Alma Mater Studiorum · University of Bologna

Statistical and Mathematical Methods for Artificial Intelligence

Optimization via Gradient Descent and Stochastic Gradient Descent

Lorenzo Cellini (lorenzo.cellini3@studio.unibo.it)

February 10, 2022

Contents

1 Summary

Mathematical optimization is the selection of a best element, with regard to some criterion, from some set of available alternatives. In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values, from within an allowed set, and computing the value of the function. More generally, optimization includes finding "best available" values of some objective function given a defined domain (or input), including a variety of different types of objective functions and different types of domains.

Depending on the function and on the real problem, optimization could mean both finding the local/global minimum or finding the local/global maximum, but for historical reasons, the accepted convention is to always talk about minimization and so, when facing a maximization problem, the first step is to change the sign of the function in order to transform it into a minimization one. However the two approaches are completely equivalent and also procedures to solve them are the same. The reason why minimization is the most common approach is related to the idea to minimize an error function when making a numerical approximation.

In this exercise two optimization technique have been explored: Gradient Descent (GD) and Stochastic Gradient Descent (SGD). Both of them are iterative methods for finding local minimum for differentiable functions, being the second a stochastic approximation of the first one.

This laboratory session consists of two distinct exercises: in the first one Gradient Descent is applied to four different functions and performances are compared for different algorithm parameters; in the second one Stochastic Gradient Descent is applied to the least square loss function in order to approximate the transformation matrix that map a random generated dataset to the true output vector.

2 Background theory

2.1 GD

An optimization problem can be formulated as: given a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, with $f \in C^1$ compute the element x^* such that

$$x^* = \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} f(x) \tag{2.1}$$

Such an x^* is a local minimum if, given $\epsilon > 0$

$$\forall x \in \mathbb{R}^n : ||x - x^*|| \leq \epsilon \Rightarrow f(x^*) \leq f(x) \tag{2.2}$$

Gradient Descent and Stochastic Gradient Descent are two techniques employed to solve such a problem. The idea is to compute the gradient of the function in an initial point, move against gradient direction (that points where the function increases) with a predefined step length and repeat until convergence is reached. Convergence here means that the increasing in the goodness of the solution is less than a certain treshold, being one of the algorithm parameters.

Gradient Descent iteration can be represented as: given an initial point $x_0 \in \mathbb{R}^n$ and a positive step length α_k, in each iteration compute

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) \tag{2.3}$$

The algorithm ends when it reaches the maximum number of iteration or when the following convergence condition is satisfied:

$$\begin{gathered} ||\nabla f(x_k)||_2 < tol_f ||\nabla f(x_0)||_2 \\ \vee \\ ||x_k - x_{k-1}||_2 < tol_x \\ \vee \\ k = k_{max} \end{gathered} \tag{2.4}$$

being $tol_f = 1e-5$ and $tol_x = 1e-4$ two algorithm parameters.

Sometimes using a fixed step length is not a good practice because the algorithm could end up jumping between the "sides" around the minimum without reaching it. In order to avoid this behaviour, there exist different techniques that update step length value during computation. Among them there is *backtracking* that can be formulated as:

$$f(x_k - \alpha_k \nabla f(x_k)) > f(x_k) - c\alpha_k ||\nabla f(x_k)||^2 \Rightarrow \alpha_{k+1} = \tau \alpha_k \tag{2.5}$$

where α is the step length, τ a parameter that control the decreasing of α and c a parameter that decides when to update α. Here $c = 0.8$ and $\tau = 0.25$.

2.2 SGD

Usually in machine learning it is not practical to implement standard gradient descent, for example when the dataset to evaluate gradients on is big. SGD aims to solve this issue by computing, at each iteration, an approximation of the true gradient by evaluating function's derivatives over a subset of samples called batch. Being all the theory the same, what changes is that, at each iteration, the algorithm does not follow the shortest path toward the minimum but an irregular one close to the it.

A classical machine learning problem can be formulated as before, with the only difference being that now the function to be minimized depends also on the dataset $f(\vec{w}, \mathbb{D})$, where $\vec{w}$ is the weights vector. The problem can be rewritten as finding $\vec{w}^*$ such that

$$\vec{w}^* = \underset{\vec{w}}{\operatorname{argmin}}\, L(\vec{w}, \mathbb{D}) = \underset{\vec{w}}{\operatorname{argmin}} \sum_{i=1}^{N} l(\vec{w}, x^{(i)}, y^{(i)}) \tag{2.6}$$

where L is the *loss function* and the summation is carried over the entire dataset $\mathbb{D}$.

Because the goal is to optimize with respect to $\vec{w}$, the training iteration becomes:

$$\vec{w}_{k+1} = \vec{w}_k - \alpha_k \sum_{i=1}^{N} \nabla l(\vec{w}, x^{(i)}, y^{(i)}) \tag{2.7}$$

3 Exercise 1 - Gradient Descent

In the first exercise it is required to implement GD for four different functions and experiment with different hyperparameters. In the following sections the four experiments are reported.

3.1 Function #1

Optimize function $f : \mathbb{R}^2 \to \mathbb{R}$ such that

$$f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 1)^2 \tag{3.1}$$

where the true optimum is $x^* = (3, 1)^T$.

In order to apply GD, the derivative of f has to be computed first. So

$$\nabla f(x_1, x_2) = [2(x_1 - 3), 2(x_2 - 1)] \tag{3.2}$$

Graphs in 3.1 and 3.2 show the value of the gradient $\nabla f(\vec{x})$ and the distance between $\vec{x}_{true}$ and $\vec{x}_k$ for each iteration and for different values of α.

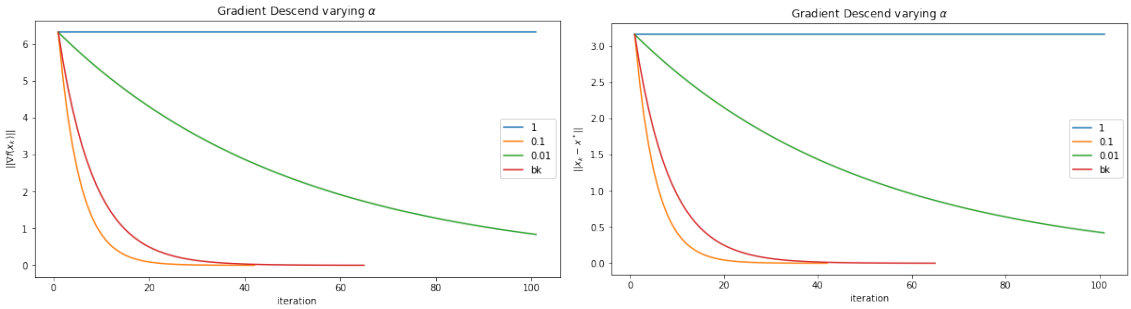

Figure 3.1: Gradient Descent applied to first function.

As it can be seen, with $\alpha = 1$ the minimum is never reached: indeed the algorithm keeps "jumping" from one side of the function to the other being the step length too big to approximate the optimum, as shown in 3.2 top-left graph. Lowering α allows the algorithm to reach the minimum, at a different speed (or equivalently iterations). $\alpha = 0.1$ is the best value in this case and outperforms *backtracking* although the latter starts exactly from the same α value. This means that, after the first iteration, the backtracking mechanism change α to a lower value, decreasing the convergence speed.

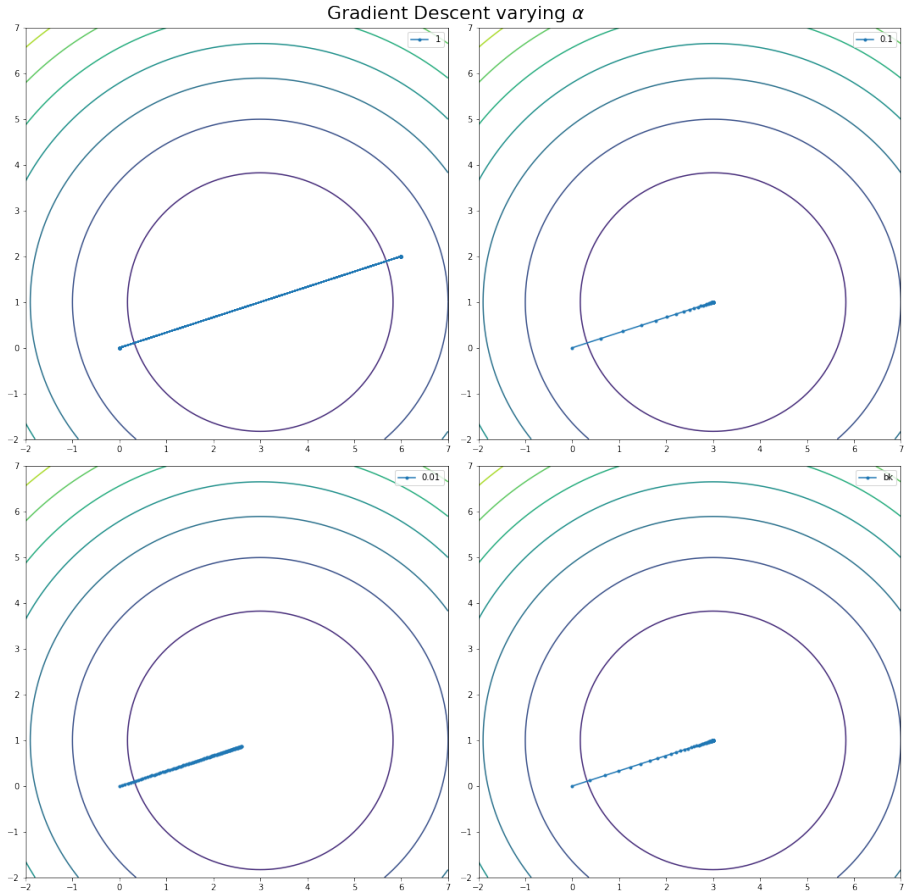

Figure 3.2: Contour plot for the first function.

3.2 Function #2

Optimize function $f : \mathbb{R}^2 \to \mathbb{R}$ such that

$$f(x_1, x_2) = 10(x_1 - 1)^2 + (x_2 - 2)^2 \tag{3.3}$$

where the true optimum is $x^* = (1, 2)^T$.

The derivative of f is

$$\nabla f(x_1, x_2) = [20(x_1 - 1), 2(x_2 - 2)] \tag{3.4}$$

Graphs in 3.3 and 3.4 show the value of the gradient $\nabla f(\vec{x})$ and the distance between $\vec{x}_{true}$ and $\vec{x}_k$ for each iteration and for different values of α.

In this case, being the function an asymmetrical paraboloid 3.4, GD can go closer to the minimum also with $\alpha = 1$, but the optimization can not go further because for the orthogonal axe the α value is too big. Here the *backtracking* is faster then the αs chosen because is able to adapt to different scale of the axes. The visible step in the backtracking plot is the change in the α value and it clearly shows the speed up in convergence.

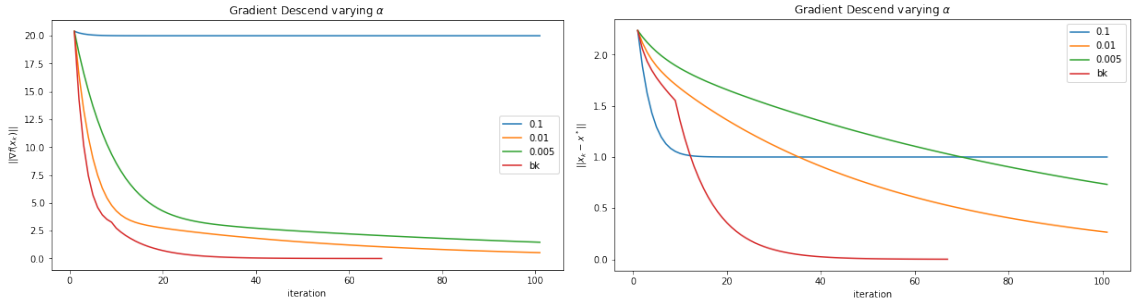

Figure 3.3: Gradient Descent applied to second function.

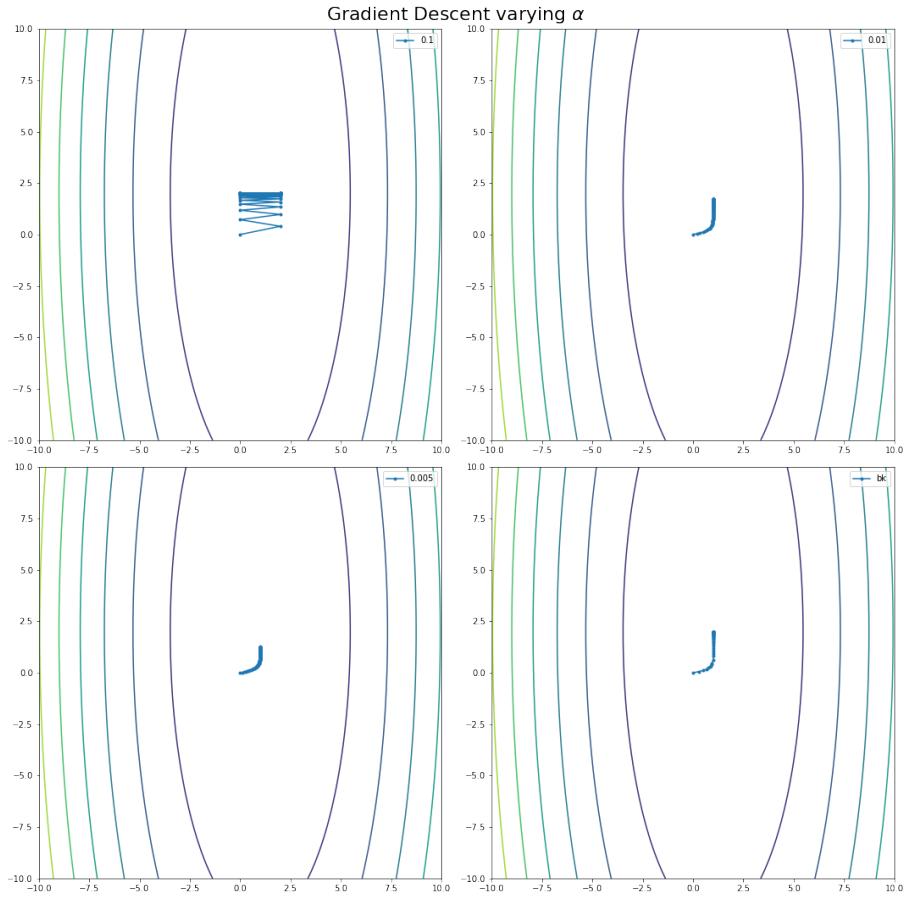

Figure 3.4: Contour plot for the second function.

3.3 Function #3

Optimize function $f : \mathbb{R}^2 \to \mathbb{R}$ such that

$$f(x) = \frac{1}{2}||Ax - b||_2^2 \tag{3.5}$$

where $A \in \mathbb{R}^{n \times n}$ is the Vandermonde matrix associated with the vector $\vec{v} \in \mathbb{R}^n$ that contains n equispaced values in the interval [0; 1], and $\vec{b} \in \mathbb{R}^n$ is computed by first setting $x_{true} = \vec{1}$ and then $\vec{b} = A\vec{x}_{true}$.

As a reminder, the Vandermonde matrix V is a matrix with the terms of a geometric progression in each row, i.e. $V_{ij} = x_i^{j-1}$.

The derivative of f is

$$\nabla f(x) = (x^T A^T - b^T)A \tag{3.6}$$

By definition, the Vandermonde matrix's shape depends on the chosen n. In this exercise I have experimented with n=(5, 10, 15). Results are shown below.

Graphs in 3.5, 3.6 and 3.7 show the value of the gradient $\nabla f(\vec{x})$ for each iteration and for different values of α and n.

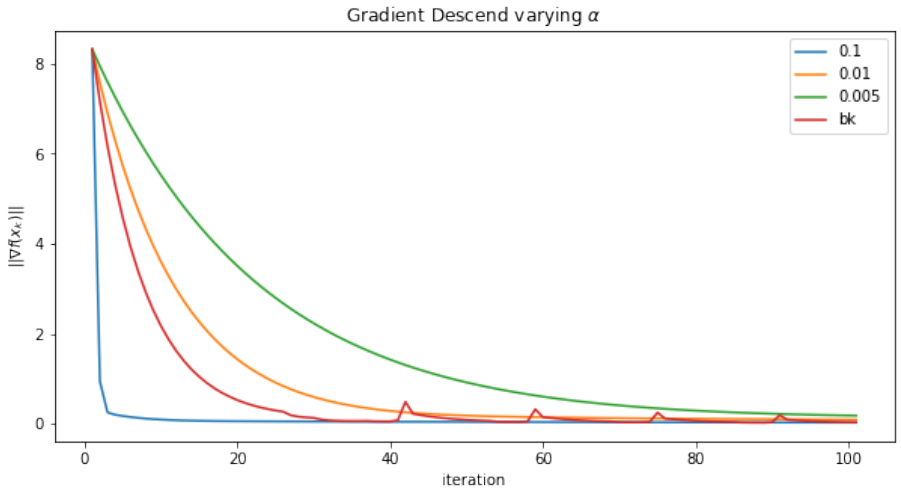

Figure 3.5: Gradient Descent applied to third function with n=5.

In this case it can be seen how the function f is sensitive to the combination of α and n: my idea is that the Vandermonde matrix is strongly ill-conditioned, being $k \approx 686, 15 * 10^6$ and $403 * 10^9$ for respectively $n = 5, 10, 15$. For this reason, when α is too big, a small error in the wrong direction leads to a divergence in the approximation error. Indeed, the Vandermonde matrix represents a linear map from the coefficient of a polinomial to its values, computed in the values of the Vandermonde matrix, so when dealing with polinomial of high grade it is easy to encounter divergent gradient if α is not correctly tuned. In 3.8 gradient component for $A_{n=10}$ are shown.

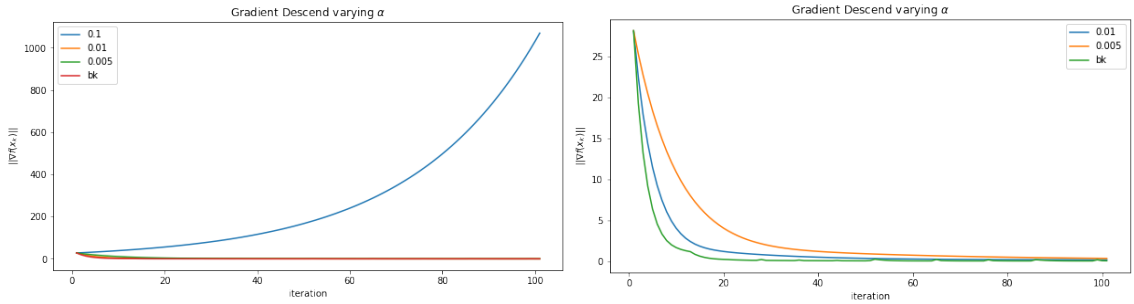

Figure 3.6: Gradient Descent applied to third function with n=10 with and without $\alpha = 0.1$.

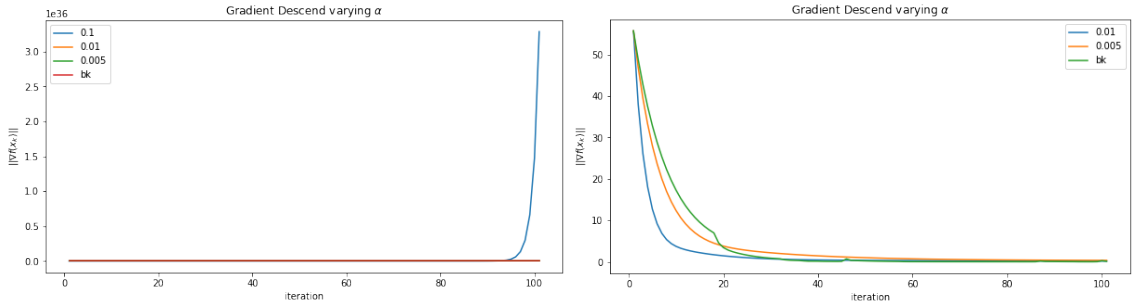

Figure 3.7: Gradient Descent applied to third function with n=15 with and without $\alpha = 0.1$.

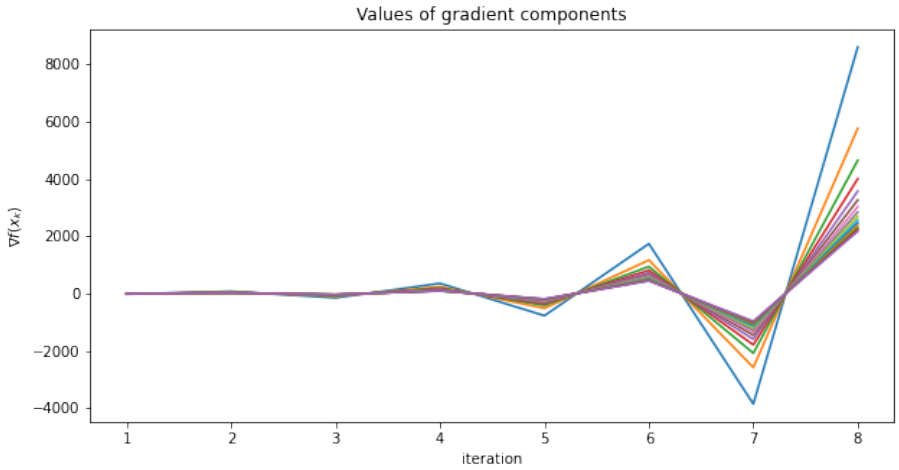

Figure 3.8: Gradient values for third function with n=5.

When using a smaller α, it can be seen that the best strategy is *backtracking* for n=10 and 15, because it reaches a better minimum faster, while for n=5 a fixed $\alpha = 0.1$ is a good choice.

3.4 Function #4

Optimize function $f : \mathbb{R}^2 \to \mathbb{R}$ such that

$$f(x) = \frac{1}{2}||Ax - b||_2^2 + \frac{\lambda}{2}||x||_2^2 \tag{3.7}$$

The derivative of f is

$$\nabla f(x) = (x^T A^T - b^T)A + \lambda x^T \tag{3.8}$$

where A in both previous equations, is the same Vandermonde matrix with n=5 used in the previous experiment and $\lambda \in [0, 1]$ is a constant. The function f is the well known Ridge regression, a regression model employed when the independent variables are highly correlated, or in other words, the data show multi-collinearity. The parameter λ is the regularization parameter, also called "penalty term", and it is used in order to prevent overfitting when training a machine learning model.

Graphs 3.9, 3.10 and 3.11 show the value of the gradient $\nabla f(\vec{x})$ for each iteration and for different values of α.

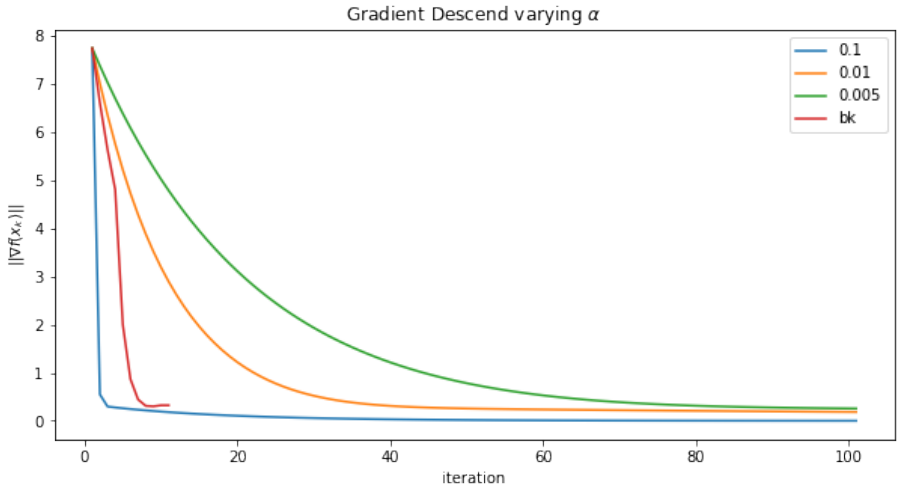

Figure 3.9: Gradient Descent applied to fourth function.

Here again ita can be seen how the ill-conditioning of the Vandermonde matrix is worse than before due to the penalty term. By removing $\alpha = 0.1$ it can be seen that the best strategy is *backtracking* again: indeed it reaches the best minimum faster and stops the algorithm thanks to the reaching of the convergence condition in 2.4.

In 3.12 it can be seen how the GD behaves varying λ. In particular, it can be seen that a value of λ close to one, prevents the algorithm from reaching a better minimum, that is instead reached when $\lambda = 0.25$. The case where $\lambda = 0$ is the same as function 3.

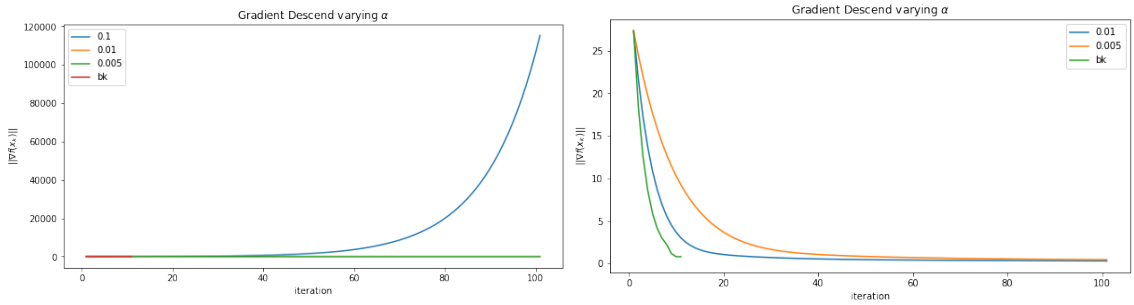

Figure 3.10: Gradient Descent applied to fourth function with n=10 with and without $\alpha = 0.1$.

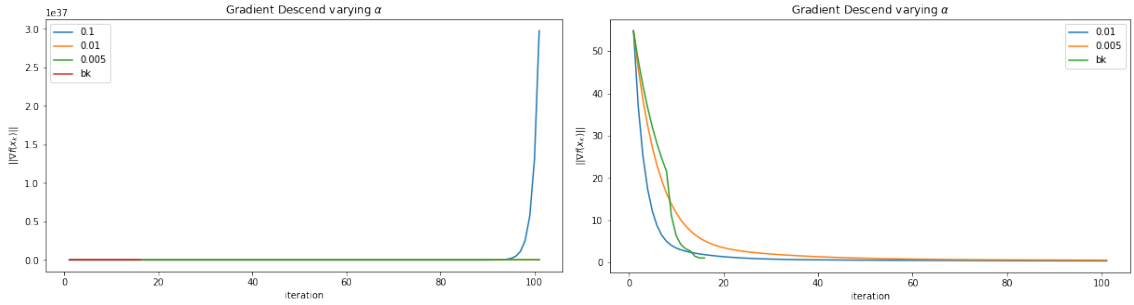

Figure 3.11: Gradient Descent applied to fourth function with n=15 with and without $\alpha = 0.1$.

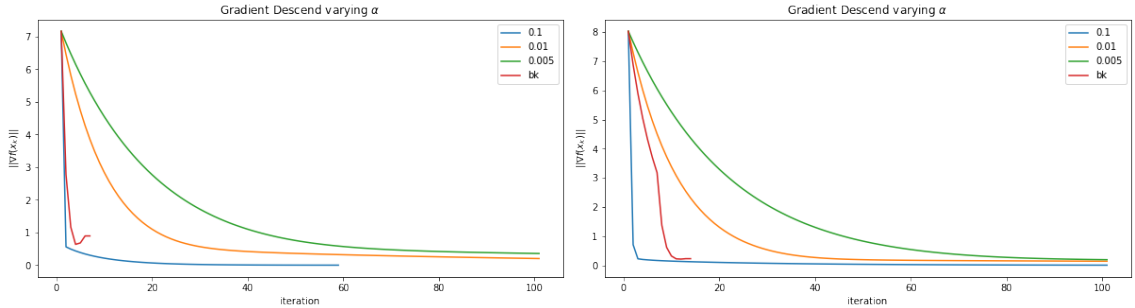

Figure 3.12: Gradient Descent applied to fourth function with n=5 $\lambda = 1$ and $\lambda = 0.25$.

4 Exercise 2 - Stochastic Gradient Descent

In the second exercise it is required to apply Stochastic Gradient Descent to a randomly generated dataset in order to simulate what is usually done in machine learning. This time optimization is carried out in the space of weights instead of the space of x: dataset samples are only used to evaluate the gradient while the derivatives are computed with respect the weights vector.

I chose to generate a dataset of $N = 1000$ samples and as space dimension for weights vector $\vec{w} \in \mathbb{R}^{10}$.

I have run two experiment with SGD: in the first I have fixed the batch size to 100 and number of epochs to 50, run SGD on diferent values of α and selected the one that made the algorithm to reach the minimum faster (results in 4.1); then I have used the α just found and run SGD for 20 epochs, each time with a diferent batch size from the list *[1,50,100,250,500]*. Results are shown in 4.2.

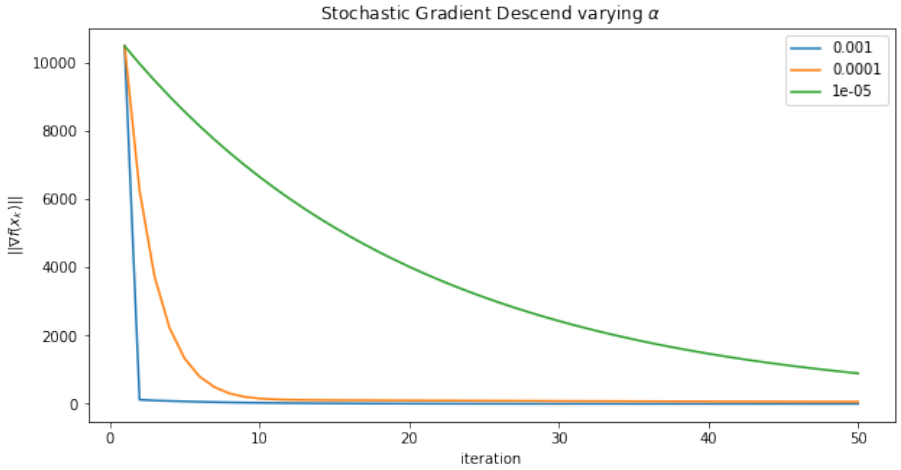

Figure 4.1: SGD varying α.

Here it can be observed that the relation between convergence speed and α value is monotonic: indeed the biggest α (among the chosen ones) allows SGD to reach the minimum faster, and when decreasing α convergence speed slows down. I tried also using $\alpha = 0.01$ but after around 20 iterations the algorithm diverged, so I removed it from the graph.

Then I have taken the best α and experimented with the batch size. In general there is no a one direction path in choosing first α and then *batch_size*, actually also this choice should be optimized by alternatively changing the two parameters until the best configuration is found.

In this case it can be observed that a different batch size can make the learning phase slower and noisier. For example, just look at the red line and the blue one in 4.2. The reason is that more smaller updates are better than one single big update: indeed the algorithm performs

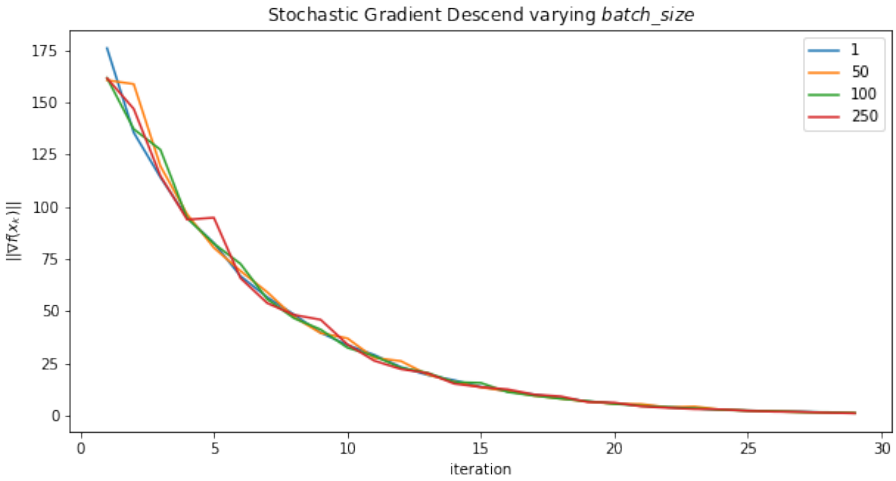

Figure 4.2: SGD with $\alpha = 1e-3$ varying *batch_size* parameter.

$|\mathbb{D}|/batch\_size$ updates at each epoch, so the smaller the batch size the greater the number of updates and so the convergence speed. The drawback is obviously the computation burden, being the mini batch size more resource demanding.

5 Conclusion

In this exercise we experimented around Gradient Descent and Stochastic Gradient Descent techniques. We saw how they are effective in optimizing almost any kind of ("optimizable") function. In case of convex oprimization, GD and SGD always reach the global optimum, while in case of non-convex function it depends on the starting point and in general it is not certain that it will reach a global minimum. A caveat that is worth to highlight here is that convergence of GD or SGD does not mean having reached the best minimum, because there would be no way to distinuish between a local and a global one.

From a machine learning point of view, we saw that is crucial to find the right configuration of hyperparameters, indeed also the best model (from a mathematical point of view) could fail to reach convergence due to a misconfiguration of hyperparameters. For this reason, a crucial step in machine learning is the validation phase, where the algorithm hyperparameter are fine-tuned on a never seen dataset partition.

In this exercise there would be a huge number of combination of hyperparameters to play with and test, so I decided to restrict the range to the value reported in this work.