Optimizing Functions by Local Search

Author: Timothy Morren

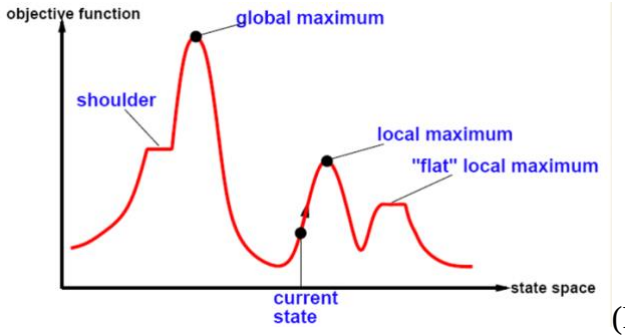
Project Description: Given a function of 1-10 dimensions, maximize the functions output:

$MAX\ (\ f(x_1, x_2, \dots, x_{10})\ )$

Approach 1: Since f is a continuous function, we pick a random starting input to f and evaluate. This produces a point or value of f (f_1). Then find a point near it but off by a small margin (or the step) and in the direction the greatest increase at that point (or the gradient) so the function produces a different point than the first but in the direction of increasing value:

$$f_{x+step*gradient} = f(x_1 + step * gradient\ , x_2 + step * gradient, \dots, x_{10} + step * gradient)$$

Now compare if the output of the new point is larger than the original, i.e. ($f_{x+step*gradient} > f_1$). If so, make that larger point the new point and repeat/loop the step addiction and the comparison process shown above until the new point is less than the current. This tells us we reached the top of the hill or a local or global maximum since moving in any direction results in a smaller value (Figure 1.). This approach is called **Hill-climbing** a.k.a. **greedy local search**: "a loop that continuously moves in the direction of increasing value".



(Figure 1.)

Disadvantages of Approach 1: Just looking at Figure 1 we can see that if we start near a "hill" but not the largest "hill", or an area called a local maximum, we will follow up that hill using greedy local search until the algorithm terminates from max iterations. This will give us a local maximum of the function but not the absolute maximum or global maximum.

Approach 2: In the first approach the step size is held constant and there is no way for the algorithm to escape a local minimum. Imagine if we start with a step size and we lower it with each iteration. This way, we began to narrow in on a value quicker and with more precision. Since when we first start, we move with larger steps because we most likely did not pick the top of a hill with the first random starting point. When we get farther along in the iterations and we begin moving closer to a maximum, the step size becomes smaller and smaller, and we approach the "peak" of

the hill. The other improvement we can make on the greedy local search is getting out of a local maximum or flat local maximum (Figure 1.) by allowing movements in a direction that doesn't improve the value of the function. These are random "bad" moves which gives the algorithm the ability to try to move past the local maximum and search for the global max. These two improvements on the hill climbing search result in a new local search algorithm called **Simulated Annealing.**

Simulated Annealing: In Simulated Annealing (SA), we can manipulate the effectiveness of the algorithm by adjusting the function that decides how the step changes based on the iteration amount. We call this function the annealing schedule.

Annealing Schedule: I chose my function to be an inverse t (t is the number of iterations) function because of the end behavior fits my needs for the SA function. The limit of t as it approaches positive infinite, step size ≅ 0. I.e.

$$step(t) = \lim_{t \to \infty} (\frac{1}{1+t}) = 0$$

This function lends to how I want the function to behave, since it starts at 1, which is relatively a large step size when we are trying to find the maximum value within a tolerance of 1e^-8, and ends with 1 divided by a very large number which is a very small number.

Comparing SA to Hill Climbing: When given the same function, number of centers(C), and dimensions(D) compare which finds the larger local maximum of the function.

Results: Below is a table DxC and reports the number of times SA produced a larger local Max.

D: **C:**	**5**	**10**	**50**	**100**
1	53	50	58	51
2	23	22	25	27
3	14	15	9	22
5	7	5	3	13

Interpreting Results: According to the table of results above, as the dimensions get lager the greedy search is a better function to find a larger max value. Also, as the number of centers increase, we generally see an increase in the SA's ability to find a larger max value for the function. This is unexpected.

Code limitation/review: Since the path is irrelevant in this problem, the space usage is very low, only 2 points are ever in memory at the same time (current point and next point) so that they can be compared. The current code is only valid for dimensions 1-10 but the code could be updated to handle any number of dimensions.

References:

https://numpy.org/doc/stable/reference/random/generated/numpy.random.uniform.html

explains numpy uniform and exponents used in code.