Python For Finance

M1 - Economie & Finance - Gradient Descent

Université Paris Dauphine

October 2025

Part I — Definition

For a function $f(x_1, \ldots, x_n)$, the gradient is the vector of partial derivatives:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right).$$

The gradient indicates the direction of the steepest local increase.

Its norm, $\|\nabla f\|$, is the local rate of increase (its magnitude / intensity).

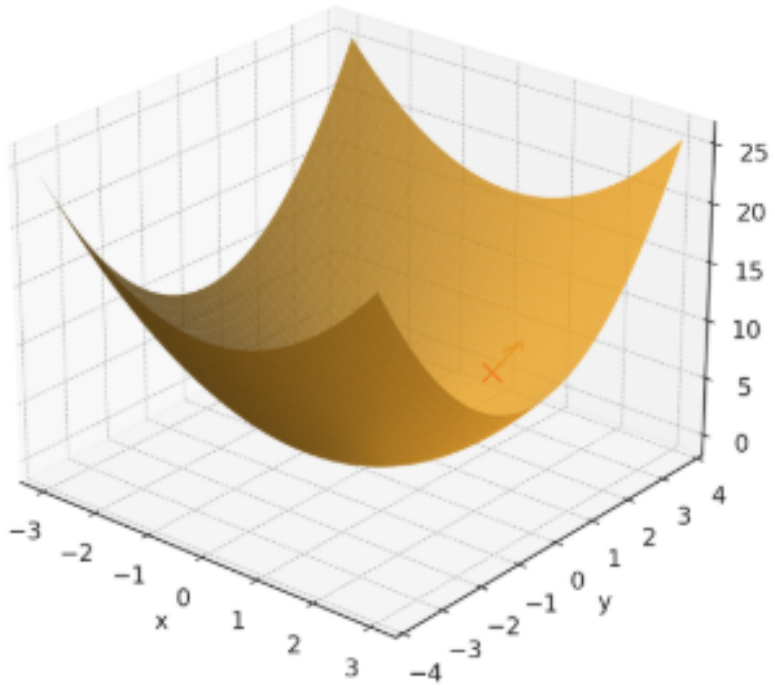
Part I — Definition (example)

Let $f(x, y) = x^2 + y^2$. Its gradient is

$$\nabla f(x, y) = (2x,\ 2y).$$

At the point $(x, y) = (1, 2)$**:**

$$\nabla f(1, 2) = (2 \cdot 1,\ 2 \cdot 2) = (2,\ 4), \qquad \left\|\nabla f(1, 2)\right\| = \|(2, 4)\| = \sqrt{2^2 + 4^2} = \sqrt{20}.$$



Surface $z = x^2 + y^2$ (gradient at $(1, 2)$ has direction $(2, 4)$ and magnitude $\sqrt{20}$)

Part II — Gradient Descent

Gradient descent is an iterative optimization method for minimizing a function $f(\theta)$.

At each step, we move the parameters θ in the direction opposite to the gradient (the steepest-ascent direction) because that yields the steepest decrease of f.

Typical update:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta f(\theta_t)$$

where $\alpha > 0$ is the learning rate and $\nabla_\theta f$ the gradient.

Part II — Multivariate example

Suppose we have a dataset giving the living areas, the number of bedrooms and prices for houses:

Living area (ft^2)	**#bedrooms**	**Price (1000$)**
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
$\vdots$	$\vdots$	$\vdots$

Here, the x's are two-dimensional vectors in $\mathbb{R}^2$. For instance, $x_1^{(i)}$ is the living area of the i-th house, and $x_2^{(i)}$ is its number of bedrooms.

Part II — Hypothesis and notation

Let's approximate y (house price) as a linear function of x (living area and nb of bedrooms):

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2.$$

Here, the θ_i are the **parameters** (also called *weights*).

To simplify notation, we drop the θ subscript in $h_\theta(x)$ and simply write $h(x)$ and we also introduce the convention $x_0 = 1$ (the *intercept* term), so that

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^\top x,$$

where on the right-hand side we view θ and x as vectors, and n is the number of input variables (not counting x_0).

Part II — Choosing the parameters θ

How do we pick, or learn, the parameters θ?

One reasonable method is to make $h(x)$ close to y, at least for the training examples we have. To formalize this, define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2.$$

With m, the number of data point in the dataset. You may recognize this as the familiar *least-squares* cost function of the ordinary least squares (OLS) regression model. Indeed, If the matrix $X^\top X$ is invertible, the unique solution is

$$\theta = (X^\top X)^{-1} X^\top y.$$

Part II — Gradient Descent Update

We want to choose θ to minimize $J(\theta)$.

We use a search algorithm that starts from an initial guess and repeatedly changes θ to make $J(\theta)$ smaller, hoping to converge to a minima => **gradient descent**

The gradient descent performs the update simultaneously for all $j = 0, \ldots, n$

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

Here, α is the **learning rate**. This algorithm repeatedly steps in the direction of steepest decrease of J.

Part II — Gradient descent update

To implement the algorithm, we need $\frac{\partial}{\partial \theta_j} J(\theta)$.

For a single data point (x, y), we have:

$$J(\theta) = \tfrac{1}{2}\big(h_\theta(x) - y\big)^2$$

$$\textit{with} \qquad h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i,$$

Thus

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \tfrac{1}{2}\big(h_\theta(x) - y\big)^2 \\
&= \big(h_\theta(x) - y\big)\, \frac{\partial}{\partial \theta_j}\big(h_\theta(x) - y\big) \\
&= \big(h_\theta(x) - y\big)\, x_j.
\end{aligned}$$

Therefore, for a single example $(x^{(i)}, y^{(i)})$ the update rule becomes

$$\boxed{\theta_j \;:=\; \theta_j \;+\; \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)\, x_j^{(i)}}$$

Part II — LMS Update

The rule is called the **LMS** update rule (Least Mean Squares), also known as the **Widrow–Hoff** learning rule.

Per-example update (simultaneous for all j):

$$\boxed{\theta_j \leftarrow \theta_j + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)\, x_j^{(i)}}$$

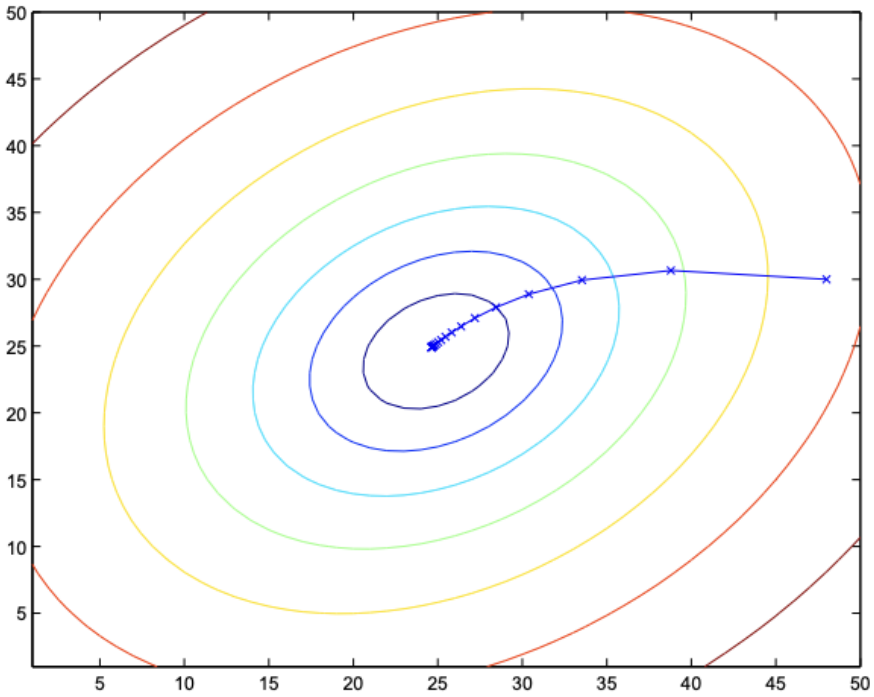
Vector form:

$$\theta \leftarrow \theta + \alpha\,\big(y^{(i)} - h_\theta(x^{(i)})\big)\, x^{(i)}.$$

Why it's intuitive:

- The update size is *proportional to the error* $\big(y^{(i)} - h_\theta(x^{(i)})\big)$.
- If the prediction nearly matches $y^{(i)}$, the change is small; if the error is large, the change is larger.

Part II — LMS Update



trajectory taken by gradient descent

Part I — Batch Gradient Descent (update rule)

Repeat until convergence:

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{for every } j = 0, \ldots, n$$

Batch gradient descent:

- Uses the *entire* data set each step
- The update direction is the negative gradient of the squared-error cost.
- Training large models is computationally intensive.

Part II — Reduce Compute: Use Stochastic Gradient Descent (SGD)

Motivation. Full-batch gradient descent processes all m examples each step $\Rightarrow$ *computationally intensive*. To reduce compute per update, use **SGD**: update with *one example at a time*.

SGD update (squared error), per example $(x^{(i)}, y^{(i)})$:

$$\boxed{\theta \leftarrow \theta + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x^{(i)}}$$

(Component-wise: $\theta_j \leftarrow \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$.*)*

How to run it (presentation):

- Shuffle the training set.
- For each example $(x^{(i)}, y^{(i)})$: compute the error $y^{(i)} - h_\theta(x^{(i)})$ and apply the update.
- Repeat for multiple epochs; optionally decay α.

Part III — Why do we use Gradient descent

Why linear regression with the *least-squares* cost function is a natural choice under some assumptions.

Assumptions.

- Data follow a linear model with noise: $y^{(i)} = \theta^\top x^{(i)} + \varepsilon^{(i)}$.
- Errors are IID Gaussian: $\varepsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$.

Implication. We can write the distribution of y(i) given x(i) and parameterized

by as follows

$$p\left(y^{(i)} \mid x^{(i)}; \theta\right) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{\left(y^{(i)} - \theta^\top x^{(i)}\right)^2}{2\sigma^2}\right)$$

Thus,

$$y^{(i)} \mid x^{(i)}; \theta \sim \mathcal{N}\left(\theta^\top x^{(i)},\, \sigma^2\right)$$

Part III — Likelihood of the Data

Given X (the matrix collecting all $x^{(i)}$) and parameters θ, the distribution of the targets $\tilde{y} = (y^{(1)}, \ldots, y^{(m)})$ is $\mathrm{p}(\ \mid X; \theta)$.

This is typically viewed as a function of $\tilde{y}$ expressed given X for fixed θ.
When we view it as a function of θ, it is called the **likelihood**:

$$L(\theta) \ = \ L(\theta; X, \tilde{y}) \ = \ p(\tilde{y} \mid X; \theta).$$

Assuming the errors $\varepsilon^{(i)}$ are IID Gaussian, we have

$$L(\theta) \ = \ \prod_{i=1}^{m} p\big(y^{(i)} \mid x^{(i)}; \theta\big) \ = \ \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{\big(y^{(i)} - \theta^{\top} x^{(i)}\big)^2}{2\sigma^2} \right).$$

Maximum likelihood principle: choose θ to make the observed data most probable, i.e.,

$$\boxed{\theta^{\star} \ \in \ \arg\max_{\theta} \ L(\theta)}.$$

Part III — Maximizing the Log-Likelihood

In practice we maximize the **log-likelihood** $\ell(\theta)$:

$$\ell(\theta) \;=\; \log L(\theta) = \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(y^{(i)} - \theta^\top x^{(i)})^2}{2\sigma^2} \right).$$

$$\ell(\theta) = \sum_{i=1}^{m} \left[\log \frac{1}{\sqrt{2\pi}\,\sigma} \;-\; \frac{(y^{(i)} - \theta^\top x^{(i)})^2}{2\sigma^2} \right]$$

$$\ell(\theta) = m \log \frac{1}{\sqrt{2\pi}\,\sigma} \;-\; \frac{1}{\sigma^2} \frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \theta^\top x^{(i)})^2.$$

Part III — From MLE to Least Squares

Maximum likelihood principle: choose θ that maximizes $\ell(\theta)$.

Observation. The first term $m \log(1/(\sqrt{2\pi}\sigma))$ does not depend on θ. Therefore,

$$\arg\max_{\theta} \ell(\theta) = \arg\min_{\theta} \sum_{i=1}^{m} \left(y^{(i)} - \theta^\top x^{(i)}\right)^2.$$

Equivalence (up to a constant factor):

$$\boxed{\arg\min_{\theta} \frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \theta^\top x^{(i)}\right)^2 = \arg\max_{\theta} \ell(\theta)}$$

Hence, **least-squares regression** = **maximum likelihood** estimation under IID Gaussian noise.

Part I — Notes and Consequences

- σ^2 **not needed to find** θ**:** the MLE of θ does not depend on the (unknown) noise variance σ^2.
- **Interpretation:** least squares minimizes the sum of squared residuals, which matches the negative log-likelihood under Gaussian noise.
- **Beyond Gaussians:** other noise models $\Rightarrow$ other loss functions (e.g., Laplace $\Rightarrow$ ℓ_1 loss; Bernoulli $\Rightarrow$ logistic loss).
- **Takeaway:** under simple, common assumptions, least squares is a *natural* and *principled* choice via maximum likelihood.

Part IV — Application

Now let's implement code to test the concepts explained above.

Follow the instructions in the Jupyter notebook titled *Gradient Descent*.

For simulating samples from a specified distribution, use the `NumPy` package.

To plot your results, use `Matplotlib`. If you need help, consult the official documentation.

Part IV — Progress Bars with `tqdm`

`tqdm` is a Python library that provides progress bars. Wrap any iterable with `tqdm(`*`iterable`*`)` to display live progress while your loop runs.

Example):

```
from tqdm import tqdm
import time

for i in tqdm(range(100), desc="Working"):
    time.sleep(0.01)  # your work here
```