Stochastic Gradient Method

Loucas Pillaud-Vivien (ENPC)

June, 2026

Motivation: Finite data set n

Fit $h_\theta(x)$ to learn the input/output function related to $(x_i, y_i)_{1 \leq i \leq n}$.

$$\min_\theta L(\theta) := \frac{1}{n} \sum_{i=1}^{n} \ell(h_\theta(x_i), y_i) \ ,$$

where ℓ is a loss function (e.g. squared loss, logistic loss, etc.).
Gradient descent is too costly as it requires to compute the gradient

$$\nabla L(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(h_\theta(x_i), y_i) \ .$$

But we can use an estimator of the gradient by picking a random point (x_i, y_i) from the dataset, and compute

$$\hat{g}(\theta) = \nabla_\theta \ell(h_\theta(x_i), y_i) \ .$$

Then we can use this estimator to update the parameter θ as

$$\theta^{(k+1)} = \theta^{(k)} - \alpha \hat{g}(\theta^{(k)}) \ ,$$

where α is a step size.

Population data set: $n = +\infty$. Want to learn a function from a model

$$y = f^*(x) + \sigma\xi \ ,$$

where $x \sim \mathcal{P}$ and $\xi \sim \mathcal{N}(0, Id)$ and have access to infinite pairs (x, y).
One idea is to minimize the population risk

$$\min_{\theta} L(\theta) = \mathbb{E}\Big[\ell(h_\theta(x), y)\Big] = \int \ell(h_\theta(x), y) dP(x, y) \ .$$

If computing the population risk is not possible, we can use a random sample $(x, y) \sim P(x, y)$ from the population to estimate the gradient

$$\hat{g}(\theta) = \nabla_\theta \ell(h_\theta(x), y) \ .$$

Then we can use this estimator to update the parameter θ as

$$\theta^{(k+1)} = \theta^{(k)} - \alpha \hat{g}(\theta^{(k)}) \ ,$$

where α is a step size.

Why should I bother to learn this stuff?

This is the main algorithm principle for training machine learning model, and in particular deep neural network.

Useful for

- understanding how the library train ML models
- specialization in optimization
- specialization in machine learning

A general succesful paradigm that dates from the 50's that has been used in many fields.

The abstract optimization problem

We consider the following optimization problem

$$\underset{\theta \in \mathbb{R}^p}{\text{Min}} \qquad F(\theta) := \mathbb{E}_{\xi}\Big[f(\theta, \xi)\Big]$$

where ξ is a random variable.
Typically,

- ξ is a random vector that represents the law of the data
- We assume we can sample ξ, but not compute the expectation exactly.
- $f(\theta, \xi)$ is a loss function that measures the quality of the model parameterized by θ on the data ξ

Computing the gradient

We want to use **first order methods** to solve the minimization problem

$$F(\theta) := \mathbb{E}\Big[f(\theta, \boldsymbol{\xi})\Big]$$

Under some regularity conditions (e.g. $f(\cdot, \xi)$ differentiable, $\dfrac{\partial f(\theta, \cdot)}{\partial \theta}$ Lipschitz in θ uniformly in ξ, and $\boldsymbol{\xi}$ integrable) we have

$$\nabla F(\theta) = \mathbb{E}\left[\frac{\partial f}{\partial \theta}(\theta, \boldsymbol{\xi})\right]$$

This is obvious if $\boldsymbol{\xi}$ is finitely supported : $\mathrm{supp}(\boldsymbol{\xi}) = \{\xi_i\}_{i \in [n]}$, and $\mathbb{P}(\boldsymbol{\xi} = \xi_i) = \frac{1}{n}$,

$$\nabla F(\theta) = \frac{\partial}{\partial \theta}\left(\sum_{i=1}^{n} \frac{1}{n} f(\theta, \xi_i)\right) = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial \theta} f(\theta, \xi_i)$$

Standard continuous optimization method

Thus, we are looking at

$$\underset{\theta \in \mathbb{R}^p}{\text{Min}}\ F(\theta)$$

where F is a convex differentiable function if $f(\cdot, \xi)$ is, and we know how to compute its gradient.

Thus, we should be able to solve our problem through the method presented in earlier courses:

- gradient algorithm
- conjugate gradient
- Newton / Quasi-Newton

Why bother with another (class of) algorithm?

Computing the gradient is costly

For a given solution $\theta \in \mathbb{R}^p$ computing the gradient

$$\nabla F(\theta) = \mathbb{E}\Big[\frac{\partial f(\theta, \boldsymbol{\xi})}{\partial \theta}\Big]$$

is costly as it requires to compute a multidimensionnal integral (if $\boldsymbol{\xi}$ admits a density), or a large sum.

Indeed, in most machine learning application, we consider that $\boldsymbol{\xi}$ is uniformly distributed over the data (*empirical risk minimization*), thus computing the gradient require a pass over every sample in the dataset.

Dataset of size $N > 10^6$ are common.

Estimating the gradient

Instead of using a true gradient

$$g^{(k)} = \nabla F(x^{(k)})$$

we can use a *statistical estimator* of the gradient

$$\hat{g}^{(k)} \rightsquigarrow g^{(k)} = \mathbb{E}\Big[\frac{\partial f(\theta^{(k)}, \boldsymbol{\xi})}{\partial \theta}\Big]$$

The most standard estimator being

$$\hat{g}^{(k)} = \frac{\partial f(\theta^{(k)}, \xi^{(k)})}{\partial \theta}$$

where $\xi^{(k)}$ is drawn randomly according to the law of $\boldsymbol{\xi}$ (i.e. it is a random datapoint).

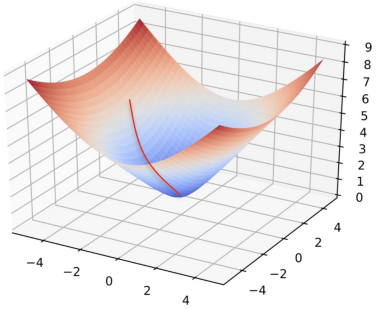
Wraping-up: GD versus SGD

Recall we want to solve the problem

$$\underset{\theta \in \mathbb{R}^p}{\text{Min}}\, F(\theta) = \mathbb{E}\Big[f(\theta, \xi)\Big].$$

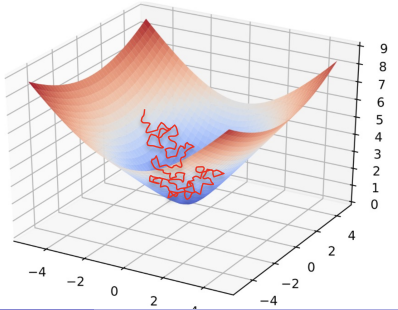
Gradient descent

$$\theta^{(k+1)} = \theta^{(k)} - \alpha\, \mathbb{E}\Big[\frac{\partial f(\theta^{(k)}, \xi)}{\partial \theta}\Big]$$

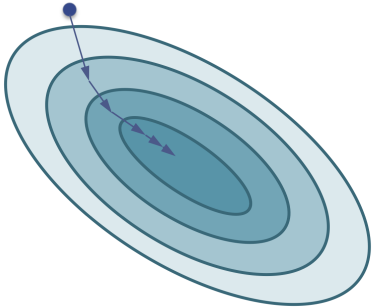


Stochastic gradient descent

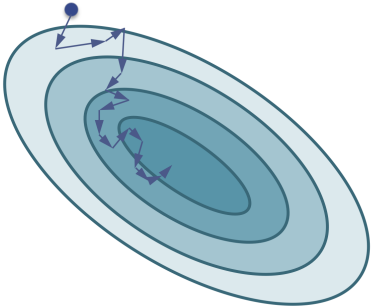
$$\theta^{(k+1)} = \theta^{(k)} - \alpha\, \frac{\partial f(\theta^{(k)}, \xi^{(k)})}{\partial \theta}$$



A noisy trajectory



Gradient Descent



Stochastic Gradient Descent

Pros and Cons

Pros:

- computing $\hat{g}^{(k)} = \frac{\partial f(\theta^{(k)}, \xi^{(k)})}{\partial \theta}$ is really easy
- we do not need to spend lots of time early to get a precise gradient
- we can stop whenever we want (do not need a full pass on the data)
- Noise can help escape poor local minima or saddle points.

Cons:

- $\hat{g}^{(k)}$ is a noisy estimator of the gradient
- requires a new convergence theory
- $\theta^{(k+1)} := \theta^{(k+1)} - \alpha \hat{g}^{(k)}$ generally does not converges almost surely to the optimal solution as this make a noisy trajectory

Convergence intuition for SGD

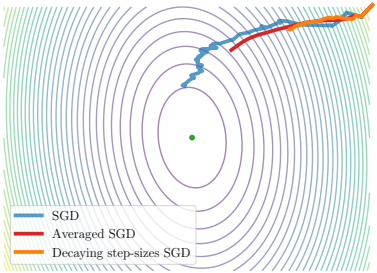
We consider the stochastic update

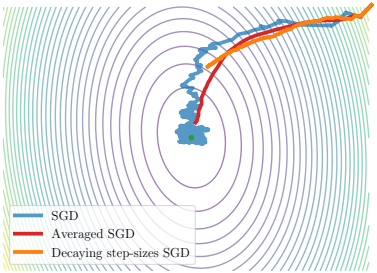
$$\theta^{(k+1)} = \theta^{(k)} - \alpha^{(k)} \hat{g}^{(k)}, \qquad \mathbb{E}[\hat{g}^{(k)} \mid \theta^{(k)}] = \nabla F(\theta^{(k)}).$$

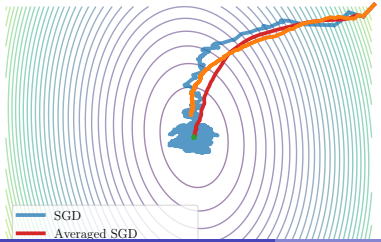
- **Constant step size** ($\alpha^{(k)} = \alpha$): fast initial progress, but the noise does not vanish.
  $\Rightarrow$ under standard assumptions, the iterates typically keep fluctuating and performance stabilizes in a neighborhood of the optimum (accuracy $\sim O(\alpha)$).
- **Decreasing step size** ($\alpha^{(k)} \searrow 0$, e.g. $\alpha^{(k)} \sim 1/k$): the noise is gradually damped.
  $\Rightarrow$ under standard assumptions, convergence to the optimum is possible, but the progress becomes slower.
- **Iterate averaging** (Polyak–Ruppert): average the iterates $\bar{\theta}^{(k)} = \frac{1}{k} \sum_{t=1}^{k} \theta^{(t)}$.
  $\Rightarrow$ improves stability and, in classical strongly convex settings, can achieve optimal asymptotic rates.

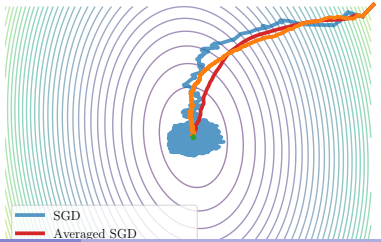
Trade-off: speed vs accuracy vs stability.

Noisy trajectory









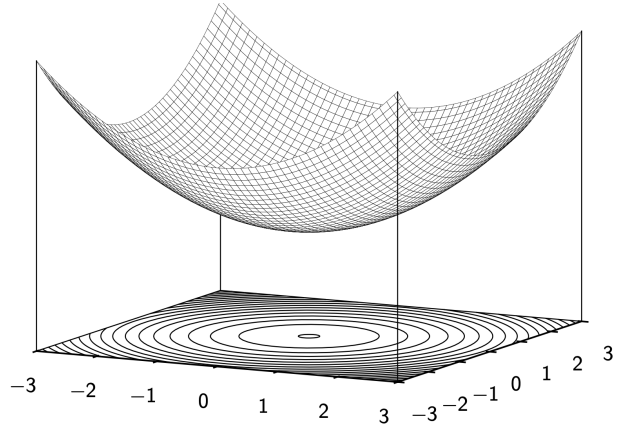
Mini-batch version

- $\hat{g}^{(k)} = \frac{\partial f(x^{(k)}, \xi^{(k)})}{\partial \theta}$ is easy to compute but noisy estimator of the gradient
- $\hat{g}^{(k)} = \frac{1}{n} \sum_{i \in [n]} \frac{\partial f(x^{(k)}, \xi_i)}{\partial \theta}$ is long to compute but perfect estimator
- minibatch aims at a middle ground : randomly draw a sample S of realizations of $\boldsymbol{\xi}$, and use

$$\hat{g}^{(k)} = \frac{1}{|S|} \sum_{\xi \in S} \frac{\partial f(x^{(k)}, \xi)}{\partial \theta}$$

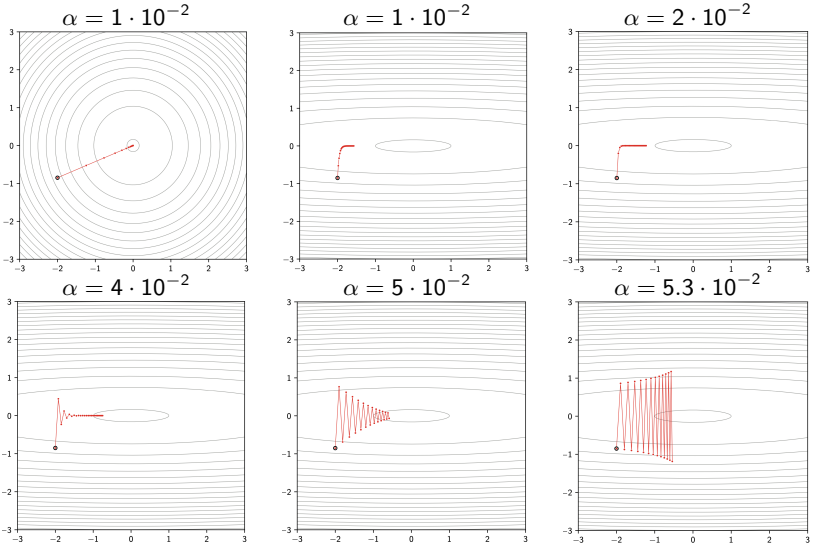
Accelerated methods

The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.



Accelerated methods

The gradient descent method makes a strong assumption about the magnitude of the "local curvature" to fix the step size, and about its isotropy so that the same step size makes sense in all directions.



Momentum methods

- Some optimization methods leverage higher-order information, in particular second order (Hessians) to use a more accurate local model of the functional to optimize. **They are too costly to use in practice.**
- However for a fixed computational budget, the complexity of these methods reduces the total number of iterations, and the eventual optimization is worse.
- Deep-learning generally relies on a smarter use of the gradient, using statistics over its past values to make a "smarter step" with the current one. **They are called momentum methods.**

The first improvement is the use of a "momentum" to add inertia in the choice of the step direction

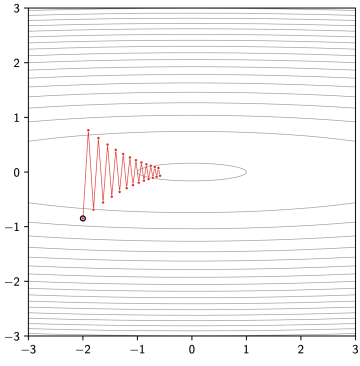
$$\eta^k = \gamma \eta^{k-1} + \alpha \widehat{g}^k$$
$$\theta^{(k+1)} = \theta^{(k)} - \eta^k \ ,$$

where $\widehat{g}^k$ is an estimator of the gradient at $\theta^{(k)}$ and $\gamma \in [0, 1]$ is a momentum parameter.
When $\gamma = 0$, this is the stochastic gradient descent method.
When $\gamma > 0$, this has nice properties:

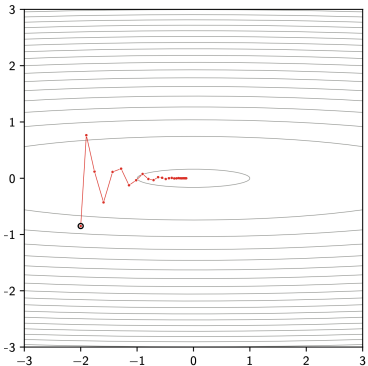
- it is a first order method
- it has a memory of the past gradients and can go through local barriers
- accelerates if the gradient does not change too much : if no change new step size is $\alpha/(1-\gamma)$
- it dampens oscillations in narrow valleys

Accelerated methods

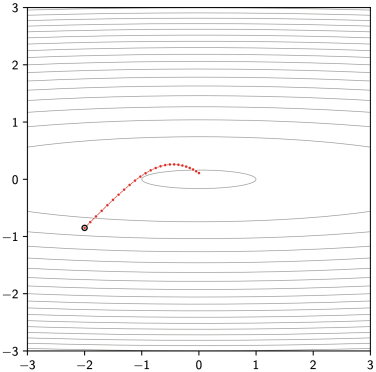
$\alpha = 5 \cdot 10^{-2},\ \gamma = 0$



$\alpha = 5 \cdot 10^{-2},\ \gamma = 0.5$



Adam: momentum method using first and second moments to adapt the step size per direction.



What you have to know

When gradient descent is good but too costly, we can use stochastic gradient descent, which is a first order method that uses a noisy estimator of the gradient.

- Reduces (a lot!) the computational cost of the algorithm
- Allows to use large datasets
- Be carefull with the convergence theory, as the trajectory is noisy
- Can be used with a decreasing step size or averaging the points
- Can be used with a mini-batch version to reduce the noise
- Can be accelerated with inertia methods
- Is a modern trick you need to think about!

What you really should know

- GD vs SGD: full gradient vs stochastic estimates via cheaper computations; mini-batch is the middle ground.
- Step-size matters: constant vs decreasing schedules; iterate averaging stabilizes noisy trajectories.
- Momentum and adaptive methods: inertia to accelerate and damp oscillations; Adam uses first/second moments.
- Practical trade-offs: batch size controls variance vs cost.

What you have to be able to do

- Implement a basic SGD/minibatch loop and parameter update.
- Tune momentum and batch size; read loss curves to detect divergence/oscillation.
- Use decreasing step sizes or iterate averaging to improve stability/convergence.

What you should be able to do

- Explain convergence intuition under noise and how step-size schedules affect it.
- Design experiments comparing GD, SGD, and mini-batch on a dataset.
- Choose momentum/Adam settings and justify when each helps.
- Diagnose/mitigate issues: noisy plateaus, narrow valleys, poor local minima.