Evolutionary Computation
Course Notes

Ernesto Costa

VersŃo 0.01

Contents

List of Tables

List of Figures

Preface

...

...

This a companion text for the course Evolutionary Computation of the Department of Informatics Engineering of the University of Coimbra. The course starts in 2006, and we think it is now time to compile the material presented to students in the form of simple notes. There are already some good general books on the subject of nature inspired computation (see [2], [3], [1]), so our aim is not to write another one.

Acknowledgments .

Chapter 1

Introduction

1.1 Problem Solving and Computers

WHEN you look around us we often ask ourselves one simple question: How such diverse and complex things did appear? Today we have a partial answer to this question.

Chapter 2

Gradient-Based Optimization

2.1 Introduction

2.2 Algorithms

2.2.1 Gradient Ascendent

This is the most simple algorithm to find the maximum of a function, as discussed in class (see the algorithm 1). It is based on the idea of, starting at a random point, iteratively improving the result by following the direction of the gradient of the function (i.e., the derivative).

Algorithm 1: Gradient Ascendent

Input : Problem,∇f, α
Output: Best

```
1 Best ← RandomSolution(Problem)
2 repeat
3 |  Best ← Best + α∇ f(Best)
4 until found ideal solution or run out of time;
5 return Best
```

The operator ∇f computes the vector of the derivatives for each dimension. In the unidimensional case it is just $\frac{\mathrm{d}f}{\mathrm{d}x}$. The way it is defined, the algorithm try to find out the **maximum** of the function. If we want the **minimum** we just have to change the add operation to a minus in the update (line 3 of the pseudo code), and we do a gradient descent algorithm. The reason why we do so is related with the sign of the derivative. In fact, if the derivative at a point is negative and we are maximizing, the value of x must be reduced, while if we are minimizing that value must be increased. The same reasoning applies when the derivative is positive (see figure 2.1).

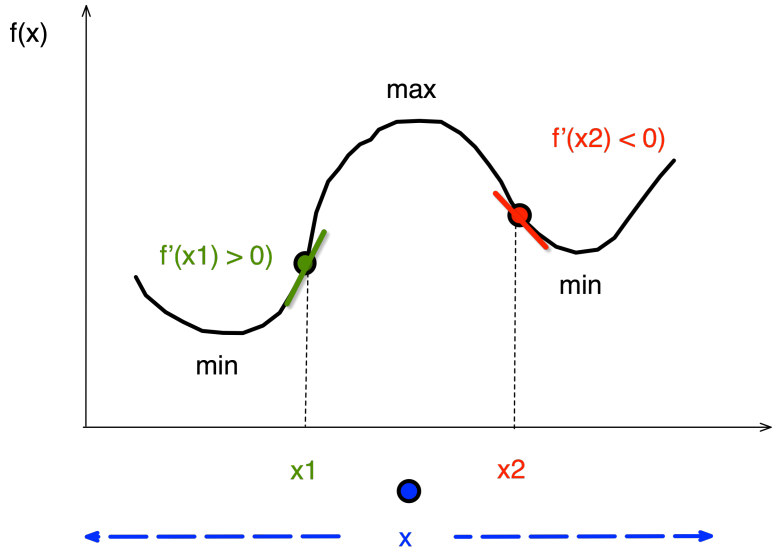

Figure 2.1: When the derivative is positive (at x_1) the new value for x must be greater if we are maximizing, smaller if we are minimizing. When the derivative is negative (at x_2) the new value for x must be lower if we are maximizing, greater if we are minimizing.

2.2.2 Random Restart

This algorithm apply the same idea but with random restart. Algorithm 2 show the pseudo-code. As expected we now have two cycles. The external one controls how many times we perform a gradient ascendent. The idea is that trying different starting points x we can avoid being trapped in a local maximum.

It should be notice here that you are not going to find a point where the derivative is **exactly** equal to zero. So you have to define an ϵ such that if $|f'(x)| < \epsilon$ you assume the derivative is zero.

2.2.3 Newton's Method

2.3 Implementation

The implementation of these methods in Python is straightforward, in particularly for the one dimensional case. As they are based in the idea of using the derivative (first and second) we start with a small program to compute it. You are free to implement your own solution!

```
def derivative(f, delta_x=0.000001):
    """Return the derivative of a function"""
    def der(x):
        return (f(x+ delta_x) - f(x)) / delta_x
    return der
```

Algorithm 2: Gradient Ascendent with random restart

Input : Problem, f, ∇f, α
Output: Best

```
1  x ← RandomSolution(Problem) ;
2  Best ← x
3  repeat
4  |  repeat
5  |  |  x ← x + α∇ f(x)
6  |  until ||∇f(x)|| = 0;
7  |  if f(x) ≥ f(Best) then
8  |  |  Best ← x
9  |  x ← RandomSolution(Problem)
10 until found ideal solution or run out of time;
11 return Best
```

If we want to visualize the functions, the derivatives and the evolution of the best solution over time, we also need some code.

```
import matplotlib.pyplot as plt

def display_function(f, x_min, x_max, delta=0.1):
    x = list(frange(x_min, x_max,delta))
    y = [f(i) for i in x]
    plt.title(f.__name__)
    plt.grid(True)
    plt.axhline(c='black')
    plt.axvline(c='black')
    plt.xlabel('X')
    plt.ylabel('Y= '+f.__name__ + '(X)')
    plt.plot(x,y, 'r')
    plt.show()
```

frange Notice the use of a function (`frange`, equivalent to `range` but that works with floats. As there is not such a thing in Python[1], let's implement it too.

```
def frange(n1,n2=None,n3=1.0):
    """
    Range with floats.
    Can be called as  with range:
    frange(n)
    frange(n1,n2)
    fange(n1,n2,n3)
    """
    if n2 == None:
```

[1] If you are using Python arrays this is not true, for we have the `arange` method in the `numpy` module.

```
        n2 = n1
        n1 = 0.0
    nextn = n1
    while (n3 >= 0.0 and nextn <= n2) or (n3 < 0.0 and nextn >= n2):
        yield nextn
        nextn += n3
```

2.4 Exercices

Chapter 3

Single-State Stochastic Search

Bibliography

[1] Anthony Brabazon, Michael O'Neill, and Sean McGarraghy. *Natural Computing Algorithms*. Springer, 2015.

[2] Leandro Nunes de Castro. *Fundamentals of Natural Computing*. Chapman & Hall, 2006.

[3] Dario Floreano and Claudio Mattiussi. *Bio-Inspired Artificial intelligence: theories, methods, and technologies*. MIT Press, 2008.