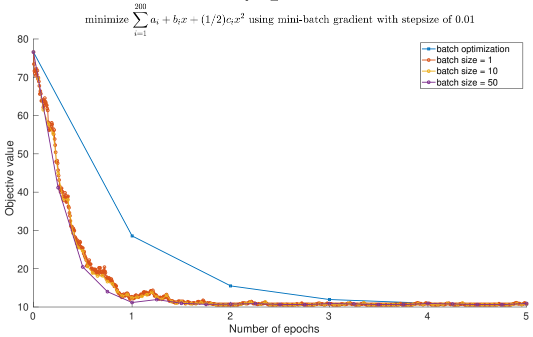
in this figure, batch size is 1



- continue this pattern on a second epoch of steps and so on
- when the batch size is bigger than 1, *e.g.*, a batch contain samples $i = 1, 3, 10, 28$, we take a descent step in minimizing $\sum_{i=1,3,10,28} \|y_i - \hat{y}_i(x)\|_2^2$
- when mini-batch size is k, an epoch will contain $\lceil N/k \rceil$ updates – an epoch means that every point in the training set has been seen exactly once

Performance of mini-batch

example: minimize $\sum_{i=1}^{200} f_i(x)$ where $f_i(x) = a_i + b_i x + (1/2)c_i x^2$

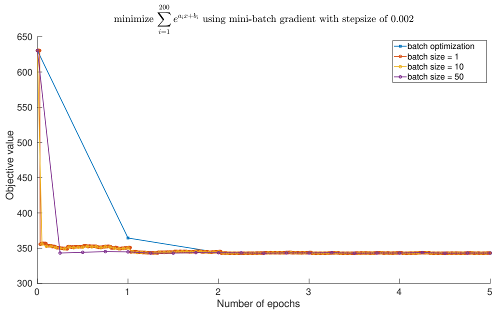


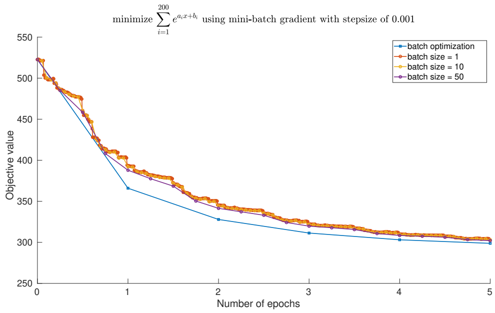
for full batch, 1 epoch is equivalent to 1 iteration

- at 1 epoch, all methods have seen data of length $N = 200$ once
- loss function of mini-batch at each step may not be monotonically decreasing
- mini-batch approaches descend significantly faster than full batch version

Performance of mini-batch

example: minimize $\sum_{i=1}^{200} f_i(x)$ where $f_i(x) = e^{a_i x + b_i}$

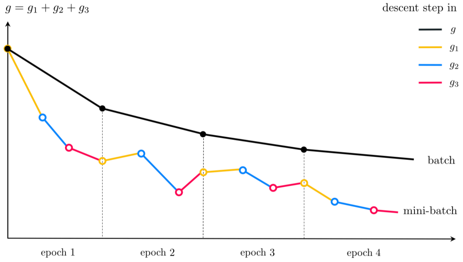




- two instances of data (different (a_i, b_i) in left and right figures)
- often observe benefits of mini-batch when the stepsize is not too small

Performance of mini-batch

mini-batch optimization often performs well when N is large



credit: Machine Learning Refined textbook, page 490

- an initial point of gradient-descent method can be insignificantly incorrect
- using only a small subset of data is sufficient to estimate the descent direction (but with less computational effort)

Convergence of mini-batch optimization

- a sufficient condition to guarantee convergence is $\sum_{k=1}^{\infty} t_k = \infty$ and $\sum_{k=1}^{\infty} t_k^2 < \infty$
- in practice, the learning rate decays linearly until iteration τ

$$t_k = (1-\alpha)t_0 + \alpha t_\tau, \quad \text{with } \alpha = k/\tau$$

  and after $k > \tau$, leave t_k constant

convergence results: measure the **excess error** $f(x) - \min_x f(x)$

- for a convex problem, the excess error is $\mathcal{O}(1/\sqrt{k})$ after k iterations
- for a strongly convex case, the error is $\mathcal{O}(1/k)$
- however, with large data sets, mini-batch can make rapid initial progress and this outweighs its slow asymptotic convergence

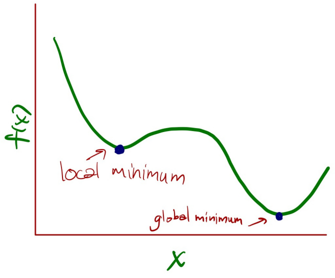
Issues of gradient methods in ML

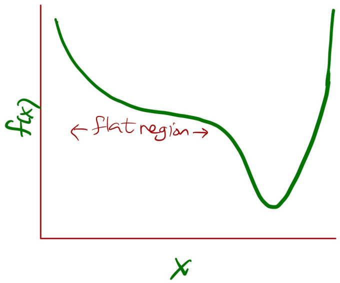
Outlines

- zig-zagging behavior
- gradient method via a change of coordinate
- momentum-accelerated gradient descent
- slow-crawing behavior
- normalized gradient descent

Main challenges

in gradient-based optimization





- zig-zagging behavior of gradient steps
- flat region (updated sequences can stuck)
- high curvature of f

factors: steplength choice, search direction

Convergence of gradient-descent method

assume f is continuously differentiable

the update $x^+ = x - t\nabla f(x)$ gives convergent sequences to a local optimum when t is from any of the following rules

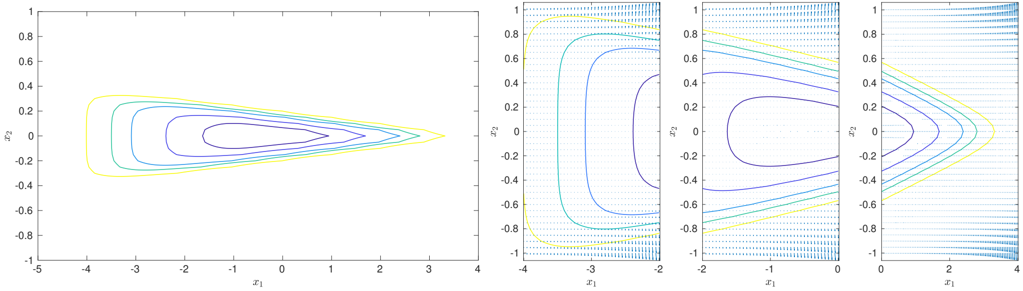
- exact line search
- backtracking line search
- fixed $0 < t < 2/L$ where L is a Lipschitz constant of ∇f (require Lipschitz continuity)

however, in ML applications,

- using a line search requires additional computation cost
- f is highly nonlinear and not possible to find L
- a fixed step length (aka learning rate) is often used and experiment on the choice

Curvature of f

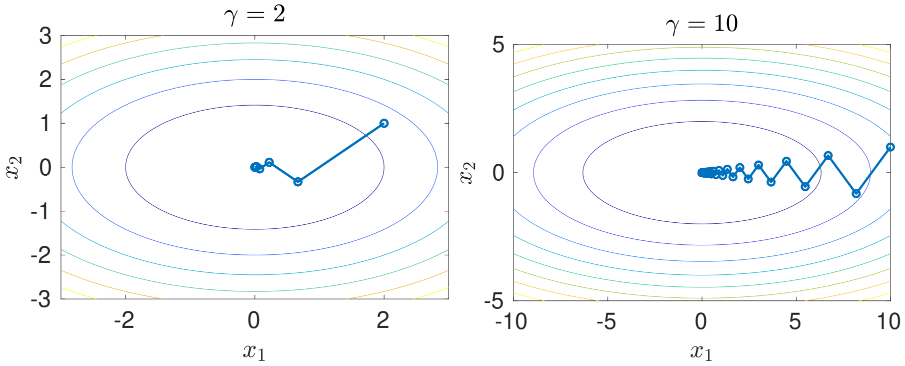
curvature of a graph provides a measure how sharply a smooth curve turns



- we want to use a large step in the direction of low curvature (to make a rapid progress) and vice versa
- when ∇f has different magnitudes in different direction, choosing t is hard for gradient steps – in this figure, a large t can make a huge progress in x_2 direction (and could be further diverge) but a small t gives a slow update in x_1 direction

Zig-zagging behavior of gradient descent

minimize $f(x) = (1/2)(x_1^2 + \gamma x_2^2)$ using exact line search and $x^{(0)} = (\gamma, 1)$



- for $\gamma > 1$, the gradient is more significant in x_2 direction, making the contour of f (which is ellipsoid) elongates in x_1 direction
- when γ is large, the gradient method can cause a zig-zagging behavior and has a slow convergence

Change of coordinates

Weighted norm and dual norm

a weighted norm (by matrix $P \succ 0$) is defined as

$$\|z\|_P = z^T P z = \|P^{1/2} z\|_2$$

(it satisfies all three properties of a norm)

the **dual norm** of a norm $\|\cdot\|$ on $\mathbf{R}^n$ is denoted by

$$\|z\|_* = \sup \{ z^T x \mid \|x\| \leq 1 \}$$

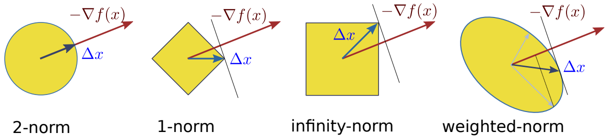
- the dual norm is a norm
- the dual norm of the Euclidean norm is the Euclidean norm itself
- the dual norm of the ℓ_1-norm is the ℓ_∞-norm (and vice versa)
- the dual norm of $\|z\|_P$ is $\|P^{-1/2} z\|_2$

Steepest descent for quadratic norm

for any norm $\|\cdot\|$, we define two steepest descent directions by considering $\nabla f^T(x)v$

- $\nabla f^T(x)v$ is the **directional derivative** of f at x in the direction v
- **normalized steepest descent**: gives the largest decrease in directional derivative

$$\Delta x_{\text{nsd}} = \operatorname{argmin} \{\nabla f(x)^T v \mid \|v\| = 1\ \}$$



- **unnormalized steepest-descent**: scaled by the dual norm of gradient

$$\Delta x_{\text{sd}} = \|\nabla f(x)\|_* \Delta x_{\text{nsd}}$$

Example

a weighted norm $\|z\|_P := \|P^{1/2}z\|_2$ for $P \succ 0$

- minimize $\quad w^T v$ over $\|P^{1/2}v\|_2 = 1$ is equivalent to minimize $w^T(P^{-1/2}y)$ over $\|y\|_2 = 1$
- the solution is $v = -\frac{P^{-1}w}{\|P^{-1/2}w\|_2}$
- from the above result, the normalized steepest descent direction is

$$\Delta x_{\text{nsd}} = -\frac{P^{-1}\nabla f(x)}{\|P^{-1/2}\nabla f(x)\|_2}$$

- dual norm is $\|z\|_* = \|P^{-1/2}z\|_2$, so the unnormalized direction is

$$x_{\text{sd}} = -P^{-1}\nabla f(x)$$

Gradient method via a change of coordinate

a change of coordinate via $z = P^{1/2}x$ where $P^{1/2}$ is a square root of $P \succ 0$

the objective function and the negative gradient step in the new coordinate are

$$\begin{array}{rcl} f(x) & = & f(P^{-1/2}z) \triangleq \tilde{f}(z), \\ \Delta z & = & -\nabla \tilde{f}(z) = -P^{-1/2}\nabla f(P^{-1/2}z) = -P^{-1/2}\nabla f(x) \end{array}$$

the gradient search in z coordinate corresponds to the direction in x variable as

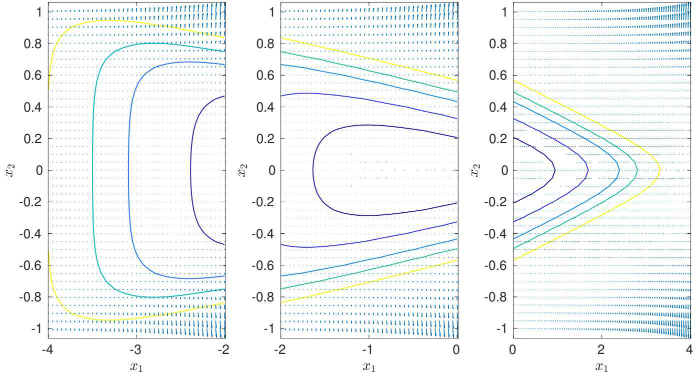
$$\Delta x = P^{-1/2}\Delta z = P^{-1/2}(-P^{-1/2}\nabla f(x)) = -P^{-1}\nabla f(x)$$

this can be interpreted as the steepest-descent direction in **quadratic norm** $\|\cdot\|_P$

data normalization in neural network (at input layer) can be seen as a change of coordinate before applying a gradient method

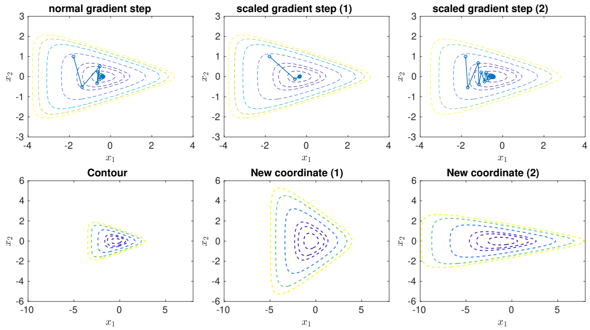
Example of gradient descent in a new coordinate

minimize $f(x) = e^{x_1+ax_2-0.1} + e^{x_1-ax_2-0.1} + e^{-x_1-0.1}$ using $x^+ = x - tP^{-1}\nabla f(x)$



- when $a > 1$, $\nabla f(x)$ has significant magnitude in x_2 direction ($a = 7$ in the figure)
- $P = I$ (the traditional steepest-descent for Euclidean norm)
- $P_1 = \mathbf{diag}(2, 8)$ and $P_2 = \mathbf{diag}(8, 2)$ (steepest-descent for P-norm)

Example of gradient descent in a new coordinate



- the experimental results when $a = 3$
- the stepsize is obtained via backtracking
- the normal gradient step has zig-zagging behavior for some initial points

- first row: using P_1 gives an improved convergence while P_2 yields a slower convergence
- the second row shows the sublevel sets in the new coordinates w.r.t P_1 and P_2

- when changing the coordinate, the Hessian of $\tilde{f}$ is $P^{-1/2}\nabla f^2 P^{-1/2}$, so we select P so that the new Hessian has a low condition number

$$P^{-1/2}\nabla f^2 P^{-1/2} \approx I$$

  (as the condition number typically appears in the convergence of $f(x^{(k)}) - p^\star$
- the sublevel set with P_1 shows a lower condition number as compared to P_2 (decrease in function value are about the same rate in both x_1 and x_2 directions)

Momentum-accelerated algorithms

Exponential averaging

some common smoothing techniques of sequences $y_0, y_1, \dots, y_K$

- **cumulative moving average:** take the sample means

$$z_k = (y_1 + \cdots + y_k)/k \quad \Leftrightarrow \quad z_k = \left(\frac{k-1}{k}\right) z_{k-1} + \frac{1}{k} y_k \qquad \text{(recursive equation)}$$

  as k is large, the weight of y_k is less

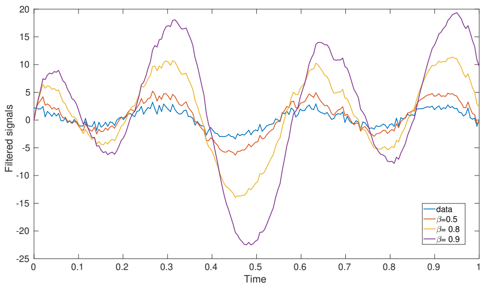
- **exponential moving average (EMA):** autoregressive sequence with y_k as input

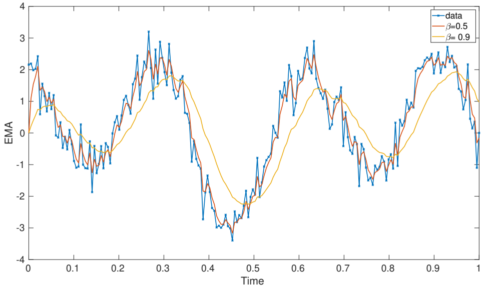
$$z_k = \beta z_{k-1} + (1-\beta) y_k, \quad \beta \in [0,1]$$

$$z_k = \beta^k z_0 + \sum_{\tau=1}^{k} \beta^{k-\tau}(1-\beta) y_\tau$$

  effect of past z_t's decays; z_k depends on the latest z_{k-1} and the recent y_k

Example: effect of β





- left: sequence $z_k = \beta z_{k-1} + y_k$
- right: EMA sequence $z_k = \beta z_{k-1} + (1-\beta) y_k$
- both z_k can be interpreted as a filtered version of y_k but with different gain
- as β increases, z_k is smoothen out but has a higher bias deviating from y_k

Physical analogy of momentum

a particle with mass m moving through a viscous medium caused by a force F

$$m\ddot{x}(t) + \mu\dot{x}(t) = F$$

- -$\mu\dot{x}(t) = -\mu v(t)$ is the viscous friction (proportional to velocity)
- discretizing $\dot{x}(t) = (x(t+h) - x(t))/h$ where h is a stepsize

  $m\left(\frac{x(t+h)+x(t-h)-2x(t)}{h^2}\right) + \mu\left(\frac{x(t+h)-x(t)}{h}\right) = F$
- arranging the term and we obtain a discretized equation of sequence $v^{(k)}$

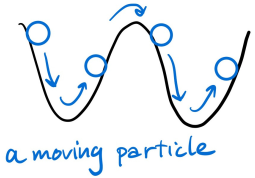
$$\begin{aligned} x(t+h) - x(t) &= \frac{h^2}{m+\mu h}F + \frac{m}{m+\mu h}(x(t) - x(t-h)) \\ v^{(k)} &\triangleq \epsilon F + \beta v^{(k-1)} \end{aligned}$$

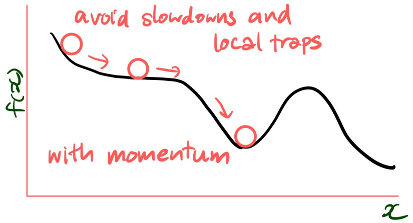
- F will be represented by the negative gradient of objective function

Negative gradient as force

net force: = negative gradient and viscous friction

momentum algorithm: $v^{(k)} = \beta v^{(k-1)} - \epsilon \nabla f(x^{(k-1)}), \quad x^{(k)} = x^{(k-1)} + v^{(k)}$





- we can think of $-\nabla f$ as a force moving a particle with velocity v which then represents the momentum for unit mass
- instead of using $-\nabla f$ to change the *position*, use it to change the **velocity**

- β is referred to a friction/damping/momentum parameter; ϵ is a **learning rate**
- as the particle rolls down and pick up the speed, it can escape from the flat region or local traps in the bowl (overshooting)
- if $-\nabla f$ is the only force, the particle never comes to rest (the hill is frictionless)
- while increasing β helps in avoiding local optima, it might also increase oscillation at the end (β should be < 1)
- as $v^{(k)}$ is EMA of past gradients, zigzagging behavior of gradients can be smoothed out for v to update x

Momentum-accelerated gradient descent

to ameliorate some zig-zagging behavior of gradient descent steps

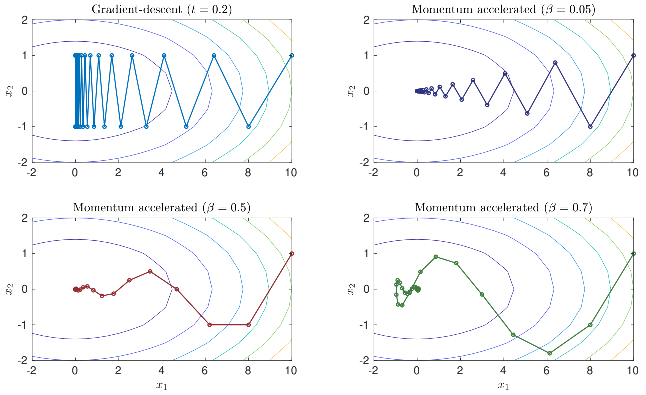
- initialize $v^{(0)} = -\nabla f(x^{(0)})$ (using the **negative** gradient direction)
- perform exponential averaging the descent direction

$$\begin{aligned} v^{(k)} &= \beta v^{(k-1)} + (1-\beta)(-\nabla f(x^{(k-1)})) \\ x^{(k)} &= x^{(k-1)} + t v^{(k)} \end{aligned}$$

- $v^{(k)}$ captures EMA or **momentum** of the directions preceding it
- we can think of $-\nabla f$ as a force moving a particle with velocity v which then represents the momentum for unit mass
- typical use of large $\beta \in [0.7, 1]$ means the *less* $v^{(k-1)}$ uses the actual negative gradient direction, but the *more* it summarizes all previous negative gradients

Example

example: minimize $(x_1^2 + 10x_2^2)$ using gradient methods with $t = 0.2$



zig-zagging behavior of gradient descent can be ameliorated using a higher value of β

Nesterov accelerated gradient (NAG)

classical momentum form

$$v^{(k)} = \beta v^{(k-1)} - \epsilon \nabla f(x^{(k-1)}), \quad x^{(k)} = x^{(k-1)} + v^{(k)}$$

we call β as **momentum coefficient** and ϵ relates to the learning rate

Sutskever et al. 2013 proposed to combine Nesterov gradient method and momentum

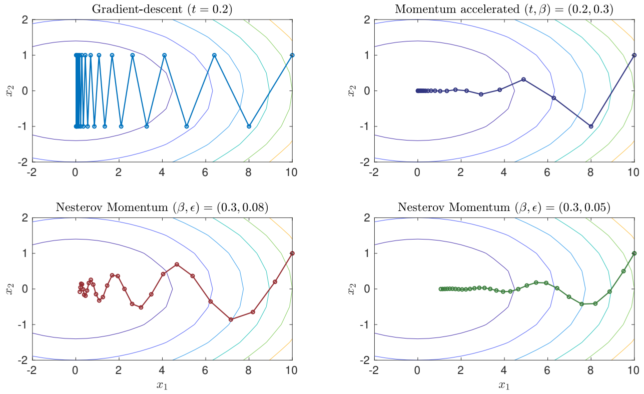
$$\begin{aligned} v^{(k)} &= \beta v^{(k-1)} - \epsilon \nabla f(x^{(k-1)} + \beta v^{(k-1)}) \\ x^{(k)} &= x^{(k-1)} + v^{(k)} \end{aligned}$$

the gradient is evaluated at the point as if $x^{(k)}$ was updated from the current velocity

we can initialize the update using $v^{(0)} = -\epsilon \nabla f(x^{(0)})$

Example

minimize $(x_1^2 + 10x_2^2)$ using gradient methods

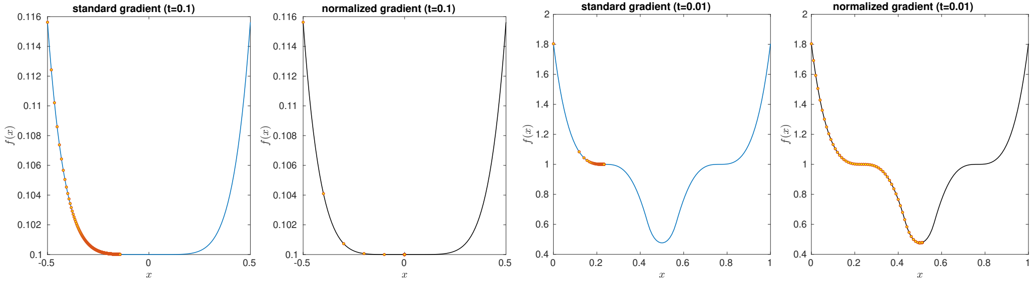


- the less ϵ, the slower the algorithm converges)
- in this experiment, no convergence when setting $\epsilon = t = 0.2$, so we show the results of using smaller ϵ

Slow-crawling behavior of gradient descent

when the objective function has a **flat region** near a local minimum

the gradient near stationary points vanishes and the sequence updates slowly

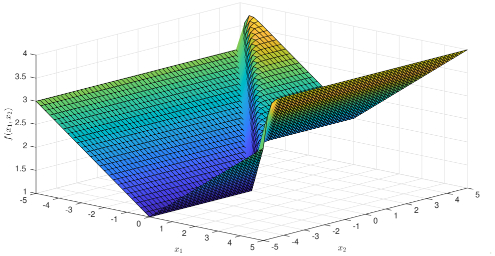


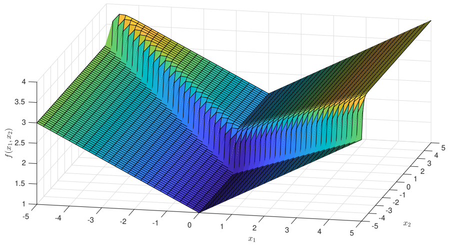
- (left) $f(x) = x^6 + 0.1$ is minimized $(0,0)$
- (right) $f(x) = \max^2(0, 1 + (3x - 2.3)^3) + \max^2(0, 1 + (-3x + 0.7)^3)$ has a minimum at $x = 1/2$ and saddle points at $x = 7/30$ and $x = 23/30$
- choice of initial guess, steplength and crawling behavior of standard gradient can prevent it from finding a local mininum

Flat region along one direction

when the function contour is flat along one direction (here, x_2)

$$f(x) = \max(0, \tanh(a(x_1 + x_2))) + |bx_1| + 1, \quad a = 4, b = 0.4$$





- the minimum is at $x = (0, 0)$ and $p^\star = 1$
- the standard gradient sequences halt (making no progress) and stuck in the valley

Normalized gradient descent

vanishing magnitude of $-\nabla f(x)$ can cause the method to halt at saddle points

we can control the magnitude of $-\nabla f(x)$ via a normalization

- norm-2 of the negative gradient

$$x^{(k)} = x^{(k-1)} - \left(\frac{t}{\|\nabla f(x^{(k-1)})\|_2 + \epsilon} \right) \nabla f(x^{(k-1)})$$

- the component-wise magnitude of the gradient

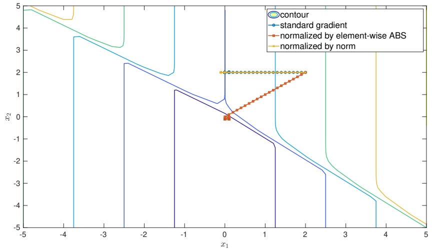
$$x_i^{(k)} = x_i^{(k-1)} - t \cdot \frac{\nabla f(x^{(k-1)})_i}{|\nabla f(x^{(k-1)})_i| + \epsilon}, \quad i = 1, 2, \ldots, n$$
$$x^{(k)} = x^{(k-1)} - t \cdot \mathbf{sign}(\nabla f(x^{(k-1)})), \;\; \text{if } \epsilon = 0$$

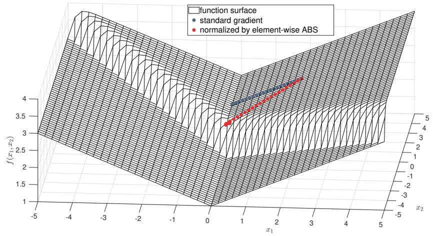
(because $y/|y| = \mathbf{sign}(y)$)

- normalized and standard gradient descent step only differ by the steplength choice

Example

result: minimize $f(x) = \max(0, \tanh(a(x_1 + x_2))) + |bx_1| + 1$ using $a = 4, b = 0.4$





- if x_2 is large enough (so that $\tanh$ becomes saturated), $\partial f / \partial x_2$ is nearly zero
- the standard gradient cannot make a progress along x_2 dimension
- the normalized gradient by its norm can be problematic (in this case) because it even makes the gradient w.r.t. x_2 smaller
- the normalized gradient by component-wise magnitude can make significant progresses in 20 steps (out of 1000 iterations, using $t = 0.1$)

Gradient-based optimizers in ML

many algorithms proposed as extension of gradient descent for ML (aka **optimizers**)

- AdaGrad (Duchi et al. 2011)
- Adadelta (extension of AdaGrad to restrict the window of gradient averaging)
- RMSProp (Tieleman and Hinton 2012)
- Adam (Adaptive moment estimation) (Kingma et al. 2015)
- AdaMax (a generalization of Adam to using ℓ_∞-norm) (Kingma et al. 2015)
- Nadam (Nesterov-Adam) (Dozat 2016)

these algorithms combine **averaging of past gradients** and **component-wise normalization of gradient**

component-wise normalized gradient is often referred to **adaptive learning rate**

Gradient descent algorithms in ML

use aggregated information of past gradients to normalize the search direction

AdaGrad (Duchi et al. 2011)

$$z_i^{(k)} = z_i^{(k-1)} + \left[\nabla f(x^{(k-1)})_i\right]^2, \quad x_i^{(k)} = x_i^{(k-1)} - t\,\frac{\nabla f(x^{(k-1)})_i}{\sqrt{z_i^{(k)}}}$$

RMSProp (Tieleman and Hinton 2012)

$$z_i^{(k)} = \beta z_i^{(k-1)} + (1-\beta)\left[\nabla f(x^{(k-1)})_i\right]^2, \quad x_i^{(k)} = x_i^{(k-1)} - t\,\frac{\nabla f(x^{(k-1)})_i}{\sqrt{z_i^{(k)}}}$$

Adam (Kingma et al. 2017)

$$\begin{aligned} y_i^{(k)} &= \beta_1 y_i^{(k-1)} + (1-\beta_1)\nabla f(x^{(k-1)})_i \\ z_i^{(k)} &= \beta_2 z_i^{(k-1)} + (1-\beta_2)\left[\nabla f(x^{(k-1)})_i\right]^2 \\ x_i^{(k)} &= x_i^{(k-1)} - t\left(\frac{\sqrt{1-\beta_2^k}}{1-\beta_1^k}\right)\frac{y_i^{(k)}}{\sqrt{z_i^{(k)}}+\epsilon} \end{aligned}$$

- $z^{(k)}$ and $y^{(k)}$ represent histories of the *first* and *second* moments of ∇f resp.
- the ith component of $-\nabla f(x^{(k)})$ is normalized *individually*

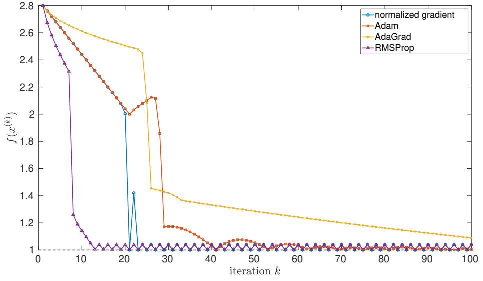
Interpretation

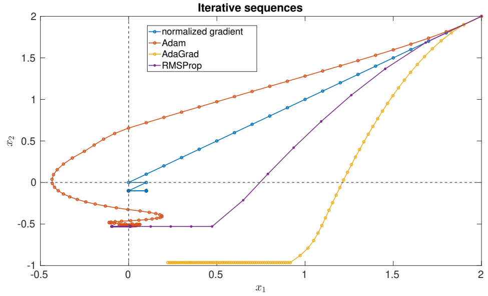
- **AdaGrad:** $z^{(k)}$ is the cumulative sum of squared gradients up to time k
  - using the square root of $z_i^{(k)}$ as normalization penalizes the ith component of the gradient that highly fluctuates
  - as $z^{(k)}$ tends to be eventually large, so the search direction tends to slow down
- **RMSProp:** $z^{(k)}$ is the EMA of squared gradient
  - the normalization by $\sqrt{z^{(k)}}$ is similar to AdaGrad
  - since $z^{(k)}$ represents EMA of squared gradient, the progress of $x^{(k)}$ is not slowed prematurely
  - however, if $z^{(0)}$ is set to 0, this causes undesirable bias in early iterations
- **Adam:** use EMAs of both gradient and its square with $\beta_1 = 0.9, \beta_2 = 0.999$
  - the search direction is the momentum of gradients scaled by the EMA of the second moment of gradient
  - the initial estimates $z^{(0)}$ and $y^{(0)}$ are zero which are biased in early iterations
  - improving over RMSProp, Adam has a bias correction term that converges to 1

Comparison of optimizers: ReLu and tanh functions

the example of f that has a flat region along x_2 direction on page 58

$$f(x) = \max(0, \tanh(a(x_1 + x_2))) + |bx_1| + 1, \quad a = 4, b = 0.4$$

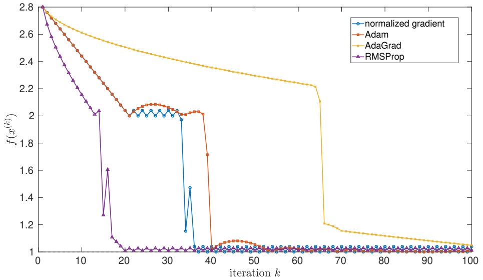


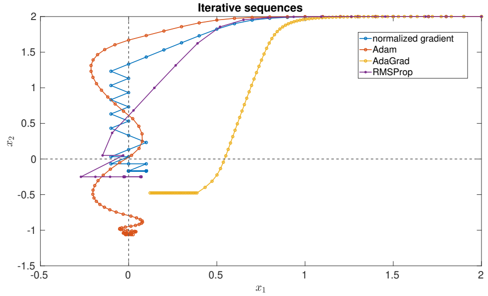


- stepsize $t = 0.1$, initialized at $x^{(0)} = (2, 2)$, β of RMSProp is set as 0.9, all auxiliary variables (z, y) were initialized at zero
- all methods can avoid being stuck the valley; only Adagrad halted at another x_2

Comparison of optimizers: ReLu and tanh functions

same setting but using $\epsilon = 10^{-8}$ in the normalization of all methods



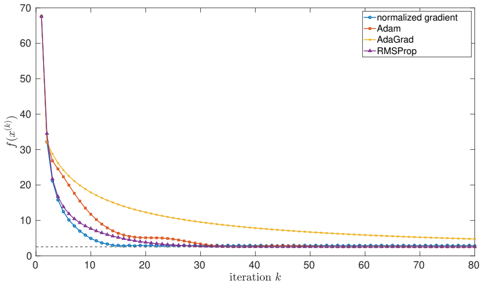


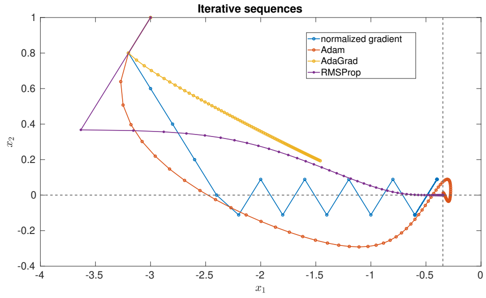
- $\nabla f(x)_2$ is zero when $|x_2|$ is large enough
- both normalized gradient and RMSProp uses $\nabla f(x)_i$ as search directions, we see ripples of x near the optimum since their $\nabla f(x)_2$ become zero
- the search direction of Adam is y_i, which is EMA of gradients; hence, the search direction becomes small near optimum, but not exact zero in x_2 direction – giving smoother path of x

Comparison of optimizers: sum of exponential

(a convex) $f(x)$ on page 44 that $\nabla f(x)$ is more significant in one direction

$$f(x) = e^{x_1+7x_2-0.1} + e^{x_1-7x_2-0.1} + e^{-x_1-0.1}$$

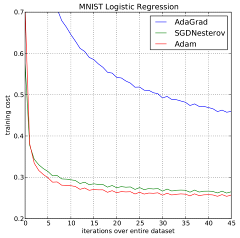


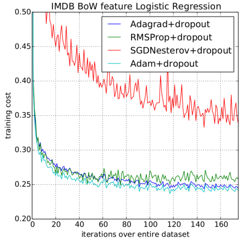


- stepsize $t = 0.2$, $x^{(0)} = (-3, 1)$, $\epsilon = 10^{-8}$, $\beta_{\text{RMSProp}} = 0.9$
- both RMSProp and Adam performed better than other methods
- AdaGrad is slowest; normalized gradient progressed quickly but oscillated

Adam's performance

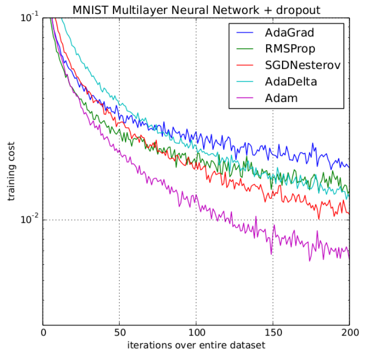
figures taken from Kingma et al. 2015 (Adam paper)



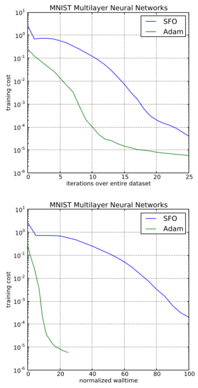


- ℓ_2-regularized multi-class logistic regression (convex problem)
- MNIST data set: Adam has similar convergence to SGD and both are faster than AdaGrad
- IMDB movie review data set: both Adam and AdaGrad (which can be efficiently deal with sparse features) have faster convergence than SGD with Nesterov

figures taken from Kingma et al. 2015 (Adam paper)



(a)



(b)

- multilayer NN with cross-entropy loss and dropout (non-convex problems)
- Adam had better convergence than others, and was compared with SFO (sum-of-functions) which is a quasi-Newton method

Summary

choosing an algorithm depends on the properties of f and problem scale

- differentiable f, medium scale: CG, quasi-Newton
- convex and smooth f: interior-point methods, first-order accelerated methods
- large-scale, non-convex: normalized momentum-accelerated algorithms
- a large sum of f_i: incorporate mini-batch optimization

notes: the performance of algorithms depend on

- initial guess of $x^{(0)}$: lead to poor local minima, saddle points, flat region
- steplength: a larger t makes $x^{(k)}$ have a significant progress; but may lead to divergence when t is too large
- algorithm parameters: different algorithms can have different proper choices (e.g., momentum coeffiicent β can be differently chosen for each method)

Convergence analysis

based on the original documents of described algorithms

- Adam and AdaGrad papers have performance analysis based on regret function (used in a context of online learning)
- RMSProp and Nadam are intuitive and empirical work

more recent literature on the analysis of these algorithms exist and should be further explored

References

gradient methods for ML

1. Chapter 3 and Appendix A in J. Watt, R. Borhani, and A.K. Katsaggelos, *Machine Learning Refined: Foundations, Algorithms, and Applications*, 2nd edition, Cambridge University Press, 2020
2. Chapter 4 and 5 in C.C. Aggarwal, *Linear Algebra and Optimization for Machine Learning: A Textbook*, Springer 2020
3. Chapter 6 in I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, The MIT Press, 2016
4. J.Duchi, E.Hazan, Y.Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research, 2011
5. I. Sutskever, J. Martens, G. Dahl and G. Hinton, *On the importance of initialization and momentum in deep learning*, ICML, 2013
6. G. Hinton, N.Srivastava, and K. Swersky, *RMSProp: Divide the gradient by a running average of its recent magnitude*, Lecture notes of Neural Networks for Machine Learning, University of Toronto [Slide:] `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf` [Video:] `https://www.youtube.com/watch?v=XhZahXzEuNo`
7. D.P. Kingma, *ADAM: A Method for stochastic optimization*, ICLR, 2015
8. T. Dozat, *Incorporating Nesterov momentum into ADAM*, ICLR workshop, 2016