A quick comparison between the previous result using simulated annealing and gradient descent

Rafael Monteiro

Mathematics for Advanced materials - Matham-Oil, Japan

May 21, 2019

Abstract: these are short notes that I have been writting alongside a paper in material informatics; they are coming naturally and I am just letting the words come, without much editing. These notes are, by no mean, intended to be original (I'm keeping the interesting things for the paper).

Introduction

The main idea of this note is to compare the previous algorithm with the classical gradient descent method.

As you might already know, GD is based on a local search, where, if you are allowed to choose directions with length $|v|$ at any interior point $x \in \Omega \subset \mathbb{R}^n$ and want to maximize a differentiable function $f(\cdot)$ then a good choice is to move to

$$x \leftarrow x + \alpha \nabla f(x). \tag{1}$$

I used the notation $\leftarrow$ to denote the "assignment statement =" used in programming (otherwise (1) is no longer true). Unlike the previous note I wrote on simulated annealing, this is a deterministic algorithm.

This is all standard, but we can still get something interesting out of this discussion. First, I would like to contrast the numerical efficience of this code with the stochastic optimization code doing simulated annealing. Second, I would like to introduce a programming technique called parallelization.

[884]:
```
import time
import numpy as np
import multiprocessing as mp # parallelization library
```

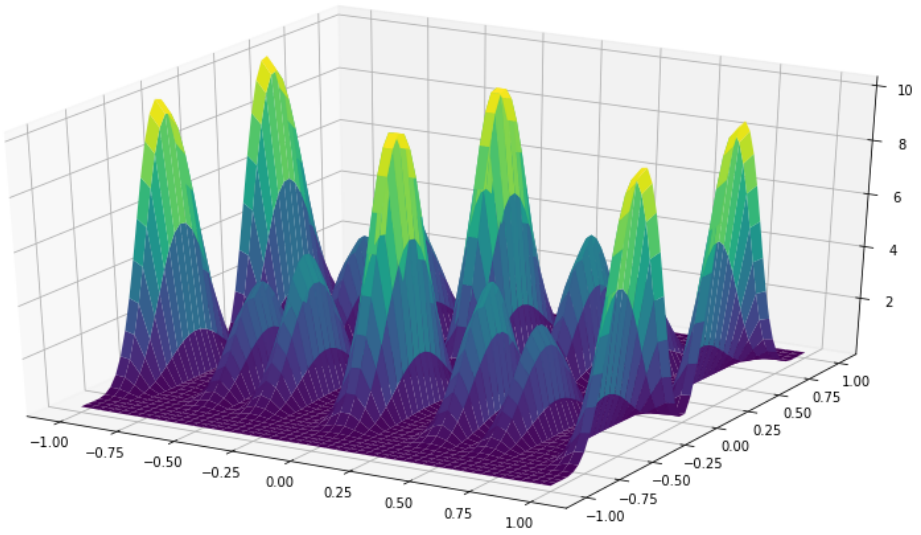
1 Gradient Descent method

To begin with, let's implement the gradient descent method. We shall need a few parameters.

a. learning rate;
b. iteration number.

The largest the learning rate, the fastest your algorithm with adapt to new data (that is, data that has a different probability distribution). The iteration number give us an upper bound of number of iterations until we stop the code.

[885]:
```
import numpy as np
from sympy import *
from sympy.plotting import plot3d
import matplotlib.pyplot as plt
import time

# define the function and its variables symbolically
x, y = symbols('x y')
f = (cos(16*x)+ cos(3*x))**2*(sin(2*y)+sin(5*y))**2
lam_f = lambdify((x,y),f)

# lambdify the gradient, ie., export it as a numpy function
f_1 = lambdify((x,y),diff(f,x))
f_2 = lambdify((x,y),diff(f,y))

def grad_g(a,b):
    return np.array([f_1(a,b),f_2(a,b)])
    #lam_g = lambdify((x,y),np.squeeze(g),'numpy')

## Changing the picture size
plt.rcParams['figure.figsize'] = 15,8
## ...and plotting it
plot3d(f, (x, -1, 1), (y, -1, 1))
```



[885]: `<sympy.plotting.plot.Plot at 0x13ad039e8>`

[984]:
```
def gradient_descent(v, numb_iterations, learning_rate,interval):
    v_prev = v
    cost=[]
    path = np.reshape(v,(2,1))
    for i in range(numb_iterations):
        v_next = v_prev+ learning_rate*grad_g(v_prev[0],v_prev[1])

        if(np.linalg.norm(v_prev-v_next)/np.linalg.norm(v_prev)<0.001): ## Error␣
↪tolerance
            break
        v_prev = v_next
        if(i%interval==0):
            cost = np.append(cost,lam_f(v_next[0],v_next[1]))
            path = np.append(path,np.reshape(v_next,(2,1)), axis=1)

    return v_next,path,cost
```

This method will clearly not work for the initial point we picked: the gradient at $(x, y) = (0, 0)$ is zero, therefore, regardless of how numb_iteration and learning rate are chosen, in the end $y = x$.

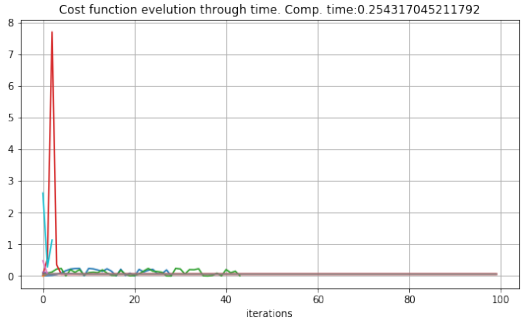
Ok... that's unfortunate, but not everything is lost: you are going to randomly initialize your code; why to start at (0,0) anyway?

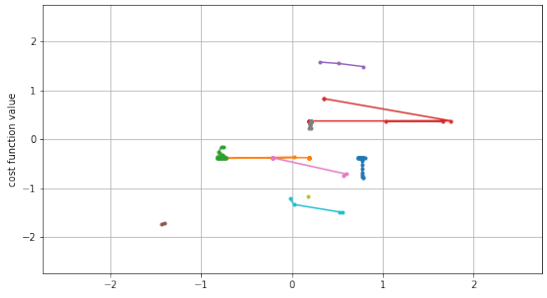
[985]:
```
def initialization():
    v = np.random.randn(2,1)
    return v
```

[986]:
```
numb_iterations=1000
learning_rate = 0.01
interval=10
N = 10

def get_gradient_descent():
    z  = initialization()
    _,path, cost = gradient_descent(z,numb_iterations,learning_rate,interval)
    return path, cost

cost= {}
path= {}

tic = time.time()
plt.figure(figsize=(20,5))
for j in range(N):

    path[str(j)] ,cost[str(j)] = get_gradient_descent()
    plt.subplot(121)
    plt.grid(True)
    plt.plot(range(len(cost[str(j)])),cost[str(j)],color="C"+str(j))
    plt.subplot(122)
    plt.set_cmap('autumn')
```

```
    plt.grid(True)
    plt.plot(path[str(j)][0,:].T,path[str(j)][1,:].T,marker='.',color="C"+str(j))

x_min= +np.min([path[str(j)].min() for j in range(N)])
x_max= +np.max([path[str(j)].max() for j in range(N)])
y_min= +np.min([path[str(j)].min() for j in range(N)])
y_max= +np.max([path[str(j)].max() for j in range(N)])

ellapsed = time.time()-tic  ##computation time
# Adding legends
plt.subplot(121)
plt.title("Cost function evelution through time. Comp. time:"+str(ellapsed))
plt.xlabel('iterations')

# Adding legends
plt.subplot(122)
plt.xlim([-1+x_min,x_max+1])
plt.ylim([-1+y_min,y_max+1])
plt.ylabel('cost function value')

plt.show()
```

```
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:22:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:36:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:41:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```





Wow... this looks like a mess!

Wait... don't panic! Let's try to figure out what is going on. First, why some cost functions terminate too early? That's because we imposed a constraint in the inner loop: if there is not enough variation from one point to the next, then we should stop (I denote that in the code as "Error tolerance").

Ok, so let's take a look at the function that we are trying to maximize (in the optimization literature this is called **cost function**); note that the function is very rough, plagued with peaks and crests. Moreover, they are also very steep (i.e., high norm derivatives).

Let's look again at the above graph, on the left hand side. In some cases the function reaches a plateau: that's a local maximum! In others (the blue curve, for instance) the cost function values keeps going up and down: the reason is that, if local maxima are sufficiently far apart from each other each iteration would stay close to a peak (if you are a dynamicist, you might thing of peak neighborhoods as "basins of attraction"). However, since there are many peaks, the orbit hops from peak to peak. Another reason for that to happen is the learning_rate parameter being too big, so it keeps "missing the point".

> **Remark:** in ML people call this a "hyperparameter", because it is not part of your model.

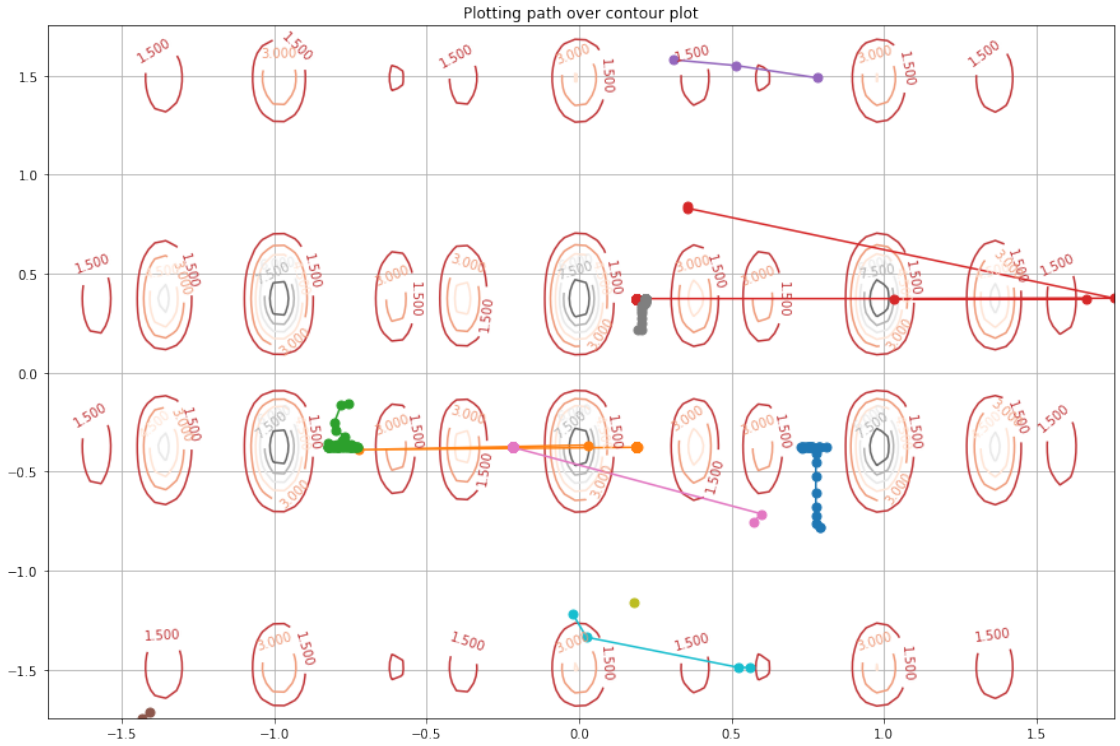
It is not hard to see that we will have to run the optimization code many times in order to get the highest value.

Let's illustrate the previous analysis seeing the path of these points on top of the contour plot of this function:

[991]:
```
import matplotlib.cm as cm
from collections import OrderedDict

x = np.linspace(x_min,x_max,100)
y = np.linspace(y_min,y_max,100)

## meshing the grids in order to do contour plot
xx, yy = np.meshgrid(x, y)
zz= lam_f(xx,yy)

## Finally, plotting the contour curve
fig, ax = plt.subplots(figsize=(15, 10))
plt.grid(True)
```

```
CS = ax.contour(xx, yy, zz,cmap='RdGy')
plt.set_cmap('RdBu')

## Below, I plot the paths on top of the previous, topography, plot
for j in range(N):
    plt.plot(path[str(j)][0,:].T,path[str(j)][1,:].T,marker='o', c="C"+str(j),␣
 ↪markersize=7)

ax.clabel(CS, inline=1, fontsize=10)
ax.set_title('Plotting path over contour plot')
```

[991]: `Text(0.5, 1.0, 'Plotting path over contour plot')`



You see? The relentless iterates "walk" around and pass through some of the local maxima: that's why we see the ups and downs in the dynamic behavior of the cost function. This is mostly due to either a big learning_rate or a high derivative.

2 Parallelization

As we would like to run many instances of the same code, we can use parallelization. In other words, let's say that we would like to try N iterations of this code (some people call them "epochs"). Without parallelization, your code would look like this:

Now we are going to parallelize it. The library you need in this case is the multiprocessing library.

Another thing that I will do in these notes is to parallelize my code. There are many instances where you can do it, for instance: 1. your data is too big and you need to chunk it into smaller pieces; 2. you have a function running on independent parts of your dataset.

We are in the second group. As we shall see later, this consists of parallelizing a "for loop".

The first thing we do consists of importing some parallelization libraries and counting the number of cores your machine has:

[992]:
```
import multiprocessing as mp

number_core = mp.cpu_count()
```

Now we initialize the parallelization. Just for you to get a feeling of what happens, I will just code one of the main parts of the parallel code (this is the parallelize for loop):

[994]:
```
pool= mp.Pool(number_core)

outcome = pool.starmap(get_gradient_descent,[() for j in range(N)])

pool.close()
```

Each one of the N elements in the outcome consists of two entries: one for the path, the other for the cost. For instante

[995]:
```
np.shape(outcome[0])
```

[995]:
```
(2,)
```

is a tuple with 2 elements, where one is the path, the other the cost.

> **Remark (on parallelization):** this is indeed one of the most complicated part in parallelizing things, namely, figuring out a way to reorganize the outcome. Most of the times the best way to do it is by putting a new argument in your function that will flag/enumerate it and, consequently, the outcome. For instance, the function get_gradient_descent() would have to be rewritten to get_gradient_descent_numbered(i), and the outcome would be (i,get_gradient_descent). Notice that i is not a useful computation value, but used only to reorganize your results.

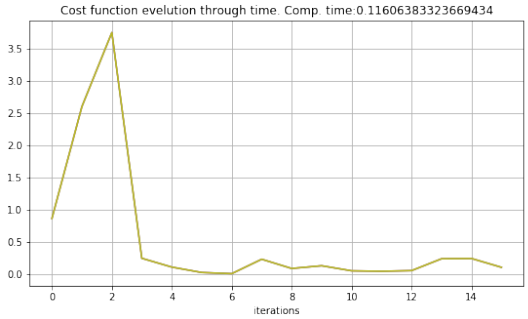
The parallelized code becomes

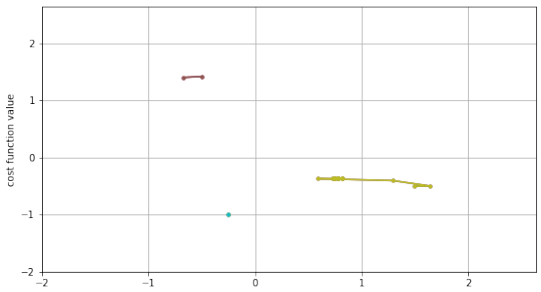
[1006]:
```
plt.figure(figsize=(20,5))

cost_par= {}
path_par= {}

tic = time.time()
## Parallel loop
pool= mp.Pool(number_core)

outcome = pool.starmap(get_gradient_descent,[() for j in range(10)])

pool.close()
```

```
## Now essetially everything is the same, we just need to reorganize the outcome

for j in range(N):
    path_par[str(j)], cost_par[str(j)] = outcome[j]
    plt.subplot(121)
    plt.grid(True)
    plt.plot(range(len(cost_par[str(j)])),cost_par[str(j)],color="C"+str(j))
    plt.subplot(122)
    plt.set_cmap('autumn')
    plt.grid(True)
    plt.plot(path_par[str(j)][0,:].T,path_par[str(j)][1,:].T,marker='.
 ↪',color="C"+str(j))

x_min_par= np.min([path_par[str(j)].min() for j in range(N)])
x_max_par= np.max([path_par[str(j)].max() for j in range(N)])
y_min_par= np.min([path_par[str(j)].min() for j in range(N)])
y_max_par= np.max([path_par[str(j)].max() for j in range(N)])

ellapsed = time.time() - tic ##computation time
# Adding legends
plt.subplot(121)
plt.title("Cost function evelution through time. Comp. time:"+str(ellapsed))
plt.xlabel('iterations')

# Adding legends
plt.subplot(122)
plt.xlim([-1+x_min_par,x_max_par+1])
plt.ylim([-1+y_min_par,y_max_par+1])
plt.ylabel('cost function value')

plt.show()
```

```
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:20:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:23:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:37:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
```

```
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
/miniconda3/lib/python3.7/site-packages/ipykernel_launcher.py:42:
MatplotlibDeprecationWarning: Adding an axes using the same arguments as a
previous axes currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this warning can be
suppressed, and the future behavior ensured, by passing a unique label to each
axes instance.
```





So...what is going on here? Is there a bug in the code? No. What is happening is that everytime we run the code in parallel we get similar random numbers. Hence, we get the same outcome. For example

[1013]:
```
pool= mp.Pool(number_core)
outcome = pool.starmap(np.random.rand,[(2,1) for j in range(10)])
pool.close()
```

[1014]:
```
np.reshape(outcome,(10,2))
```

[1014]:
```
array([[0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065],
       [0.62160452, 0.88370065]])
```

On the other hand, if we run it sequentially we get something totally different

[1015]:
```
v = [np.random.rand(2,1) for j in range(10)]
```

[1016]:
```
np.reshape(v,(10,2))
```

[1016]:
```
array([[0.62160452, 0.88370065],
       [0.16477124, 0.64091096],
       [0.09751658, 0.73630907],
```

```
[0.65190727, 0.06782594],
[0.85203047, 0.60350091],
[0.59059537, 0.43319377],
[0.38609587, 0.63415255],
[0.59011052, 0.37923407],
[0.71431675, 0.57656474],
[0.77855409, 0.06521806]])
```

You see?

In fact, this is a common problem in parallelized code that has (pseudo) random numbers involved (see, for instance, this post or this post.

I will stop at this point, but it is clear how one should go from here: rewrite your code and add an internal seed() function that depends on your iteration argument, or use another parallelization library (there are others available!).

> **Last remark:** I don't want to play the devil's advocate here. Parallelization is extremely useful, and for sure it has some caveats. At least in the deterministic case you will not stumble upon the issues I point out here.