Instituto Superior Técnico

Departamento de Engenharia Electrotécnica e de Computadores

Machine Learning

2nd Lab Assignment

Shift 4ºF Group number 8

Number 78719 Name Afonso Barroso Costa

Number 79069 Name Bruno Miguel Freitas Gonçalves

Function Optimization – The Gradient Descent Method

1 Introduction

In many situations, it is necessary to optimize a given function, i.e., to minimize or maximize it. Most machine learning methods are based on optimizing a function that measures the performance of the system that we want to train.

This function is generically called *objective function,* because it indirectly defines the objective of the training. Frequently, this function measures how costly are the errors made by the system. In that case, the function is called c*ost function*, and the purpose of training is to minimize it. Since this is the most common case, in this assignment we'll study function minimization. However, all the conclusions can be applied, with the appropriate changes, to the case of function maximization.

In most cases of practical interest, the function that we want to optimize is very complex. Therefore, solving the system of equations that is obtained by setting to zero the partial derivatives of the function with respect to all variables, is not practicable. In fact, these equations are usually highly nonlinear, and the number of variables is often very large, on the order of hundreds, thousands, or even more. In those cases, iterative optimization methods have to be used.

2 The gradient descent method

One of the simplest and most frequently used optimization methods is the method of gradient descent. Consider a function $f(\boldsymbol{x})$ that we want to minimize, where the vector $\boldsymbol{x} = (x_1, x_2, \cdots, x_N)$ is the set of arguments. The gradient of *f*, denoted by ∇f, points in the direction that makes *f* increase fastest. Therefore, it makes sense that, in order to minimize the function, we take steps in the direction of the negative gradient, $-\nabla f$, which is the direction that makes *f* decrease fastest. The gradient method consists of the following steps:

- Choose an initial vector $\boldsymbol{x}^{(0)}$.
- Update $\boldsymbol{x}$ iteratively, according to the equation:

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} - \eta \, \nabla f[\boldsymbol{x}^{(n)}].$$

The parameter η is chosen by the user, and must be positive. It is clear, from the previous equation, that the method consists of a succession of steps, each taken in the direction that $-\nabla f$ has at the current location. The iterations stop when a given stopping criterion, chosen by the user, is met.

In this lab we'll study the gradient method in order to gain experience about the way it works. We'll also study some modifications to this method, which are intended to increase its efficiency.

2.1 Minimization of functions of one variable

We'll start by studying the gradient method in the simplest situation, which corresponds to minimizing functions of only one variable. Namely, we'll minimize a function of the form $f(x) = a\, x^2/2$. The parameter *a* controls the functions' curvature.

Start Matlab and change to the directory where you placed the files related to this lab assignment. Then type the command **`quad1ini`**, which initializes the necessary parameters for the tests that you will run. This command initializes the following values:

$$a = 1,\ \eta = 0.1,\ x^{(0)} = -9.$$

Type the command **`quad1`**, which performs the optimization and graphically shows its evolution. The stopping criterion consists in finding a value of f under 0.01 (this value can be controlled by the variable **`threshold`**), with a maximum of 1000 iterations (this value can be controlled by the variable **`maxiter`**).

The variable **`anim`** controls the graphic animation. Setting **`anim=1`** makes the evolution visible as it progresses. This allows us to get a better idea of the evolution, but also makes it take longer. Setting **`anim=0`** shows the plot only at the end of the evolution, which makes it go considerably faster.

1. Fill the following table with the numbers of iterations needed to optimize the function, for different values of a (**`a`** in Matlab) and η (**`eta`** in Matlab). If more than 1000 iterations were needed, write ">1000". If the optimization method diverged, write "div". In the last two lines, instead of the number of iterations, write the approximate values of η that correspond to the fastest optimization and to the threshold of divergence.

η	$a = 0.5$	$a = 1$	$a = 2$	$a = 5$
.001	> 1000	> 1000	> 1000	990
.01	750	414	223	97
.03	242	137	73	31
.1	75	40	21	8
.3	24	12	5	8
1	6	1	> 1000	> 1000
3	6	> 1000	> 1000	> 1000
Fastest	1 ou 3	1	0.3	0.1 ou 0.3
Divergence threshold	4	2	1	0.4

Table 1

2. Comment on the results from the table.

Para os primeiros casos, onde eta=0.001, este é ainda demasiado pequeno e, portanto, não se consegue chegar ao valor mínimo da função. No entanto, após o sucessivo aumento de eta, acaba-se por chegar a alguns resultados.
Algo interessante é por exemplo para a=0.5, e eta=1 e 3, onde se chega exatamente ao mesmo mínimo, no mesmo número de passos, mas com etas diferentes. Neste caso, para eta=1 estamos a ir sempre pelo lado esquerdo da parábola, mas para eta=3 estamos sucessivamente a oscilar entre o lado esquerdo e direito.
Os melhores valores da função de custo foram obtidos para eta=1,3; eta=1; eta=0.1 e eta=0.1,0.3; respetivamente para cada um dos valores de a. Neste caso, para a=2, curiosamente o melhor valor da função de custo acaba por não ser o obtido no menor número de passos.
Está-se perante o divergence threshold, neste caso, quando a optimização obtida fica sucessivamente a oscilar entre a componente crescente e decrescente da parábola. Assim. podemos calcular o eta para o qual esta situação acontece:
x_(t+1) = - x_t <=> x_t - eta*a*x_t = - x_t <=> x_t (2-eta*a) = 0 <=> eta = 2/a

3. How many steps correspond to the fastest optimization, for each value of a? Does there exist, for every differentiable function of one variable (even if the function is not quadratic), and for each given starting point $\mathbf{x}^{(0)}$, a value of η that optimizes the function in that number of steps? Assume that the function grows to $+\infty$ when $\|\mathbf{x}\| \to \infty$.

Para cada valor de a, os passos que correspondem à optimização mais rápida são, respetivamente, 6, 1, 5 e 8.
A optimização mais rápida acontece quando em apenas um passo se consegue chegar ao mínimo da função.
Para o caso em que a função tem apenas uma variável, a direção para a qual o gradiente aponta (onde a função está a crescer mais rapidamente) será sempre aquela que se afasta mais rapidamente do mínimo daquela função. Assim, independentemente do ponto inicial, consegue-se sempre saber qual a direção de um mínimo e, por isso, existe sempre um valor de eta que nos leva até ao mínimo em um passo.

Para o caso desta função, descobrir o valor de eta que leva à optimização em apenas um passo é resolver a seguinte equação:
x_1 = 0 <=> x_1 = x_0 - eta *a* x_0 <=> 0 = x_0 (1-eta*a) <=> eta = 1/a

2.2. Minimization of functions of more than one variable

When we deal with functions of more than one variable, new phenomena occur in the gradient method. We'll study them by minimizing functions of two variables.

We'll start by studying a simple class of functions: quadratic functions of the form $f(x_1, x_2) = (ax_1^2 + x_2^2)/2$. For these functions, the second derivative with respect to x_1 is a, and the second derivative with respect to x_2 is 1.

Type the command **`clear`**, to eliminate the variables used in the previous test, and then type the command **`quad2ini`**, which initializes the necessary parameters for the tests that you'll run next. This command initializes the following values:

$$a = 2,\ \eta = 0.1,\ \mathbf{x}^{(0)} = (-9,9).$$

Type the command **`quad2`**, which performs the optimization and shows the results. The stopping criterion corresponds to finding a value of f smaller than 0.01 (this value can be controlled by the variable **`threshold`**), with a maximum of 1000 iterations (this value can be controlled by the variable **`maxiter`**).

Observe that, along the trajectory, the steps are always taken in the direction orthogonal to the level curves of f. In fact, the gradient is always orthogonal to these lines.

1. Fill the first column of the following table for the various values of η. Then set $a = 20$ (which creates a relatively narrow valley) and fill the second column. Use the same rules for filling the table as in the preceding case. You may find the values for η that correspond to the fastest optimization and to the threshold of divergence by trial and error.

η	$a = 2$	$a = 20$
.01	448	563
.03	148	186
.1	43	> 1000
.3	13	> 1000
1	> 1000	> 1000
3	> 1000	> 1000
Fastest	0.3	186
Divergence threshold	1	~ 0.1

Table 2

2. Comment on the results from the table.

Neste caso não conseguimos calcular analiticamente o divergence threshold. Obtivemos então estes valores por experimentação.
No caso de a=2, quando eta=1, através da figura obtida, verificou-se que as otimizações ficam exatamente a oscilar entre dois pontos, nas paredes do vale, após uma primeira optimização que levou a um desses pontos. No caso de a=20 e eta=0.1, as otimizações não ficam exatamente a oscilar entre dois pontos, mas o modo como a oscilação acontece, em que após algumas otimizações, estas ficam a oscilar à volta de dois pontos, leva-nos a acreditar que o divergence threshold é muito próximo deste valor, eta=0.1.
Neste caso, o valor de eta que conduz a uma melhor otimização, onde a função de custo tem o valor mais baixo, é respetivamente 0.3 e 0.03, que são também os que o conseguem fazer em menos passos, para cada a.
Para os restantes, o valor de eta acaba por ser demasiado grande, e a otimização acaba por divergir, não se chegando ao mínimo.

3. Is it always possible, for differentiable functions of more than one variable, to achieve, for any given $\mathbf{x}^{(0)}$, the same minimum number of iterations that was reached for functions of one variable? What is the qualitative relationship between the valley's width and the minimum number of iterations that can be achieved?

Para funções com mais do que uma variável, dado um ponto arbitrário, o gradiente dá-nos o sentido em que a função está a crescer mais. No entanto, neste caso, este sentido, ao contrário das funções com só uma variável, poderá não estar relacionado com o mínimo da função. A função poderá simplesmente estar a crescer muito rapidamente naquele sentido, independentemente de se estar a afastar/aproximar de um mínimo. Neste sentido, é possível chegar ao mínimo da função em apenas um passo para alguns pontos iniciais, mas dependendo do ponto inicial, pode nem sempre existir um eta para o qual isto é possível.

Qualitativamente, quando maior for a largura de um vale, para um mesmo eta, mais iterações são necessárias para de chegar ao mínimo, isto porque a função de custo vai diminuindo muito mais devagar.

3. Momentum term

In order to accelerate the optimization in situations in which the function has narrow valleys (situations which are very common in machine learning problems), one of the simplest solutions is to use the so called *momentum term*. The previous examples showed how the divergence in the gradient descent method is normally oscillatory. The aim of the momentum term is to attenuate

the oscillations by using, at each step, a fraction of the previous one. The iterations are described by:

$$\Delta \mathbf{x}^{(n+1)} = \alpha \Delta \mathbf{x}^{(n)} - \eta \nabla f[\mathbf{x}^{(n)}]$$
$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)}$$

or, alternatively, by

$$\Delta \mathbf{x}^{(n+1)} = \alpha \Delta \mathbf{x}^{(n)} - (1-\alpha)\eta \nabla f[\mathbf{x}^{(n)}]$$
$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \Delta \mathbf{x}^{(n+1)}$$

.

We'll use this second version.

The parameter α should satisfy $0 \le \alpha < 1$. Using $\alpha = 0$ corresponds to optimizing without the momentum term. The term $\alpha \Delta \boldsymbol{x}^{(n)}$, in the update equation for $\Delta \boldsymbol{x}^{(\boldsymbol{n+1})}$, attenuates the oscillations and adds a kind of inertia to the process, which explains the name *momentum term*, given to this term.

The students that are knowledgeable on digital filters, can readily verify that the equation that computes $\Delta \boldsymbol{x}^{(n+1)}$ corresponds to filtering the gradient with a first order low-pass filter, with a pole at $z = \alpha$. This low-pass filtering attenuates rapid oscillations.

1. Still using the function $f(x_1, x_2) = (ax_1^2 + x_2^2)/2$, fill the following table, using $a = 20$, and varying the momentum parameter α. (in Matlab, the parameter α corresponds to the variable **`alfa`**). Use the same rules for filling the table as in the preceding cases.

η	$\alpha = 0$	$\alpha = .5$	$\alpha = .7$	$\alpha = .9$	$\alpha = .95$
.003	> 1000	> 1000	> 1000	> 1000	> 1000
.01	563	558	552	516	448
.03	186	181	174	115	175
.1	> 1000	48	35	91	122
.3	> 1000	> 1000	29	83	92
1	> 1000	> 1000	> 1000	92	146
3	> 1000	> 1000	> 1000	> 1000	147
10	> 1000	> 1000	> 1000	> 1000	> 1000
Divergence threshold	~ 0.1	~ 0.3	~ 1	~ 3	~ 10

Table 3

2. Comment on the results from the table.

Com a existência do momentum term, passaram a existir valores de eta para os quais passou a ser possível encontrar o valor mínimo. Para maiores valores de alfa, a convergência do método não depende tanto do eta.
Ainda assim, para um mesmo eta, é inconclusivo se o aumento de alfa acaba por trazer melhores ganhos, sendo que, tal como demonstrado nos resultados obtidos, acaba existir um alfa ideal, mas para diferentes etas, este alfa é diferente.
A introdução deste parâmetro acabou então por melhorar bastante o número de passos com que se chegava ao mínimo.

4. Adaptive step sizes

The previous examples showed how narrow valleys create difficulties in the gradient descent method, and how the momentum term alleviates these problems. However, in complex problems, the optimization can take a long time even when the momentum term is used. Another acceleration technique that is quite effective relies on the use of adaptive step sizes. This technique will not be explained here, given its complexity. Nevertheless, we'll test its efficiency.

As an example of a function which is difficult to optimize, we'll use the Rosenbrock function, which is one of the common benchmarks used for testing optimization techniques. This function is given by:

$$f(x_1, x_2) = (x_1 - 1)^2 + a(x_2 - x_1^2)^2 .$$

This function has a valley along the parabola $x_2 = x_1^2$, with a minimum at (1,1). The parameter *a* controls the width of the valley. The original Rosenbrock function uses the value $a = 100$, which creates a very narrow valley. Initially, we'll use $a = 20$, which creates a wider valley, so that we can run our tests faster.

Type the command **`clear`**, followed by **`rosenini`**, which initializes the parameters for the tests that will follow. This command disables the adaptive step sizes, and initializes the following values:

$$a = 20, \ \eta = 0.001, \ \alpha = 0, \ \boldsymbol{x}^{(0)} = (-1.5, 1),$$

Type the command **`rosen`**, which performs the optimization. The stopping criterion corresponds to having two consecutive iterations with *f* smaller than 0.001, with a maximum of 1000 iterations.

1. Try to find a pair of values for α and η that leads to a number of iterations below 200. If, the number of tests is becoming too large, stop and use the best result obtained so far. Write down how many tests you performed in order to find that pair of values, and fill the following table, using the best value that you found for η, and also values 10% and 20% higher and lower than the best.

N. of tests	α	$\eta \rightarrow$	-20%	-10%	best	+10%	+20%
510	0.86	N. of iterations→	204	191	165	121	> 1000

eta=0.05

Table 4

2. Basing yourself on the results that you obtained in the table above, give a qualitative explanation of why it is hard to find values of the parameters that yield a relatively small number of iterations.

Em primeiro lugar, como normalmente não conhecemos com certeza o comportamento da função, acabamos por experimentar valores um pouco aleatórios, o que acaba por acrescentar alguma dificuldade à resolução do problema. Pelos resultados obtidos, neste caso para eta=0.05, podemos ver que uma pequena variação de +20% conduziu a que já não fosse possível a convergência para um mínimo.
Neste sentido, uma pequena variação pode conduzir logo para um valor de eta que já não conduz a um mínimo e, por isso acaba por ser um pouco diíicil achar valores que o façam.

Note that the total time that it takes to optimize a function corresponds to both the time it takes to perform the tests that needed to find a fast enough optimization, plus the time it takes for that optimization to run.

3. Next, we'll test the optimization using adaptive step sizes. Type the command **`assact`**, which activates the adaptive step sizes (**`assdeact`** deactivates them). Fill the following table with the numbers of iterations necessary for the optimization under different situations.

η	$\alpha = 0$	$\alpha = .5$	$\alpha = .7$	$\alpha = .9$	$\alpha = .95$	$\alpha = .99$
.001	596	298	236	140	198	167
.01	565	287	221	190	200	165
.1	769	389	214	183	172	152
1	729	396	233	160	137	173
10	672	383	239	173	124	133

Table 5

Observe how the number of iterations depends little on the value of η, contrary to what happened when fixed step sizes were used (note that, in the table above, η varies by four orders of magnitude). The relative insensitivity to the value of η is due to the fact that the step sizes vary automatically during the minimization (the value given to η represents only the initial value of the step size parameter). The little dependency on the initial value of η makes it easier to choose values that result in efficient optimization.

4. Finally, we'll test the optimization of the Rosenbrock function with the original value of *a*. Set **a=100**. Try to find values of η and α such that the convergence is reached in less than 500 steps, first without adaptive step sizes, and then with adaptive step sizes. Write down, for each case, the number of tests required to find the final values of η and α. If, in any case, the number of tests is becoming too large, stop and use the best result obtained so far.

   For both cases change the best value of eta by about 10% up and down, without changing α, and write down the corresponding numbers of iterations. Fill table 6 with the values that you obtained.

eta = 0.02

	N. of tests	η	α	N. of iterations
Without adaptive step sizes	55	-10%	0.95	385
		best		412
		+10%		> 1000
With adaptive step sizes	1	-10%	0.95	383
		best		366
		+10%		523

Table 6

5. Comment on the efficiency of the acceleration methods that you have tested in this assignment.

Com a=20, repara-se que a diferença entre o número de iterações para diferentes alphas varia mais do que variando o valor de eta. Tal deve-se ao facto de esta função ter um vale estreito em x2 = x1^2.

No entanto, o método no qual se verificou um melhores melhorias é o dos adaptative steps, onde se verifica que os valores respetivos do número de steps diminui acentuadamente. Isto deve-se ao facto do valor do eta se adaptar ao comportamento da função. Para a=100 reparou-se que o fator que mais contribui foi o valor de alpha, que não é alterado quando se usa adaptative step sizes.

Com estes resultados mostramos que o método com mais impacto na optimização é o adaptative step sizes

---

Do not write below this line.