Optimization for Machine Learning

Aymeric Dieuleveut

March 11, 2026

Teaching team

Lectures

1. Aymeric Dieuleveut: main teacher optimization.
2. Adrien Taylor: main teacher deep learning, practicals

Practicals

1. Adrien Taylor
2. Pascal Bianchi
3. Baptiste Goujaud
4. Daniel Berg Thomsen

Rooms: PC 16-19 this afternoon, BEM afterwards

Ressources

Information on Moodle: → moodle
APM_52445_EP - Optimization / Deep Learning (2024-2025).

Communication on Slack (corrections, etc.) and interactive polls via Wooclap

1. Slack: join link.
2. Wooclap:

Also shared on Moodle.

Planning

1. Thu 22 Jan – Lecture 1 – Optimization
2. Wed 28 Jan – Lecture 2 – Deep Learning
3. Wed 4 Feb – Lecture 3 – Optimization
4. Wed 11 Feb – Lecture 4 – Deep Learning
5. Wed 18 Feb – Lecture 5 – Deep Learning
6. Wed 25 Feb – Lecture 6 – Deep Learning
7. Wed 4 Mar no lecture
8. Wed 11 Mar – Lecture 7 – Optimization

Planning

1. Thu 22 Jan – Lecture 1 – Optimization
2. Wed 28 Jan – Lecture 2 – Deep Learning
3. Wed 4 Feb – Lecture 3 – Optimization
4. Wed 11 Feb – Lecture 4 – Deep Learning
5. Wed 18 Feb – Lecture 5 – Deep Learning
6. Wed 25 Feb – Lecture 6 – Deep Learning
7. Wed 4 Mar no lecture
8. Wed 11 Mar – Lecture 7 – Optimization

Grading:

1. Attendance, lab participation
2. Graded Lab at home (groups of 3 students)
3. Exam

Outline

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD $(b = 1)$ rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

For $\epsilon = 1/\sqrt{n}$

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$
For $\epsilon = 1/\sqrt{n}$	$O\left(n^{3/2}d\right)$	

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$
For $\epsilon = 1/\sqrt{n}$	$O\left(n^{3/2}d\right)$	$O(nd)$

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$
For $\epsilon = 1/\sqrt{n}$	$O\left(n^{3/2}d\right)$	$O(nd)$
For $\epsilon = 1/n$	$O\left(n^2 d\right)$	$O\left(n^2 d\right)$

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$
For $\epsilon = 1/\sqrt{n}$	$O\left(n^{3/2}d\right)$	$O\left(nd\right)$
For $\epsilon = 1/n$	$O\left(n^2d\right)$	$O\left(n^2d\right)$
For $\epsilon = 1/n^2$	$O\left(n^3d\right)$	$O\left(n^4d\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

Comparison of GD and SGD - Class of L-smooth convex functions.

SGD ($b = 1$) rate

$$\mathbb{E}\left[f\left(\bar{\theta}_T\right)\right] - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|\sigma}{\sqrt{T}}.$$

$\Rightarrow$ Optimization complexity

$$T_\epsilon = \frac{4\|\theta_0 - \theta_\star\|^2\sigma^2}{\epsilon^2}.$$

$\rightarrow$ 1. For a given target ϵ

Method	GD	SGD
# of iter. T_ϵ for ϵ precision	$O\left(\frac{1}{\epsilon}\right)$	$O\left(\frac{1}{\epsilon^2}\right)$.
Cost per iteration C	$O(nd)$	$O(d)$
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$

$\rightarrow$ The total complexity does not depend on n for SGD!

$\rightarrow$ For any given ϵ, the total complexity of SGD is smaller than the one of GD if n *is large enough.*

$\rightarrow$ 2. If ϵ is fixed relatively to n?

Method	GD	SGD
Total complexity $= T_\epsilon \times C$	$O\left(\frac{nd}{\epsilon}\right)$	$O\left(\frac{d}{\epsilon^2}\right)$
For $\epsilon = 1/\sqrt{n}$	$O\left(n^{3/2}d\right)$	$O\left(nd\right)$
For $\epsilon = 1/n$	$O\left(n^2d\right)$	$O\left(n^2d\right)$
For $\epsilon = 1/n^2$	$O\left(n^3d\right)$	$O\left(n^4d\right)$

$\rightarrow$ SGD is faster than GD when aiming for a moderate precision w.r.t. n!

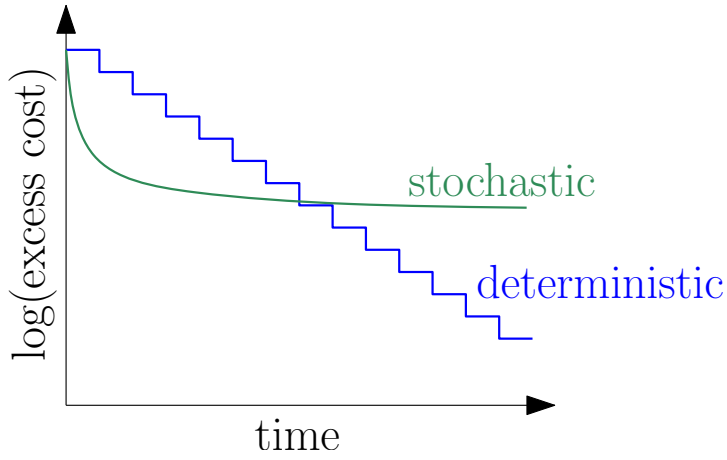
$\rightarrow$ For a high precision w.r.t. n, GD has a lower complexity.

$\rightarrow$ Recall that in ML the typical precision needed is *moderate*.

Comparison of convergence : SGD vs GD

Which one to choose?

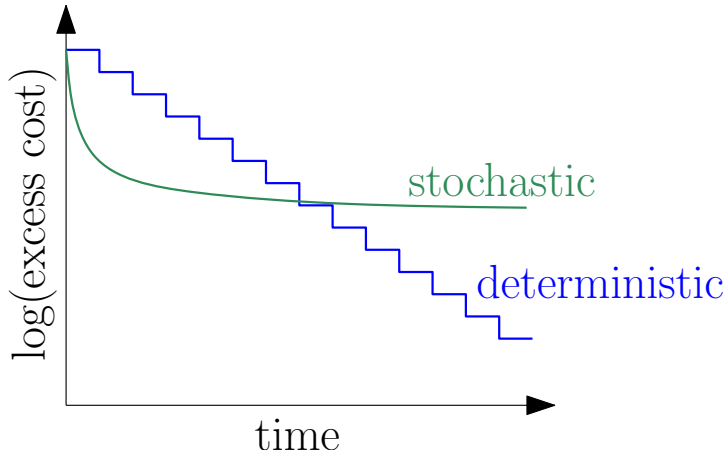
1. Depends on the precision we want.



Comparison of convergence : SGD vs GD

Which one to choose?

1. Depends on the precision we want.



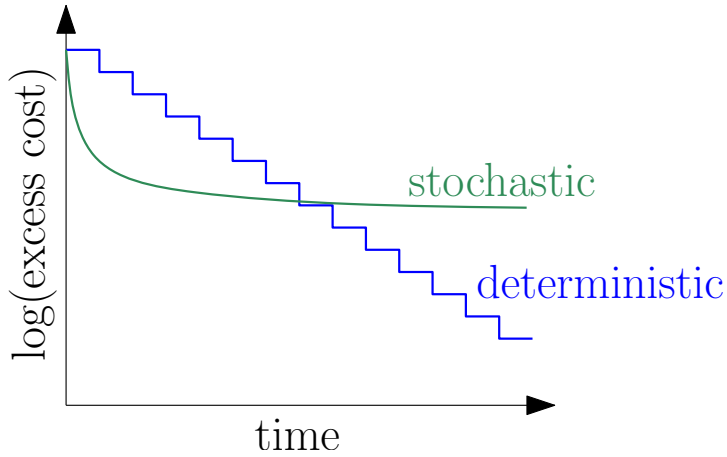
   Example: non strongly convex case.

2. If our goal is to get a convergence of $1/\sqrt{n}$, then
   - Complexity of GD: $n^{3/2}d$
   - Complexity of SGD: nd.

Comparison of convergence : SGD vs GD

Which one to choose?

1. Depends on the precision we want.



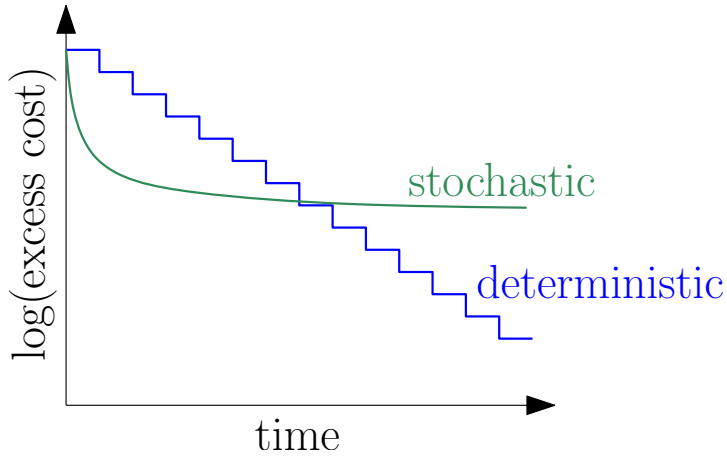
   Example: non strongly convex case.

2. If our goal is to get a convergence of $1/\sqrt{n}$, then
   - Complexity of GD: $n^{3/2}d$
   - Complexity of SGD: nd.
3. If our goal is to get a convergence of $1/n^2$, then
   - Complexity of GD: $n^3 d$ (n^2 iterations)
   - Complexity of SGD: $n^4 d$ (n^4 iterations).

Comparison of convergence : SGD vs GD

Which one to choose?

1. Depends on the precision we want.



   Example: non strongly convex case.

2. If our goal is to get a convergence of $1/\sqrt{n}$, then
   - Complexity of GD: $n^{3/2}d$
   - Complexity of SGD: nd.
3. If our goal is to get a convergence of $1/n^2$, then
   - Complexity of GD: $n^3 d$ (n^2 iterations)
   - Complexity of SGD: $n^4 d$ (n^4 iterations).

   Same comparison can be made, with the same conclusions, - on the class of L-smooth μ strongly-convex functions (left as an exercise).

Comparison of GD and SGD .

$\rightarrow$ Overall, SGD has a lower computational complexity when

- n is large
  or
- for ϵ moderate w.r.t. n.

In other words:

- SGD is extremely fast in the early iterations (first few (1-10) passes on the data)
- But it fails to converge accurately to the minimum

$\rightsquigarrow$ GD vs SGD – Which one to choose?

Comparison of GD and SGD .

→ Overall, SGD has a lower computational complexity when

- n is large
  or
- for ϵ moderate w.r.t. n.

In other words:

- SGD is extremely fast in the early iterations (first few (1-10) passes on the data)
- But it fails to converge accurately to the minimum

⇝ GD vs SGD – Which one to choose?

There is no uniformly best algorithm! The choice between SGD and GD depends on the context!

But, as a moderate precision is often enough in ML and n typically very large, SGD is always preferred to GD in ML applications.

SGD: Summary

First-order method **Stochastic**

Update

$$\theta_t \leftarrow \theta_{t-1} - \eta g_t(\theta_{t-1})$$

with g_t unbiased gradient oracle based on b observations

Hyperparameters: $(\eta_t)_t, T, b$.

$\rightsquigarrow$ choose η decreasing of small, to balance noise and optimization.

$\rightsquigarrow$ choose b small to limit complexity, typically 32 to 256.

Complexity/iter in ML:

$$O(b \times d)$$

Intuition & motivation:

Challenge: n large $\rightarrow$ Solution: stochastic alg.

Behavior

- noise - initial condition tradeoff
- averaging helps reduce the noise.

Comparison to GD.

Lower complexity IF we aim for moderate precision or n is large.

Theory:

- Rate for smooth and (strongly) convex functions
- 💡 Impact of T.
- 💡 Impact of (η_t).

$\rightarrow$ SGD is the building block of all ML algorithms

Outline

1. Extensions of GD and SGD
   - Momentum and Acceleration
   - Second-order algorithms
   - Variance reduced methods for ERM
   - Coordinate methods
   - Adaptivity

2. Wrapping up: Summary of GD extensions

Roadmap of Part 1

We are going to study 5 directions into which GD and SGD can be improved.

1. **Variance reduction.**
   Hybrid methods between GD and SGD: computational cost of SGD, convergence of GD!
2. **Accelerated methods.**
   Builds on GD to improve the convergence for poorly conditioned functions (κ very large).
3. **Second order methods.**
   Whenever the dimension is not too large, use also the Hessian matrix information.
4. **Coordinate dependent-step size.**
   To take into account multiple possible scales, we use a different step for each coordinate.
5. **Adaptivity.**
   The size of the step-sizes depends on the observed values of the function, gradients, etc., not a predefined schedule.

Outline

Acceleration / Momentum algorithms

Setup: deterministic (at first, algorithms behave well also in stochastic regimes).

Goal: taking into account the previous updates as *additional velocity*.

→  If I have been moving *consistently* in one direction over the few last updates, I should move *faster* in that direction.

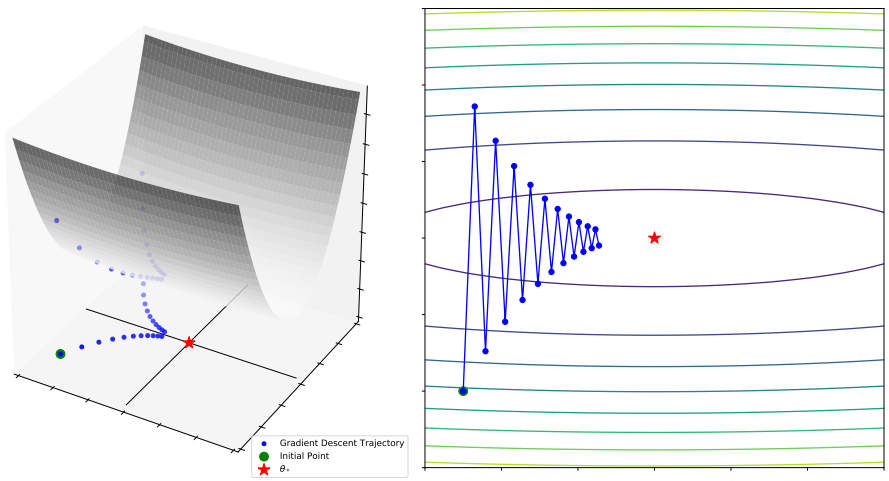
Acceleration / Momentum algorithms

Setup: deterministic (at first, algorithms behave well also in stochastic regimes).

Goal: taking into account the previous updates as *additional velocity*.

→  If I have been moving *consistently* in one direction over the few last updates, I should move *faster* in that direction.

Gradient Descent



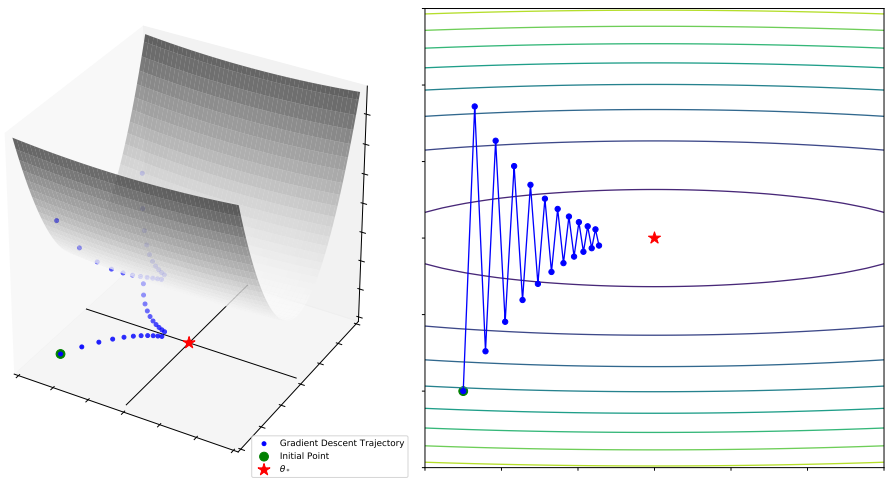
Acceleration / Momentum algorithms

Setup: deterministic (at first, algorithms behave well also in stochastic regimes).

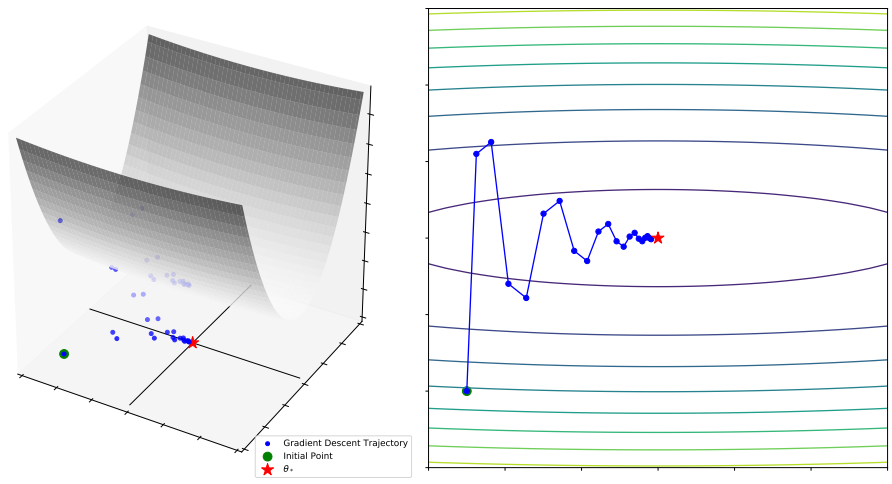
Goal: taking into account the previous updates as *additional velocity*.

→  If I have been moving *consistently* in one direction over the few last updates, I should move *faster* in that direction.

Gradient Descent



Accelerated Gradient Descent



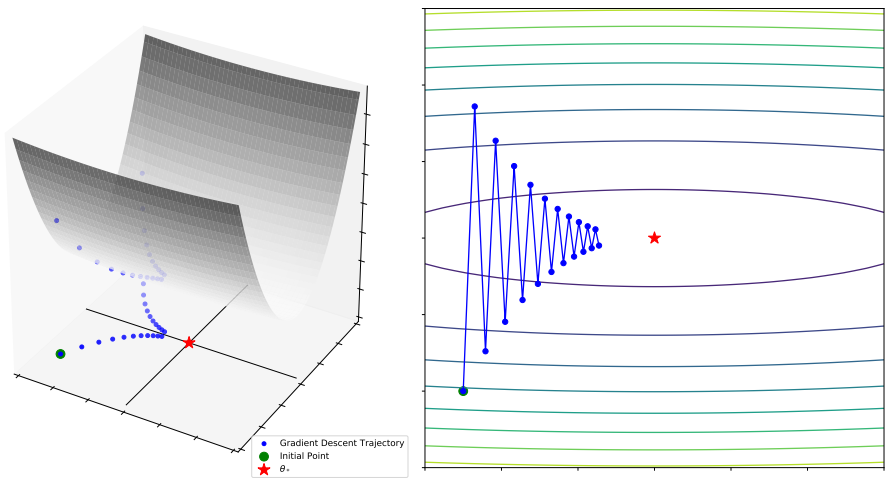
Acceleration / Momentum algorithms

Setup: deterministic (at first, algorithms behave well also in stochastic regimes).

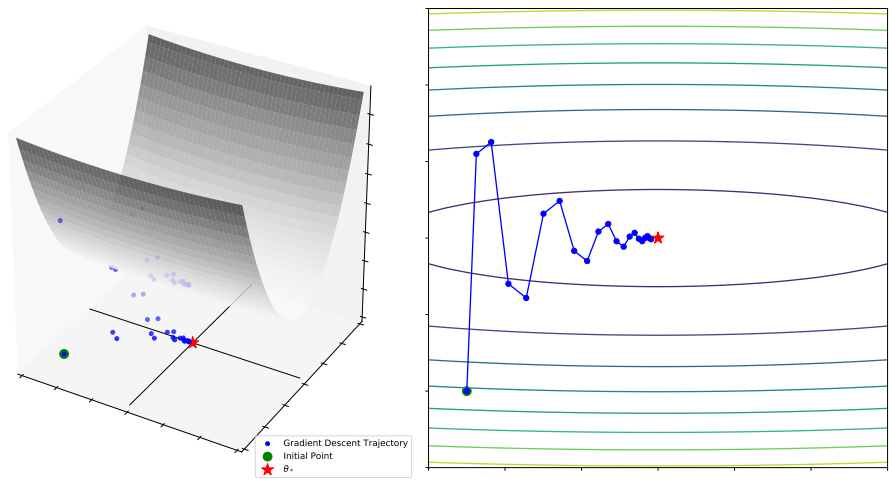
Goal: taking into account the previous updates as *additional velocity*.

→  If I have been moving *consistently* in one direction over the few last updates, I should move *faster* in that direction.

Gradient Descent



Accelerated Gradient Descent



1. Physics inspired approach: the GD update has no "memory": even if we have moved consistently in a direction, we do not *accelerate*.
   → rolling ball analogy.
2. Helpful to improve convergence for *flat areas* (saddle points, bad conditioning), and *local minimum*.

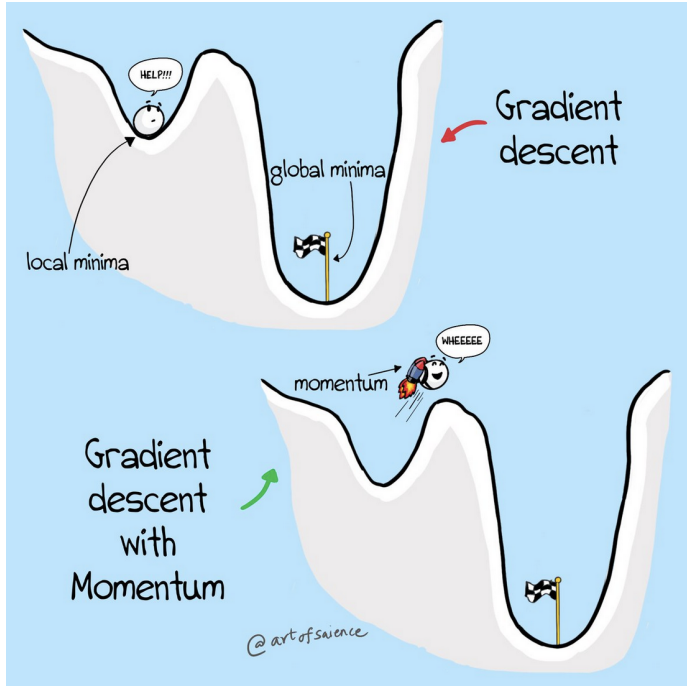


Figure: Classical illustration of HB method for local minima - Source: Twitter @DSaience

Polyak's momentum algorithm (1964) - Heavy ball (HB) method

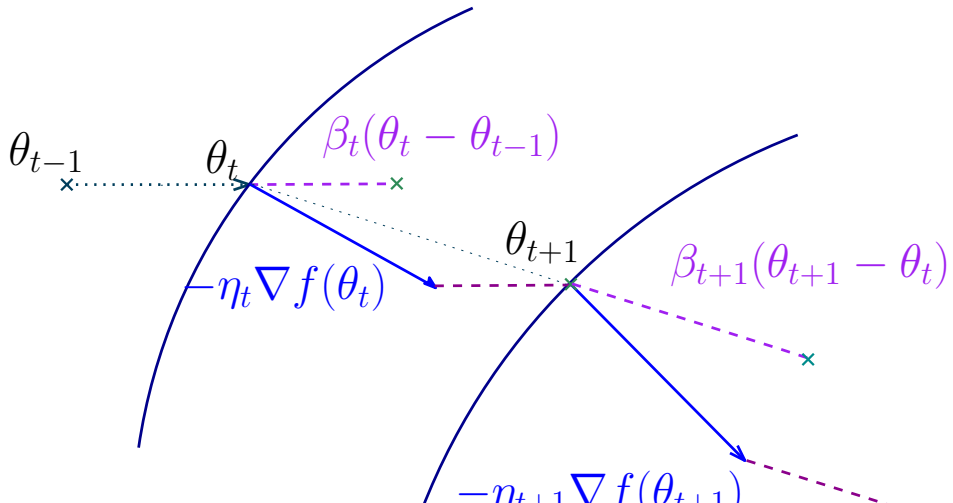
We add to the gradient update a fraction of the difference between the last two iterates, $\beta(\theta_t - \theta_{t-1})$, with $\beta \simeq 1$ typically.

Polyak's momentum algorithm (1964) - Heavy ball (HB) method

We add to the gradient update a fraction of the difference between the last two iterates, $\beta(\theta_t - \theta_{t-1})$, with $\beta \simeq 1$ typically.

Algorithm : Polyak's momentum algorithm - a.k.a. Heavy ball method

Hyperparameters: # steps T, learning rate $(\eta_t)_t > 0$, momentum $(\beta_t)_t \in [0, 1]$



Polyak's momentum algorithm (1964) - Heavy ball (HB) method

We add to the gradient update a fraction of the difference between the last two iterates, $\beta(\theta_t - \theta_{t-1})$, with $\beta \simeq 1$ typically.

Algorithm : Polyak's momentum algorithm - a.k.a. Heavy ball method

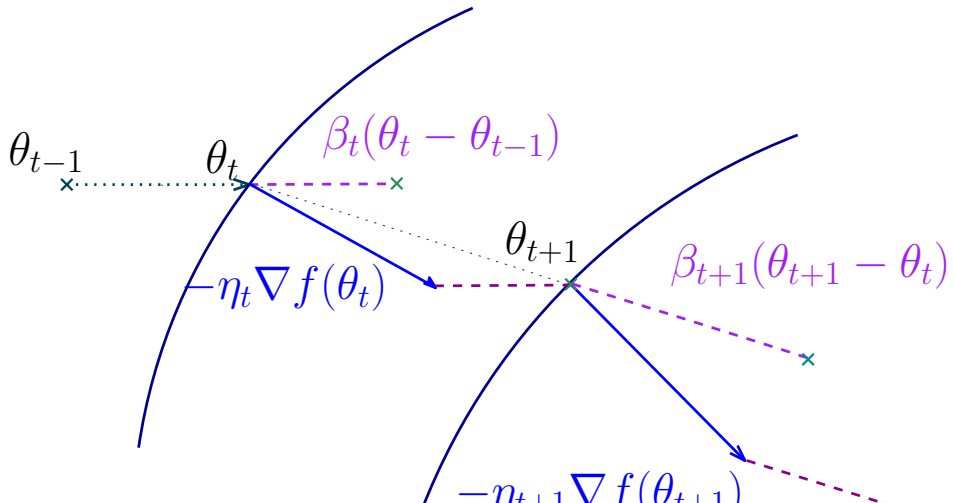
Hyperparameters: # steps T, learning rate $(\eta_t)_t > 0$, momentum $(\beta_t)_t \in [0, 1]$

Initialization: initial weight vector $\theta_1 = \theta_0$

For $t = 1, 2, \ldots, T$ do

$$\theta_{t+1} = \theta_t - \underbrace{\eta_t \nabla f(\theta_t)}_{\text{GD step}} + \underbrace{\beta_t(\theta_t - \theta_{t-1})}_{\text{momentum step / velocity}}$$

Output: Return last θ_T.



Nesterov's Acceleration (NAG, 1983)

Same idea but we first apply the momentum step, *then* take the gradient.

Algorithm : Nesterov accelerated GD

Hyperparameters: # steps T, learning rate $(\eta_t)_t > 0$, momentum $(\beta_t)_t \in [0, 1]$

Initialization: initial weight vector $\theta_1 = \theta_0$.

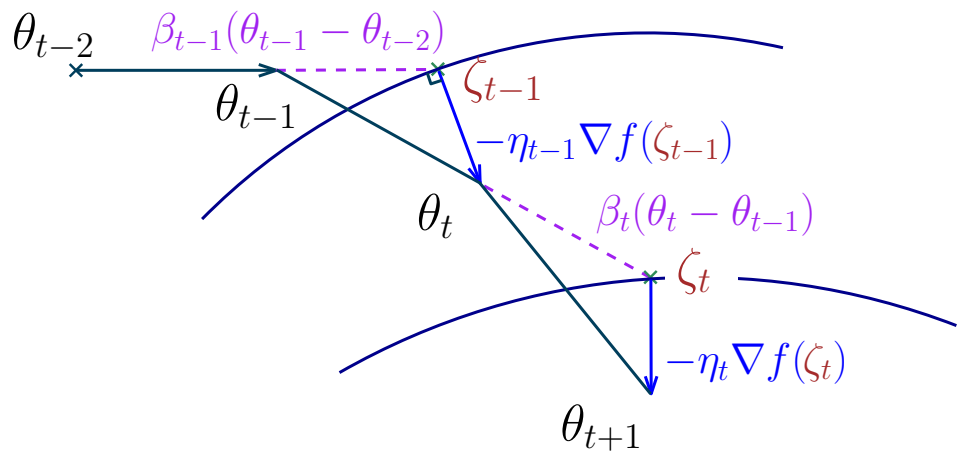
For $t = 1, 2, \dots, T$ do

$$\theta_{t+1} \leftarrow \theta_t + \underbrace{\beta_t(\theta_t - \theta_{t-1})}_{\text{momentum step}} - \underbrace{\eta_t \nabla f(\theta_t + \beta_t(\theta_t - \theta_{t-1}))}_{\text{GD step at } \textit{look-ahead} \text{ point}}$$

Output: Return last θ_T.

Alternative 2-steps description: we introduce the look-ahead point as (ζ_t)

$$\begin{cases} \zeta_t & \leftarrow & \theta_t + \beta_t(\theta_t - \theta_{t-1}) \\ \theta_{t+1} & \leftarrow & \zeta_t - \eta_t \nabla f(\zeta_t) \end{cases}$$

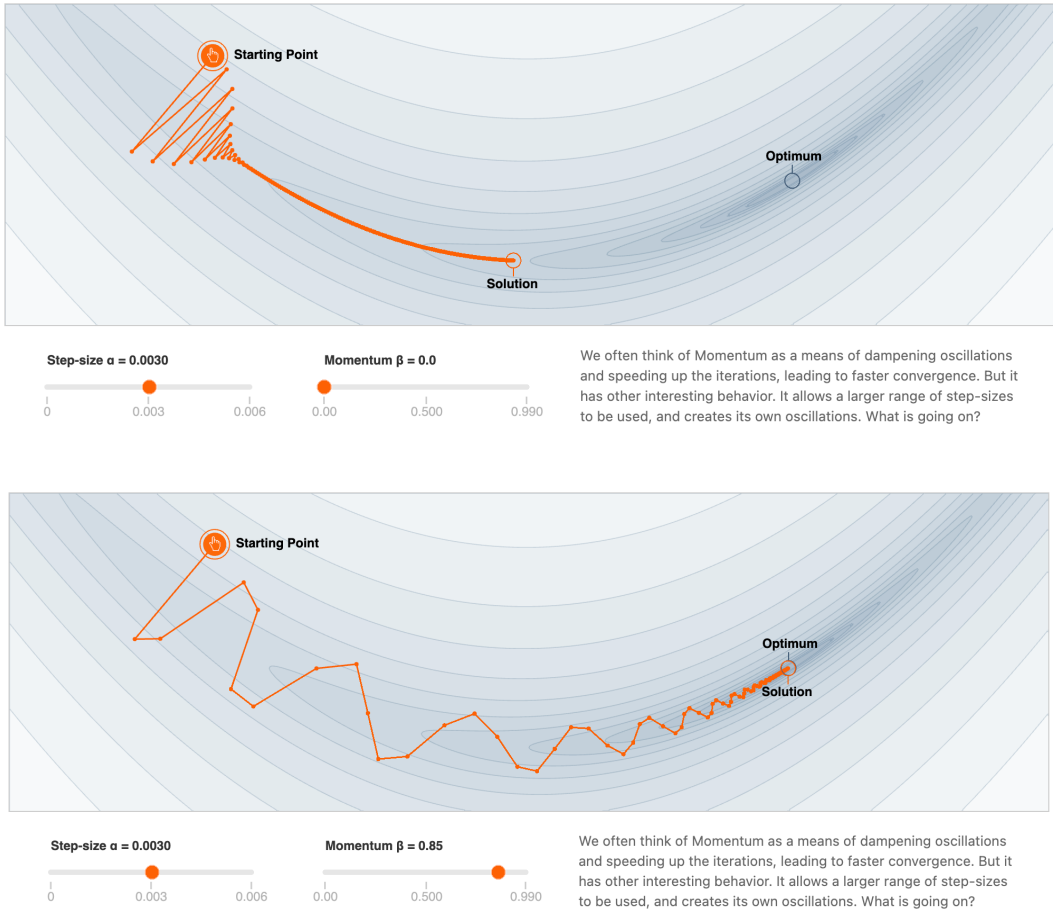


Accelerated methods: more visualization

Polyak Heavy-Ball and Nesterov accelerated gradient descent are both referred to as *accelerated methods* or *momentum methods*.

Interactive visualization with tunable η, β, non convex function,

$\rightarrow$ https://distill.pub/2017/momentum/



Same intuition: directions in which we accelerate consistent progress.

Rate of convergence of Nesterov Accelerated Gradient (NAG) - smooth case

Theorem : NAG rate of convergence - smooth convex functions

If f is L-smooth and convex with a minimum at $\theta_\star$. Then, if for all t, $\beta_{t+1} = \frac{t}{t+3}$, $\eta \leqslant \frac{1}{L}$

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|_2^2}{\eta(t+1)^2}.$$

Rate of convergence of Nesterov Accelerated Gradient (NAG) - smooth case

Theorem : NAG rate of convergence - smooth convex functions

If f is L-smooth and convex with a minimum at $\theta_\star$. Then, if for all t, $\beta_{t+1} = \frac{t}{t+3}$, $\eta \leqslant \frac{1}{L}$

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|_2^2}{\eta(t+1)^2}.$$

Observations:

1. We use a constant step size η, but an increasing momentum parameter $\beta_t \to 1$.

Rate of convergence of Nesterov Accelerated Gradient (NAG) - smooth case

Theorem : NAG rate of convergence - smooth convex functions

If f is L-smooth and convex with a minimum at $\theta_\star$. Then, if for all t, $\beta_{t+1} = \frac{t}{t+3}$, $\eta \leqslant \frac{1}{L}$

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|_2^2}{\eta(t+1)^2}.$$

Observations:

1. We use a constant step size η, but an increasing momentum parameter $\beta_t \to 1$.
2. **Comparison to GD.** With $\eta = \frac{1}{L}$
   - For GD, we had:
   $$f(\theta_t) - f(\theta_\star) \leqslant \frac{L\|\theta_0 - \theta_\star\|_2^2}{t}.$$
   - For NAG, we have:
   $$f(\theta_t) - f(\theta_\star) \leqslant \frac{2L\|\theta_0 - \theta_\star\|_2^2}{(t+1)^2}.$$

Rate of convergence of Nesterov Accelerated Gradient (NAG) - smooth case

Theorem : NAG rate of convergence - smooth convex functions

If f is L-smooth and convex with a minimum at $\theta_\star$. Then, if for all t, $\beta_{t+1} = \frac{t}{t+3}$, $\eta \leqslant \frac{1}{L}$

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{2\|\theta_0 - \theta_\star\|_2^2}{\eta(t+1)^2}.$$

Observations:

1. We use a constant step size η, but an increasing momentum parameter $\beta_t \to 1$.
2. **Comparison to GD.** With $\eta = \frac{1}{L}$
   - For GD, we had:
   $$f(\theta_t) - f(\theta_\star) \leqslant \frac{L\|\theta_0 - \theta_\star\|_2^2}{t}.$$
   - For NAG, we have:
   $$f(\theta_t) - f(\theta_\star) \leqslant \frac{2L\|\theta_0 - \theta_\star\|_2^2}{(t+1)^2}.$$

$\to$ The convergence is much faster!!

Rate of convergence of NAG - smooth strongly-cvx

Theorem : NAG rate of convergence - smooth & strongly-convex functions

If f is L-smooth and μ-strongly- convex with a minimum at $\theta_\star$. Then, if for all t, if

$$\beta_t = \beta = \frac{1-\sqrt{\mu/L}}{1+\sqrt{\mu/L}}, \qquad \eta_t = \eta = \frac{1}{L}$$

we have

$$f(\theta_t) - f(\theta_\star) \leqslant L\left(1-\sqrt{\frac{\mu}{L}}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right).$$

Rate of convergence of NAG - smooth strongly-cvx

Theorem : NAG rate of convergence - smooth & strongly-convex functions

If f is L-smooth and μ-strongly- convex with a minimum at $\theta_\star$. Then, if for all t, if

$$\beta_t = \beta = \frac{1-\sqrt{\mu/L}}{1+\sqrt{\mu/L}}, \qquad \eta_t = \eta = \frac{1}{L}$$

we have

$$f(\theta_t) - f(\theta_\star) \leqslant L\left(1-\sqrt{\frac{\mu}{L}}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right).$$

1. **Comparison to GD.** With $\eta = \frac{1}{L}$, we get resp.
   - For GD, we had:

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{L}{2}\left(1-\frac{\mu}{L}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\kappa}\right)\right).$$

Rate of convergence of NAG - smooth strongly-cvx

Theorem : NAG rate of convergence - smooth & strongly-convex functions

If f is L-smooth and μ-strongly- convex with a minimum at $\theta_\star$. Then, if for all t, if

$$\beta_t = \beta = \frac{1-\sqrt{\mu/L}}{1+\sqrt{\mu/L}}, \qquad \eta_t = \eta = \frac{1}{L}$$

we have

$$f(\theta_t) - f(\theta_\star) \leqslant L\left(1-\sqrt{\frac{\mu}{L}}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right).$$

1. **Comparison to GD.** With $\eta = \frac{1}{L}$, we get resp.
   - For GD, we had:

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{L}{2}\left(1-\frac{\mu}{L}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\kappa}\right)\right).$$

$\rightarrow$ The convergence is much faster: we need $\sqrt{\kappa}$ times as less iterations to achieve the same upper bound!

2. Recall that in ML, κ can be of the order of 10^6 to 10^9 $\rightarrow$ acceleration can reduce the number of necessary iterations by a factor $1000+$.

Rate of convergence of NAG - smooth strongly-cvx

Theorem : NAG rate of convergence - smooth & strongly-convex functions

If f is L-smooth and μ-strongly- convex with a minimum at $\theta_\star$. Then, if for all t, if

$$\beta_t = \beta = \frac{1-\sqrt{\mu/L}}{1+\sqrt{\mu/L}}, \qquad \eta_t = \eta = \frac{1}{L}$$

we have

$$f(\theta_t) - f(\theta_\star) \leqslant L\left(1-\sqrt{\frac{\mu}{L}}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\sqrt{\kappa}}\right)\right).$$

1. **Comparison to GD.** With $\eta = \frac{1}{L}$, we get resp.
   - For GD, we had:

$$f(\theta_t) - f(\theta_\star) \leqslant \frac{L}{2}\left(1-\frac{\mu}{L}\right)^t \|\theta_0 - \theta_\star\|_2^2 = O\left(\exp\left(-\frac{t}{\kappa}\right)\right).$$

$\rightarrow$ The convergence is much faster: we need $\sqrt{\kappa}$ times as less iterations to achieve the same upper bound!

2. Recall that in ML, κ can be of the order of 10^6 to 10^9 $\rightarrow$ acceleration can reduce the number of necessary iterations by a factor $1000+$.

Proposition (informal): Actually, NAG achieves the *optimal* rate for first order methods on smooth and (strongly)-convex functions.

Meaning that no other method using only first-order information can perform significantly faster (more than a constant times) on all functions of the class under consideration

Theoretical results, Heavy ball method

For HB method, the results are more complicated.

Positive results:

1. It achieves acceleration (similar to NAG) if we consider only *quadratic* functions.
2. It is then even *twice* faster than NAG.

Theoretical results, Heavy ball method

For HB method, the results are more complicated.

Positive results:

1. It achieves acceleration (similar to NAG) if we consider only *quadratic* functions.
2. It is then even *twice* faster than NAG.

Negative results:

1. It is known to have pathological behaviors on non-quadratic functions.
2. Especially, it fails to systematically *accelerate* on this class.

Theoretical results, Heavy ball method

For HB method, the results are more complicated.

Positive results:

1. It achieves acceleration (similar to NAG) if we consider only *quadratic* functions.
2. It is then even *twice* faster than NAG.

Negative results:

1. It is known to have pathological behaviors on non-quadratic functions.
2. Especially, it fails to systematically *accelerate* on this class.

Nevertheless, HB is more used in practice, especially in deep-learning contexts, where it is used by default, and using NAG requires to change the default setting.

Tuning of β

The momentum parameter $\beta \in [0; 1]$ defines the strength of the acceleration.

1. $\beta = 0$: no acceleration, we get GD.
2. $\beta = 1$: divergence (no friction).

The good value is thus in between, typically close to 1.

- The two theorems before give a theoretically grounded tuning for particular class of functions.
- In practice, or if μ, L are unknown, difficult question. Often use first a default value of 0.9, or 0.99.

Summary: Accelerated methods - NAG and HB.

First-order method

Deterministic
automatic adaptation for stochastic.

Accelerated: use a *momentum term*.

Intuition & motivation:
Challenge: GD suffers from bad conditioning, local minima
→ Solution: use *acceleration*
If we make consistent progress in one direction, accelerate in that direction.

Update:

$$\theta_t \overset{\text{NAG}}{\leftarrow} \theta_{t-1} - \eta \nabla f(\theta_{t-1}) + \beta(\theta_{t-1} - \theta_{t-2}).$$

$$\theta_t \overset{\text{HB}}{\leftarrow} \theta_{t-1} - \eta \nabla f(\zeta_{t-1}) + \beta(\theta_{t-1} - \theta_{t-2}).$$

Two strategies: only diff. is the point at which the gradient is taken

- HB: used more often in practice, optimal theoretically on quadratics
- NAG: optimal theoretically on smooth and (strongly) convex functions.

Hyperparameters: η, T, β,
$\rightsquigarrow \eta$: same tradeoffs as GD.
$\rightsquigarrow \beta$ theory or 0.9, 0.99

Complexity/iter for ERM

- nd if deterministic
- d if stochastic.

Theory:

- Rate for smooth and strongly convex functions for NAG
- NAG is "optimal" among first order method
- Impact of bad conditioning (κ large) reduced.

Take-aways:
→ Acceleration/momentum is fundamental to improve convergence, for poorly condition functions, on flat regions, saddle points, and to escape local minima.
→ It is systematically used in DL.

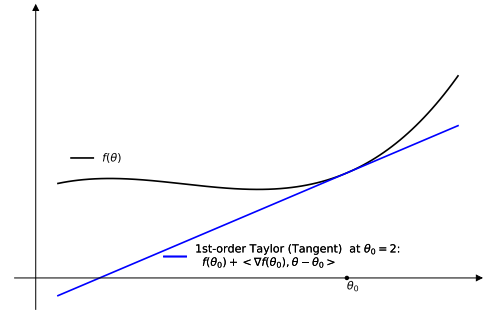
Let's recap! Wooclap quizz!

Outline

Intuition - Second order methods - Newton method

Reminder - Proposition

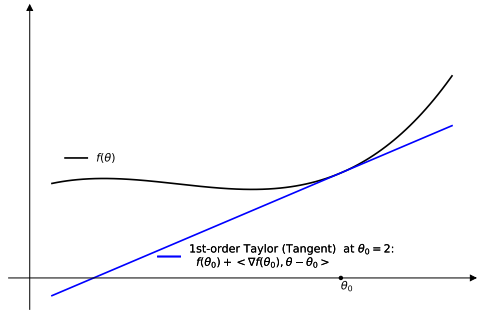
$$\underset{\theta \in B(\theta_0, R)}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle}_{\text{First-order approximation}} = \underbrace{\theta_0 - \frac{R}{\|\nabla f(\theta_0)\|} \nabla f(\theta_0)}_{\text{GD!}}$$



Intuition - Second order methods - Newton method

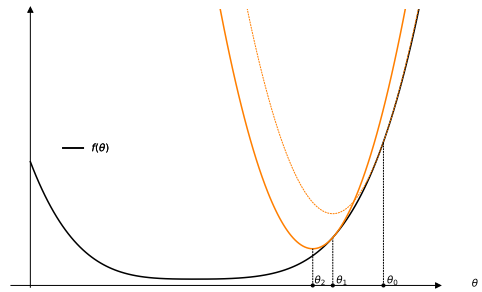
Reminder - Proposition

$$\underset{\theta \in B(\theta_0, R)}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle}_{\text{First-order approximation}} = \underbrace{\theta_0 - \frac{R}{\|\nabla f(\theta_0)\|} \nabla f(\theta_0)}_{\text{GD!}}$$



Reminder - Proposition

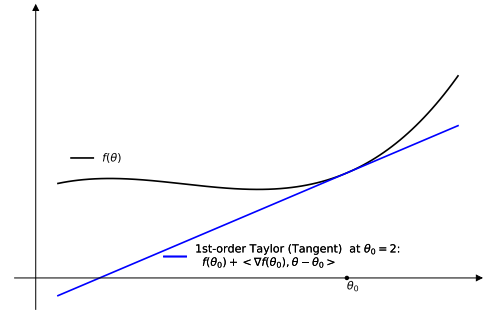
$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{L}{2} \|\theta - \theta_0\|^2}_{\text{Quadratic upper bound}} = \underbrace{\theta_0 - \frac{1}{L} \nabla f(\theta_0)}_{\text{GD!}}$$



Intuition - Second order methods - Newton method

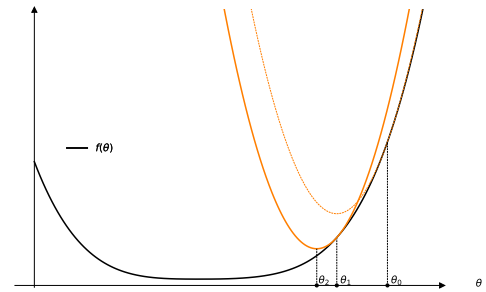
Reminder - Proposition

$$\underset{\theta \in B(\theta_0, R)}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle}_{\text{First-order approximation}} = \underbrace{\theta_0 - \frac{R}{\|\nabla f(\theta_0)\|} \nabla f(\theta_0)}_{\text{GD!}}$$



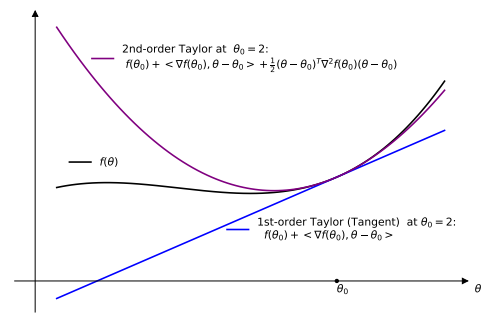
Reminder - Proposition

$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{L}{2} \|\theta - \theta_0\|^2}_{\text{Quadratic upper bound}} = \underbrace{\theta_0 - \frac{1}{L} \nabla f(\theta_0)}_{\text{GD!}}$$



💡 **Newton**: global minimization Second-order Taylor approximation

$$\underbrace{\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2} (\theta - \theta_0)^\top \nabla^2 f(\theta_0) (\theta - \theta_0)}_{\text{Second-order approximation}}}_{\textbf{Newton update}}$$



→ **Second order method:** requires to know the Hessian matrix $\nabla^2 f(\theta_0)$

Newton algorithm - closed form solution

- The Newton's direction minimizes the best locally quadratic approximation of f.
- For a point θ_0, such that $\nabla^2 f(\theta_0) \succ 0$

$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation} =: h_{\theta_0}(\theta)} = \underbrace{\theta_0 - [\nabla^2 f(\theta_0)]^{-1} \nabla f(\theta_0)}_{\textbf{Newton update}}$$

Newton algorithm - closed form solution

- The Newton's direction minimizes the best locally quadratic approximation of f.
- For a point θ_0, such that $\nabla^2 f(\theta_0) \succ 0$

$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation} =: h_{\theta_0}(\theta)} = \underbrace{\theta_0 - [\nabla^2 f(\theta_0)]^{-1} \nabla f(\theta_0)}_{\textbf{Newton update}}$$

Proof:

$$\nabla h_{\theta_0}(\theta) = \nabla f(\theta_0) + \nabla^2 f(\theta_0)\ (\theta - \theta_0)$$

$$\nabla h_{\theta_0}(\theta) = 0 \overset{\nabla^2 f(\theta_0) \text{invertible}}{\Leftrightarrow} \theta - \theta_0 = -(\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$

$$\Leftrightarrow \theta = \theta_0 - (\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$

Newton algorithm - closed form solution

- The Newton's direction minimizes the best locally quadratic approximation of f.
- For a point θ_0, such that $\nabla^2 f(\theta_0) \succ 0$

$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation} =: h_{\theta_0}(\theta)} = \underbrace{\theta_0 - [\nabla^2 f(\theta_0)]^{-1} \nabla f(\theta_0)}_{\textbf{Newton update}}$$

Proof:

$$\nabla h_{\theta_0}(\theta) = \nabla f(\theta_0) + \nabla^2 f(\theta_0)\ (\theta - \theta_0)$$

$$\nabla h_{\theta_0}(\theta) = 0 \overset{\nabla^2 f(\theta_0) \text{invertible}}{\Leftrightarrow} \theta - \theta_0 = -(\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$

$$\Leftrightarrow \theta = \theta_0 - (\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$

Critical point and $\forall \theta$, $\nabla^2 h_{\theta_0}(\theta) = \nabla^2 f(\theta_0) \succ 0 \Rightarrow$ global minimum

Newton algorithm - closed form solution

- The Newton's direction minimizes the best locally quadratic approximation of f.
- For a point θ_0, such that $\nabla^2 f(\theta_0) \succ 0$

$$\underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation} =: h_{\theta_0}(\theta)} = \underbrace{\theta_0 - [\nabla^2 f(\theta_0)]^{-1} \nabla f(\theta_0)}_{\textbf{Newton update}}$$

Proof:

$$\nabla h_{\theta_0}(\theta) = \nabla f(\theta_0) + \nabla^2 f(\theta_0)\ (\theta - \theta_0)$$

$$\nabla h_{\theta_0}(\theta) = 0 \overset{\nabla^2 f(\theta_0) \text{invertible}}{\Leftrightarrow} \theta - \theta_0 = -(\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$
$$\Leftrightarrow \theta = \theta_0 - (\nabla^2 f(\theta_0))^{-1} \nabla f(\theta_0)$$

Critical point and $\forall \theta$, $\nabla^2 h_{\theta_0}(\theta) = \nabla^2 f(\theta_0) \succ 0 \Rightarrow$ global minimum

Algorithm : Newton Descent.

Input: Twice differentiable function f to minimize.
Initialization: initial weight vector θ_0
Hyper-parameters: number of iterations T.
For $t \in \{1, \dots, T\}$:

- $\theta_{t+1} \leftarrow \theta_t - (\nabla^2 f(\theta_t))^{-1} \nabla f(\theta_t)$

Output: θ_T.

Visualization

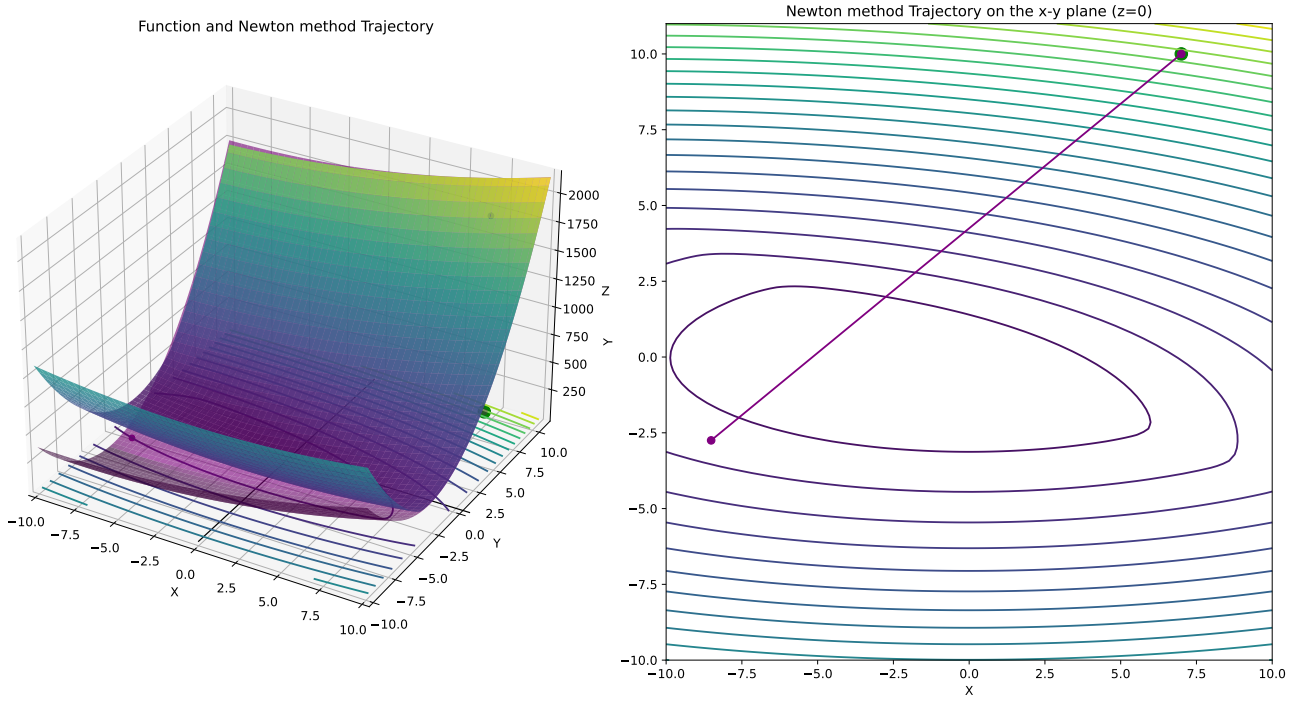


Figure: Newton method, step $T = 1$

Visualization

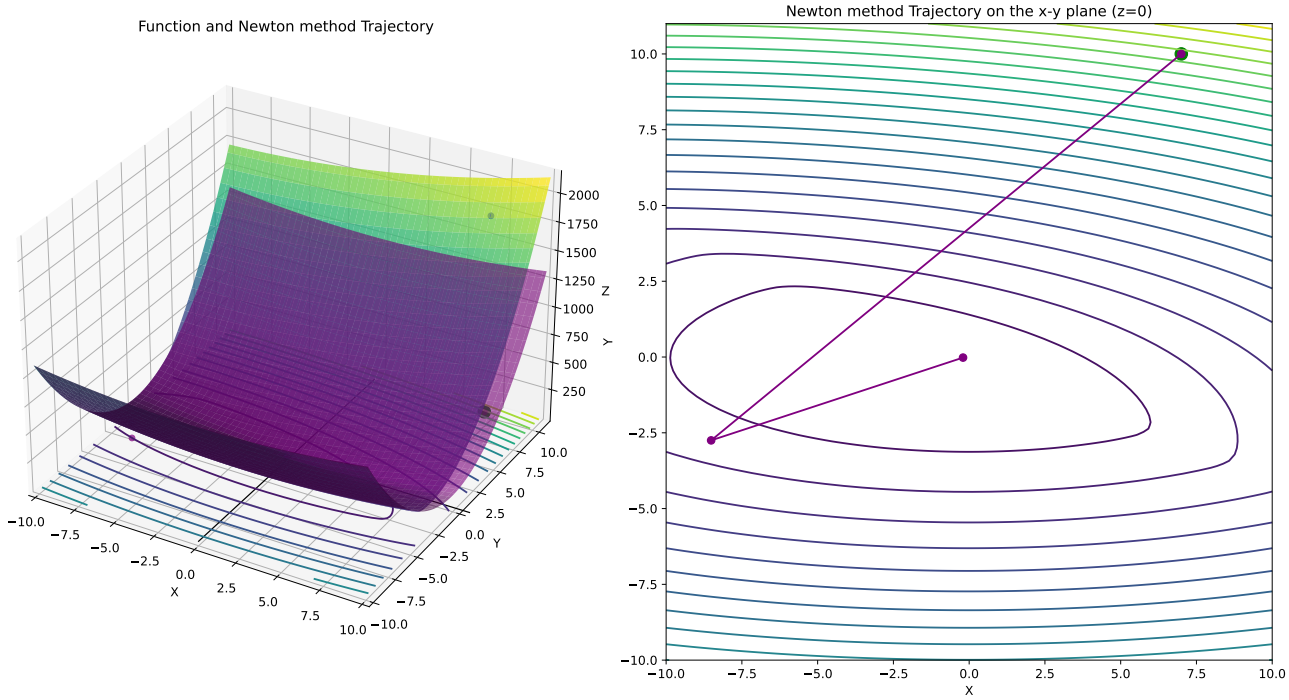


Figure: Newton method, step $T = 2$

Visualization

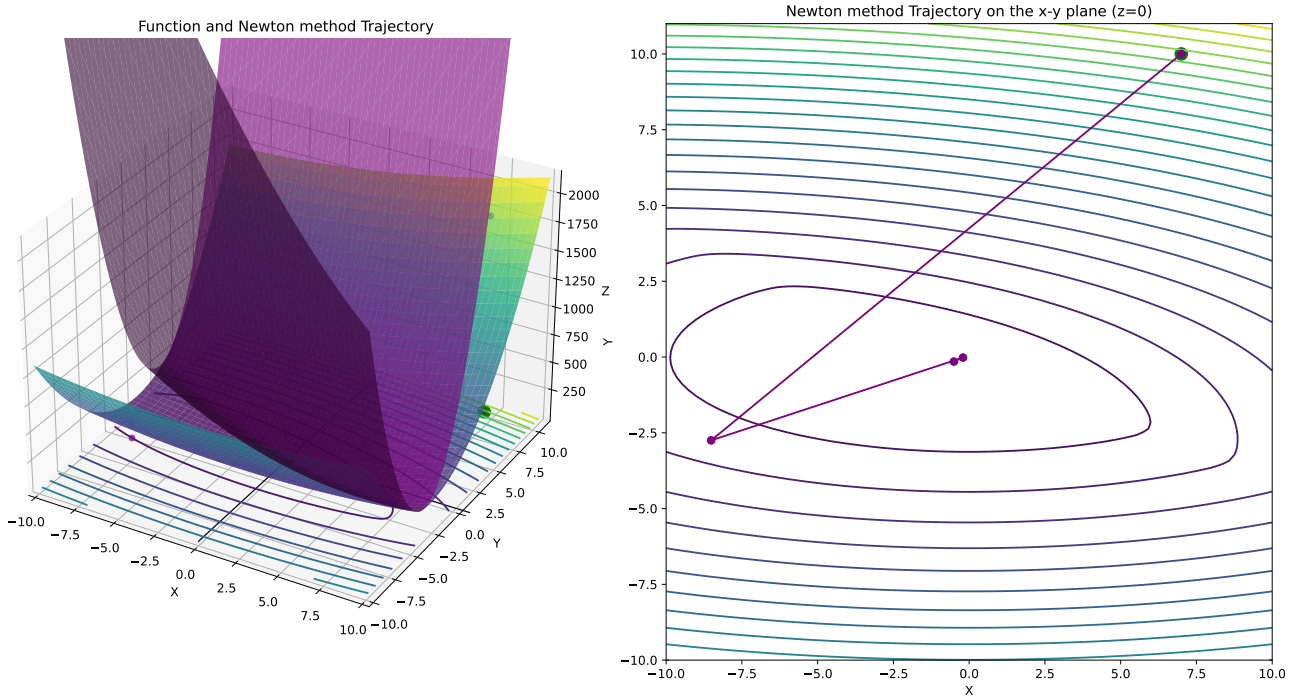


Figure: Newton method, step $T = 3$

Visualization

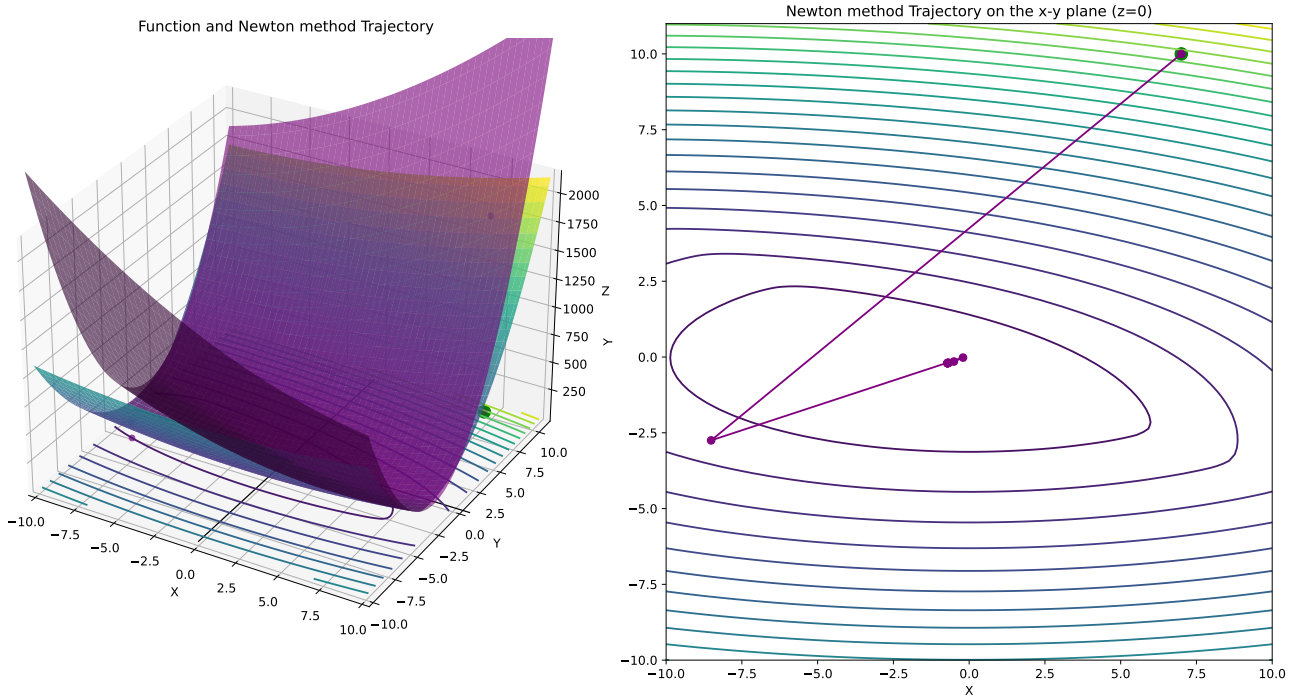
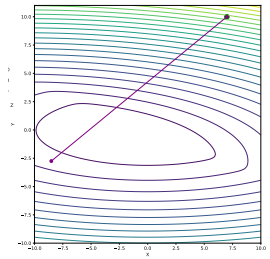
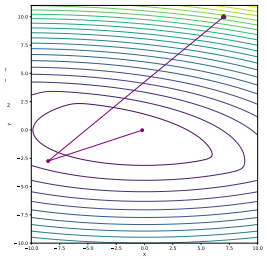
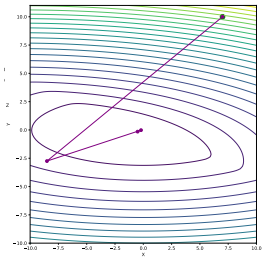
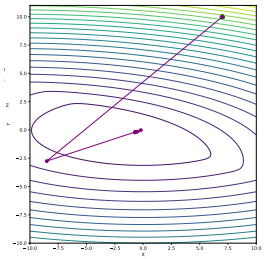
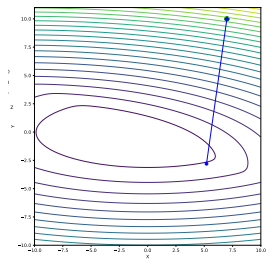
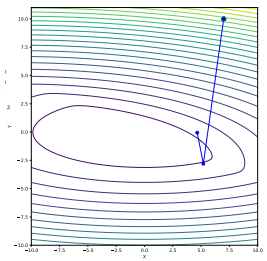
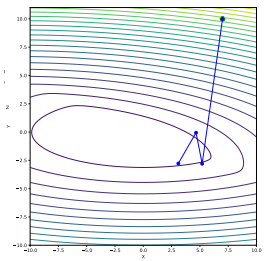
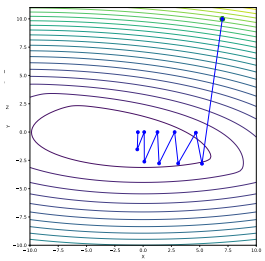
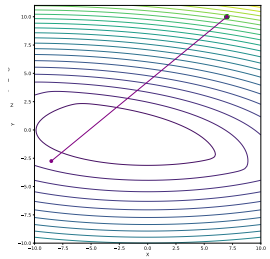
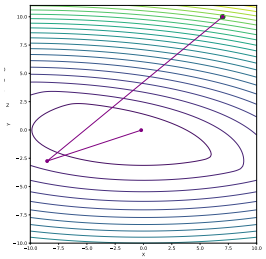
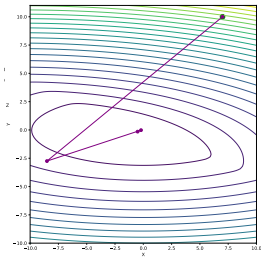
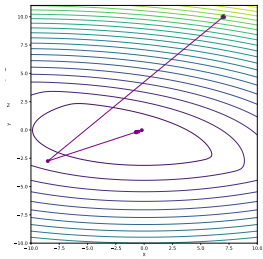
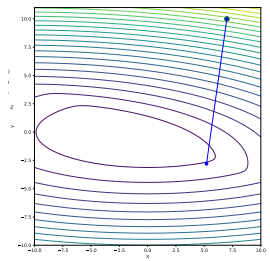
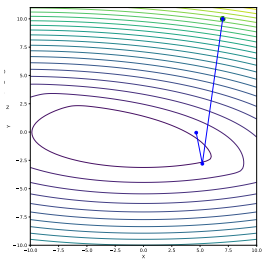
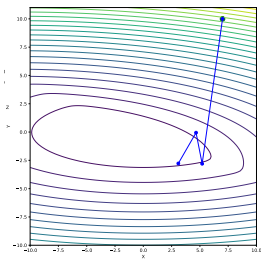
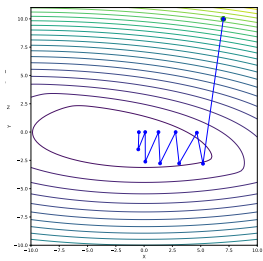


Figure: Newton method, step $T = 10$

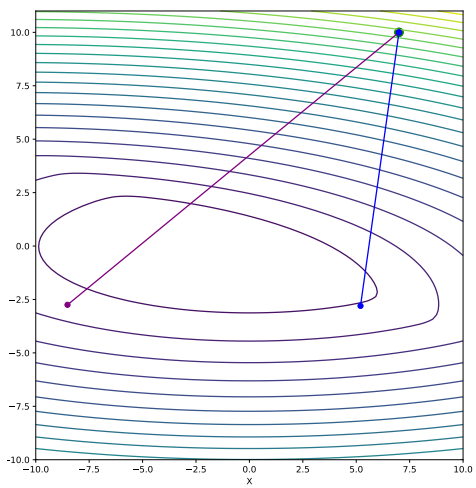
Comparison to GD

	$T = 1$	$T = 2$	$T = 3$	$T = 10$	
Newton					→ very fast convergence
GD					→ much slower

Comparison to GD

	$T = 1$	$T = 2$	$T = 3$	$T = 10$	
Newton					→ very fast convergence
GD					→ much slower

First step:



1. Gradient is orthogonal to the level set:

$$\theta_{t+1} \leftarrow \theta_t - \underbrace{\frac{1}{L}}_{\text{step}} \nabla f(\theta_t)$$

2. Newton step takes the curvature into account: "sees also how the gradient varies locally".

$$\theta_{t+1} \leftarrow \theta_t - \underbrace{\left(\nabla^2 f(\theta_t)\right)^{-1}}_{\text{pseudo-step}} \nabla f(\theta_t)$$

Theoretical result - Newton on quadratics

Case of quadratic functions.

Reminder: if there exists a matrix $A \in \mathbb{R}^{d \times d}$, $b \in \mathbb{R}^d$ and $c \in \mathbb{R}$ such that

$$f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$$

then, for all $\theta \in \mathbb{R}^d$, then the second-order Taylor approximation is **exact** at all points: for all θ, θ_0

$$f(\theta) = \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}}.$$

Theoretical result - Newton on quadratics

Case of quadratic functions.

Reminder: if there exists a matrix $A \in \mathbb{R}^{d \times d}$, $b \in \mathbb{R}^d$ and $c \in \mathbb{R}$ such that

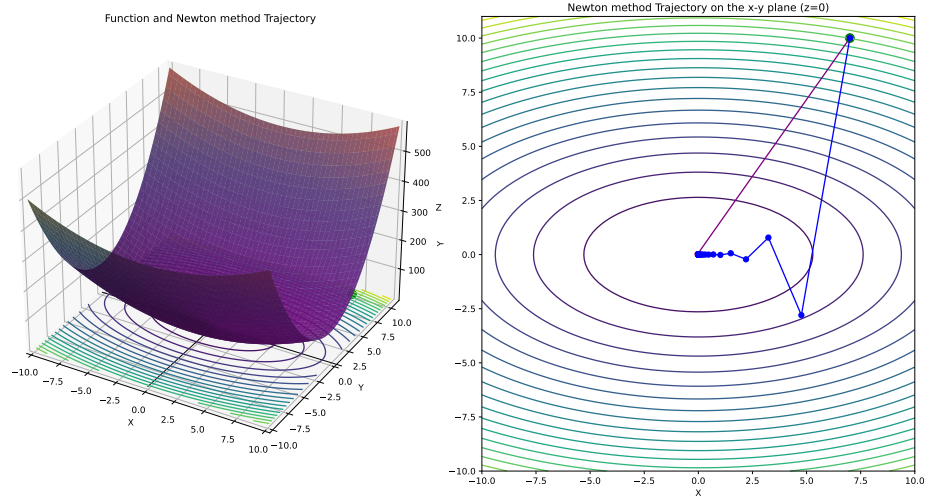
$$f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$$

then, for all $\theta \in \mathbb{R}^d$, then the second-order Taylor approximation is **exact** at all points: for all θ, θ_0

$$f(\theta) = \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}}.$$

💡 **Newton method converges after one step:** for any starting point θ_0,

$$\theta_1 = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}} = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} f(\theta) = \theta_\star$$



Theoretical result - Newton on quadratics

Case of quadratic functions.

Reminder: if there exists a matrix $A \in \mathbb{R}^{d \times d}$, $b \in \mathbb{R}^d$ and $c \in \mathbb{R}$ such that

$$f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$$

then, for all $\theta \in \mathbb{R}^d$, then the second-order Taylor approximation is **exact** at all points: for all θ, θ_0

$$f(\theta) = \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}}.$$

Newton method converges after one step: for any starting point θ_0,

$$\theta_1 = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}} = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} f(\theta) = \theta_\star$$

$$= \theta_0 - \underbrace{A^{-1}}_{(\nabla^2 f(\theta_0))^{-1}} \underbrace{A(\theta_0 - \theta_\star)}_{\nabla f(\theta_0)} = \theta_\star$$

Theoretical result - Newton on quadratics

Case of quadratic functions.

Reminder: if there exists a matrix $A \in \mathbb{R}^{d \times d}$, $b \in \mathbb{R}^d$ and $c \in \mathbb{R}$ such that

$$f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$$

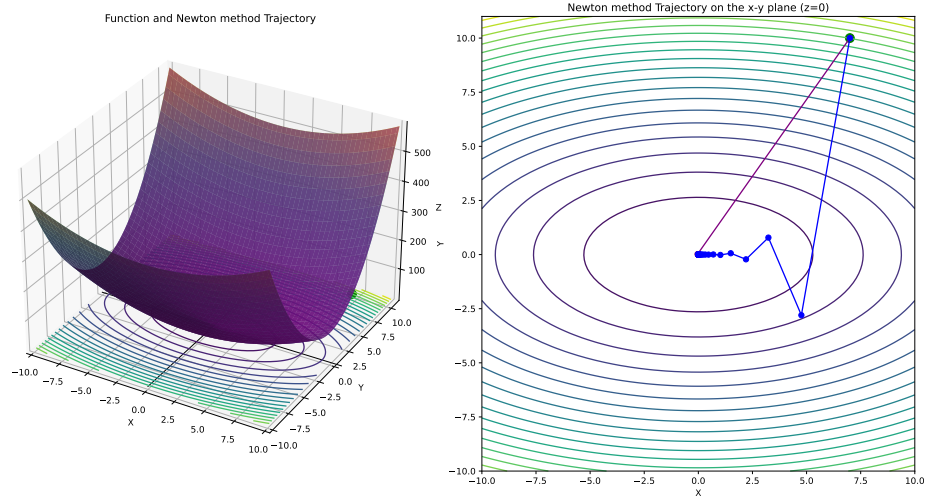
then, for all $\theta \in \mathbb{R}^d$, then the second-order Taylor approximation is **exact** at all points: for all θ, θ_0

$$f(\theta) = \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}}.$$

Newton method converges after one step: for any starting point θ_0,

$$\theta_1 = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \underbrace{f(\theta_0) + \langle \nabla f(\theta_0), \theta - \theta_0 \rangle + \frac{1}{2}(\theta - \theta_0)^\top \nabla^2 f(\theta_0)\ (\theta - \theta_0)}_{\text{Second-order approximation}} = \operatorname*{argmin}_{\theta \in \mathbb{R}^d} f(\theta) = \theta_\star$$

$$= \theta_0 - \underbrace{A^{-1}}_{(\nabla^2 f(\theta_0))^{-1}} \underbrace{A(\theta_0 - \theta_\star)}_{\nabla f(\theta_0)} = \theta_\star$$



Theoretical result - Newton on smooth-convex functions

1. **Conditioning:** Newton's method is not affected by a problem's conditioning κ .

2. **Converges extremely fast once a neighborhood has been reached**:

Proposition : Newton method – local convergence rate

If θ_0 is "close enough" to $\theta_\star$ (in a certain sense), and f is L-smooth, μ-strongly convex and $\nabla^2 f$ is M Lipschitz, then,

$$f(\theta_t) - f_\star \leqslant C_{\mu,L,M} \left(\frac{1}{2}\right)^{2^t} .$$

Theoretical result - Newton on smooth-convex functions

1. **Conditioning:** Newton's method is not affected by a problem's conditioning κ .

2. **Converges extremely fast once a neighborhood has been reached**:

Proposition : Newton method – local convergence rate

If θ_0 is "close enough" to $\theta_\star$ (in a certain sense), and f is L-smooth, μ-strongly convex and $\nabla^2 f$ is M Lipschitz, then,

$$f(\theta_t) - f_\star \leqslant C_{\mu,L,M} \left(\frac{1}{2}\right)^{2^t} .$$

- $\rightsquigarrow$ error is **squared** at each step!
  Once a neighborhood of the solution has been reached, the excess risk decays at a quadratic rate: typically

Theoretical result - Newton on smooth-convex functions

1. **Conditioning:** Newton's method is not affected by a problem's conditioning κ .

2. **Converges extremely fast once a neighborhood has been reached**:

Proposition : Newton method – local convergence rate

If θ_0 is "close enough" to $\theta_\star$ (in a certain sense), and f is L-smooth, μ-strongly convex and $\nabla^2 f$ is M Lipschitz, then,

$$f(\theta_t) - f_\star \leqslant C_{\mu,L,M} \left(\frac{1}{2}\right)^{2^t} .$$

- $\rightsquigarrow$ error is **squared** at each step!
  Once a neighborhood of the solution has been reached, the excess risk decays at a quadratic rate: typically

iteration	1	2	3	4	5	...
error	10^{-1}	10^{-2}	10^{-4}	10^{-8}	10^{-16}	...

- Machine precision is then obtained in a few steps.

Newton: drawbacks

1. ⚠ Complexity: each iteration requires $O(d^3)$ operations (solving a $d \times d$-linear system).
2. ⚠ Memory: each iteration requires $O(d^2)$ (Hessian size) storage.
   $\rightsquigarrow$ may simply be prohibitive in ML setups.

Newton: drawbacks

1. ⚠ Complexity: each iteration requires $O(d^3)$ operations (solving a $d \times d$-linear system).
2. ⚠ Memory: each iteration requires $O(d^2)$ (Hessian size) storage.
   $\rightsquigarrow$ may simply be prohibitive in ML setups.

3. Requires the function to be twice differentiable.
4. Requires the Hessian to be invertible (numerical problems may arise otherwise)

5. Requires convexity:
   $\rightsquigarrow$ ⚠ The sequence of iterates is the same if we change f into $-f$.
   $\rightsquigarrow$ ⚠ Newton method cannot distinguish minima and maxima!

Newton: drawbacks

1. ⚠ Complexity: each iteration requires $O(d^3)$ operations (solving a $d \times d$-linear system).
2. ⚠ Memory: each iteration requires $O(d^2)$ (Hessian size) storage.
   $\rightsquigarrow$ may simply be prohibitive in ML setups.

3. Requires the function to be twice differentiable.
4. Requires the Hessian to be invertible (numerical problems may arise otherwise)

5. Requires convexity:
   $\rightsquigarrow$ ⚠ The sequence of iterates is the same if we change f into $-f$.
   $\rightsquigarrow$ ⚠ Newton method cannot distinguish minima and maxima!

6. May diverge: no global; convergence is ensured.
   $\rightsquigarrow$ solutions exist damped-Newton, cubic-regularization, etc.

Newton: drawbacks

1. ⚠ Complexity: each iteration requires $O(d^3)$ operations (solving a $d \times d$-linear system).
2. ⚠ Memory: each iteration requires $O(d^2)$ (Hessian size) storage.
   $\rightsquigarrow$ may simply be prohibitive in ML setups.

3. Requires the function to be twice differentiable.
4. Requires the Hessian to be invertible (numerical problems may arise otherwise)

5. Requires convexity:
   $\rightsquigarrow$ ⚠ The sequence of iterates is the same if we change f into $-f$.
   $\rightsquigarrow$ ⚠ Newton method cannot distinguish minima and maxima!

6. May diverge: no global; convergence is ensured.
   $\rightsquigarrow$ solutions exist damped-Newton, cubic-regularization, etc.

💡 Extremely high precision, at a very high cost per iteration, only for convex smooth problems.
$\rightarrow$ only useful in specific ML cases.

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in R$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in R$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t,

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.
   - Remark: $\forall \zeta$ in a neighborhood of θ_t:

$$A_t(\zeta - \theta_t) = \nabla^2 f(\theta_t)(\zeta - \theta_t) \simeq \nabla f(\zeta) - \nabla f(\theta_t)$$

$$\text{in particular} \quad A_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} \simeq \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in R$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t,

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.
   - Remark: $\forall \zeta$ in a neighborhood of θ_t:

$$A_t(\zeta - \theta_t) = \nabla^2 f(\theta_t)(\zeta - \theta_t) \simeq \nabla f(\zeta) - \nabla f(\theta_t)$$

$$\text{in particular} \quad A_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} \simeq \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in R$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Quasi-Newton methods

Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - A_{t-1}^{-1} \nabla f(\theta_{t-1})$,
2. with $A_{t-1} = \nabla^2 f(\theta_{t-1})$
   - Complexity: $O(d^3)$ per iteration.
   - Remark: $\forall \zeta$ in a neighborhood of θ_t:

$$A_t(\zeta - \theta_t) = \nabla^2 f(\theta_t)(\zeta - \theta_t) \simeq \nabla f(\zeta) - \nabla f(\theta_t)$$

$$\text{in particular} \quad A_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} \simeq \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Quasi-Newton method:

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in R$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

💡 In quasi-Newton's methods, the Newton direction is approximated by using only first order information (gradient)

→ ⚠ **QN methods are first-order methods, (but with omplexity d^2)**

💡 Successive iterates and gradients yield second order information

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}.$$

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}}_{\text{rank-1 matrix}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}_{\text{rank-1 matrix}}.$$

✓ B_t is a rank-2 update.

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}}_{\text{rank-1 matrix}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}_{\text{rank-1 matrix}}.$$

- ✓ B_t is a rank-2 update.
- ✓ We have the "key property":

$$B_t(\Delta\theta)_t = B_{t-1}(\Delta\theta)_t - \frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}(\Delta\theta)_t}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t} + \frac{(\Delta G)_t (\Delta G)_t^T (\Delta\theta)_t}{(\Delta G)_t^T (\Delta\theta)_t}$$

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}}_{\text{rank-1 matrix}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}_{\text{rank-1 matrix}}.$$

- ✓ B_t is a rank-2 update.
- ✓ We have the "key property":

$$\begin{aligned} B_t(\Delta\theta)_t &= B_{t-1}(\Delta\theta)_t - \frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}(\Delta\theta)_t}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t} + \frac{(\Delta G)_t (\Delta G)_t^T (\Delta\theta)_t}{(\Delta G)_t^T (\Delta\theta)_t} \\ &= B_{t-1}(\Delta\theta)_t - B_{t-1}(\Delta\theta)_t + (\Delta G)_t \end{aligned}$$

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. $B_{t-1} \rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}}_{\text{rank-1 matrix}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}_{\text{rank-1 matrix}}.$$

- ✓ B_t is a rank-2 update.
- ✓ We have the "key property":

$$\begin{aligned} B_t(\Delta\theta)_t &= B_{t-1}(\Delta\theta)_t - \frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}(\Delta\theta)_t}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t} + \frac{(\Delta G)_t (\Delta G)_t^T (\Delta\theta)_t}{(\Delta G)_t^T (\Delta\theta)_t} \\ &= B_{t-1}(\Delta\theta)_t - B_{t-1}(\Delta\theta)_t + (\Delta G)_t \\ &= (\Delta G)_t \end{aligned}$$

Quasi-Newton's methods: (BFGS) algorithm

1. $\theta_t \leftarrow \theta_{t-1} - \eta_{t-1} B_{t-1}^{-1} \nabla f(\theta_{t-1})$
2. $\eta_{t-1} \in \mathbb{R}$ (line-search), B_{t-1} to be chosen such that
   - B_t is a rank-1/rank-2 update w.r.t. B_{t-1} $\rightarrow$ complexity: $O(d^2)$ per iteration.
   - B_t "mimics" A_t, by ensuring "key property"

$$B_t \underbrace{(\theta_t - \theta_{t-1})}_{\text{update direction } (\Delta\theta)_t} = \underbrace{\nabla f(\theta_t) - \nabla f(\theta_{t-1})}_{\text{gradient difference } (\Delta G)_t}$$

Example: Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm

$$B_t = B_{t-1} - \underbrace{\frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}}{(\Delta\theta)_t^T B_{t-1} (\Delta\theta)_t}}_{\text{rank-1 matrix}} + \underbrace{\frac{(\Delta G)_t (\Delta G)_t^T}{(\Delta G)_t^T (\Delta\theta)_t}}_{\text{rank-1 matrix}}.$$

- ✓ B_t is a rank-2 update.
- ✓ We have the "key property":

$$\begin{aligned} B_t(\Delta\theta)_t &= B_{t-1}(\Delta\theta)_t - \frac{B_{t-1}(\Delta\theta)_t (\Delta\theta)_t^T B_{t-1}(\Delta\theta)_t}{(\Delta\theta)_t^T B_{t-1}(\Delta\theta)_t} + \frac{(\Delta G)_t (\Delta G)_t^T (\Delta\theta)_t}{(\Delta G)_t^T (\Delta\theta)_t} \\ &= B_{t-1}(\Delta\theta)_t - B_{t-1}(\Delta\theta)_t + (\Delta G)_t \\ &= (\Delta G)_t \end{aligned}$$

Considered as the state-of-the-art quasi-Newton's algorithm!
Extremely efficient in practice for smooth convex optimization! Often the default method!
Extension: **L-BFGS** reduces per-iteration complexity to Md, M extra hyper-parameter.

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.

⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

*extensions to stochastic case exist, see e.g., Stochastic-L-BFGS

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.
⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

*extensions to stochastic case exist, see e.g., Stochastic-L-BFGS

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.
⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

solver : *{'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'}, default='lbfgs'*

Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited
- 'newton-cholesky' is a good choice for `n_samples` >> `n_features`, especially with one-hot encoded categorical features with rare categories. Note that it is limited to binary classification and the one-versus-rest reduction for multiclass classification. Be aware that the memory usage of this solver has a quadratic dependency on `n_features` because it explicitly computes the Hessian matrix.

Warning: The choice of the algorithm depends on the penalty chosen. Supported penalties by solver:

- 'lbfgs' - ['l2', None]
- 'liblinear' - ['l1', 'l2']
- 'newton-cg' - ['l2', None]
- 'newton-cholesky' - ['l2', None]
- 'sag' - ['l2', None]
- 'saga' - ['elasticnet', 'l1', 'l2', None]

Figure: Logistic-regression solvers - sklearn documentation - link

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.
⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

solver : {'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'}, default='lbfgs'

Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

Fastest in most situations

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited
- 'newton-cholesky' is a good choice for `n_samples` >> `n_features`, especially with one-hot encoded categorical features with rare categories. Note that it is limited to binary classification and the one-versus-rest reduction for multiclass classification. Be aware that the memory usage of this solver has a quadratic dependency on `n_features` because it explicitly computes the Hessian matrix.

Warning: The choice of the algorithm depends on the penalty chosen. Supported penalties by solver:

- 'lbfgs' - ['l2', None]
- 'liblinear' - ['l1', 'l2']
- 'newton-cg' - ['l2', None]
- 'newton-cholesky' - ['l2', None]
- 'sag' - ['l2', None]
- 'saga' - ['elasticnet', 'l1', 'l2', None]

Figure: Logistic-regression solvers - sklearn documentation - link

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.

⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

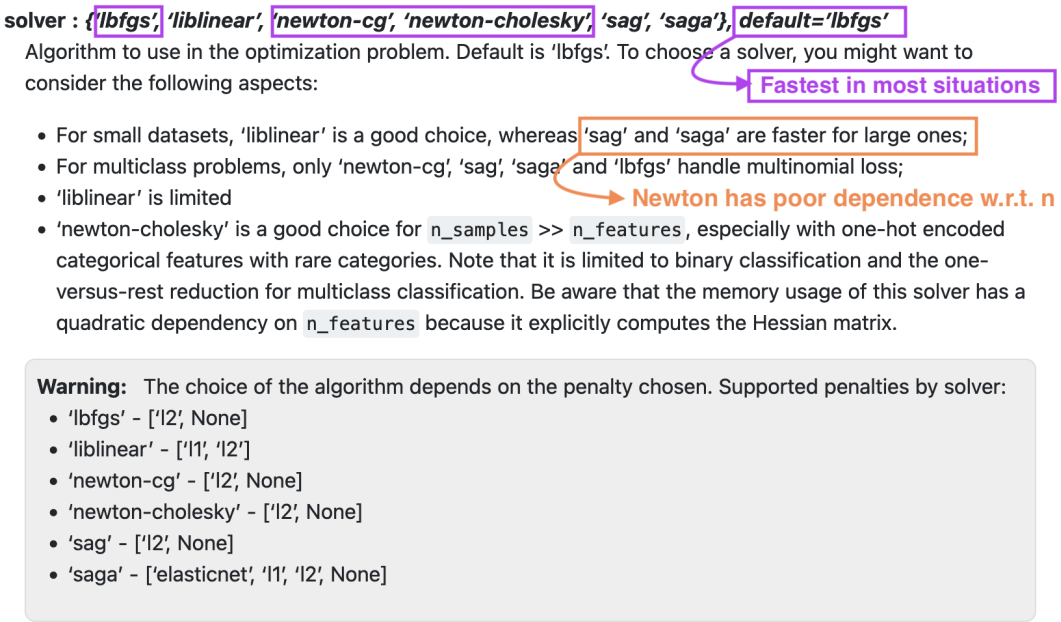


Figure: Logistic-regression solvers - sklearn documentation - link

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.

⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

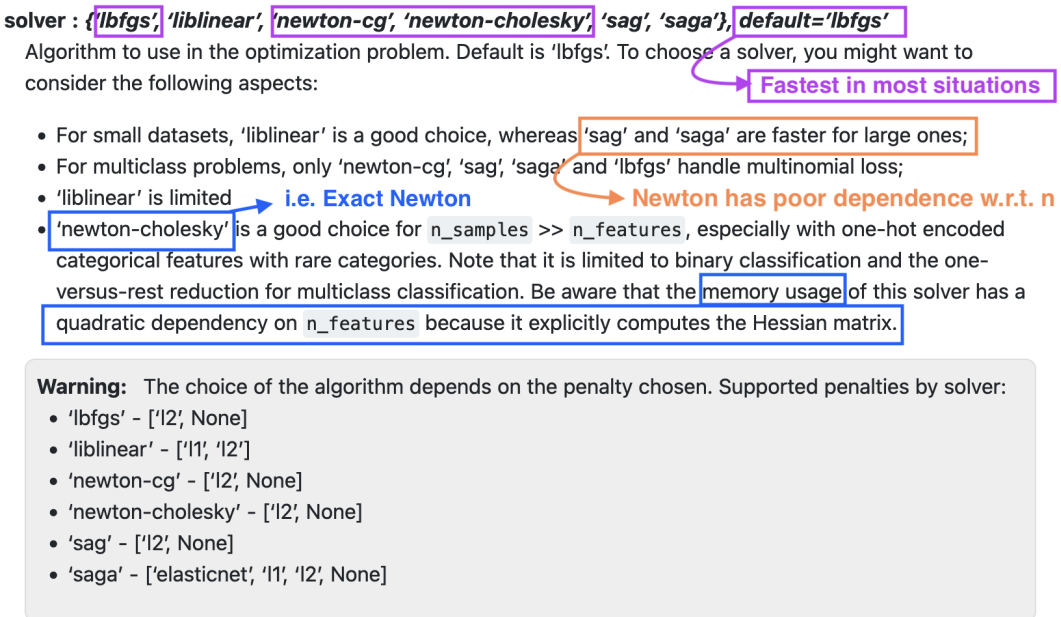


Figure: Logistic-regression solvers - sklearn documentation - link

Newton and L-BFGS in practical applications? Ideal scenario

⚠ Methods above are deterministic.

⚠ ⚠ In supervised ML, computing the Hessian or gradient has a cost proportional to n, typically $O(nd^2 + d^3)$ for Newton.*

- Ideal scenario ? Need convex and smooth problem, n not too large (deterministic method), d not too large for Newton

→ Logistic regression

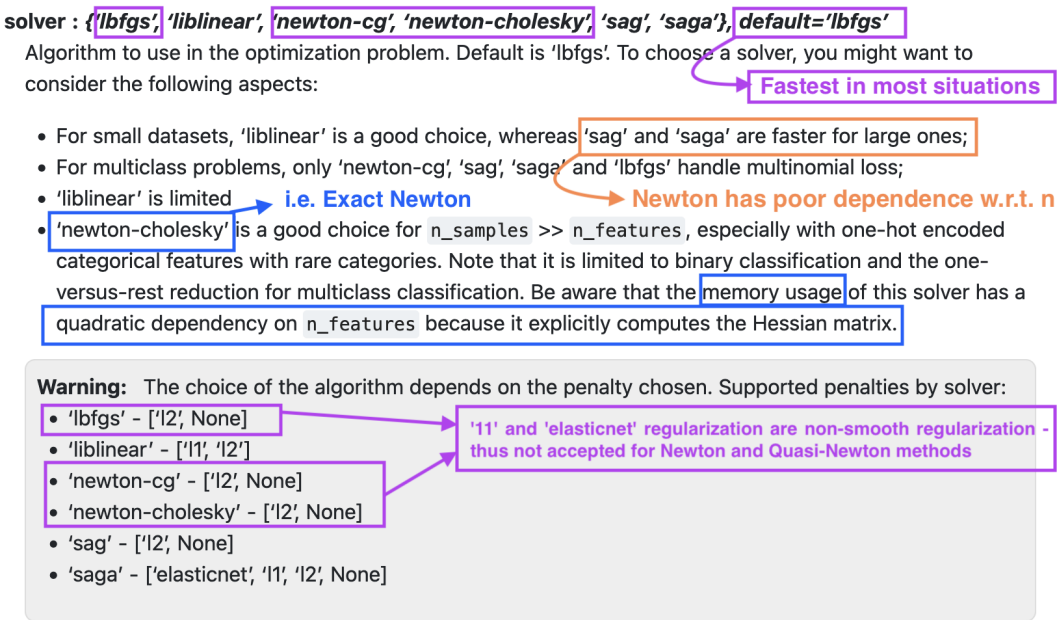


Figure: Logistic-regression solvers - sklearn documentation - link

*extensions to stochastic case exist, see e.g., Stochastic-L-BFGS

Second order methods: Summary

Newton: second-order method
Quasi-Newton: first-order method

Deterministic
(Stochastic extensions)

~~Variance reduction~~
~~Acceleration~~

Newton update
$\theta_{t+1} \leftarrow \theta_t - (\nabla^2 f(\theta_t))^{-1} \nabla f(\theta_t)$

BFGS update
$\theta_{t+1} \leftarrow \theta_t - \eta (B_t)^{-1} \nabla f(\theta_t)$

Hyperparameters:
T
T, η
$\rightsquigarrow \eta$ line search

Complexity/iter in ML:
$O(nd^2 + d^3)$
$O(nd + d^2)$

Intuition & motivation:
Challenge:

- gradient doesn't point in the best direction

→ Solution:

- Use Hessian information
- Minimize 2nd order approx.

Theory:

- Super fast rate for smooth and (strongly) convex functions, **locally**
- One-step convergence for quadratics

Take away:

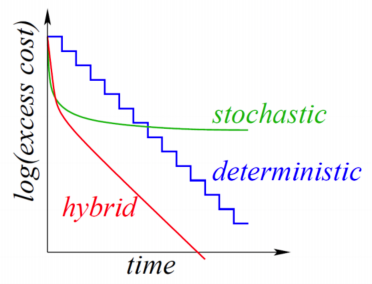
→ (L)-BFGS is SOTA method for convex smooth optimization if

1. n, d are not too large
2. high precision is needed

Outline

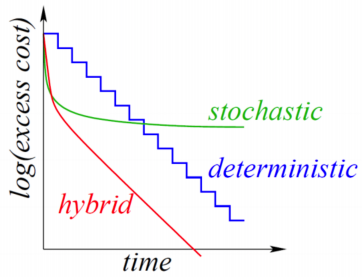
Motivation

1. Can we get the best of both worlds between SGD and GD?



Motivation

1. Can we get the best of both worlds between SGD and GD?



→ Objective of Variance reduced method.

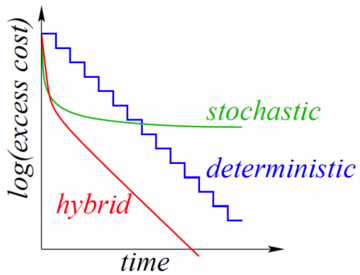
2. How to improve on SGD?

No memory / reusing of information from the past in SGD.

→ Key idea of Variance reduced method.

Motivation

1. Can we get the best of both worlds between SGD and GD?



→ Objective of Variance reduced method.

2. How to improve on SGD?

No memory / reusing of information from the past in SGD.

→ Key idea of Variance reduced method.

Variance reduction methods build on stochastic methods. They can thus ONLY be applied if the objective is a sum, such as

$$\frac{1}{n}\sum_{i=1}^{n} f_i(\theta).$$

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5

✓ $\leftrightarrow \nabla f_i(\theta_t) =$ one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t = 1$	✓	✓	✓	✓	✓

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t = 1$	✓	✓	✓	✓	✓
$t = 2$	✓	✓	✓	✓	✓
$t = 3$	✓	✓	✓	✓	✓
$t = 4$	✓	✓	✓	✓	✓
$t = 5$	✓	✓	✓	✓	✓
$t = 6$	✓	✓	✓	✓	✓

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t = 1$	✓	✓	✓	✓	✓
$t = 2$	✓	✓	✓	✓	✓
$t = 3$	✓	✓	✓	✓	✓
$t = 4$	✓	✓	✓	✓	✓
$t = 5$	✓	✓	✓	✓	✓
$t = 6$	✓	✓	✓	✓	✓

SGD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t = 1$: $i_1 = 3$			✓		
$t = 2$: $i_2 = 1$	✓				

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$	✓	✓	✓	✓	✓
$t=2$	✓	✓	✓	✓	✓
$t=3$	✓	✓	✓	✓	✓
$t=4$	✓	✓	✓	✓	✓
$t=5$	✓	✓	✓	✓	✓
$t=6$	✓	✓	✓	✓	✓

SGD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$: $i_1=3$			✓		
$t=2$: $i_2=1$	✓				
$t=3$: $i_3=4$				✓	
$t=4$: $i_4=1$	✓				
$t=5$: $i_5=5$					✓
$t=6$: $i_6=2$		✓			

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$	✓	✓	✓	✓	✓
$t=2$	✓	✓	✓	✓	✓
$t=3$	✓	✓	✓	✓	✓
$t=4$	✓	✓	✓	✓	✓
$t=5$	✓	✓	✓	✓	✓
$t=6$	✓	✓	✓	✓	✓

SGD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$: $i_1=3$			✓		
$t=2$: $i_2=1$	✓				
$t=3$: $i_3=4$				✓	
$t=4$: $i_4=1$	✓				
$t=5$: $i_5=5$					✓
$t=6$: $i_6=2$		✓			

SAG/SVRG

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=0$	✓	✓	✓	✓	✓
$t=1$: $i_1=3$	∼	∼	✓	∼	∼

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$	✓	✓	✓	✓	✓
$t=2$	✓	✓	✓	✓	✓
$t=3$	✓	✓	✓	✓	✓
$t=4$	✓	✓	✓	✓	✓
$t=5$	✓	✓	✓	✓	✓
$t=6$	✓	✓	✓	✓	✓

SGD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$: $i_1=3$			✓		
$t=2$: $i_2=1$	✓				
$t=3$: $i_3=4$				✓	
$t=4$: $i_4=1$	✓				
$t=5$: $i_5=5$					✓
$t=6$: $i_6=2$		✓			

SAG/SVRG

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=0$	✓	✓	✓	✓	✓
$t=1$: $i_1=3$	∼	∼	✓	∼	∼
$t=2$: $i_2=1$	✓	∼	∼	∼	∼
$t=3$: $i_3=4$	∼	∼	∼	✓	∼
$t=4$: $i_4=1$	✓	∼	∼	∼	∼
$t=5$: $i_5=5$	∼	∼	∼	∼	✓
$t=6$: $i_6=2$	∼	✓	∼	∼	∼

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

Variance reduced methods for finite sum minimization: SAG/SVRG

Objective: $\min \left\{ f(\theta) = \frac{1}{n} \sum_{i=0}^{n} f_i(\theta) \right\}$.

Update: $\theta_{t+1} = \theta_t - \eta_t \underbrace{g_t}_{??}$

GD: at step $t+1$

1. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_t)$

SGD: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$
2. use $g_t = \nabla f_{i_t}(\theta_t)$

SAG/SVRG: at step $t+1$,

1. sample $i_t \sim \mathcal{U}[1; n]$,
2. use $g_t = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t_i})$
3. with $\theta_{t_i} \in \{\theta_1, \dots \theta_t\}$ "one iterate in the past",
4. and $\theta_{t_{i_t}}$ updated to θ_t.

Example: $n = 5$:

$$\frac{1}{5} \sum_{i=1}^{5} f_i(\theta)$$

GD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$	✓	✓	✓	✓	✓
$t=2$	✓	✓	✓	✓	✓
$t=3$	✓	✓	✓	✓	✓
$t=4$	✓	✓	✓	✓	✓
$t=5$	✓	✓	✓	✓	✓
$t=6$	✓	✓	✓	✓	✓

SGD:

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=1$: $i_1=3$			✓		
$t=2$: $i_2=1$	✓				
$t=3$: $i_3=4$				✓	
$t=4$: $i_4=1$	✓				
$t=5$: $i_5=5$					✓
$t=6$: $i_6=2$		✓			

SAG/SVRG

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5
$t=0$	✓	✓	✓	✓	✓
$t=1$: $i_1=3$	∼	∼	✓	∼	∼
$t=2$: $i_2=1$	✓	∼	∼	∼	∼
$t=3$: $i_3=4$	∼	∼	∼	✓	∼
$t=4$: $i_4=1$	✓	∼	∼	∼	∼
$t=5$: $i_5=5$	∼	∼	∼	∼	✓
$t=6$: $i_6=2$	∼	✓	∼	∼	∼

✓ $\leftrightarrow \nabla f_i(\theta_t)$ = one gradient at the current iterate θ_t.

∼ $\leftrightarrow \nabla f_i(\theta_{t_i})$ = one gradient an old iterate θ_{t_i} ($\rightsquigarrow$ difference between SVRG and SAG.)

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t = 0$	✓	✓	✓	✓	✓	θ_0	$\theta_1 \leftarrow \theta_0 - \eta/n \sum$

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t = 0$	✓	✓	✓	✓	✓	θ_0	
$t = 1$: $i_1 = 3$	$\sim$	$\sim$	✓	$\sim$	$\sim$	θ_1	$\theta_2 \leftarrow \theta_1 - \eta/n \sum$ □

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t = 0$	✓	✓	✓	✓	✓	θ_0	
$t = 1$: $i_1 = 3$	∼	∼	✓	∼	∼	θ_1	
$t = 2$: $i_2 = 1$	✓	∼	∼	∼	∼	θ_2	$\theta_3 \leftarrow \theta_2 - \eta/n \sum$ □

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t = 0$	✓	✓	✓	✓	✓	θ_0	
$t = 1$: $i_1 = 3$	$\sim$	$\sim$	✓	$\sim$	$\sim$	θ_1	
$t = 2$: $i_2 = 1$	✓	$\sim$	$\sim$	$\sim$	$\sim$	θ_2	
$t = 3$: $i_3 = 4$	$\sim$	$\sim$	$\sim$	✓	$\sim$	θ_3	
$t = 4$: $i_4 = 1$	✓	$\sim$	$\sim$	$\sim$	$\sim$	θ_4	$\theta_5 \leftarrow \theta_4 - \eta/n \sum$ □

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t = 0$	✓	✓	✓	✓	✓	θ_0	
$t = 1$: $i_1 = 3$	$\sim$	$\sim$	✓	$\sim$	$\sim$	θ_1	
$t = 2$: $i_2 = 1$	✓	$\sim$	$\sim$	$\sim$	$\sim$	θ_2	
$t = 3$: $i_3 = 4$	$\sim$	$\sim$	$\sim$	✓	$\sim$	θ_3	
$t = 4$: $i_4 = 1$	✓	$\sim$	$\sim$	$\sim$	$\sim$	θ_4	
$t = 5$: $i_5 = 5$	$\sim$	$\sim$	$\sim$	$\sim$	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i = 1, \dots, n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i = 1, \dots, n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.
2. Sample $i_t \in \{1, \dots, n\}$ with replacement, and update: $g_{i_t,t} = \nabla f_{i_t}(\theta_t)$.

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i = 1, \dots, n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.
2. Sample $i_t \in \{1, \dots, n\}$ with replacement, and update: $g_{i_t,t} = \nabla f_{i_t}(\theta_t)$.
3. For $i \neq i_t$, propagate the previous value: $g_{i,t} = g_{i,t-1}$

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i = 1, \dots, n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.
2. Sample $i_t \in \{1, \dots, n\}$ with replacement, and update: $g_{i_t,t} = \nabla f_{i_t}(\theta_t)$.
3. For $i \neq i_t$, propagate the previous value: $g_{i,t} = g_{i,t-1}$
4. Iteration: $\theta_{t+1} = \theta_t - \dfrac{\eta}{n} \displaystyle\sum_{i=1}^{n} g_{i,t}$.

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i=1,\dots,n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.
2. Sample $i_t \in \{1,\dots,n\}$ with replacement, and update: $g_{i_t,t} = \nabla f_{i_t}(\theta_t)$.
3. For $i \neq i_t$, propagate the previous value: $g_{i,t} = g_{i,t-1}$
4. Iteration: $\theta_{t+1} = \theta_t - \dfrac{\eta}{n} \displaystyle\sum_{i=1}^{n} g_{i,t}$.

Equivalently: $\sum_{i=1}^n g_{i,t} = \left(\sum_{i=1}^n \nabla f_i(\theta_{t_i}) - \nabla f_{i_t}(\theta_{t_{i_t}}) + \nabla f_{i_t}(\theta_t)\right)$.

SAG: precise definition

In SAG, we use as the *old iterate* the last time the corresponding function was picked.

Step \ Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate"	Update
$t=0$	✓	✓	✓	✓	✓	θ_0	
$t=1$: $i_1=3$	∼	∼	✓	∼	∼	θ_1	
$t=2$: $i_2=1$	✓	∼	∼	∼	∼	θ_2	
$t=3$: $i_3=4$	∼	∼	∼	✓	∼	θ_3	
$t=4$: $i_4=1$	✓	∼	∼	∼	∼	θ_4	
$t=5$: $i_5=5$	∼	∼	∼	∼	✓	θ_5	$\theta_6 \leftarrow \theta_5 - \eta/n \sum$ □

In other words, at each step t:

1. For each function f_i, $i=1,\dots,n$, keep in memory a gradient (represented as □) $g_{i,t} = \nabla f_i(\theta_{t_i})$ of f_i at the last point it was computed.
2. Sample $i_t \in \{1,\dots,n\}$ with replacement, and update: $g_{i_t,t} = \nabla f_{i_t}(\theta_t)$.
3. For $i \neq i_t$, propagate the previous value: $g_{i,t} = g_{i,t-1}$
4. Iteration: $\theta_{t+1} = \theta_t - \dfrac{\eta}{n} \sum_{i=1}^{n} g_{i,t}$.

Equivalently: $\sum_{i=1}^n g_{i,t} = \left(\sum_{i=1}^n \nabla f_i(\theta_{t_i}) - \nabla f_{i_t}(\theta_{t_{i_t}}) + \nabla f_{i_t}(\theta_t)\right)$.

↬ $\oplus$ update costs the same as SGD → complexity per iteration $O(d)$.
↬ $\ominus$ needs to store all gradients $\nabla f_i(\theta_{t_i})$ at "points in the past" → storage cost $O(nd)$.

Variance reduced algorithms: SAG

Algorithm : Stochastic Average Gradient (SAG)

Hyperparameters: learning rate $\eta > 0$, number of steps T.

Initialization: initial weight vector θ_0.

Iteration 0. For all $i = 1, \dots, n$, compute $g_{i,0} \leftarrow \nabla f_i(\theta_{(0)})$. Update $\theta_1 = \theta_0 - \eta \sum_{i=1}^n g_{i,0}$.

For $t = 1, 2, \dots, T$ do

- Pick uniformly at random i_t in $\{1, \dots, n\}$
- Update $g_{i_t,t} \leftarrow \nabla f_{i_t}(\theta_{t-1})$
- Propagate for all $i \neq i_t$, $g_{i,t} \leftarrow g_{i,t-1}$.
- Apply

$$\theta_t \leftarrow \theta_{t-1} - \frac{\eta}{n} \sum_{i=1}^n g_{i,t}$$

Output: Return last θ_T

Variant of SAG: SAGA $\rightarrow$ similar but with more weight on the current gradient / current point.

Solving the memory problem: SVRG

SAG method is promising, but its implementation is limited to "moderate n" regime:

- n large enough for stochastic methods to be relevant.
- the memory cost $n \times d$ needs not to be too large.

To solve the memory issue, SVRG was introduced. SVRG enjoys:

- $\oplus$ A negligible memory cost: $O(d)$
- $\ominus$ A computational cost twice as large as SGD ($2d$ per iteration).
- $\ominus$ Has an extra hyperparameter m, not always easy to tune.

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

Outer loop – Inner loop / Obs. used	f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t = 1$ $k = 0$	✓	✓	✓	✓	✓	$\theta_{1,0}$	$\theta_{1,1} \leftarrow \theta_{1,0} - \frac{\eta}{n} \sum \square$.	$O(nd)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop / Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$	$\theta_{1,2} \leftarrow \theta_{1,1} - \frac{\eta}{n} \sum \square$.	$O(d)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$	$\theta_{1,3} \leftarrow \theta_{1,2} - \frac{\eta}{n} \sum \square$.	$O(d)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$		$O(d)$
	⋮						⋮		
	$k=m$: $i_m=4$	∼	∼	∼	✓	∼	$\theta_{1,m}$	$\theta_{1,m+1} \leftarrow \theta_{1,m} - \frac{\eta}{n} \sum \square$.	$O(d)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$		$O(d)$
	⋮						⋮		
	$k=m$: $i_m=4$	∼	∼	∼	✓	∼	$\theta_{1,m}$		$O(d)$
$t=2$	$k=0$:	✓	✓	✓	✓	✓	$\theta_{2,0} \leftarrow \theta_{1,m}$	$\theta_{2,1} \leftarrow \theta_{2,0} - \frac{\eta}{n} \sum \square$	$O(nd)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$		$O(d)$
	⋮						⋮		
	$k=m$: $i_m=4$	∼	∼	∼	✓	∼	$\theta_{1,m}$		$O(d)$
$t=2$	$k=0$:	✓	✓	✓	✓	✓	$\theta_{2,0} \leftarrow \theta_{1,m}$		$O(nd)$
	$k=1$: $i_1=5$	∼	∼	∼	∼	✓	$\theta_{2,1}$	$\theta_{2,2} \leftarrow \theta_{2,1} - \frac{\eta}{n} \sum \square$.	$O(d)$

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1, \dots, T \\ k \text{ inner loop index, in } 1, \dots, m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n} \sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$		$O(d)$
	⋮						⋮		
	$k=m$: $i_m=4$	∼	∼	∼	✓	∼	$\theta_{1,m}$		$O(d)$
$t=2$	$k=0$:	✓	✓	✓	✓	✓	$\theta_{2,0} \leftarrow \theta_{1,m}$		$O(nd)$
	$k=1$: $i_1=5$	∼	∼	∼	∼	✓	$\theta_{2,1}$		$O(d)$
	$k=2$: $i_2=2$	∼	✓	∼	∼	∼	$\theta_{2,2}$	$\theta_{2,3} \leftarrow \theta_{2,2} - \frac{\eta}{n} \sum \square$	$O(d)$
	⋮					⋮			

Solving the memory problem: SVRG

How SVRG works: two loops structure: Iterate $\theta_{t,k}$: $\begin{cases} t \text{ outer loop index, in } 1,\dots,T \\ k \text{ inner loop index, in } 1,\dots,m \end{cases}$

$\rightarrow$ Outer loop: reset every m iter. We compute and store

$$\nabla f(\theta_{t+1,0}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\theta_{t+1,0})$$

$\rightarrow$ Inner loop update: $\theta_{t,k+1} \leftarrow \theta_{t,k} - \frac{\eta}{n}\sum \square$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$	Update	Iter. complexity
$t=1$	$k=0$	✓	✓	✓	✓	✓	$\theta_{1,0}$		$O(nd)$
	$k=1$: $i_1=3$	∼	∼	✓	∼	∼	$\theta_{1,1}$		$O(d)$
	$k=2$: $i_2=1$	✓	∼	∼	∼	∼	$\theta_{1,2}$		$O(d)$
	⋮						⋮		
	$k=m$: $i_m=4$	∼	∼	∼	✓	∼	$\theta_{1,m}$		$O(d)$
$t=2$	$k=0$:	✓	✓	✓	✓	✓	$\theta_{2,0} \leftarrow \theta_{1,m}$		$O(nd)$
	$k=1$: $i_1=5$	∼	∼	∼	∼	✓	$\theta_{2,1}$		$O(d)$
	$k=2$: $i_2=2$	∼	✓	∼	∼	∼	$\theta_{2,2}$	$\theta_{2,3} \leftarrow \theta_{2,2} - \frac{\eta}{n}\sum \square$	$O(d)$
	⋮					⋮			

The "old iterate" is always the one at the beginning of the loop $\theta_{t,0}$. That way, we only have to store *one* gradient $\nabla f(\theta_{t,0})$. Indeed:

$$\sum \square = \left(\underbrace{\nabla f(\theta_{t,0})}_{\text{stored}} + \underbrace{\nabla f_{i_k}(\theta_{t,k})}_{\text{computed} \rightarrow O(d)} - \underbrace{\nabla f_{i_k}(\theta_{t,0})}_{\text{computed} \rightarrow O(d)} \right)$$

Solving the memory problem: SVRG

Algorithm : Stochastic Variance Reduced Gradient (SVRG)

Hyperparameters: step $\eta > 0$, # of outer loops T, # of inner loop iterations m.
Initialization: initial weight vector θ_0
Outer loop: for $t = 1, 2, \ldots, T$ do

- Compute $\nabla f(\theta_{t,0})$
- **Inner loop:** for $k = 1, \ldots, m$
  - Sample uniformly at random i_k in $\{1, \ldots, n\}$
  - Apply the step

$$\theta_{t,k+1} \leftarrow \theta_{t,k} - \eta(\nabla f_{i_k}(\theta_{t,k}) - \nabla f_{i_k}(\theta_{t,0}) + \nabla f(\theta_{t,0}))$$

- Set

$$\theta_{t+1,0} \leftarrow \theta_{t,m+1} \qquad \left(\text{or } \frac{1}{m} \sum_{k=1}^{m} \theta_{t,k} \right)$$

Output: Return $\theta_{T+1,0}$.

Outer loop – Inner loop \ Obs. used		f_1	f_2	f_3	f_4	f_5	"Current iterate" $\theta_{t,k}$
$t = 1$	$k = 0$	✓	✓	✓	✓	✓	$\theta_{1,0}$
	$k = 1$: $i_1 = 3$	∼	∼	✓	∼	∼	$\theta_{1,1}$
	$k = 2$: $i_2 = 1$	✓	∼	∼	∼	∼	$\theta_{1,2}$
	⋮						⋮
	$k = m$: $i_m = 4$	∼	∼	∼	✓	∼	$\theta_{1,m}$
$t = 1$	$k = 0$:	✓	✓	✓	✓	✓	$\theta_{2,0} = \theta_{1,m}$
	$k = 1$: $i_1 = 5$	∼	∼	∼	∼	✓	$\theta_{2,1}$
	⋮					⋮	

Remark: the choice of m for SVRG is not always easy. By default, use $m = n$.

Theoretical results for variance reduced methods

Theorem : Convergence of SAGA

Assume that *all* functions f_i are L-smooth and f is μ-strongly convex. Let $\theta_\star$ be the minimum of f on $\mathbb{R}^d$. Then with $\eta = \frac{1}{4L}$, after T iterations, SAGA satisfies:

$$\mathbb{E}\left[f(\theta_T) - f(\theta_\star)\right] \leqslant \frac{L}{2}\left(1 - \min\left\{\frac{1}{3n}, \frac{3\mu}{16L}\right\}\right)^T \left(1 + \frac{n}{4}\right)\|\theta_0 - \theta_\star\|_2^2.$$

Theoretical results for variance reduced methods

Theorem : Convergence of SAGA

Assume that *all* functions f_i are L-smooth and f is μ-strongly convex. Let $\theta_\star$ be the minimum of f on $\mathbb{R}^d$. Then with $\eta = \frac{1}{4L}$, after T iterations, SAGA satisfies:

$$\mathbb{E}\left[f(\theta_T) - f(\theta_\star)\right] \leqslant \frac{L}{2}\left(1 - \min\left\{\frac{1}{3n}, \frac{3\mu}{16L}\right\}\right)^T \left(1 + \frac{n}{4}\right)\|\theta_0 - \theta_\star\|_2^2.$$

Similar contraction to GD (recall Theorem on GD under same assumptions, $\eta = \frac{1}{L}$):

$$f(\theta_T) - f(\theta_\star) \leqslant \frac{L}{2}\left(1 - \frac{\mu}{L}\right)^T \|\theta_0 - \theta_\star\|_2^2. \qquad \text{(Reminder)}$$

Theoretical results for variance reduced methods

Theorem : Convergence of SAGA

Assume that *all* functions f_i are L-smooth and f is μ-strongly convex. Let $\theta_\star$ be the minimum of f on $\mathbb{R}^d$. Then with $\eta = \frac{1}{4L}$, after T iterations, SAGA satisfies:

$$\mathbb{E}\left[f(\theta_T) - f(\theta_\star)\right] \leqslant \frac{L}{2}\left(1 - \min\left\{\frac{1}{3n}, \frac{3\mu}{16L}\right\}\right)^T \left(1 + \frac{n}{4}\right)\|\theta_0 - \theta_\star\|_2^2.$$

Similar contraction to GD (recall Theorem on GD under same assumptions, $\eta = \frac{1}{L}$):

$$f(\theta_T) - f(\theta_\star) \leqslant \frac{L}{2}\left(1 - \frac{\mu}{L}\right)^T \|\theta_0 - \theta_\star\|_2^2. \qquad \text{(Reminder)}$$

- Choice of hyperparameter η:
  - $\rightarrow$ similar to GD: we use η constant.
  - $\rightarrow$ here $\eta = \frac{1}{4L}$ is suggested by theory.
- no noise/initial condition tradeoff

$\rightarrow$ the dynamic thus compares to the one of GD.

Theoretical results for variance reduced methods

Theorem : Convergence of SAGA

Assume that *all* functions f_i are L-smooth and f is μ-strongly convex. Let $\theta_\star$ be the minimum of f on $\mathbb{R}^d$. Then with $\eta = \frac{1}{4L}$, after T iterations, SAGA satisfies:

$$\mathbb{E}\left[f(\theta_T) - f(\theta_\star)\right] \leqslant \frac{L}{2}\left(1 - \min\left\{\frac{1}{3n}, \frac{3\mu}{16L}\right\}\right)^T \left(1 + \frac{n}{4}\right)\|\theta_0 - \theta_\star\|_2^2.$$

Similar contraction to GD (recall Theorem on GD under same assumptions, $\eta = \frac{1}{L}$):

$$f(\theta_T) - f(\theta_\star) \leqslant \frac{L}{2}\left(1 - \frac{\mu}{L}\right)^T \|\theta_0 - \theta_\star\|_2^2. \qquad \text{(Reminder)}$$

- Choice of hyperparameter η:
  - $\rightarrow$ similar to GD: we use η constant.
  - $\rightarrow$ here $\eta = \frac{1}{4L}$ is suggested by theory.
- no noise/initial condition tradeoff

$\rightarrow$ the dynamic thus compares to the one of GD.

(Advanced): proof is of interest, see e.g. Francis Bach's book, Th. 5.6.

- *Idea of the proof:* the gradient oracle has a variance that goes to 0 as $t \to \infty$.
- In words, although the method is stochastic, the amount of noise reduces to 0 as we advance along the iterations.

Theoretical results for variance reduced methods

Theorem : Convergence of SAGA

Assume that *all* functions f_i are L-smooth and f is μ-strongly convex. Let $\theta_\star$ be the minimum of f on $\mathbb{R}^d$. Then with $\eta = \frac{1}{4L}$, after T iterations, SAGA satisfies:

$$\mathbb{E}\left[f(\theta_T) - f(\theta_\star)\right] \leqslant \frac{L}{2}\left(1 - \min\left\{\frac{1}{3n}, \frac{3\mu}{16L}\right\}\right)^T \left(1 + \frac{n}{4}\right)\|\theta_0 - \theta_\star\|_2^2.$$

Similar contraction to GD (recall Theorem on GD under same assumptions, $\eta = \frac{1}{L}$):

$$f(\theta_T) - f(\theta_\star) \leqslant \frac{L}{2}\left(1 - \frac{\mu}{L}\right)^T \|\theta_0 - \theta_\star\|_2^2. \qquad \text{(Reminder)}$$

- Choice of hyperparameter η:
  - $\rightarrow$ similar to GD: we use η constant.
  - $\rightarrow$ here $\eta = \frac{1}{4L}$ is suggested by theory.
- no noise/initial condition tradeoff

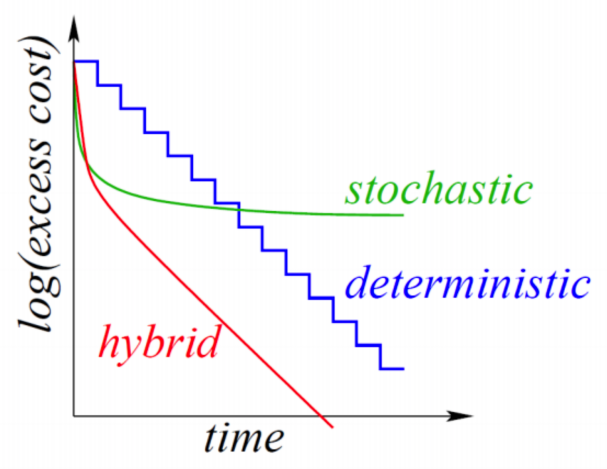
$\rightarrow$ the dynamic thus compares to the one of GD.

(Advanced): proof is of interest, see e.g. Francis Bach's book, Th. 5.6.

- *Idea of the proof:* the gradient oracle has a variance that goes to 0 as $t \to \infty$.
- In words, although the method is stochastic, the amount of noise reduces to 0 as we advance along the iterations.

Variance reduction enables to achieve the best of both worlds.

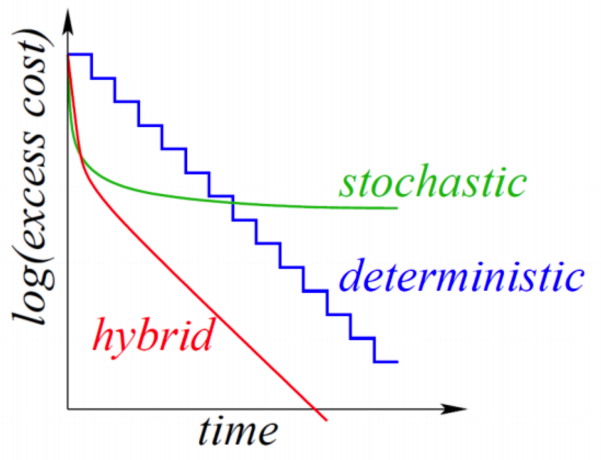
	SGD	GD	SAG/SVRG/SAGA
Complexity per iteration	$O(d)$	$O(n \times d)$	$O(d)$
Rate for L-smooth & μ-strongly-convex	$O\left(\frac{\kappa}{t}\right)$	$O\left(\exp\left(-\frac{t}{\kappa}\right)\right)$	$O\left(\exp\left(-t(\frac{1}{\kappa} \wedge \frac{1}{n})\right)\right)$



(Fig. from Schmidt, Le Roux, Bach 2023)

Variance reduction enables to achieve the best of both worlds.

	SGD	GD	SAG/SVRG/SAGA
Complexity per iteration	$O(d)$	$O(n \times d)$	$O(d)$
Rate for L-smooth & μ-strongly-convex	$O\left(\frac{\kappa}{t}\right)$	$O\left(\exp\left(-\frac{t}{\kappa}\right)\right)$	$O\left(\exp\left(-t(\frac{1}{\kappa} \wedge \frac{1}{n})\right)\right)$



(Fig. from Schmidt, Le Roux, Bach 2023)

Extensions

1. **Other schemes:** The study of variance reduced methods is still an active field of research. Many novel methods are regularly presented.
   $\rightsquigarrow$ SDCA, Spider, etc.
2. **Mini-batches:** it is straightforwardly possible to extend SAG, SVRG, etc. to b-minibatches at each step.

Summary: variance reduction (VR) methods.

First-order method **Stochastic**

⚠ Only applicable to finite sums problems

Update

$$\theta_t \leftarrow \theta_{t-1} - \eta g_t(\theta_{t-1})$$

with g_t gradient oracle based on one *fresh* gradient and gradients *from the past*.

Hyperparameters: η, T, ($+$ m for SVRG.)
$\rightsquigarrow$ η: same tradeoffs as GD.
$\rightsquigarrow$ m not easy to choose

Complexity/iter in ML:

d for SAG, SAGA

$2d$ for SVRG

Intuition & motivation:
Challenge: SGD suffers from the noise
$\rightarrow$ Solution: use *previously observed gradients* to reduce the noise

Theory:

- Rate for smooth and strongly convex functions
- 💡 VR achieves the best of both worlds between SGD and GD.
- 💡 VR achieves high precision.

Several strategies:

- SAG: high mem. cost $O(nd)$, single loop
- SVRG: normal mem. cost $O(d)$, two loops.
- SAGA, Spider, etc.

$\rightarrow$ VR is the one of the best candidates for n large, convex and smooth problems, where we want to achieve a high precision.

Variance reduction: take-aways and limits

Variance reduced methods are often the best ones in large n, convex, high precision, scenarios.

solver : *{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'*

Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited to one-versus-rest schemes.

Figure: sklearn documentation

See also:

1. Benchmarking of multiple strategies - *benchopt*.
2. Implementation of SAG/SAGA/SVRG in *lightning*.

Stochastic averaged gradient (SAG and SAGA)

`classification.SAGClassifier`, `classification.SAGAClassifier`, `regression.SAGRegressor`, `regression.SAGARegressor`

- Main idea: instead of using the full gradient (average of sample-wise gradients), compute gradient for a randomly selected sample and use out-dated gradients for other samples
- Non-smooth losses: Yes (`classification.SAGAClassifier` and `regression.SAGARegressor`)
- Penalties: L1, L2, Elastic-net
- Learning rate: Yes (not very sensitive)
- Multiclass: one-vs-rest

Stochastic variance-reduced gradient (SVRG)

`classification.SVRGClassifier`, `regression.SVRGRegressor`

- Main idea: compute full gradient periodically and use it to center the gradient estimate (this can be shown to reduce the variance)
- Non-smooth losses: No
- Penalties: L2
- Learning rate: Yes (not very sensitive)
- Multiclass: one-vs-rest

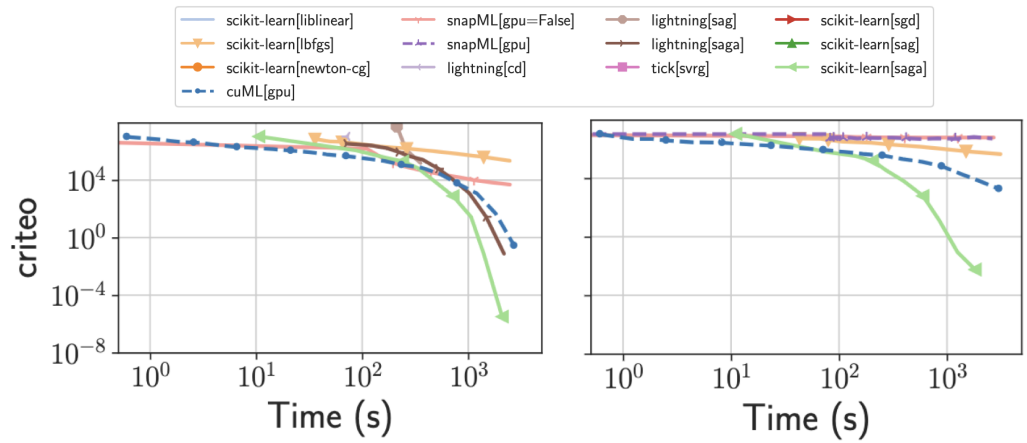


Figure: lighting documentation - *Benchopt* results - link

Variance reduction: take-aways and limits

Variance reduced methods are often the best ones in large n, convex, high precision, scenarios.

solver : *{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'*

Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited to one-versus-rest schemes.

Figure: sklearn documentation

See also:

1. Benchmarking of multiple strategies - *benchopt*.
2. Implementation of SAG/SAGA/SVRG in *lightning*.

However:

1. High precision is not always useful
2. Typically not used in deep learning.
   - Those methods require some regularity of the functions f_i (for the "past gradients" to be good estimators of the gradients at the current point).
   - Empirically, not successful $\rightsquigarrow$ hypothesis: convergence to "bad minima" $\rightarrow$ bad generalization (advanced).

$\rightarrow$ variance reduced methods are not used in deep learning contexts.

$\rightarrow$ not even implemented in keras, pytorch, tensorflow...

Let's recap! Wooclap quizz!

Outline

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t)$$

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix} \underbrace{\odot}_{\text{coord-wise product}} \nabla f(\theta_t)$$

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \underbrace{\begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix}}_{\text{coord-wise product}} \odot \nabla f(\theta_t) = \underbrace{\begin{pmatrix} \theta_{t,1} - a_1^{-1}(\nabla f(\theta_t))_1 \\ \theta_{t,2} - a_2^{-1}(\nabla f(\theta_t))_2 \\ \dots \\ \theta_{t,d} - a_d^{-1}(\nabla f(\theta_t))_d \end{pmatrix}}_{\text{one step-size per coordinate}}$$

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \underbrace{\begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix}}_{\text{coord-wise product}} \odot \nabla f(\theta_t) = \underbrace{\begin{pmatrix} \theta_{t,1} - a_1^{-1}(\nabla f(\theta_t))_1 \\ \theta_{t,2} - a_2^{-1}(\nabla f(\theta_t))_2 \\ \dots \\ \theta_{t,d} - a_d^{-1}(\nabla f(\theta_t))_d \end{pmatrix}}_{\text{one step-size per coordinate}}$$

   - The good step-size may be coordinate dependent.

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix} \underbrace{\odot}_{\text{coord-wise product}} \nabla f(\theta_t) = \underbrace{\begin{pmatrix} \theta_{t,1} - a_1^{-1}(\nabla f(\theta_t))_1 \\ \theta_{t,2} - a_2^{-1}(\nabla f(\theta_t))_2 \\ \dots \\ \theta_{t,d} - a_d^{-1}(\nabla f(\theta_t))_d \end{pmatrix}}_{\text{one step-size per coordinate}}$$

   - The good step-size may be coordinate dependent.
2. Similarly, if for any $\theta \in \mathbb{R}^d$, $f(\theta) = \sum_{j=1}^d f_i(\theta_j)$, then
   - we can minimize each coordinate independently

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top \theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \underbrace{\begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix}}_{\text{coord-wise product}} \odot \underbrace{\nabla f(\theta_t) = \begin{pmatrix} \theta_{t,1} - a_1^{-1}(\nabla f(\theta_t))_1 \\ \theta_{t,2} - a_2^{-1}(\nabla f(\theta_t))_2 \\ \dots \\ \theta_{t,d} - a_d^{-1}(\nabla f(\theta_t))_d \end{pmatrix}}_{\text{one step-size per coordinate}}$$

   - The good step-size may be coordinate dependent.

2. Similarly, if for any $\theta \in \mathbb{R}^d$, $f(\theta) = \sum_{j=1}^d f_i(\theta_j)$, then
   - we can minimize each coordinate independently
   - the "good" step-size may strongly depend on the coordinate.

Coordinate Gradient Descent - Motivation

1. Consider a quadratic function, $f : \theta \mapsto \frac{1}{2}\theta^T A\theta + b^\top\theta + c$
   - Newton method converges in one step.
   - If A is diagonal: $A = \begin{pmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \dots \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & a_d \end{pmatrix}$, then Newton step writes as:

$$\theta_{t+1} = \theta_t - A^{-1}\nabla f(\theta_t) = \theta_t - \underbrace{\begin{pmatrix} a_1^{-1} \\ a_2^{-1} \\ \dots \\ a_d^{-1} \end{pmatrix}}_{\text{coord-wise product}} \odot \nabla f(\theta_t) = \underbrace{\begin{pmatrix} \theta_{t,1} - a_1^{-1}(\nabla f(\theta_t))_1 \\ \theta_{t,2} - a_2^{-1}(\nabla f(\theta_t))_2 \\ \dots \\ \theta_{t,d} - a_d^{-1}(\nabla f(\theta_t))_d \end{pmatrix}}_{\text{one step-size per coordinate}}$$

   - The good step-size may be coordinate dependent.

2. Similarly, if for any $\theta \in \mathbb{R}^d$, $f(\theta) = \sum_{j=1}^d f_i(\theta_j)$, then
   - we can minimize each coordinate independently
   - the "good" step-size may strongly depend on the coordinate.

E.g., consider $d = 2$ and $f(\theta) = \underbrace{\frac{\theta_1^2}{2} + 4\log(1 + \exp(\theta_1))}_{\substack{L=2\text{-smooth} \\ \text{good coord step-size } \eta_1=1/2}} + \underbrace{10\frac{\theta_2^2}{2}}_{\substack{L=10\text{-smooth} \\ \text{good coord step-size } \eta_2=1/10}}$

Good algorithm choice: $\theta_{t+1} = \theta_t - \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix} \odot \begin{pmatrix} (\nabla f(\theta_t))_1 \\ (\nabla f(\theta_t))_2 \end{pmatrix}$

$\rightarrow$ **In Coordinate Descent methods** act differently (and iteratively) on *each* coordinate.

Three algorithms: 1. Coordinate dependent step size (CDSS)

Idea 1. *Update all coordinates* simultaneously with one step size for each coordinate:

Three algorithms: 1. Coordinate dependent step size (CDSS)

Idea 1. *Update all coordinates* simultaneously with one step size for each coordinate:

Algorithm : Coordinate dependent step size

Hyperparameters: # steps T, step-size vector $\eta = (\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Update all coordinates $\#j$

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$

Output: Return last θ_T

Three algorithms: 1. Coordinate dependent step size (CDSS)

Idea 1. *Update all coordinates* simultaneously with one step size for each coordinate:

Algorithm : Coordinate dependent step size

Hyperparameters: # steps T, step-size vector $\eta = (\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Update all coordinates $\#j$

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$

Output: Return last θ_T

- Equivalently, the update writes as

$$\theta_{t+1} = \theta_t - \eta \odot \nabla f(\theta_t)$$

Three algorithms: 1. Coordinate dependent step size (CDSS)

Idea 1. *Update all coordinates* simultaneously with one step size for each coordinate:

Algorithm : Coordinate dependent step size

Hyperparameters: # steps T, step-size vector $\eta = (\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Update all coordinates $\#j$

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$

Output: Return last θ_T

- Equivalently, the update writes as

$$\theta_{t+1} = \theta_t - \eta \odot \nabla f(\theta_t)$$

- The step-size η_j for coordinate $\#j$, can be taken as $\eta_j = 1/L_j$ where L_j is the smoothness of

$$z \mapsto f_j(z) = f(\theta + z e_j) = f(\theta_1, \ldots, \theta_{j-1}, \theta_j + z, \ \theta_{j+1}, \ldots, \theta_d)$$

- Complexity $O(d)$ per iteration.

Three algorithms: 2. CGD - Coordinate gradient descent

Idea 2. Do a gradient step on one coordinate at a time (keeping all others fixed):

Three algorithms: 2. CGD - Coordinate gradient descent

Idea 2. Do a gradient step on one coordinate at a time (keeping all others fixed):

Algorithm : Coordinate gradient descent (CGD)

Hyperparameters: # steps T, step-sizes $(\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ with a gradient step.

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$
$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

Three algorithms: 2. CGD - Coordinate gradient descent

Idea 2. Do a gradient step on one coordinate at a time (keeping all others fixed):

Algorithm : Coordinate gradient descent (CGD)

Hyperparameters: # steps T, step-sizes $(\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ with a gradient step.

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$
$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Complexity per iteration: $O(1)$. Indeed $(\nabla f(\theta_t))_j = \frac{\partial f}{\partial \theta_j}(\theta)$.

Three algorithms: 2. CGD - Coordinate gradient descent

Idea 2. Do a gradient step on one coordinate at a time (keeping all others fixed):

Algorithm : Coordinate gradient descent (CGD)

Hyperparameters: # steps T, step-sizes $(\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ with a gradient step.

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$
$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Complexity per iteration: $O(1)$. Indeed $(\nabla f(\theta_t))_j = \frac{\partial f}{\partial \theta_j}(\theta)$.
- Several strategies regarding choice of coordinates order: e.g., fixed loop over coordinates, or random sampling uniformly at each step, or greedy (steepest one).

Three algorithms: 2. CGD - Coordinate gradient descent

Idea 2. Do a gradient step on one coordinate at a time (keeping all others fixed):

Algorithm : Coordinate gradient descent (CGD)

Hyperparameters: # steps T, step-sizes $(\eta_j)_{1 \leqslant j \leqslant d}$.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ with a gradient step.

$$(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$$
$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Complexity per iteration: $O(1)$. Indeed $(\nabla f(\theta_t))_j = \frac{\partial f}{\partial \theta_j}(\theta)$.
- Several strategies regarding choice of coordinates order: e.g., fixed loop over coordinates, or random sampling uniformly at each step, or greedy (steepest one).
- As for CDSS, the step-size η_j for coordinate $\#j$, can be taken as $\eta_j = 1/L_j$ where L_j is the smoothness of

$$z \mapsto f_j(z) = f(\theta + z e_j) = f(\theta_1, \ldots, \theta_{j-1}, \theta_j + z, \ \theta_{j+1}, \ldots, \theta_d)$$

Three algorithms: 3. ECD - Exact Coordinate Descent

Idea 3. *Exactly* minimize one coordinate at a time (keeping all others fixed):

Three algorithms: 3. ECD - Exact Coordinate Descent

Idea 3. *Exactly* minimize one coordinate at a time (keeping all others fixed):

Algorithm : Exact coordinate descent (ECD)

Hyperparameters: # steps T.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ by exact minimization

$$(\theta_{t+1})_j = \underset{z \in \mathbb{R}}{\text{argmin}} \; f((\theta_t)_1, \ldots, (\theta_t)_{j-1}, z, (\theta_t)_{j+1}, \ldots, (\theta_t)_d)$$

$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

Three algorithms: 3. ECD - Exact Coordinate Descent

Idea 3. *Exactly* minimize one coordinate at a time (keeping all others fixed):

Algorithm : Exact coordinate descent (ECD)

Hyperparameters: # steps T.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ by exact minimization

$$(\theta_{t+1})_j = \underset{z \in \mathbb{R}}{\text{argmin}} \; f((\theta_t)_1, \ldots, (\theta_t)_{j-1}, z, (\theta_t)_{j+1}, \ldots, (\theta_t)_d)$$

$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Several strategies regarding choice of coordinates order, as before.

Three algorithms: 3. ECD - Exact Coordinate Descent

Idea 3. *Exactly* minimize one coordinate at a time (keeping all others fixed):

Algorithm : Exact coordinate descent (ECD)

Hyperparameters: # steps T.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ by exact minimization

$$(\theta_{t+1})_j = \underset{z \in \mathbb{R}}{\text{argmin}} \; f((\theta_t)_1, \ldots, (\theta_t)_{j-1}, z, (\theta_t)_{j+1}, \ldots, (\theta_t)_d)$$

$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Several strategies regarding choice of coordinates order, as before.

Three algorithms: 3. ECD - Exact Coordinate Descent

Idea 3. *Exactly* minimize one coordinate at a time (keeping all others fixed):

Algorithm : Exact coordinate descent (ECD)

Hyperparameters: # steps T.

Initialization: initial weight vector θ_0

For $t = 1, 2, \ldots$ until *convergence* do

- Choose $j \in \{1, \ldots, d\}$
- Update coordinate $\#j$ by exact minimization

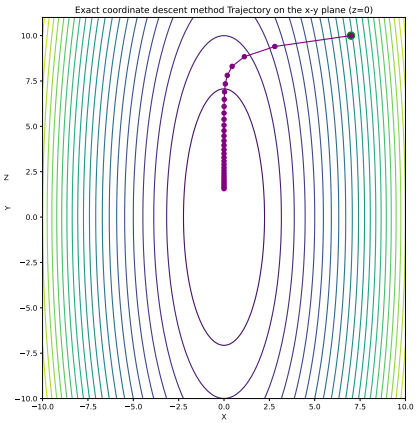
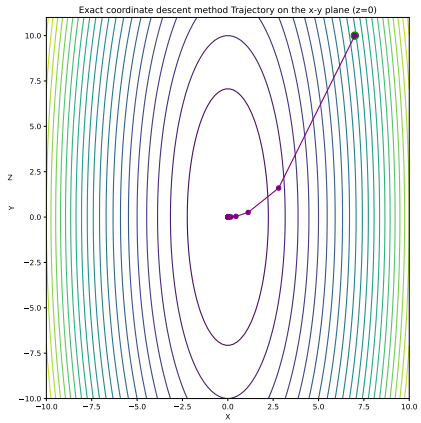
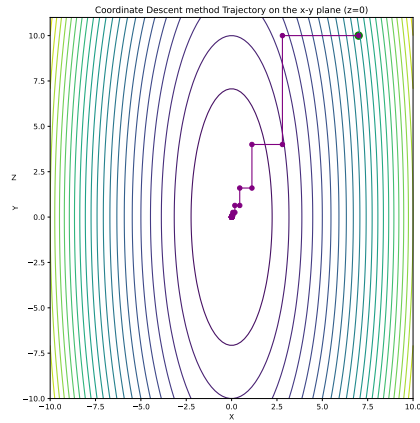
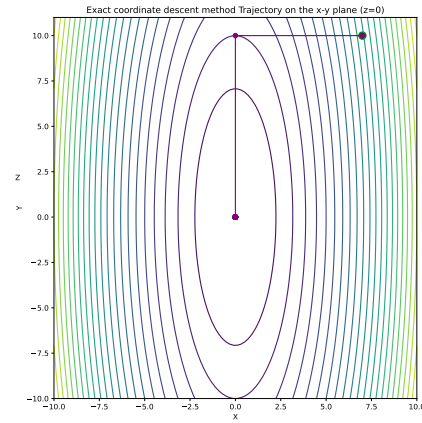
$$(\theta_{t+1})_j = \operatorname*{argmin}_{z \in \mathbb{R}} f((\theta_t)_1, \ldots, (\theta_t)_{j-1}, z, (\theta_t)_{j+1}, \ldots, (\theta_t)_d)$$

$$(\theta_{t+1})_{j'} = (\theta_t)_{j'} \quad \text{for } j' \neq j$$

Output: Return last θ_T

- Several strategies regarding choice of coordinates order, as before.
- "Exact" minimization: as the problem, is in dimension 1, possible to use fast methods, e.g. Newton method.
  $\rightsquigarrow$ Complexity per iteration typically constant (independent of d).

Coordinate methods - illustration

GD	CDSS	CGD	ECD
			
✗	✓	✓	✓

1. GD: Gradient Descent
2. CDSS: Coordinate dependent step size
3. CGD: Coordinate gradient descent
4. ECD: Exact Coordinate descent

Coordinate method: impact and theory

Advantages:

- 💡 Improves *a lot* when the "optimal" step size strongly depends on the direction

→ Among the most widely used method

→ Extremely successful, in particular for deep learning, where variation/gradient regularity is of different scales depending on e.g., the layers

Coordinate method: impact and theory

Advantages:

- Improves *a lot* when the "optimal" step size strongly depends on the direction

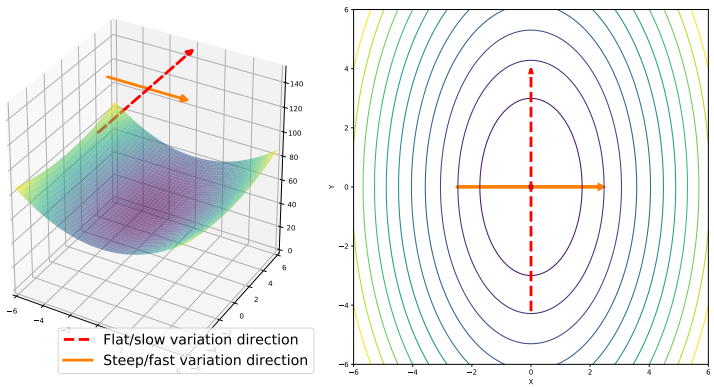
→ Among the most widely used method

→ Extremely successful, in particular for deep learning, where variation/gradient regularity is of different scales depending on e.g., the layers

Reminder (see *Regularity*): L, μ, κ capture regularity *uniformly* over space and directions.

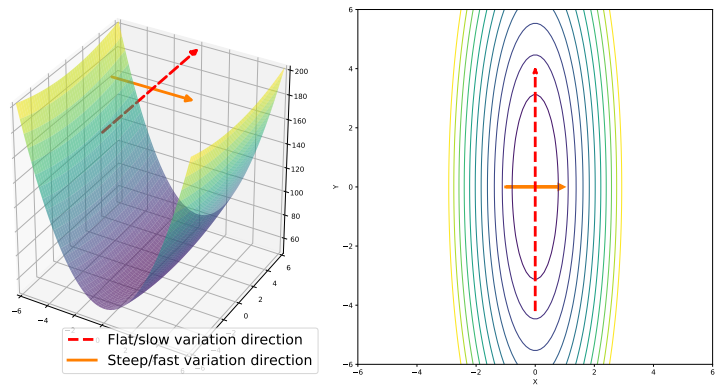
Well conditioned function

$\mu = 1, L = 1.5$



Poorly conditioned function

$\mu = 1, L = 10$



→ L large ↔ the curvature on the x-axis is much bigger than the one on the y-axis

Coordinate method: impact and theory

Advantages:

- Improves *a lot* when the "optimal" step size strongly depends on the direction

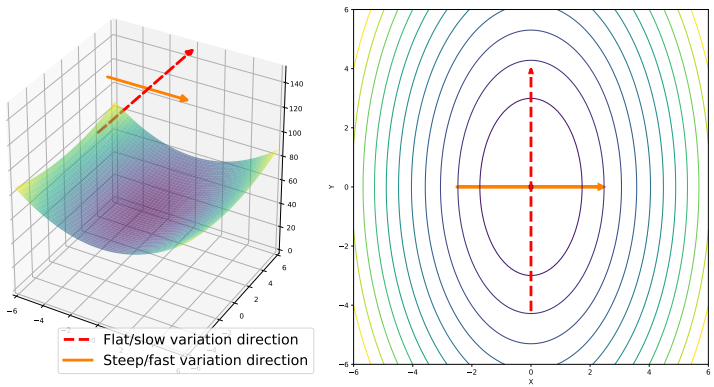
→ Among the most widely used method

→ Extremely successful, in particular for deep learning, where variation/gradient regularity is of different scales depending on e.g., the layers

Reminder (see *Regularity*): L, μ, κ capture regularity *uniformly* over space and directions.

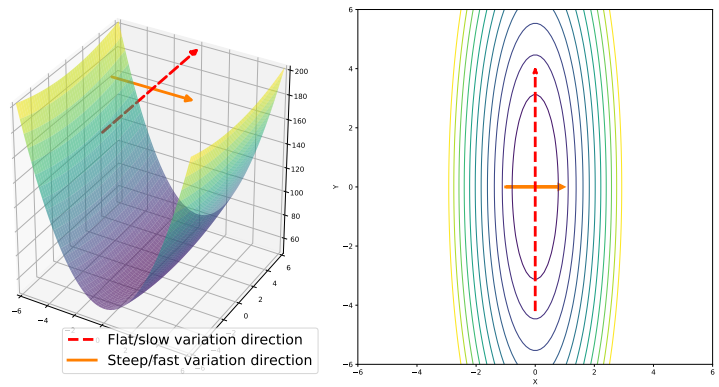
Well conditioned function

$\mu = 1, L = 1.5$



Poorly conditioned function

$\mu = 1, L = 10$



→ L large ↔ the curvature on the x-axis is much bigger than the one on the y-axis

⇝ if f is L-smooth, its curvature is upper-bounded *at any point*, *in all directions*, by L.

⇝ the choice of a step $\leqslant \frac{2}{L}$ may be far too small in some directions.

Theory: Rate of Coordinate Gradient Descent (CGD)

Theorem : Convergence for CGD, uniform sampling, smooth functions

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth and convex function, such that each f_j is L_j-smooth. Assume $\theta_\star$ to be the minimum of f on $\mathbb{R}^d$.
Consider the sequence $\{\theta^{(k)}\}$ given by CGD with $\eta_j = 1/L_j$ and coordinates $\#j_1, \#j_2, \ldots$ sampled independent and uniformly over $\{1, \ldots, d\}$. Then, after any number t of iterations:

$$\mathbb{E}\left[f(\theta_{t+1}) - f(\theta_\star)\right] \leqslant \frac{d}{1+t}\left(f(\theta_0) - f(\theta_\star) + \frac{1}{2}\|\theta_0 - \theta_\star\|_L^2\right),$$

with $\|\theta\|_L^2 = \sum_{j=1}^d L_j \theta_j^2$, and

Theory: Rate of Coordinate Gradient Descent (CGD)

Theorem : Convergence for CGD, uniform sampling, smooth functions

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth and convex function, such that each f_j is L_j-smooth. Assume $\theta_\star$ to be the minimum of f on $\mathbb{R}^d$.
Consider the sequence $\{\theta^{(k)}\}$ given by CGD with $\eta_j = 1/L_j$ and coordinates $\#j_1, \#j_2, \ldots$ sampled independent and uniformly over $\{1, \ldots, d\}$. Then, after any number t of iterations:

$$\mathbb{E}\left[f(\theta_{t+1}) - f(\theta_\star)\right] \leqslant \frac{d}{1+t}\left(f(\theta_0) - f(\theta_\star) + \frac{1}{2}\|\theta_0 - \theta_\star\|_L^2\right),$$

with $\|\theta\|_L^2 = \sum_{j=1}^d L_j \theta_j^2$, and

- Bound in expectation, since coordinates are taken at random.

Theory: Rate of Coordinate Gradient Descent (CGD)

Theorem : Convergence for CGD, uniform sampling, smooth functions

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth and convex function, such that each f_j is L_j-smooth. Assume $\theta_\star$ to be the minimum of f on $\mathbb{R}^d$.
Consider the sequence $\{\theta^{(k)}\}$ given by CGD with $\eta_j = 1/L_j$ and coordinates $\#j_1, \#j_2, \ldots$ sampled independent and uniformly over $\{1, \ldots, d\}$. Then, after any number t of iterations:

$$\mathbb{E}\left[f(\theta_{t+1}) - f(\theta_\star)\right] \leqslant \frac{d}{1+t}\left(f(\theta_0) - f(\theta_\star) + \frac{1}{2}\|\theta_0 - \theta_\star\|_L^2\right),$$

with $\|\theta\|_L^2 = \sum_{j=1}^d L_j \theta_j^2$, and

- Bound in expectation, since coordinates are taken at random.
- Compares to GD:

$$f(\theta_{t+1}) - f(\theta_\star) \leqslant \frac{1}{1+t}\frac{L}{2}\|\theta_0 - \theta_\star\|^2,$$

with $L = \max_{j=1,\ldots,d}\{L_j\}$.

Theory: Rate of Coordinate Gradient Descent (CGD)

Theorem : Convergence for CGD, uniform sampling, smooth functions

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth and convex function, such that each f_j is L_j-smooth. Assume $\theta_\star$ to be the minimum of f on $\mathbb{R}^d$.
Consider the sequence $\{\theta^{(k)}\}$ given by CGD with $\eta_j = 1/L_j$ and coordinates $\#j_1, \#j_2, \ldots$ sampled independent and uniformly over $\{1, \ldots, d\}$. Then, after any number t of iterations:

$$\mathbb{E}\left[f(\theta_{t+1}) - f(\theta_\star)\right] \leqslant \frac{d}{1+t}\left(f(\theta_0) - f(\theta_\star) + \frac{1}{2}\|\theta_0 - \theta_\star\|_L^2\right),$$

with $\|\theta\|_L^2 = \sum_{j=1}^d L_j \theta_j^2$, and

- Bound in expectation, since coordinates are taken at random.
- Compares to GD:

$$f(\theta_{t+1}) - f(\theta_\star) \leqslant \frac{1}{1+t}\frac{L}{2}\|\theta_0 - \theta_\star\|^2,$$

  with $L = \max_{j=1,\ldots,d}\{L_j\}$.
- Cost per iteration is d times smaller for CGD

Theory: Rate of Coordinate Gradient Descent (CGD)

Theorem : Convergence for CGD, uniform sampling, smooth functions

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth and convex function, such that each f_j is L_j-smooth. Assume $\theta_\star$ to be the minimum of f on $\mathbb{R}^d$.
Consider the sequence $\{\theta^{(k)}\}$ given by CGD with $\eta_j = 1/L_j$ and coordinates $\#j_1, \#j_2, \ldots$ sampled independent and uniformly over $\{1, \ldots, d\}$. Then, after any number t of iterations:

$$\mathbb{E}\left[f(\theta_{t+1}) - f(\theta_\star)\right] \leqslant \frac{d}{1+t}\left(f(\theta_0) - f(\theta_\star) + \frac{1}{2}\|\theta_0 - \theta_\star\|_L^2\right),$$

with $\|\theta\|_L^2 = \sum_{j=1}^d L_j \theta_j^2$, and

- Bound in expectation, since coordinates are taken at random.
- Compares to GD:

$$f(\theta_{t+1}) - f(\theta_\star) \leqslant \frac{1}{1+t}\frac{L}{2}\|\theta_0 - \theta_\star\|^2,$$

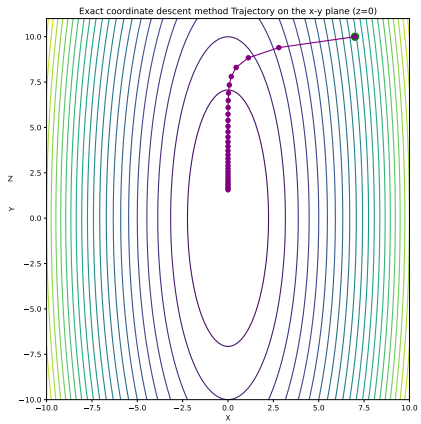
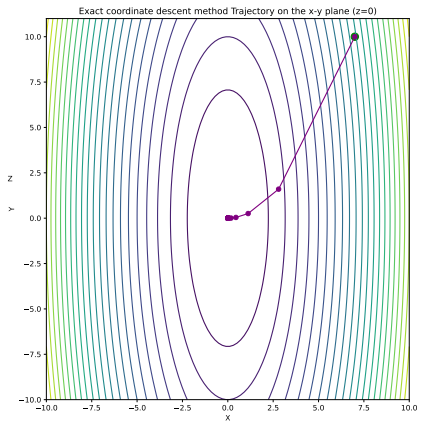
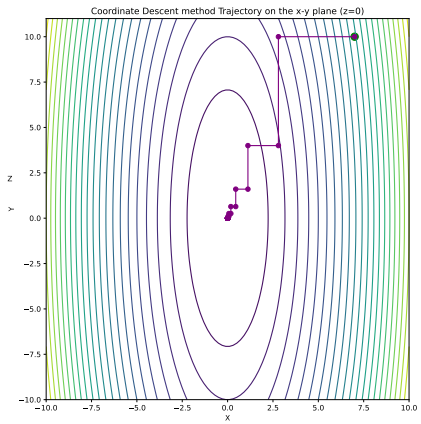
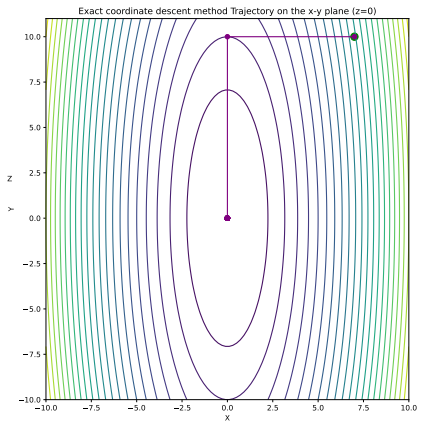
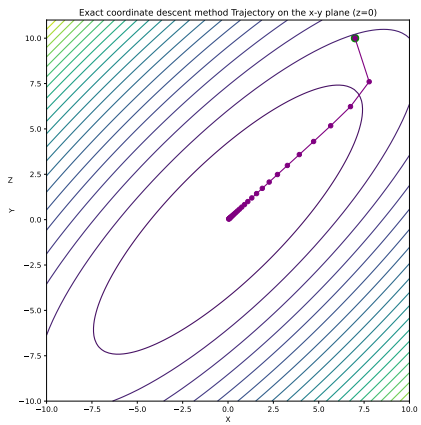
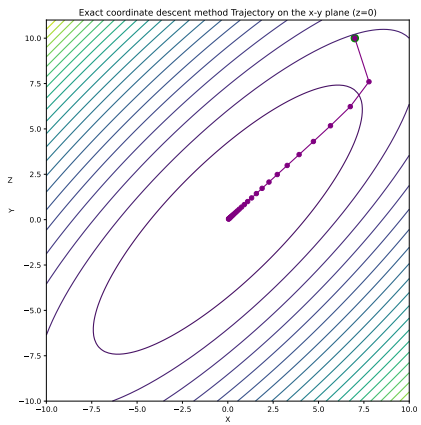
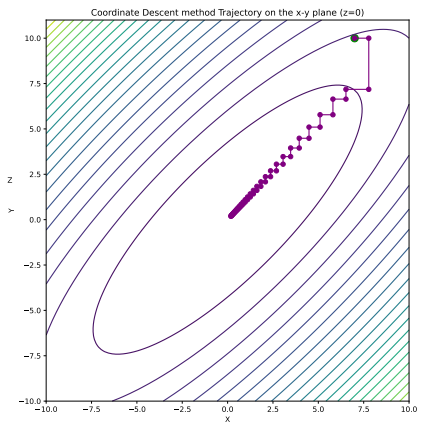
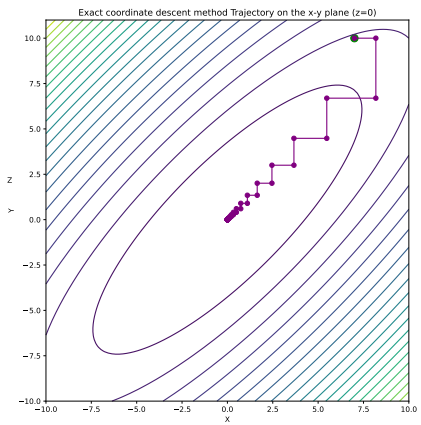
  with $L = \max_{j=1,\ldots,d}\{L_j\}$.
- Cost per iteration is d times smaller for CGD
  - $\frac{1}{2}\|\theta_0 - \theta_\star\|_L^2 \ll \frac{L}{2}\|\theta_0 - \theta_\star\|^2$ potentially.
  - $f(\theta_0) - f(\theta_\star) \ll \frac{L}{2}\|\theta_0 - \theta_\star\|^2$ potentially.

  $\rightarrow$ much better bound in some situations

Drawback

⚠: The behavior depends on the choice of basis.

- If we "rotate" the function, the behavior of coordinate-based methods change
- Consider the second row: Coordinate methods do not really improve w.r.t. GD.

GD	CDSS	CGD	ECD
			
✗	✓	✓	✓
			
~	~	~	~

Coordinate descent in sklearn

solver : *{'lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag', 'saga'}, default='lbfgs'*

Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited to one-versus-rest schemes.
- 'newton-cholesky' is a good choice for `n_samples` >> `n_features`, especially with one-hot encoded categorical features with rare categories. Note that it is limited to binary classification and the one-versus-rest reduction for multiclass classification. Be aware that the memory usage of this solver has a quadratic dependency on `n_features` because it explicitly computes the Hessian matrix.

Warning: The choice of the algorithm depends on the penalty chosen. Supported penalties by solver:

- 'lbfgs' - ['l2', None]
- 'liblinear' - ['l1', 'l2']
- 'newton-cg' - ['l2', None]
- 'newton-cholesky' - ['l2', None]
- 'sag' - ['l2', None]
- 'saga' - ['elasticnet', 'l1', 'l2', None]

Figure: Logistic-regression solvers - sklearn documentation - link

1. The solver "liblinear" uses a coordinate descent (CD) algorithm

Coordinate methods - Summary

First-order method

Deterministic gradients, possible stochastic coordinate sampling
Natural extension to stochastic gradients instead.

~~Variance reduction~~
~~Acceleration~~

CDSS: $\theta_{t+1} = \theta_t - \eta \odot \nabla f(\theta_t)$

CGD: $(\theta_{t+1})_j = (\theta_t)_j - \eta_j (\nabla f(\theta_t))_j$

ECD: $(\theta_{t+1})_j = \text{argmin}_{z \in \mathbb{R}} \quad f((\theta_t)_1, \dots, z, \dots, (\theta_t)_d)$

Hyperparameters:
All: T
CGD, CDSS: $\eta = (\eta_j)_{j=1,\dots,d}$
CGD, ECD: Coord. $\#j$ pick strategy

Complex/iter in ML:
CGD : $O(n)$
ECD: $O(n + \dots)$
CDSS: $O(nd)$

Intuition & motivation:
Challenge:

- The curvature may strongly vary depending on the direction

→ Solution:

- Use coordinate dependent update rules

Theory:

- Strongly improves upon GD if L_i varies a lot

Take away:

→ Coordinate methods are essential in many applications, in particular deep learning (scale varies a lot depending on e.g., layer)

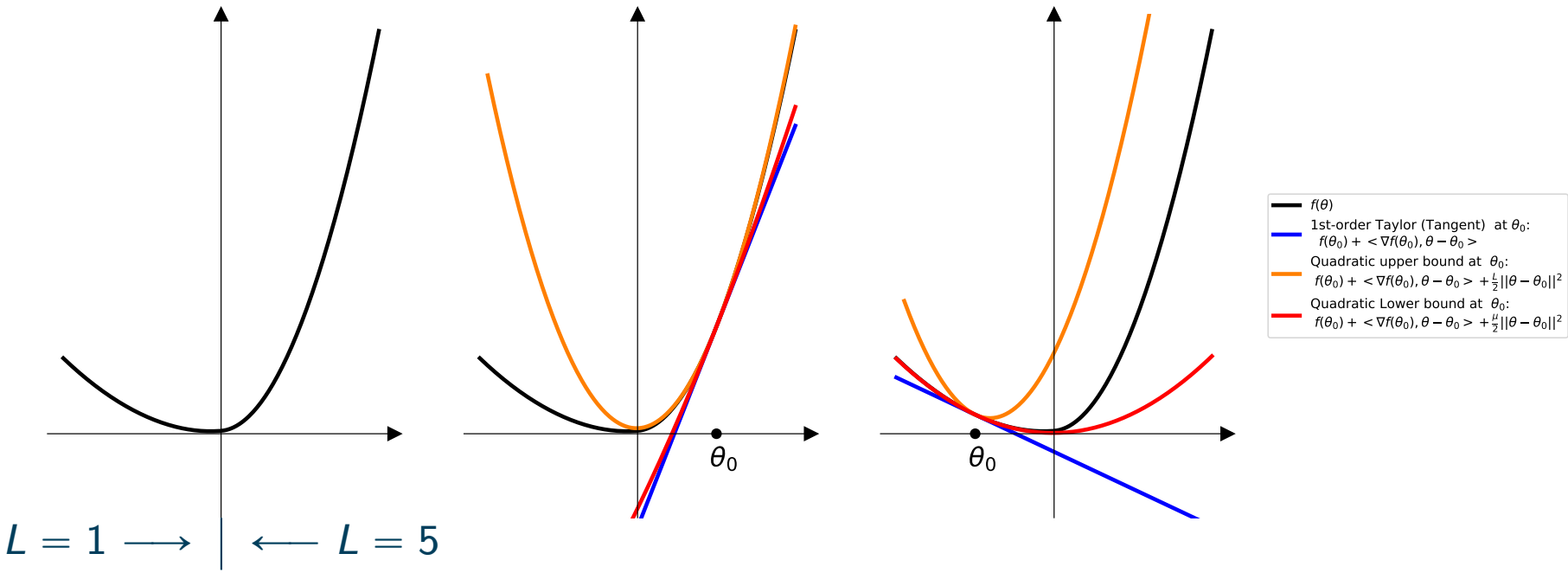
Outline

1. Extensions of GD and SGD
   - Momentum and Acceleration
   - Second-order algorithms
   - Variance reduced methods for ERM
   - Coordinate methods
   - Adaptivity
2. Wrapping up: Summary of GD extensions

Adaptivity: Intuition

The curvature can depend on the direction, but also the point we are at.

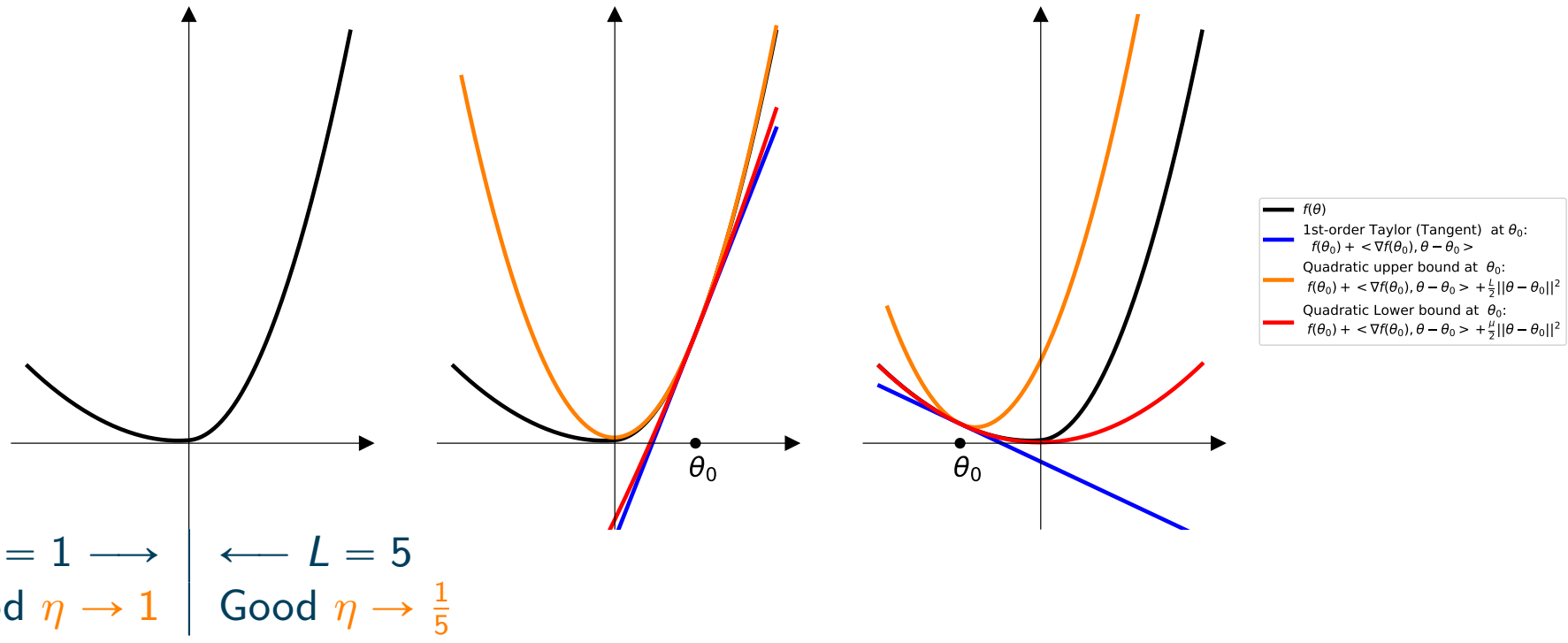
Poorly conditioned function $\kappa = 5$



Adaptivity: Intuition

The curvature can depend on the direction, but also the point we are at.

Poorly conditioned function $\kappa = 5$



$L = 1 \longrightarrow$	$\longleftarrow L = 5$
Good $\eta \to 1$	Good $\eta \to \frac{1}{5}$

In adaptive methods, we use information to **adapt the step size to the point**, leveraging information on the function, gradients, to *estimate the local curvature*.

The step size then automatically **evolves over time**.

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

The step size that minimizes the right hand side is called the Polyak step-size:

$$\eta_t^{\mathrm{PS}} := 2\frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

The step size that minimizes the right hand side is called the Polyak step-size:

$$\eta_t^{\mathrm{PS}} := 2\frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

✓ Adapts to the current point's local curvature.

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

The step size that minimizes the right hand side is called the Polyak step-size:

$$\eta_t^{\text{PS}} := 2\frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

- ✓ Adapts to the current point's local curvature.
- ✓ Can be used for non quadratic functions as well.

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

The step size that minimizes the right hand side is called the Polyak step-size:

$$\eta_t^{\mathrm{PS}} := 2\frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

- ✓ Adapts to the current point's local curvature.
- ✓ Can be used for non quadratic functions as well.
- ✓ We always have $\eta_t^{\mathrm{PS}} \geqslant \frac{1}{L}$: Indeed (Descent Lemma+)

$$f(\theta_t) - f(\theta_\star) \geqslant \frac{1}{2L}\|\nabla f(\theta_t)\|^2$$

$\rightarrow$ the step size is larger than the "pessimistic" $\frac{1}{L}$ which takes into account the highest curvature over the entire space.

Adaptivity: two techniques - 1. Polyak step size

Consider a quadratic function: there exists a matrix $A \in \mathbb{R}^{d \times d}$, such that

$$f(\theta) - f(\theta_\star) = \frac{1}{2}(\theta - \theta_\star)^T A(\theta - \theta_\star)$$

Gradient descent with step size η_t:

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t) = \theta_t - \eta_t A(\theta_t - \theta_\star).$$

Then

$$\begin{aligned}\|\theta_{t+1} - \theta_\star\|^2 &= \|\theta_t - \theta_\star\|^2 - 2\eta_t(\theta_t - \theta_\star)^\top A(\theta_t - \theta_\star) + \eta_t^2 \|A(\theta_t - \theta_\star)\|^2 \\ &= \|\theta_t - \theta_\star\|^2 - 4\eta_t(f(\theta_t) - f(\theta_\star)) + \eta_t^2 \|\nabla f(\theta_t)\|^2\end{aligned}$$

The step size that minimizes the right hand side is called the Polyak step-size:

$$\eta_t^{\mathrm{PS}} := 2\frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

- ✓ Adapts to the current point's local curvature.
- ✓ Can be used for non quadratic functions as well.
- ✓ We always have $\eta_t^{\mathrm{PS}} \geqslant \frac{1}{L}$: Indeed (Descent Lemma+)

$$f(\theta_t) - f(\theta_\star) \geqslant \frac{1}{2L}\|\nabla f(\theta_t)\|^2$$

  $\rightarrow$ the step size is larger than the "pessimistic" $\frac{1}{L}$ which takes into account the highest curvature over the entire space.
- ✗ Require the knowledge of $f_\star$

Adaptivity: two techniques - 2. Barzilai-Borwein

- ✓ Same idea: approximate locally the curvature.

Adaptivity: two techniques - 2. Barzilai-Borwein

- ✓ Same idea: approximate locally the curvature.
- ✓ Relying *only* on the last two points.
  → Use that, at any step (cocoercivity) :

$$\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle \geqslant \frac{1}{L} \|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2$$

Adaptivity: two techniques - 2. Barzilai-Borwein

- ✓ Same idea: approximate locally the curvature.
- ✓ Relying *only* on the last two points.
  → Use that, at any step (cocoercivity) :

$$\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle \geqslant \frac{1}{L} \|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2$$

  → Define

$$\eta_t^{\mathrm{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2}$$

Adaptivity: two techniques - 2. Barzilai-Borwein

- ✓ Same idea: approximate locally the curvature.
- ✓ Relying *only* on the last two points.
  → Use that, at any step (cocoercivity) :

$$\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle \geqslant \frac{1}{L} \|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2$$

  → Define

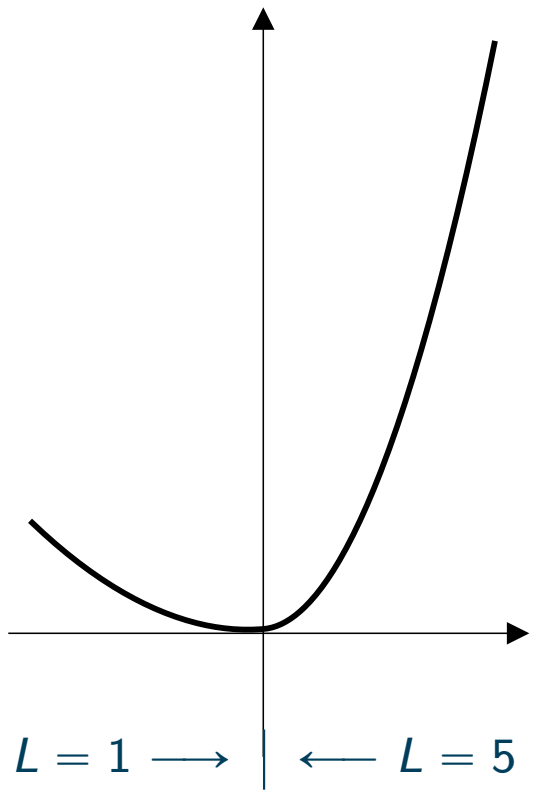
$$\eta_t^{\mathrm{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2}$$

- ✓ Again, we ensure that always, $\eta_t^{\mathrm{BB}} \geqslant \frac{1}{L}$: the step size is larger than the "pessimistic" $\frac{1}{L}$ which takes into account the highest curvature over the entire space.

Polyak step size and Barzilai-Borwein on our initial example

$$\eta_t^{\text{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2} \qquad \eta_t^{\text{PS}} := \frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

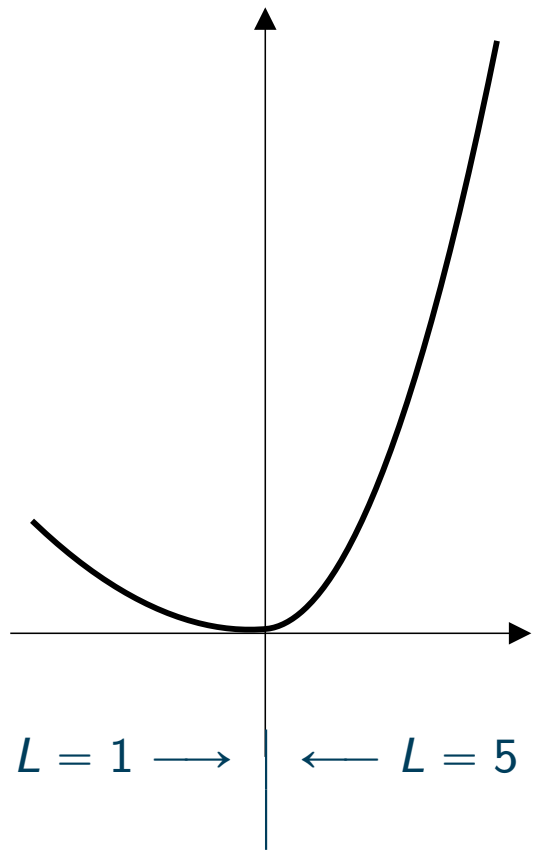
Poorly conditioned function $\kappa = 5$



Polyak step size and Barzilai-Borwein on our initial example

$$\eta_t^{\text{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2} \qquad \eta_t^{\text{PS}} := \frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

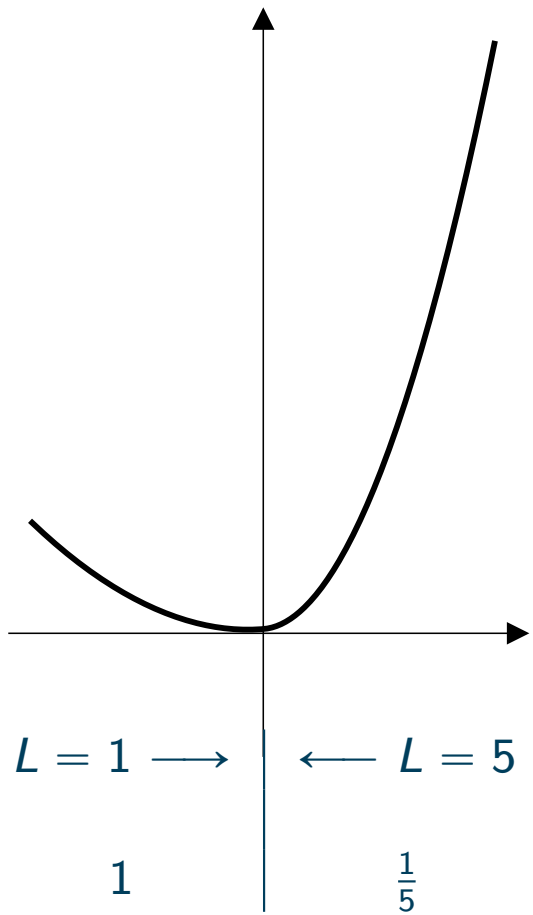
Poorly conditioned function $\kappa = 5$



Polyak step size and Barzilai-Borwein on our initial example

$$\eta_t^{\text{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2} \qquad \eta_t^{\text{PS}} := \frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

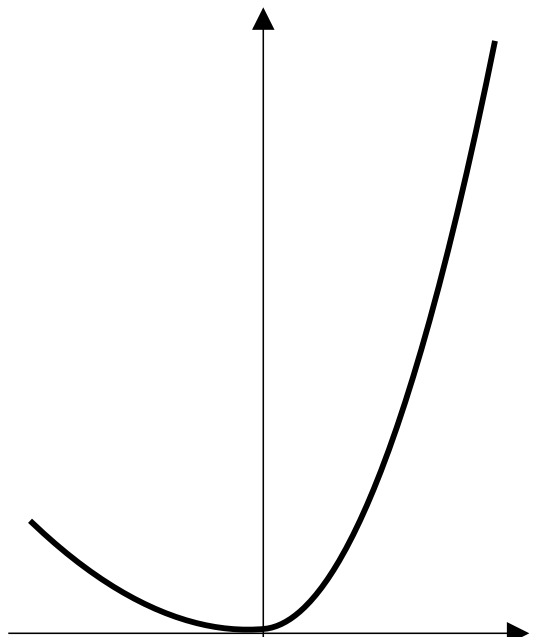
Poorly conditioned function $\kappa = 5$



Polyak step size and Barzilai-Borwein on our initial example

$$\eta_t^{\text{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2} \qquad \eta_t^{\text{PS}} := \frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

Poorly conditioned function $\kappa = 5$



η_t^{PS}	1	$\frac{1}{5}$
η_t^{BB}	1	$\frac{1}{5}$

Adaptivity - Summary

First-order | **Deterministic, natural extension to stochastic** | **Adaptive step size** | ~~Variance reduction~~ ~~Acceleration~~

BB:

$$\eta_t^{\text{BB}} := \frac{\langle \theta_t - \theta_{t-1}, \nabla f(\theta_t) - \nabla f(\theta_{t-1}) \rangle}{\|\nabla f(\theta_t) - \nabla f(\theta_{t-1})\|^2}$$

Polyak:

$$\eta_t^{\text{PS}} := \frac{f(\theta_t) - f(\theta_\star)}{\|\nabla f(\theta_t)\|^2}$$

Intuition & motivation:

Challenge:

- The curvature may strongly vary depending on the **point**

→ Solution:

- Use a step size that varies with time, estimating the **local curvature**

Take away:

→ Adaptive methods are essential in many applications, in particular deep learning (scale varies a lot along iterations)

Outline

Outline

All algorithms

	Deterministic/Stochastic	Variance Reduction	Momentum	Second order	Coordinate dependence	Adaptivity	Low/High precision
GD	Deterministic						High
SGD	Stochastic						Low
SVRG	Stochastic	✓					High
SAG	Stochastic	✓					High
Nesterov	Deterministic		✓				High
Heavy Ball	Deterministic		✓				High
Newton	Deterministic			✓			Highest
BFGS	Deterministic			quasi			High+
CDSS	Deterministic				✓		High
CGD	Deterministic*				✓		High
ECD	Deterministic*				✓		High
Polyak step	Deterministic					✓	High
Barzilai-Borwein	Deterministic					✓	High

Techniques can be combined, in particular stochastic versions of deterministic algorithms can be created for most algorithms.

Challenges and solutions

Needs or benefits from →

Requires ⇠

Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

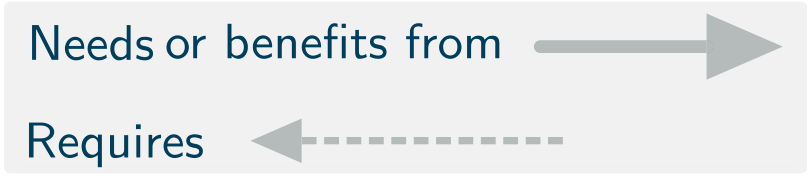
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

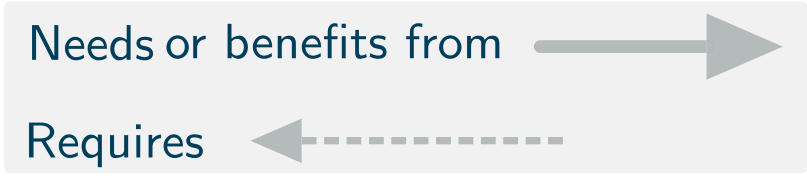
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

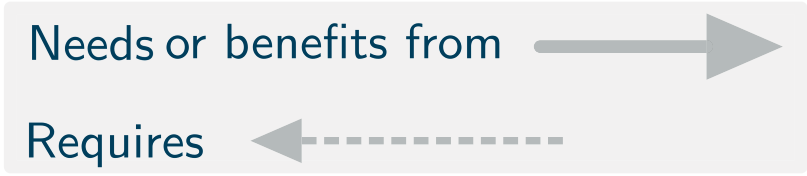
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

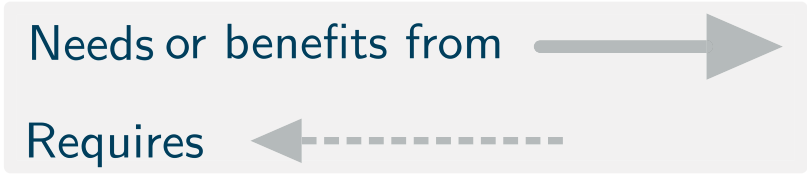
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

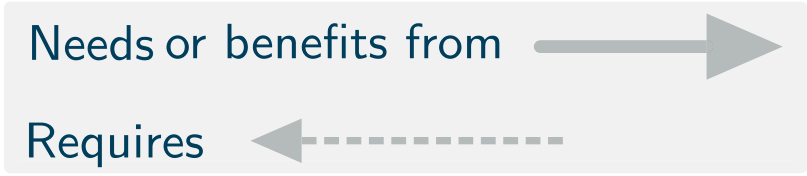
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

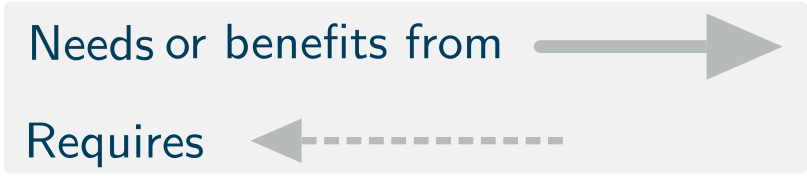
Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Question:

- Low precision needed?
- High precision needed?

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

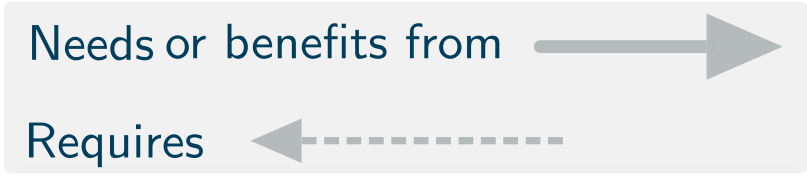
Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Challenges and solutions



Challenges

- Extremely large number of observations n
- Extremely large number of parameters d
- Moderate d
- Bad conditioning, flat areas, saddle points
- Non convex $\Rightarrow$ Local minima
- Convex case
- Different scales
  - depending on directions (e.g., layers)
  - depending on iteration

Question:

- Low precision needed?
- High precision needed?

Partial solutions:

- First order
- Stochastic
- Variance reduction
- Momentum
- Second order
- Different steps per coordinates
- Adaptive methods

Outline

Loss Landscape in deep learning

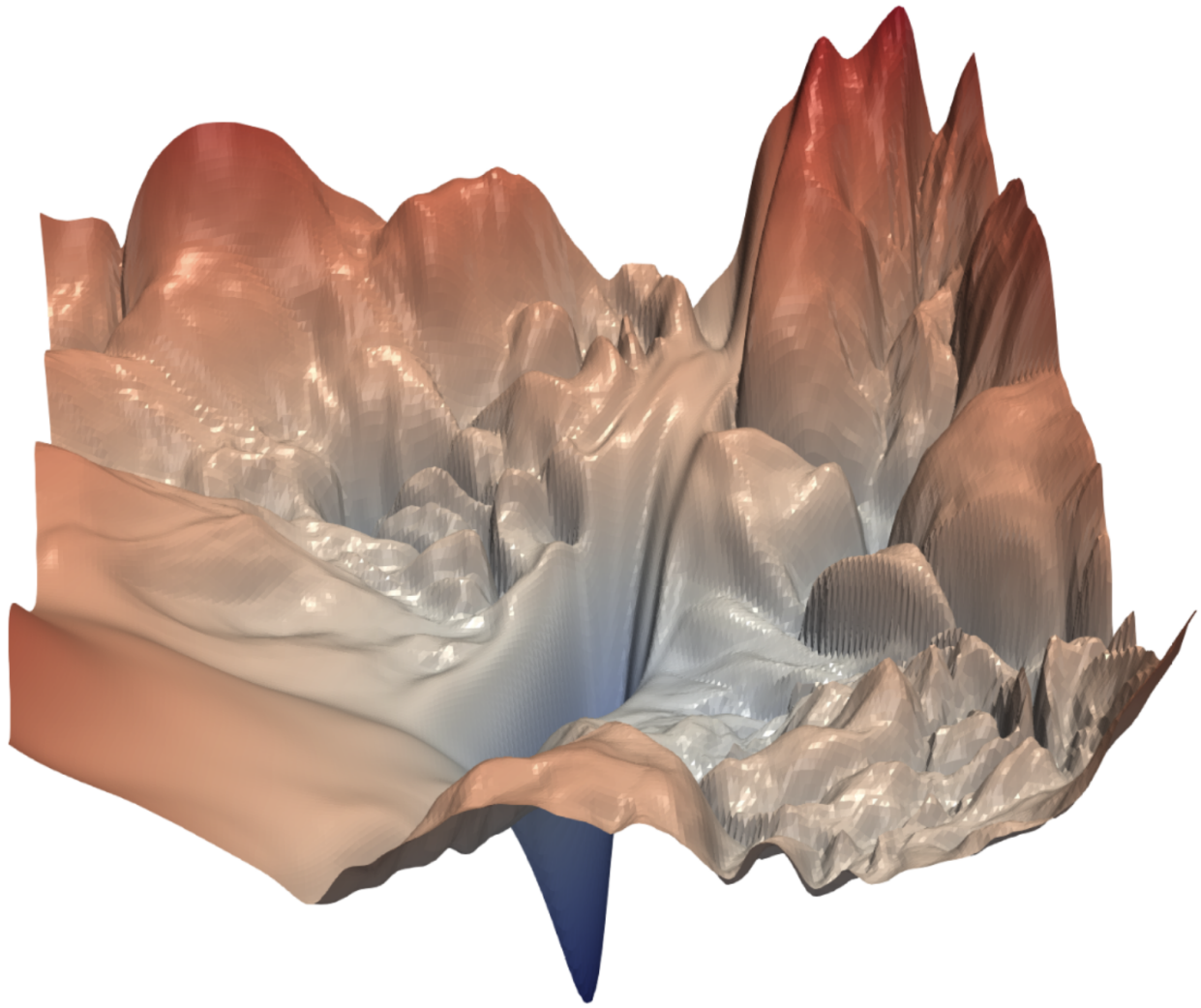


Figure: Loss surfaces of ResNet-56. 2 dimensions out of 25 millions. Source

Challenges in Deep Learning

Needs or benefits from →

Requires ⇠

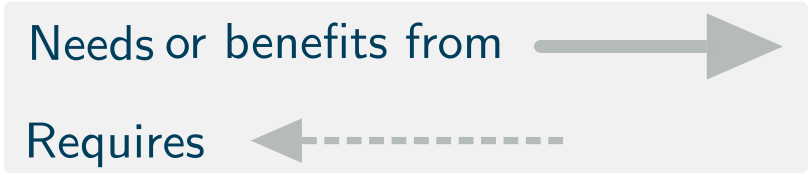
Challenges

1. Extremely large number of observations
2. Extremely large number of parameters
3. Bad conditioning + flat areas + saddle points
4. Non convex $\Rightarrow$ Local minima
5. Different scales (depending on layers, iteration)

Ingredients of popular algorithms:

1. First order
2. Stochastic
3. Momentum
4. Different steps per coordinates
5. Adaptive methods

Challenges in Deep Learning



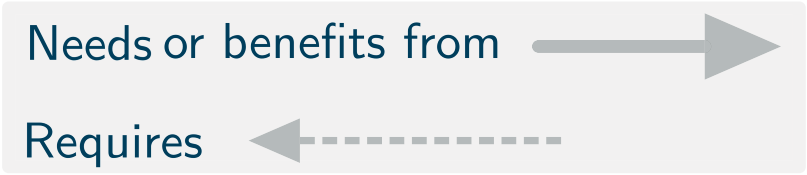
Challenges

1. Extremely large number of observations
2. Extremely large number of parameters
3. Bad conditioning + flat areas + saddle points
4. Non convex $\Rightarrow$ Local minima
5. Different scales (depending on layers, iteration)

Ingredients of popular algorithms:

1. First order
2. Stochastic
3. Momentum
4. Different steps per coordinates
5. Adaptive methods

Challenges in Deep Learning



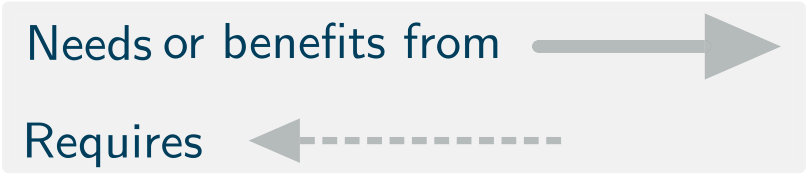
Challenges

1. Extremely large number of observations
2. Extremely large number of parameters
3. Bad conditioning + flat areas + saddle points
4. Non convex $\Rightarrow$ Local minima
5. Different scales (depending on layers, iteration)

Ingredients of popular algorithms:

1. First order
2. Stochastic
3. Momentum
4. Different steps per coordinates
5. Adaptive methods

Challenges in Deep Learning



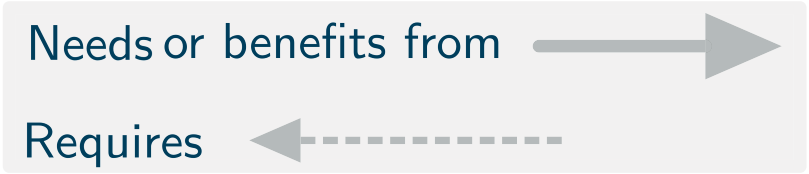
Challenges

1. Extremely large number of observations
2. Extremely large number of parameters
3. Bad conditioning + flat areas + saddle points
4. Non convex $\Rightarrow$ Local minima
5. Different scales (depending on layers, iteration)

Ingredients of popular algorithms:

1. First order
2. Stochastic
3. Momentum
4. Different steps per coordinates
5. Adaptive methods

Challenges in Deep Learning



Challenges

1. Extremely large number of observations
2. Extremely large number of parameters
3. Bad conditioning + flat areas + saddle points
4. Non convex $\Rightarrow$ Local minima
5. Different scales (depending on layers, iteration)

Ingredients of popular algorithms:

1. First order
2. Stochastic
3. Momentum
4. Different steps per coordinates
5. Adaptive methods

Generalization and overfitting problems are poorly understood but:

1. Noise helps
2. "Too precise" methods (e.g. variance reduction, second order) are not used. e.g.: SVRG is great for convex, but not even implemented in Keras.

Optimizers in practice in Deep Learning libraries

Algorithms in Deep Learning:

1. combine the necessary techniques that we have just described
2. multiple "recipes" coexist

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

Available optimizers

- SGD
- RMSprop
- Adam
- AdamW
- Adadelta
- Adagrad
- Adamax
- Adafactor
- Nadam
- Ftrl
- Lion
- Loss Scale Optimizer

Figure: Pytorch (Link) and Keras (Link) optimizers

Generic update form for Adagrad, RMSprop, Adam

	Deterministic/Stochastic	Variance Reduction	Momentum	Second order	Coordinate dependence	Adaptivity	Low/High precision
Adagrad	Stochastic	✗	((✓))	✗	✓	✓	Low
RMSPROP	Stochastic	✗	(✓)	✗	✓	✓	Low
Adam	Stochastic	✗	✓	✗	✓	✓	Low

Adagrad, RMSprop, Adam all admit un update of the following form:

$$\theta_t \leftarrow \theta_{t-1} - \underbrace{\eta_t \odot \underbrace{g_t(\theta_{t-1})}_{\text{Using a stochastic (minibatch) gradient}}}_{\text{Using a coordinate dependent and adative step size}} + \underbrace{\beta_t(\theta_{t-1} - \theta_{t-2})}_{\text{Using Momentum}}$$

Adagrad

We use the following vector of step sizes:

$$\eta_t = \frac{\gamma_0}{\sqrt{\sum_{\tau=1}^{t}(g_\tau(\theta_{\tau-1}))^2 + \epsilon}} \in \mathbb{R}^d$$

- The $\sqrt{\cdot}$ and $(\cdot)^2$ operations are coefficient-wise computations.
- Basic hyperparameters: γ_0 (initial learning rate), ϵ (avoid numerical errors, very small)

Adagrad

We use the following vector of step sizes:

$$\eta_t = \frac{\gamma_0}{\sqrt{\sum_{\tau=1}^{t}(g_\tau(\theta_{\tau-1}))^2 + \epsilon}} \in \mathbb{R}^d$$

- The $\sqrt{\cdot}$ and $(\cdot)^2$ operations are coefficient-wise computations.
- Basic hyperparameters: γ_0 (initial learning rate), ϵ (avoid numerical errors, very small)

Pros:

- Different dynamic rates on each coordinate
- Dynamic rates grow as the inverse of the gradient magnitude:
  1. Large/small gradients have small/large learning rates
  2. The dynamic over each dimension tends to be of the same order
  3. Interesting for neural networks in which gradient at different layers can be of different order of magnitude.

Adagrad

We use the following vector of step sizes:

$$\eta_t = \frac{\gamma_0}{\sqrt{\sum_{\tau=1}^{t}(g_\tau(\theta_{\tau-1}))^2 + \epsilon}} \in \mathbb{R}^d$$

- The $\sqrt{\cdot}$ and $(\cdot)^2$ operations are coefficient-wise computations.
- Basic hyperparameters: γ_0 (initial learning rate), ϵ (avoid numerical errors, very small)

Pros:

- Different dynamic rates on each coordinate
- Dynamic rates grow as the inverse of the gradient magnitude:
  1. Large/small gradients have small/large learning rates
  2. The dynamic over each dimension tends to be of the same order
  3. Interesting for neural networks in which gradient at different layers can be of different order of magnitude.
- Accumulation of gradients in the denominator act as a decreasing learning rate.

Adagrad

We use the following vector of step sizes:

$$\eta_t = \frac{\gamma_0}{\sqrt{\sum_{\tau=1}^{t}(g_\tau(\theta_{\tau-1}))^2 + \epsilon}} \in \mathbb{R}^d$$

- The $\sqrt{\cdot}$ and $(\cdot)^2$ operations are coefficient-wise computations.
- Basic hyperparameters: γ_0 (initial learning rate), ϵ (avoid numerical errors, very small)

Pros:

- Different dynamic rates on each coordinate
- Dynamic rates grow as the inverse of the gradient magnitude:
  1. Large/small gradients have small/large learning rates
  2. The dynamic over each dimension tends to be of the same order
  3. Interesting for neural networks in which gradient at different layers can be of different order of magnitude.
- Accumulation of gradients in the denominator act as a decreasing learning rate.

Cons:

- Very sensitive to initial condition: large initial gradients lead to small learning rates.

Adagrad - documentation from Pytorch

ADAGRAD

CLASS torch.optim.Adagrad(*params*, *lr=0.01*, *lr_decay=0*, *weight_decay=0*, *initial_accumulator_value=0*, *eps=1e-10*, *foreach=None*, ***, *maximize=False*, *differentiable=False*) [SOURCE]

Implements Adagrad algorithm.

---

input : γ (lr), θ_0 (params), $f(\theta)$ (objective), λ (weight decay),
τ (initial accumulator value), η (lr decay)

initialize : $state\_sum_0 \leftarrow 0$

---

for $t = 1$ **to** $\ldots$ **do**

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

$\tilde{\gamma} \leftarrow \gamma/(1 + (t-1)\eta)$

if $\lambda \neq 0$

$g_t \leftarrow g_t + \lambda\theta_{t-1}$

$state\_sum_t \leftarrow state\_sum_{t-1} + g_t^2$

$\theta_t \leftarrow \theta_{t-1} - \tilde{\gamma}\dfrac{g_t}{\sqrt{state\_sum_t} + \epsilon}$

---

return θ_t

Figure: Pytorch documentation:link

- Optional extra learning rate decay, weight decay
- Check default values!
- ⚠ No momentum by default.

Improving upon AdaGrad: RMS-prop

$$\eta_t^{\text{Adagrad}} = \frac{\gamma_0}{\sqrt{\sum_{\tau=1}^{t}(g_\tau(\theta_{\tau-1}))^2 + \epsilon}} \in \mathbb{R}^d$$

⚠ Very sensitive to initial condition: large initial gradients lead to small learning rates.

Idea: restricts the window of accumulated past gradients to some limited size through moving average.

→ new params: decay rate $\alpha > 0$

$$\eta_t^{\text{RMSProp}} = \frac{\gamma_0}{\sqrt{\underbrace{(1-\alpha)\sum_{\tau=1}^{t}\alpha^{t-\tau}(g_\tau(\theta_{\tau-1}))^2}_{v_t} + \epsilon}} \in \mathbb{R}^d$$

Equivalently:

$$v_t = \alpha v_{t-1} + (1-\alpha)(g_t(\theta_{t-1}))^2.$$

with $v_0 = 0$

RMSprop - documentation from Pytorch

CLASS torch.optim.RMSprop(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*, *foreach=None*, *maximize=False*, *differentiable=False*) [SOURCE]

Implements RMSprop algorithm.

$$
\begin{aligned}
&\textbf{input} : \alpha \text{ (alpha)}, \gamma \text{ (lr)}, \theta_0 \text{ (params)}, f(\theta) \text{ (objective)} \\
&\qquad \lambda \text{ (weight decay)}, \mu \text{ (momentum)}, \textit{centered} \\
&\textbf{initialize} : v_0 \leftarrow 0 \text{ (square average)}, \mathbf{b}_0 \leftarrow 0 \text{ (buffer)}, g_0^{ave} \leftarrow 0 \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad if\ \lambda \neq 0 \\
&\quad\quad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad v_t \leftarrow \alpha v_{t-1} + (1-\alpha) g_t^2 \\
&\quad \tilde{v_t} \leftarrow v_t \\
&\quad if\ centered \\
&\quad\quad g_t^{ave} \leftarrow g_{t-1}^{ave} \alpha + (1-\alpha) g_t \\
&\quad\quad \tilde{v_t} \leftarrow \tilde{v_t} - \left(g_t^{ave}\right)^2 \\
&\quad if\ \mu > 0 \\
&\quad\quad \mathbf{b}_t \leftarrow \mu \mathbf{b}_{t-1} + g_t / \left(\sqrt{\tilde{v_t}} + \epsilon\right) \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} - \gamma \mathbf{b}_t \\
&\quad else \\
&\quad\quad \theta_t \leftarrow \theta_{t-1} - \gamma g_t / \left(\sqrt{\tilde{v_t}} + \epsilon\right) \\
&\textbf{return } \theta_t
\end{aligned}
$$

Figure: RMSprop - Pytorch documentation - link

- Heavy-ball type momentum in option (coefficient μ)

ADAM: ADAptive Moment estimation

1. RMSProp update:
$$\theta_t \leftarrow \theta_{t-1} - \eta_t^{\text{RMSProp}} \odot g_t(\theta_{t-1}) + \underbrace{\beta_t(\theta_{t-1} - \theta_{t-2})}_{\text{Optional Momentum}}$$
2. Adam update:
$$\theta_t \leftarrow \theta_{t-1} - \eta_t^{\text{Adam}} \odot m_t$$

→ As for RMSProp, v_t is an exponentially decaying average of past square gradients:
$$v_t = \alpha v_{t-1} + (1-\alpha)(g_t(\theta_{t-1}))^2.$$
and
$$\eta_t^{\text{RMSProp}} = \eta_t^{\text{Adam}} = \frac{\gamma_0}{\sqrt{v_t} + \epsilon}$$

→ m_t be an exponentially decaying average over the past gradients
$$m_t = (1-\beta_1)g_t(\theta_{t-1}) + \beta_1 m_{t-1}$$
corresponds to adding a momentum term directly.

→ Initialization: $m_0 = v_0 = 0$.

⚠ There is a small bias in the equations above, we use instead of m_t and v_t:
$$\tilde{m}_t = \frac{m_t}{1-\beta_1^t} \quad \tilde{v}_t = \frac{v_t}{1-\alpha^t}.$$

Adam - documentation from Pytorch

CLASS torch.optim.Adam(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*, *amsgrad=False*, ***, *foreach=None*, *maximize=False*, *capturable=False*, *differentiable=False*, *fused=None*) [SOURCE]

Implements Adam algorithm.

$$
\begin{aligned}
&\textbf{input} : \gamma \text{ (lr)}, \beta_1, \beta_2 \text{ (betas)}, \theta_0 \text{ (params)}, f(\theta) \text{ (objective)} \\
&\qquad \lambda \text{ (weight decay)}, \ \textit{amsgrad}, \ \textit{maximize} \\
&\textbf{initialize} : m_0 \leftarrow 0 \text{ (first moment)}, v_0 \leftarrow 0 \text{ (second moment)}, \ \widehat{v_0}^{max} \leftarrow 0 \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad \textbf{if } \textit{maximize} : \\
&\qquad g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{else} \\
&\qquad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{if } \lambda \neq 0 \\
&\qquad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big) \\
&\quad \widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big) \\
&\quad \textbf{if } \textit{amsgrad} \\
&\qquad \widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t}) \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big) \\
&\quad \textbf{else} \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big) \\
&\textbf{return } \theta_t
\end{aligned}
$$

Figure: Adam - Pytorch documentation - link

Summary - methods for Deep Learning

Combine

- Coordinate dependent
- Stochastic
- Momentum
- Adaptivity
- Low precision

$$\theta_t \leftarrow \theta_{t-1} - \eta_t \odot g_t(\theta_{t-1}) + \beta_t(\theta_{t-1} - \theta_{t-2})$$

Different recipes: among the most widely used are:

- Adam, Adagrad, RMSProp