Xiangyu Zheng 21331025 CS7CS2 Optimisation for Machine Learning week6 assignment

**(a)(i)**

def globalRandomSearch(x, y, iters, f):

First of all, the algorithm has four parameters,

the array x stores the maximum and minimum values of x,

the array y is the maximum and minimum values of y,

iters is the number of iterations,

f is the incoming equation.

for k in range(iters):  
 cx = random.uniform(x[0], x[1])  
 cy = random.uniform(y[0], y[1])  
 value = f(cx, cy)

In the number of iterations, randomly select points in the range of x and y, and calculate the function value

if minValue > value:  
 minValue = value

record the smallest function value

**(a)(ii)**

Here are the two functions from my week4:

Function1: 1 \* (x1 - 5) \*\* 4 + 3 \* (y1 - 0) \*\* 2

Function2: Max(x1 - 5, 0) + 3 \* abs(y1 - 0)

|  |  |
| --- | --- |
| Chart, line chart  Description automatically generated  Figure 1 The comparison of global random search and gradient descent with different parameter ranges is in function1 | Figure 2 Global random search and gradient descent function calculation comparison in function1 |

|  |  |  |
| --- | --- | --- |
| Figure 3 The comparison of global random search and gradient descent with different parameter ranges is in function2 | Figure 4 Global random search and gradient descent function calculation comparison in function2 | Figure 5 Time required for global random search and gradient descent under 1000 iterations. |

We can see from Figure 1 and Figure 3. With the passage of time, the changes of the optimal function values of the two algorithms, global random search and gradient descent, will be inaccurate when measured by time. This is because the content calculated by each iteration of the two algorithms is inconsistent, resulting in the required iterations. time is different.

We can see that the performance of global random search seems to be better than gradient descent, no matter which function. Global random search can always find better function values, and gradient descent slows down to a certain extent in function1, while it keeps oscillating in function2, and global random search does not have this problem. Global random search also performs well in iteration speed. But global random search seems to suffer from the range problem, the closer the range is to the optimal point, the better the algorithm performs.

We can see from Figures 2 and 4 that no matter which function it is, global random search computes the function value more times in the same time. This is because although the global random search only calculates the function value once in one iteration, and the gradient descent calculates the partial derivative value twice, the global random search iteration takes less time, so the number of times the function value is calculated per unit time is more. Figure 5 shows the time required for each of the two algorithms at 1000 iterations

**(b)(i)**

population based search has a total of 7 parameters

def populationBasedSearch(x, y, iters, Nsample, bestP, range, exploitP,):

**x** represents the range of x.

**y** represents the range of y.

**iters** indicates the number of times the function needs to be iterated.

**Nsample** represents how many random samples the algorithm needs to take at the beginning.

**bestP** indicates how many optimal points are left.

**range** indicates how many points are randomly selected in the range of the best point.

**exploitP** indicates how many points to take near the best point.

First create a small top heap for accessing the best function value and xy

stp = BtmkHeap(bestP)

random sample:

while i < Nsample:  
 cx = random.uniform(x[0], x[1])  
 cy = random.uniform(y[0], y[1])  
 value = f1(cx, cy)  
 stp.Push((value, [cx, cy]))  
 i = i + 1

start iterating：

i = 0  
while i < iters:

Obtain the optimal point, start to randomly obtain points near the optimal point within the specified range, push the function to the small top heap, and automatically calculate the best and latest points:

datas = stp.BtmK()

for data in datas:

while j < exploitP:  
 cx = random.uniform(xy[0] - range, xy[0] + range)  
 cy = random.uniform(xy[1] - range, xy[1] + range)  
 value = f1(cx, cy)  
 stp.Push((value, [cx, cy]))

**(b)(ii)**

|  |  |
| --- | --- |
| Figure 6 Gradient descent comparison of global random search and population iters = 1000 function1 | Figure 7 Gradient descent compares global random search and population x and y in function1 |

|  |  |
| --- | --- |
| Figure 10 Gradient descent comparison of global random search and population iters = 1000 function2 | Figure 11 Gradient descent compares global random search and population x and y in function2 |

It can be seen from Figure 6 and Figure 8 that no matter which function it is, the performance of population based search is the best because it records several optimal points and continues to search for points near the optimal point. But its overhead is relatively high.

As can be seen from Figures 7 and 9, the iterative paths of various algorithms, gradient descent is at a fixed starting point, global random has strong randomness, the starting point is far away, and the fluctuation is also large. Population based search selects the optimal points, the starting point is close, and the iteration is stable.

**(c)**

|  |  |  |
| --- | --- | --- |
|  |  |  |