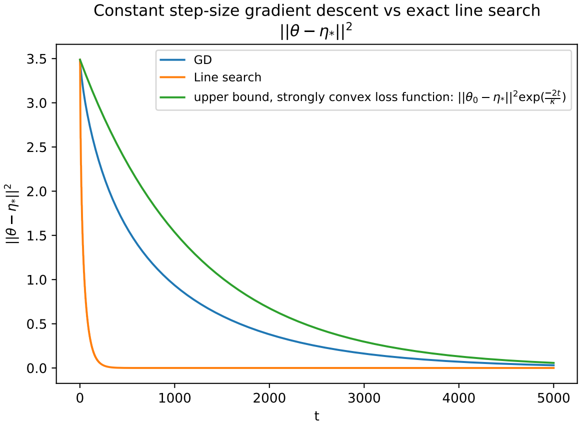
# Machine learning I, supervised learning: gradient algorithms



# Minimization of functions

In machine learning, we face **function minimization problems**. Typically, the function to minimize if the empirical risk on the train set, that will typically depend on a parameter $\theta \in \mathbb{R}^d$.

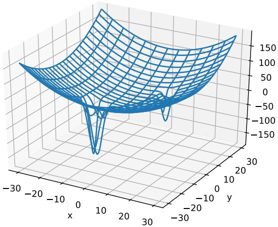



Figure – Example of a function to minimize

# Analytic minimum

What is the minimum of the function

$$f : x \to (x - 1)^2 + 3.5 \tag{1}$$

And for what value $x$ is it obtained ?

# Context

In machine learning, we often encounter problems in high dimension, where closed-form solutions to the **empirical risk minimization** problem are **not** available (e.g. for logistic regression), or where even if they are available, the necessary computation time is too large (OLS).

# Context

In machine learning, we often encounter problems in high dimension, where closed-form solutions are not avaible, or where even if they are available, the necessary computation time is too large.

**Example 1 :** Computing the OLS estimator requires a matrix inversion, which is $\mathcal{O}(d^3)$.

$$\hat{\theta} = (X^T X)^{-1} X^T y \tag{2}$$

# Context

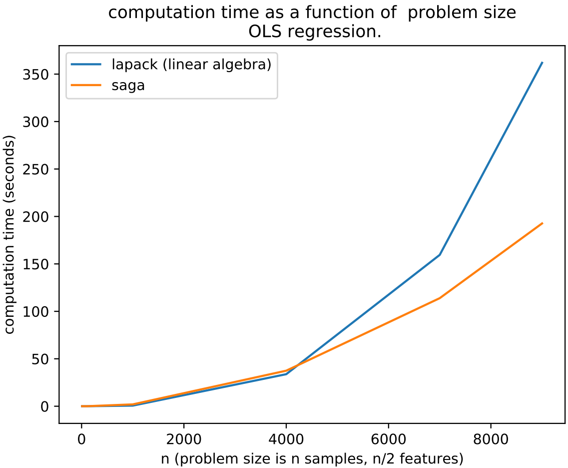
In machine learning, we often encounter problems in high dimension, where closed-form solutions are not avaible, or where even if they are available, the necessary computation time is too large.

**Example 2 :** The cancellation of the gradient of the objective function with logistic loss has no closed-form solution.

# Context

Instead, we often use **iterative** algorithm such as Gradient descent (GD) or Stochastic gradient descent (SGD). SGD is the standard optimization algorithm for large-scale machine learning.

# SGD vs Lapack



# Derivation and variation

- In the case a function $f : \mathbb{R} \to \mathbb{R}$, we can study its variations by computing its derivative $f'$, **if it exists**
- If $f'(x) > 0$, the function grows around $x$.
- If $f'(x) < 0$, the function decreases around $x$.
- If $x$ is a local extremum, $f'(x) = 0$
- Is the reciprocal true ?

## One dimensional functions

If $f : \mathbb{R} \mapsto \mathbb{R}$ is differentiable (dérivable) :
(Landau notation)

$$f(a+h) = f(a) + hf'(a) + o(h) \tag{3}$$

Physicist notation :

$$f(a+h) \simeq f(a) + hf'(a) \tag{4}$$

# Gradient update

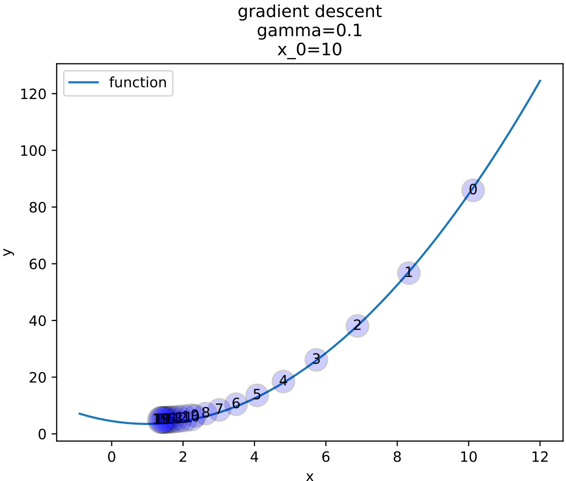
In one dimension :

$$x \leftarrow x - \gamma f'(x) \tag{5}$$

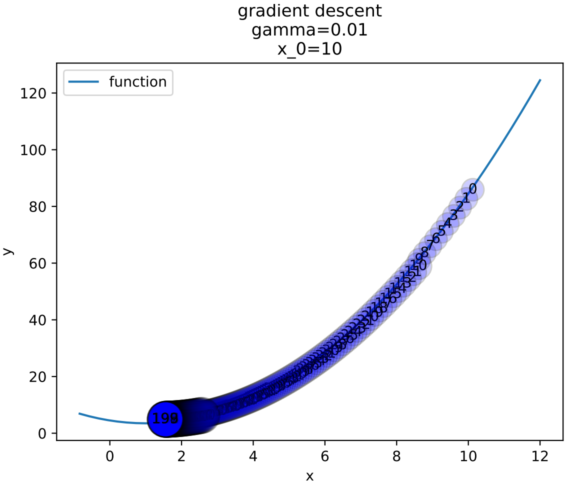
- $\leftarrow$ means "is substituted by".
- $\gamma > 0$ is a real number called the **learning rate** (hyperparameter of the algorithm).

# Examples in 1 dimension

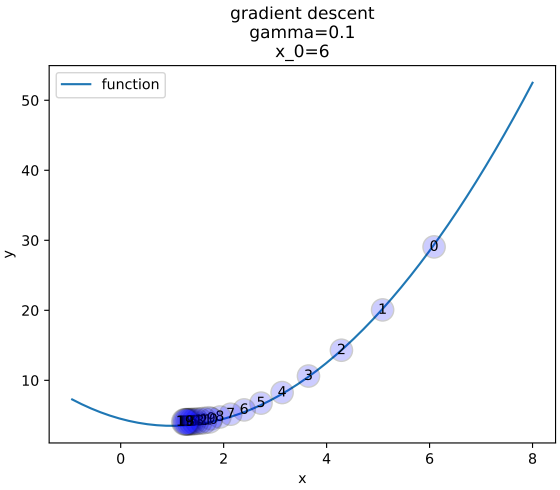
In the following examples, we use the script **gradients/1d_function.py**



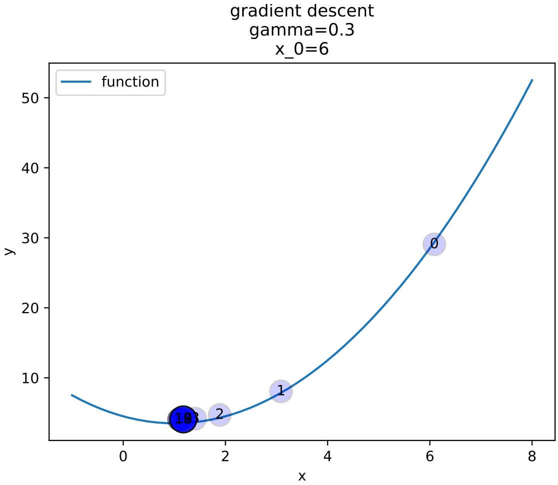
# Examples in 1 dimension



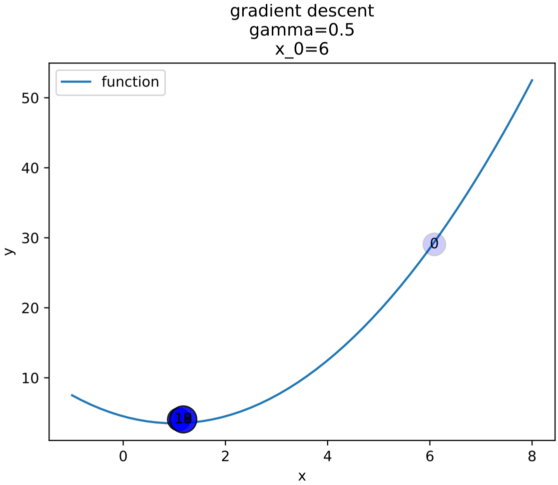
# Examples in 1 dimension



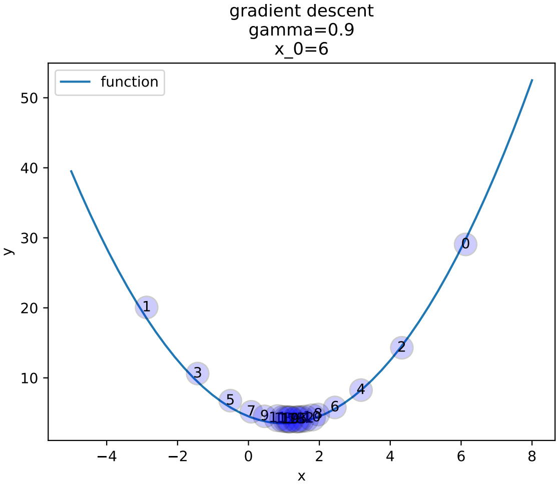
# Examples in 1 dimension



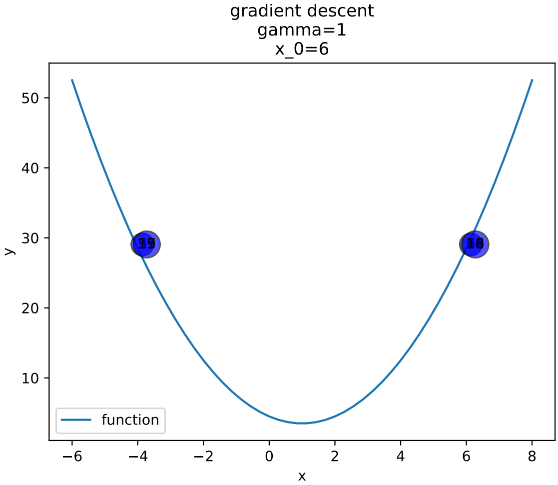
# Examples in 1 dimension



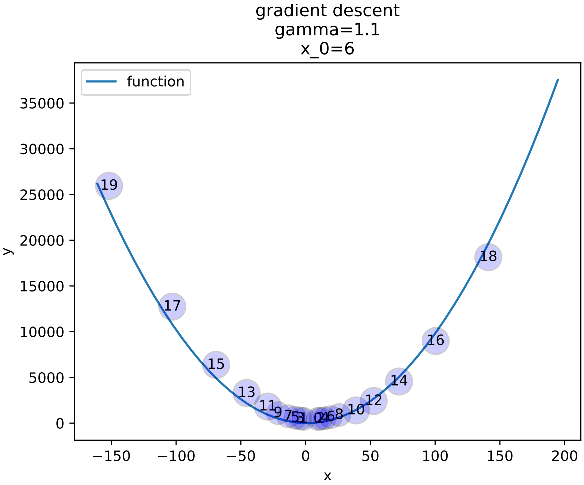
# Examples in 1 dimension



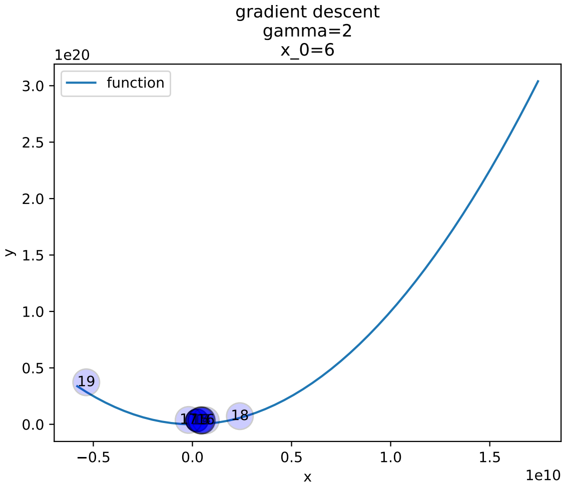
# Examples in 1 dimension



# Examples in 1 dimension



# Examples in 1 dimension



# Gradients

The **gradient** is the generalization of the derivative to functions with more than 1 input variable.
Example : consider a function $f$ that has 2 parameters as inputs. If $f$ is differentiable, the gradient writes :

$$\nabla f(x, y) = (\frac{\delta f}{\delta x}(x, y), \frac{\delta f}{\delta y}(x, y)) \tag{6}$$

$\frac{\delta f}{\delta x}(x, y)$ is the **partial derivative** with respect to $x$, commputed in $(x, y)$.

# Example gradient

If

$$f(x, y) = x^2 + 3xy + 1 \tag{7}$$

Then

$$\forall (x, y) \in \mathbb{R}^2, \nabla f(x, y) = (2x + 3y, 3x) \tag{8}$$

# Multiple-variable functions

If $f : \mathbb{R}^d \mapsto \mathbb{R}$ is differentiable (dérivable) :
(Landau notation)

$$f(\theta + h) = f(\theta) + \langle \nabla f(\theta) | h \rangle + o(h) \tag{9}$$

Physicist notation :

$$f(\theta + h) \simeq f(\theta) + \langle \nabla f(\theta) | h \rangle \tag{10}$$

# Gradient descent

$$\theta \leftarrow \theta - \gamma \nabla f(\theta) \tag{11}$$

$\gamma$ must be carefully chosen.

- too large $\gamma$ : the minimization might not work
- too small $\gamma$ : the minimization will be too slow

## Gradient descent algorithm summary :

- In one dimension ($x \in \mathbb{R}$) :

$$x \leftarrow x - \gamma f'(x) \tag{12}$$

- In $d > 1$ dimensions, $\theta \in \mathbb{R}^d$ :

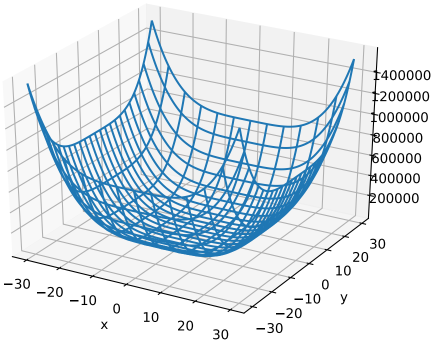
$$\theta \leftarrow \theta - \gamma \nabla f(\theta) \tag{13}$$

- $\leftarrow$ means "is substituted by".
- $\gamma > 0$ is a the **learning rate**.

# Gradient

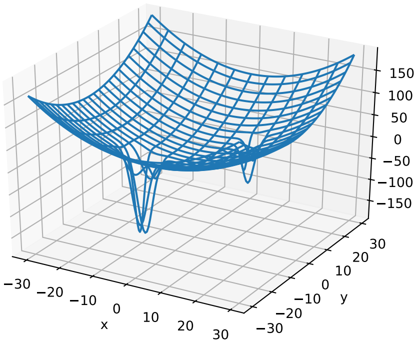
**Exercice 1 : Implementing the gradient algorithm in $\mathbb{R}^2$**

We will use the algorithm on two functions defined over $\mathbb{R}^2$.



# Gradient

Exercice 1 : **Implementing the gradient algorithm in $\mathbb{R}^2$** We will use the algorithm on two functions defined over $\mathbb{R}^2$.

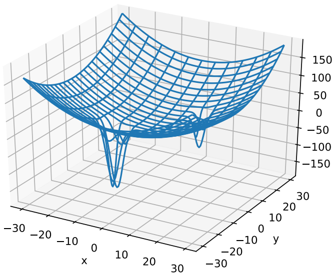


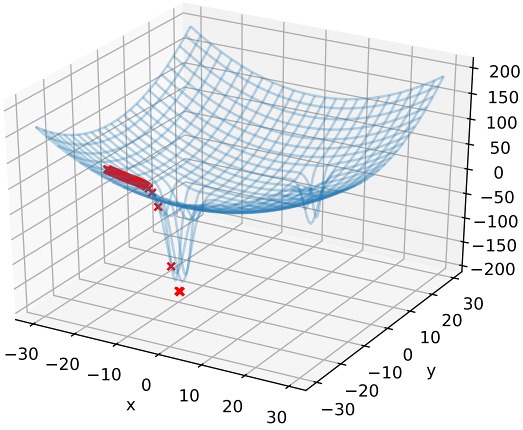
# Gradient

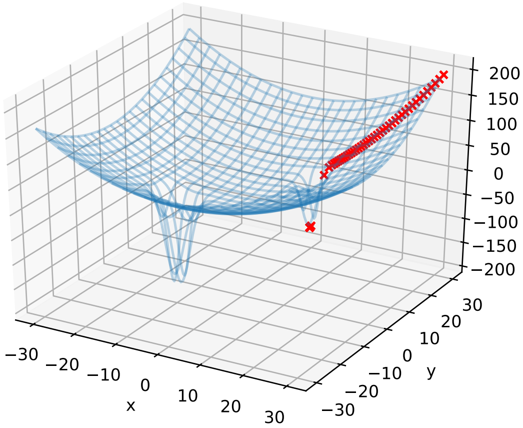
Exercice 1 : **Implementing the gradient algorithm**

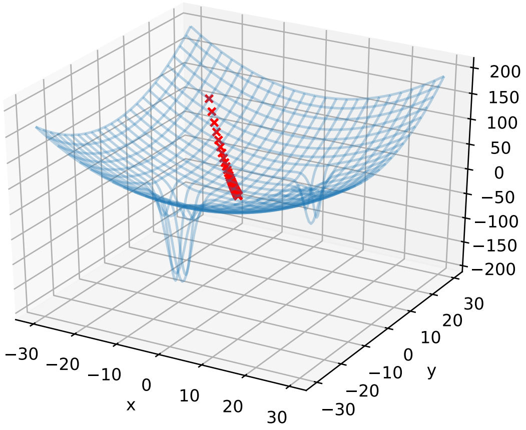
**cd ./gradient** and use the files **gradient_algo_1.py** and **gradient_algo_2.py** in order to implement the algorithm to find **minima**.

Experiment with all the parameters that you consider relevant (several are) to assess their impact on the algorithm.




200
150
100
50
0
−50
−100
−150
−200
−30
−20
−10
0
10
20
30
x
30
20
10
0
−10
−20
−30
y


200
150
100
50
0
−50
−100
−150
−200
−30
−20
−10
0
10
20
30
x
30
20
10
0
−10
−20
−30
y


200
150
100
50
0
−50
−100
−150
−200
−30
−20
−10
0
10
20
30
x
30
20
10
0
−10
−20
−30
y

# Convergence speed

For some problems, it is possible to have garantees on the convergence speed of gradient descent. The results will depend on the following properties of the objective function :

- convexity or strong convexity
- smoothness (Lipshitz-continuous gradients) or non-smoothness
- condition number

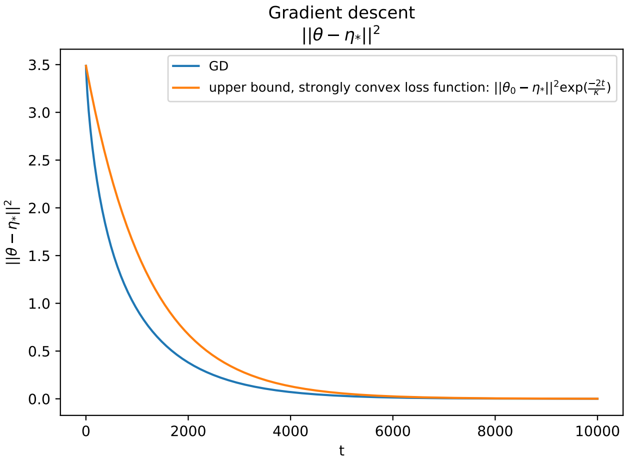
# Smoothness (Example of theoretical criterion)

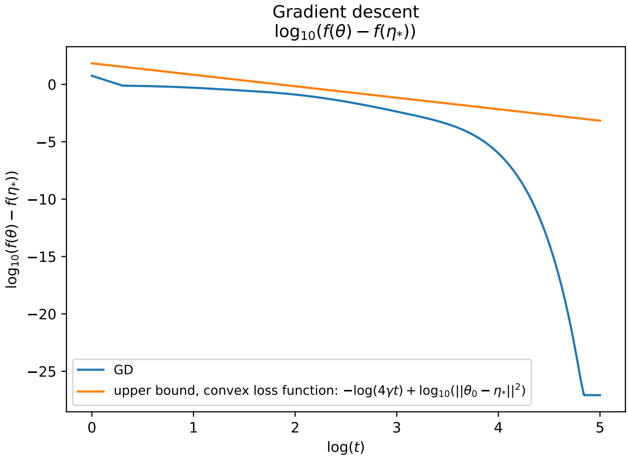
## Définition

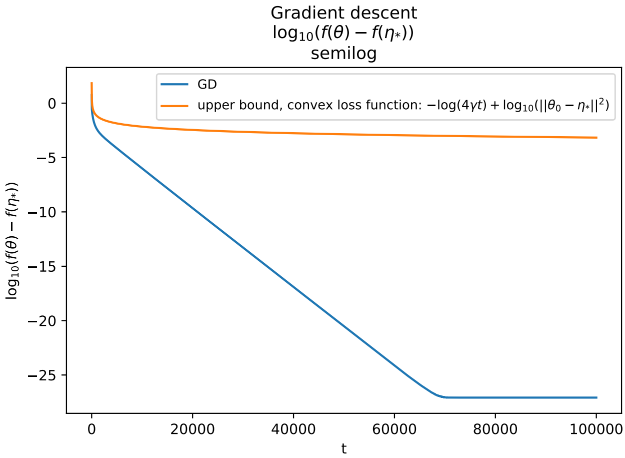
Smoothness

A differentiable function $f$ with real values is said $L$-smooth if and only if

$$\forall x, y \in \mathbb{R}^d, |f(y) - f(x) - \nabla_x f(y - x)| \leq \frac{L}{2} ||y - x||^2$$

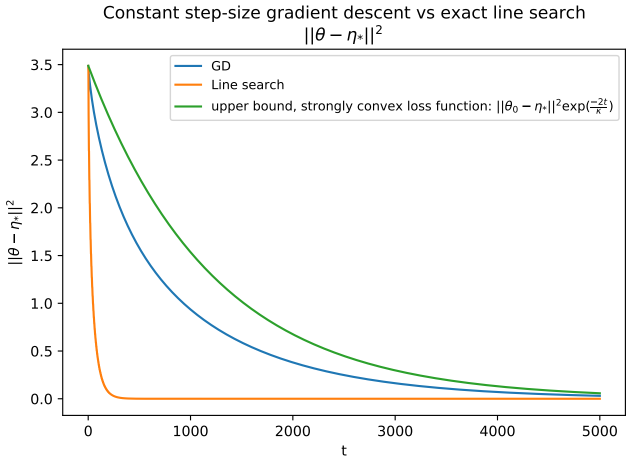

Gradient descent
$||\theta - \eta_*||^2$
GD
upper bound, strongly convex loss function: $||\theta_0 - \eta_*||^2 \exp(\frac{-2t}{\kappa})$
$||\theta - \eta_*||^2$
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000
t


Gradient descent
$\log_{10}(f(\theta) - f(\eta_*))$
0
−5
−10
−15
−20
−25
$\log_{10}(f(\theta) - f(\eta_*))$
GD
upper bound, convex loss function: $-\log(4\gamma t) + \log_{10}(||\theta_0 - \eta_*||^2)$
0
1
2
3
4
5
$\log(t)$


Gradient descent
$\log_{10}(f(\theta) - f(\eta_*))$
semilog
GD
upper bound, convex loss function: $-\log(4\gamma t) + \log_{10}(||\theta_0 - \eta_*||^2)$
0
−5
−10
−15
−20
−25
$\log_{10}(f(\theta) - f(\eta_*))$
0
20000
40000
60000
80000
100000
t

# Extensions

- Line search
- Nesterov accleration (optimal rates among algorithms that linearly combine gradients)


Constant step-size gradient descent vs exact line search
$||\theta - \eta_*||^2$
GD
Line search
upper bound, strongly convex loss function: $||\theta_0 - \eta_*||^2 \exp(\frac{-2t}{\kappa})$
$||\theta - \eta_*||^2$
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1000
2000
3000
4000
5000
t

# Stochastic gradient descent

In machine learning, we often consider an objective function of the form

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f_\theta(x_i)) + \Omega(\theta) \tag{14}$$

# Batch gradient

In machine learning, we often consider an objective function of the form

$$f(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f_\theta(x_i)) + \Omega(\theta) \tag{15}$$

Computing the gradient of $f$ requires at least $n$ calculations, and each calculation also has a complexity that depends on the dimension $d$. When $n$ and $d$ are large, this can be quite slow.

# Stochastic gradient descent

We consider an objective function of the form

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f_\theta(x_i)) + \Omega(\theta) \tag{16}$$

Instead of computing the **batch gradient** $\nabla_\theta f$, we will compute :

$$u(i, \theta) = \nabla_\theta [l(y_i, f_\theta(x_i)) + \Omega(\theta)] \tag{17}$$

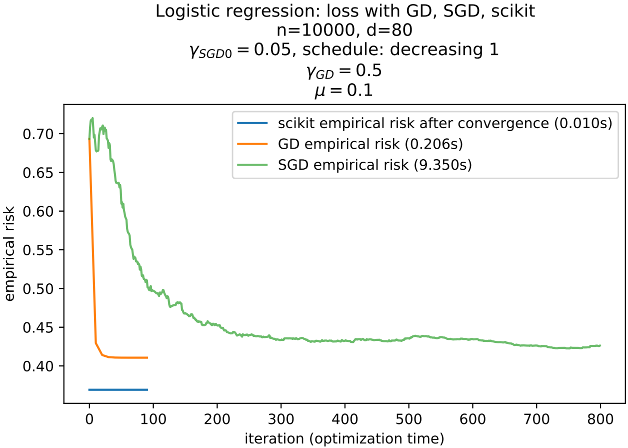
for a randomly sampled $i \in [1, n]$.
$u(i, \theta)$ is an **estimation** of the full (batch) gradient $\nabla_\theta f$.

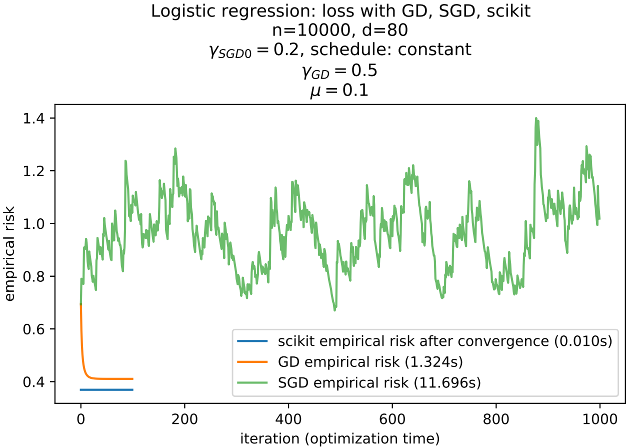
# Tradeoff

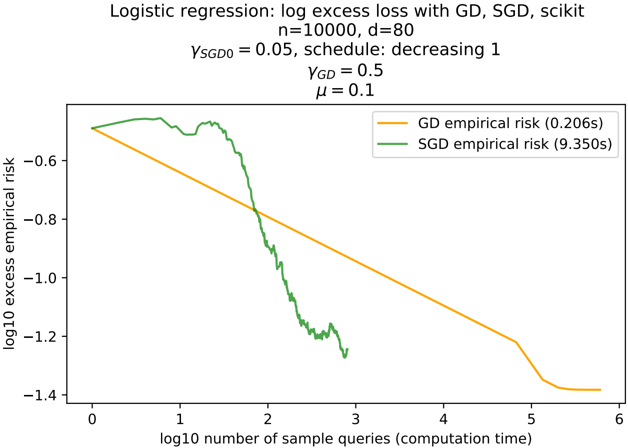
By replacing the computation of the full gradient (GD) by this estimation (SGD) :

- we reduce the computation time (divide it by $n$)
- but reduce precision of the optimization

To summarize : if $n$ is really large, SGD is often better.


Logistic regression: loss with GD, SGD, scikit
n=10000, d=80
$\gamma_{SGD0} = 0.05$, schedule: decreasing 1
$\gamma_{GD} = 0.5$
$\mu = 0.1$
scikit empirical risk after convergence (0.010s)
GD empirical risk (0.206s)
SGD empirical risk (9.350s)
empirical risk
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0
100
200
300
400
500
600
700
800
iteration (optimization time)


Logistic regression: loss with GD, SGD, scikit
n=10000, d=80
$\gamma_{SGD0} = 0.2$, schedule: constant
$\gamma_{GD} = 0.5$
$\mu = 0.1$
empirical risk
1.4
1.2
1.0
0.8
0.6
0.4
0
200
400
600
800
1000
iteration (optimization time)
scikit empirical risk after convergence (0.010s)
GD empirical risk (1.324s)
SGD empirical risk (11.696s)


Logistic regression: log excess loss with GD, SGD, scikit
n=10000, d=80
$\gamma_{SGD0} = 0.05$, schedule: decreasing 1
$\gamma_{GD} = 0.5$
$\mu = 0.1$
GD empirical risk (0.206s)
SGD empirical risk (9.350s)
log10 excess empirical risk
−0.6
−0.8
−1.0
−1.2
−1.4
0
1
2
3
4
5
6
log10 number of sample queries (computation time)

# Comparison (advanced)

The ridge regression problem is smooth and strongly convex.

- GD has a convergence rate of $\mathcal{O}(\exp(-\frac{t}{\kappa}))$. To get an error of $\epsilon$, we must have $t = \mathcal{O}(\kappa \log \frac{1}{\epsilon})$. Since each iteration requires $\mathcal{O}(nd)$ computations, the computation time will be $\mathcal{O}(\kappa nd \log \frac{1}{\epsilon})$.
- SGD has a convergence rate of $\mathcal{O}(\frac{\kappa}{t})$. To get an error of $\epsilon$, we must have $t = \mathcal{O}(\frac{\kappa}{\epsilon})$. Since each iteration is $\mathcal{O}(d)$, we have a computation time of $\mathcal{O}(\frac{\kappa d}{\epsilon})$.

# Comparison (advanced)

As a consequence :

- When $n$ is large and $\epsilon$ not too small, GD will need more computation time to reach error $\epsilon$. An order of magnitude can be obtained by studying the value $\epsilon^*$ such that

$$\kappa n d \log \frac{1}{\epsilon^*} = \frac{\kappa d}{\epsilon^*}$$

  Which translates to

$$\epsilon^* \log \epsilon^* = -\frac{1}{n}$$

- When $\epsilon \to 0$, GD becomes faster than SGD to reach this precision.

# Conclusion

For lower precision and large $n$, SGD is a preferable.
In machine learning, due to the estimation error that is $\mathcal{O}(\frac{1}{\sqrt{n}})$, a very high precision is often not needed

# Extensions of SGD

See also :

- Variance reduction methods (SAG, SAGA)