Gradient Descent exercises

The exercises 1 and 2 are on the 1-dimensional case. It is intended that you will work within small test scripts or (for quick experiments) at the console. You can copy-and-paste pieces of code from our "GradDesc.py" file, as needed. They should be done in sequence.

Exercises from 3 onwards are on the N-dimensional case.

Exercises 1 to 3 are on the computation of derivatives with finite differences. The reason is not that this is an extremely important topic (it isn't); instead, it simply offers a nice opportunity for exercises that cover a number of topics at once. These exercises have a number of goals, among which are getting accustomed to analyze data, getting a hands-on idea on numerical errors introduced by approximations, using lambda functions, working with N-dimensional data, etc.

Exercise 4 is about refactoring (i.e. reorganizing and improving) the code.

Exercise 5 is about going beyond the pure gradient descent algorithm. Feel free to try to implement further extensions as you studied in the theory part, if you want.

Exercise 6 is about learning to use `minimize`, parsing its documentation, etc. (with the idea that if you learn to understand `minimize` documentation you'll be a long way towards being able to use most of the scipy library by yourself).

Exercise 7 is about performing several optimization attempts, changing the starting point, which is something that you may want to do when you have no clue about the function that you're trying to optimize and what might be a good starting point.

1. Second derivative with finite differences

Say that you want to compute the second derivative, in the 1-dimensional case, using finite differences. Say your function is called `g`. You can pick whatever function you want, but if you want a concrete example use the function `g` from `"GradDesc1D_final/function_examples_1d.py"`, which is a 4th degree polynomial and is adequate for this exercise and the following.

Try to get the second derivative it in 3 different ways:

1. Suppose that you have the analytical expression for the gradient of `g`, in a function called `grad_g`. Use the generic finite-differences function `grad` that we wrote (the one in `"GradDesc1D_final/GradDesc.py"`), to get the second derivative. You may call this version `grad2_naive1`.

2. Suppose that you don't have (or don't want to use) the analytical expression for the first derivative of `g`. Write a function `grad2_naive2` that takes `f`, `x` and `delta` as arguments, like our `grad` function. Inside that function, use a lambda expression to define a function `gf(x)` that returns the derivative of `f` computed at `x`, using the `grad` function (with the step `delta`). This is precisely what we did in `"GradDesc1D_final/GradDesc.py"`, except that you should pass the `delta` argument too. You can then apply `grad` again to `gf`, with the same `x` argument and the same step `delta`, and get the second derivative.

3. The previous method uses 4 function evaluations of its argument `f` to derive the result (you should be able to easily figure out why it is so; you should also be able to verify this by inserting calls to `print` inside `g`). Write a function `grad2` that does the same but with only 3 function evaluations (and without using `grad`). It should return the same result as the previous version, given the same arguments. To do this, write down on paper the finite-differences expression that we have written in class, and apply it to the first derivative:

$$f''(x) \simeq \frac{f'(x+\delta x) - f'(x-\delta x)}{2\delta x}$$

Then substitute the analogous expression for f' and simplify. You should end up with an expression that evaluates f only three times, instead of four.

You should compare the results obtained with the last method with an analytical expression, like we did for the 1st derivative. Pick a value of `x` and try to change the value of `delta`: start from a large value (e.g. `1e-1`) and decrease it (e.g. in steps of `10`, so try `1e-2`, `1e-3` etc.), and monitor the error compared to the analytical expression. Observe at which value the error stops decreasing and starts decreasing. Of course you can do all of this programmatically, writing a small script (or even just one line of code…). This gives you an optimal `delta`. If you do this for both the 1st and 2nd derivatives, for a given `x`, which one incurs into numerical precision issues first (has the smaller optimal `delta`)? Which one can be estimated more accurately? Can you make sense of why?

2. Third, fourth, n-th derivatives

Same as exercise 1, but go up to the third and then fourth derivatives.

So for example, you should write the numerical 3rd derivative in several different ways, like in exercise 2, first using a single `grad` over the analytical expression of the second derivative, then using lambda functions and numerical derivatives all the way, then writing the expression on paper and simplifying it. Analyze the numerical errors that you make when using the numerical derivatives, and inspect the results and try to make sense of what you see.

1. Instead of doing the analysis of the error manually, you should definitely use some general enough code that allows you to inspect the error vs `delta` without much effort, something that works for a generic `n`-th derivative (see the next points…)

2. First, write a recursive function `gradn(n, f, x, delta)` that computes the `n`-th order derivative by applying `grad` to the `(n-1)`-th order derivative. The base case of the recursion is `n==0`, which corresponds to no derivative at all, i.e. `gradn(0, f, x)` is the same as `f(x)`. Then `gradn(1, f, x)` is the same as `grad(f, x)`, which is to say, it's `grad` applied to `gradn(0, f, x)`. Then `gradn(2, f, x)` is `grad` applied to `gradn(1, f, x)`, etc. You will need to use `lambda` expressions to write this. Also try to work out how many times your function `f` will be evaluated every time you call `gradn`, as a function of `n`. (Hint: too many!)

3. (This is optional and rather tricky but at least think about it.) First of all, once you work out the simplified expressions for the 2nd, 3rd and 4th derivatives using finite differences, you should see that a pattern emerges with which you can write the `n`-th derivative with finite differences. If you don't see the pattern, read along. First, consider how many function evaluations the `n`-th derivative computation requires (Hint: in exercise 1, point 3, you should have obtained that it's `3` for `n=2`; if you repeated the same exercise for `n=3` you should see that it requires `4` computations, etc.). Then look at the positions where the function is evaluated (e.g. for `n=1` it's at `x+delta` and `x-delta`; for `n=2` it's `x+2*delta`, `x` and `x-2*delta`; etc.) Next, look at the coefficients in front of those function evaluations, neglecting their sign. For `n=1`, they are `(1,1)`. For `n=2`, you should have got `(1,2,1)`. For `n=3`, you should have got `(1,3,3,1)`, etc. Finally, look at the pattern of the signs in front of the coefficients.

   If you manage to put the above observations together, you could exploit the pattern and write an alternative version of `gradn` that does not use recursion and performs fewer evaluations of the function. You may want to use the `scipy.special.binom` function. If you exploit broadcasting, you can do it in 5 lines of code of fewer (even a really long and unreadable single line of code, actually, but I advise against that).

3. Hessian matrix

We now consider the N-dimensional case.

Write a function called e.g. `grad2` that works like our `grad` function (uses finite differences) but returns the Hessian matrix, i.e. the matrix of all second order partial derivatives of a function:

$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

Of course you need to have a finite-differences formula: that is just the finite differences formula applied along the component i and then again along the component j (or vice versa, it's the same). Here is the expression, assuming that δ_i is a **vector** that represents a small change of δ (a scalar) along the component i, and the same for δ_j:

$$H_{ij}(x) \approx \frac{f(x+\delta_i+\delta_j) - f(x+\delta_i-\delta_j) - f(x-\delta_i+\delta_j) + f(x-\delta_i-\delta_j)}{4\delta^2}$$

So compared to the 1-d case, it's 4 terms instead of 2, and you end up dividing by `delta**2`.

The most straightforward implementation is similar to the function `grad`, but you will need 2 for loops instead of 1. But **be careful**: make sure that you deal correctly with the diagonal case $i = j$.

Because it's incredibly easy to make mistakes here, be sure to verify that it works correctly on the function `g` that is defined in the file `"GradDescND_final/function_examples.py"`, by writing the Hessian matrix of `g` by hand (it's super easy).

4. Interface improvements to GradDescND

Let's take a hint from scipy's `minimize` and improve a little the interface of our `grad_desc` algorithm, and also organize things a little better.

1. Make the algorithm more verbose. Add an optional argument to `grad_desc` called `verbose` that defaults to `False` and determines whether the algorithm should print what it's doing at each iteration or not (the current version doesn't print anything). When it's on, it should print some message at each iteration, about how many iterations were performed, what's the value of `x`, of `f(x)` of the gradient, of the gradient norm… Also, print a message at the end to tell you whether the optimization succeeded or not (inspecting the `converged` variable).

2. Create a class inside the `"GradDescND.py"` file, called `GDResults`, that keeps the results of the gradient descent algorithm. This class should have the following attributes: the final `x`, the final value of the function, the number of iterations, a bool that records whether the algorithm was successful, and finally the list of intermediate values `xs`. So add a constructor to the class that fills in all of these attributes. The idea is that you would construct such an object at the very end of the `grad_desc` function, so the constructor just gets the quantities and "packs them up" inside the object, and then returns that.

3. Define a `__repr__` method for the class that prints all of this information, except for the intermediate values. Take inspiration from scipy's `minimize` output.

4. Now modify the `grad_desc` function and options. First, at the end build a `GDResults` object and return that from the function, as we were saying. Then, add an option called `keep_intermediate`, with default `False`, that determines whether to store lists with the intermediate results of the computation or not. If not, just leave the lists empty when you put them in the `GDResults` object. This should speed up things (and save memory) in case you don't need the values (and assuming `verbose` is off, printing takes time too). Finally, change the `verbose` option to a `verbosity` option that can take integer values, where `0` means no verbosity at all, `1` or more means that the final result is printed (just call print on the `GDResults` object so that the `__repr__` method is invoked), `2` or more means that the intermediate computations are also printed.

   NOTE: if you don't need the intermediate values, then using `x -= alpha * p` instead of `x = x - alpha * p` is more efficient, since it works in-place without creating new arrays. So you can write this that way. But then you have a problem if you do want to keep the intermediate values, because you need to make sure that whenever you append `x` to the list it's not going to get updated at the next iteration… See the comments in the original file. Solving this is easy, but understanding it is tricky. It's a good exercise to see whether you understand the issues with mutability and references.

Having done these modifications, let's now modify the `"run_test.py"` and `"fit_test.py"` scripts and change

them so as to adapt them to the new return value of `grad_desc`. In both cases you will want to use the `keep_intermediate` functionality as they're used for plotting.

This overall organization is much cleaner, since everything is logically separated but all necessary interactions between different parts of the code are possible, and time-consuming parts of the code can be disabled if they're not strictly needed. It's classical refactoring step.

As a final touch, you may try to add documentation to the `grad_desc` function. This is done by simply writing a triple-quoted string between the `def` line and the body of the function. Whatever you write there, should appear in the Spyder help panel when you use the help, e.g. if you write `grad_desc` on the console and then press `CTRL+I` (on my laptop at least, I'm not sure what's the shortcut on Macs…)

5. Nesterov's accelerated gradient (AKA Nesterov's momentum)

A simple but effective variant of the steepest descent method uses Nesterov's accelerated gradient, as you saw in the theory part. Here's a remainder.

The original steepest descent formula is:

$$x^{t+1} = x^t - \alpha \, \nabla f \, \left(x^t\right)$$

where t is the iteration step and the initial point x^0 is chosen arbitrarily.

The modified formula involves an auxiliary vector v, which is also updated during the iteration and it's initialized to all zeros. It also involves an additional scalar parameter $\beta \in [0, 1]$. The update rules are:

$$v^{t+1} = \beta \, v^t - \alpha \, \nabla f \, \left(x^t + \beta \, v^t\right)$$

$$x^{t+1} = x^t + v^{t+1}$$

Notice that the gradient is not computed in the current point x^t, but in a shifted position (this is Nesterov's momentum; if it were computed in the current point it would be "standard" momentum).

If `beta` is `0`, this reduces to the original steepest descent algorithm. Implement this algorithm (in the N-dimensional version of the code) and experiment with different values of `beta`. Check the effect on convergence speed and see what's happening. You can use the improved interface and organization performed in the previous exercise and compare the trajectories with and without `beta`, also varying `alpha` and observing the effect on the convergence and the number of iterations required.

Hint: run the algorithm with different parameters, record the results and plot the trajectories with different colors to compare them.

If you're so inclined, you may want to add an option to `grad_desc` (call it `nesterov`) that determines whether to use Nesterov's momentum or standard momentum.

Note: There are many equivalent formulations of the update rule. You may want to work out what is the simplest form in which you can write it. Also, our example functions may not be the best showcases for this method: it gives an improvement, but in other cases the improvement is larger. You may want to experiment with different functions…

6. Experiment with `minimize`

You can look at the `minimize` part of `"runtests.py"` and experiment with different optimization methods. You can do more or less the same sort of experiments as done in exercise 5, e.g. you can try different optimization methods and look at their trajectories using different colors, etc.

Also, you can try to pass the arguments `jac` (the explicit gradient, since you have the analytical expressions) and maybe (but it only makes sense for the `Newton-CG` method) even the `hess` argument (but you need to write the Hessians yourself, or you can use the `grad2` function of exercise 3; in all those cases the results should be

basically identical). (Please note that outside of these toy examples having the explicit expressions for the first and second order derivatives can make a significant difference in terms of computational time, and sometimes in terms of precision of the result, so it's useful to know how to do it. Also how to interpret the documentation.)

You can also try to pass the option for constrained optimization, by adding some bounds: say for example that you want to find a solution not in the whole $\mathbb{R}^2$, but you want to restrict yourself to $[0, 3] \times [1, 2]$. Read the documentation for the `bounds` option and use it. Inspect the result visually and see if it makes sense. Maybe you can try to plot a colored rectangle on the surface plot or on the contour plot that indicates the acceptable region, and check whether the optimization algorithm actually stays there. Experiment.

You should also try to figure out how exactly to pass the other options that we had in our code, like the maximum number of iterations. You might notice from the documentation that most solvers (e.g. "BFGS") use the "infinity-norm", instead of the 2-norm. (The "infinity-norm" simply returns the maximum of the abolute value over an array, you could also implement that by the way.) Figure out how to make them use the 2-norm instead. Note however that none of the methods in `minimize` require us to specify the step size `alpha` (they all compute it automatically in a sense), so there is no equivalent for that.

7. Random guesses

Do the following exercise with `grad_desc` first, then repeat it with `minimize` (only minimal changes are required).

As we said in class, gradient methods perform an optimization operation that can go very wrong (produce garbage) if you start from the wrong guess and the function that you want to optimize is nasty. The one that we have in `"fit_test.py"` is, indeed, quite nasty, in that it has a lot of local minima and if you guess wrong the resulting fit is horribly wrong. Check by yourself that this is true.

Choose some range of values for the parameters `a`, `b`, `c` (however you want, but make them farily large, and ensure that at least one between `b` and `c` is always non-negative). Then write a `for` loop that performs a number of trials (say, `20`, or `100`…). At each loop, pick at random a value for `a`, `b` and `c` within their range, and use that as the initial guess for the optimization algorithm. After each optimization attempt, check that the algorithm has converged and the resulting `loss` (assuming you did exercises 4, that's part of the returned result for both algorithms, so you don't need to recompute it). Keep the best loss value and its corresponding optimal parameters `a`, `b` and `c`. At the end of the trials, print the overall optimal value that you have found.

You can do the above with several methods (at least for `minimize`) and see if anything changes. You can also try to do some statistical analysis: how often does `minimize` produce a better result than `grad_desc`? To test that, keep a list with all the optimization attempt results, not just the last one. Compare the resulting lists for `minimize` and `grad_desc`: for each, how many times did you get a value that's close to the absolute minimum, for example? Plot a histogram of the two lists too.