Instituto Superior Técnico

Departamento de Engenharia Electrotécnica e de Computadores

Machine Learning

1st Lab Assignment

Group number _____

Number _____   Name _____

Number _____   Name _____

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 $f(x)$ that we want to minimize, where the vector $x = (x_1, x_2, \cdots, x_N)$ 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 $x^{(0)}$.
- Update x iteratively, according to the equation:

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

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. Find these values by trial and error.

η	$a = 0.5$	$a = 1$	$a = 2$	$a = 5$
.001				
.01				
.03				
.1				
.3				
1				
3				
Fastest				
Divergence threshold				

Table 1

2. What is the relationship between a (the function's second derivative) and the value of η that corresponds to the fastest optimization? Is it possible to theoretically prove that relationship for this class of functions?

3. What is the relationship between the function's second derivative and the value of η that corresponds to the divergence threshold? Is it possible to theoretically prove that relationship for this class of functions?

4. 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), a value of η that optimizes the function in that number of steps, for a given starting point $x^{(0)}$?

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:

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

Type the command **quad2**, which performs the optimization and shows the results. The stopping criterion corresponds to having two consecutive iterations with 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 $a = 20$ and fill the second column (use ">1000" and "div" as indicated above). In the row 'Fastest' write the best value for η that you found, as well as the corresponding number of iterations. In the row "Divergence threshold" write the approximate value of η that corresponds to the threshold of divergence. Find these values by trial and error.

η	$a = 2$	$a = 20$
.01		
.03		
.1		
.3		
1		
3		
Fastest		
Divergence threshold		

Table 2

Observe that, with $a = 20$, the function has a relatively narrow valley. The example that you analyzed illustrates how difficult it is for the simple gradient descent method to deal with valleys.

2. Is it possible, for functions of more than one variable, to achieve 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?

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 $\alpha \Delta x^{(n)}$, in the update equation for $\Delta x^{(n+1)}$, 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 $\Delta x^{(n+1)}$ 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.

3. 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**). Observe that using the momentum term enables the acceleration of the optimization, but that choosing an excessive value for α can make the results worse.

η	$\alpha = 0$	$\alpha = .5$	$\alpha = .7$	$\alpha = .9$	$\alpha = .95$
.003					
.01					
.03					
.1					
.3					
1					
3					
10					
Divergence threshold					

Table 3

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:

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

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	$\eta \rightarrow$					
		N. of iterations$\rightarrow$					

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.

Note that the total time that it takes to optimize a function corresponds to both the time it takes for the fastest optimization to run, and the time it took to perform the previous tests that were needed to find that fastest optimization.

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						
.01						
.1						
1						
10						

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				
With adaptive step sizes				

Table 6

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

Do not write below this line.