Instituto Superior Técnico

Departamento de Engenharia Electrotécnica e de Computadores

Machine Learning

2nd Lab Assignment

Shift 4ª - 14h Group number 1

Number 78486 Name Luís Bernardo de Brito Mendes Rei

Number 78761 Name João Miguel Limpo de Lacerda Pestana Girão

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	760	414	223	97
.03	252	137	73	31
.1	75	40	21	8
.3	24	12	5	8
1	6	1	div	div
3	6	div	div	div
Fastest	1	1	0.3	0.1
Divergence threshold	3	1	0.3	0.3

Table 1

2. Comment on the results from the table.

Analyzing the results we can see that the effect of the η parameter increases somewhat exponentially with its growth as it reaches, and surpasses, its optimal value; it behaves almost linearly for values smaller than, and further away from, the optimal value. We can also conclude that the fastest and the divergence threshold values of η are in accordance to what was expected, that is:

Fastest value: $X^{(1)} = X^{(0)} - \eta * \nabla f(X^{(0)}) <=> 0 = X^{(0)}(1 - \eta a) <=> \eta a = 1 <=> \eta = \frac{1}{a}$

Divergence threshold: $|f(X^{(1)})| = |f(X^{(0)})| <=> X^{(0)}|(1 - \eta a)| = X^{(0)} <=> \eta = \frac{2}{a}.$

(note that $f(x)$ is an even function so $f(X^{(0)}) = f(-X^{(0)})$)

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 $\|\mathbf{x}\| \to \infty$.

For this simple cost function, the fastest optimization is obtained in one iteration and we can compute the optimal η by applying the following formula and substituting $X^{(1)}$ with the value that leads to a global minimum: $X^{(1)} = X^{(0)} - \eta * \nabla f(X^{(0)})$, $X^{(1)} = 0$, $\nabla f(x^{(t)}) = ax^{(t)}$. From this we take that the optimal η does not depend on the initial point but instead just on the parameter a. For more complex functions the optimal η may be dependent on the starting point and the algorithm may not even converge. An example is the following function $f(x) = x^4 - x^2 + 0.5x$ which has two minima and a maximum. If the starting point is at said maximum the obtained cost function's value will never change, even though it is not minimal.

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	415	415
.03	138	138
.1	41	div
.3	13	div
1	div	div
3	div	div
Fastest	0.3	0.03
Divergence threshold	0.3	0.03

Table 2

2. Comment on the results from the table.

The difference in efficiency in both situations due to the width variation of the valley is easily detected by looking at the values displayed in the table above.The wider valley (a = 2) is more easily optimized as it allows for bigger step sizes to be taken without diverging, leading to more space being covered towards our goal (minimum cost) in the same amount of iterations.

A narrower valley not only translates in a slower convergence using the tabled η, but is also less forgiving in providing with a valid η value that doesn't diverge.

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?

No, the fact that the gradient can now be pointed in a direction such that it doesn't intercept the point that minimizes optimally the cost function makes it so that the number of iterations necessary for convergence is, in most cases, much bigger than that of it's one variable counterparts. In general, for functions of one or more variables the optimal values of η depend on the starting point and can't be analytically obtained.

A narrower valley implies a more restricted interval of parameter's values that are capable of providing a convergence of the cost function. This implies that the convergence will either be slower (when comparing with a wider valley) due to the small step sizes or because of the oscillation generated by bigger values of η.

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 $\Delta \boldsymbol{x}^{(n+1)}$, 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	415	412	407	383	339
.03	138	135	130	97	172
.1	div	37	32	92	162
.3	div	div	34	102	193
1	div	div	div	114	223
3	div	div	div	div	229
10	div	div	div	div	div
Divergence threshold	0.03	0.1	0.3	1	3

Table 3

2. Comment on the results from the table.

This technique is known to improve the convergence rate, specially if the cost function exhibits deep valleys in which the gradient method is slow. Also the objective function f is evaluated at each iteration to check if it decreases as expected. If not, the memory of the moment term is set to zero. As the parameter α gets closer to 1 the acceleration technique starts depending much less on η and more on its last iteration's value; this attenuates the impact that the step sizes have in the algorithm. Such an effect can be perceived by studying the results in the table above, where a clear difference in the iterations and convergence interval of the same function with various values for α is seen.

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%
90	0.7963	N. of iterations→	264	245	120	div	div

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.

As stated above, the function used to test the algorithm is an adaptation of the Rosenbrock function, known to be difficult to optimize. Although the version we're using has a wider valley, it is nonetheless still a challenge to obtain good results. This is mainly due to the fact that the valley isn't straight and completely aligned with the x axes, but curved, originating multiple intervals of convergence and divergence, and making it hard to pinpoint a good starting point for the test. In the table above we can see the contrast in changes of η: lowering its value by 10% and 20% of the optimal value more than doubles the steps needed to reach a cost smaller than the chosen threshold; increasing it by 10% or 20% makes the algorithm diverge.

NOTE: In 4.1 and 4.4 the tests were conducted this way: fill with the values of the steps for each combination a table with axes η and α (each with periodically spaced values); then, if there were no acceptable results, zoom in on the most promising interval and repeat until completion.

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.

	N. of tests	η	α	N. of iterations
Without adaptive step sizes	120	-10%	0.9215	194
		best		159
		+10%		972
With adaptive step sizes	30	-10%	0.99	312
		best		202
		+10%		329

Table 6

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

Both acceleration techniques play a very important role in reducing the number of iterations needed for convergence and together they make a powerful tool for function optimization.
The momentum term works best for functions with deep valleys and slow gradient but can cause some instability when the gradient becomes faster. The adaptive step size performs very well if the cost function contains valleys aligned with the x axes. It also widens the intervals of η in which the function converges. Using both techniques simultaneously one can test and use their optimization algorithms faster and without worrying as much about them diverging.
In Table 6 there is a noticeable distinction between the influence of η when the adaptive step size is applied and when it isn't. Although the values of α were such that the impact of the η parameter was severely dampened, the effects of varying the latter by $\pm 10\%$ resulted in very different iteration values in both cases: the changes of $\pm 10\%$ with adaptive steps led to a 50% increase in the number of iterations needed, while without adaptive steps a decrease of 10% resulted in an increased 20% of iteration steps, and the increase of 10% in an increase of 500%. The results obtained from the two best values of η didn't differ much from one another, so the deciding factor between choosing one technique over the other relies on the results of the $\pm 10\%$ η variation, and on the difficulty of achieving a reasonable result without many tests being performed. We took a much larger amount of tests to achieve a reasonable amount of iteration steps without the adaptive steps technique, in comparison to the case where we applied the technique.

Do not write below this line.