Homework 1 : Optimization, Coordinate descent

You should submit source a python notebook along with a pdf of the notebook for all exercises. For students using phys-ugrad, If any files are distributed for this assignment they will be available in /home/3G03/HW1 on phys-ugrad

Due: Friday January 28 2022, 11:30pm on avenue, with a grace period till the following Monday at 11:30pm

☛**Reading:** Parts of the first lectures are based on *The Hundred Page Machine Learning Book* by A. Burkov. Found at http://themlbook.com. This week and next we will focus on mathematical optimization following Watt, Borhani and Katsaggelos *Machine Learning Refined* (MLR). Many of the exercises in HW1 are modifications of some of the problems in Chapter 2 of MLR.

Exercise 1 *Random Search and Rosenbrock Banana function*

In this exercise we you will implement a version of the random search algorithm. Your python routine should take several aguments so that the beginning should look like this:

```
# random search function
def random_search(g,alpha_choice,max_its,w,num_samples):
    # run random search
    w_history = []         # container for w history
    cost_history = []           # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice
```

Here, `g` is the function we're trying to minimize. `alpha_choice` is the step length/learning rate which can take 2 values, either the string `'diminishing'` or simply a numerical constant. `max_its` is the maximum number of iterations to be taken and `w` the starting vector. `num_samples` is the number of random directions that we choose. In addition we like to be able to monitor the progression of the algorithm so you should keep track of the points the algorithm visits in `w_history` as well as the associated cost in `cost_history`. These two containers should be returned by the function so you should end with:

```
    return w_history,cost_history
```

☛**Note:** For generating the random directions you might want to consult `numpy.random.randn`.

1.1 Finish the above implementation of `random_search`. You should *normalize* the directions and use `alpha` as step length. If `directions` contain *all* the generated and normalized directions

you should be able to generate *all* the candidates (`w_candidates`) for new points simply by using `w_candidates = w + alpha*directions`. Try to find an efficient way for evaluating the function in all the points, `w_candidates`.

1.2 Now we want to test the `random search` function. We want to find the minimum of the function wich is simply:

$$g(\vec{w}) = \mathbf{w}^T\mathbf{w} + 2 \;\; (= \vec{w} \cdot \vec{w} + 2) \tag{1}$$

in python we can implement this in a straight forward manner as shown below. In addition, we start the search with the following parameters:

```
g = lambda w: np.dot(w.T,w) + 2
alpha_choice = 1; w = np.array([3,4]); num_samples = 1000; max_its = 5;
```

Here, `w = np.array([3,4])` specify the starting point for the minimization. Show plots of the resulting `w_history,cost_history`.

1.3 Repeat the analysis but with a steplength of `alpha=0.3`. Did the results improve ? Explain your results.

1.4 Next we want to study a function in two dimensions with a very long, narrow and curved valley, the Rosenbrock Banana function:

$$g(w_1, w_2) = 100(w_2 - w_1^2)^2 + (w_1 - 1)^2 \tag{2}$$

This function has a minimun at $(w_1, w_2) = (1, 1)$. Make two runs with `num_samples=1000;max_its = 50;` starting from the point $(-2, -2)$. First run with a fixed step length of $\alpha = 1$ then with a diminishing steplength $\alpha = 1/k$. Compare the two runs by plotting the `cost_history`. Try to make a contour plot of the $g(w_1, w_2)$ and overlay `w_history`. The github repository for *Machine Laerning Refined* has some nice tools for making plots.
See `https://github.com/jermwatt/machine_learning_refined` .

1.5 Now let's try something a little more challenging, the six-hump camel function:

$$g(w_1, w_2) = (4 - 2.1w_1^2 + w_1^4/3)w_1^2 + w_1w_2 + (-4 + 4w_2^2)w_2^2. \tag{3}$$

In the domain $w_1 \in [-3, 3]$, $w_2 \in [-2, 2]$ it has 2 global minima at $(0.0898, -0.7126)$ and $(-0.0898, 0.7126)$. Select a starting point and use random search to locate the minima. Do you converge to a global minima or stuck elsewhere ?

Exercise 2 *Curse of Dimensionality*

Let us explore the curse of dimensionality using the above $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$. We start with an initial vector $\mathbf{w}^0 = (1, 0, \ldots, 0, 0)$ but the length of the vector (the dimension of the space) is no longer 2. Instead let's call it N. We're not going to search for the minimum but instead we shall just just look around the neighborhood of $\mathbf{w}^0$ and determine how many of P randomly generated directions are descent directions. For $N = 1, 2, \ldots, 25$ make a plot of the *fraction* of the P directions that are descent directions. Make a plot of the fraction versus N for the 4 values of $P = 10, 100, 1000, 10000$. Comment on your observations.

Exercise 3 *Coordinate Search and Coordinate Descent*

In high dimensions random search is no longer efficient. We therefore want to look at coordinate search and coordinate descent. For coordinate search, instead of generating P random directions we only look for descent among the N coordinate directions. For coordinate descent

we randomly go through the coordinate directions and the first time we find a descent we use it.

3.1 Modify your implementation of `random_search` to instead implement `coordinate_search` and `coordinate_descent`. In both cases you should use normalized directions (multiplied by the steplength). Make sure you can go both in the postive and negative direction for each coordinate. In one dimension there would be two directions, forward and backward. In two dimensions there would be 4 directions, north, south, east and west.

☛**Hint:** For coordinate descent have a look at `numpy.random.permutation`

☛**Watch out:** For a fair comparison between the 2 algorithms you should allow `coordinate_descent` to *attempt* to take N steps whenever `coordinate_search` takes one step. One way to do that is to generate a random permutation of the N directions and let `coordinate_descent` run through them sequentially before generating another permutation, letting `max_its` count the number of permutations generated. So, if `max_its` is the maximum number of steps that `coordinate_search` can take then `coordinate_descent` could *possibly* take $N \times$`max_its` step.

3.2 Consider the function $g(w_1, w_2) = 0.26(w_1^2 + w_2^2) - 0.48w_1w_2$. Starting from $\mathbf{w}^0 = (3, 4)$, using a maximum number of itereations $P = 40$ and a diminishing alpha, compare coordinate search with coordiate descent by plotting the cost history.

Exercise 4 *Gradient Descent*

Complete the notebook on autograd and Gradient Descent labelled HW1.4.ipynb on avenue.