IMT574 PS04

Your name:

Deadline: Tue, Feb 26, 11:59pm

Introduction

In this homework you are asked to implement the Gradient Descent algorithm and apply it to a simple function, and to linear regression loss function.

Please submit a) your code (notebooks) and b) the final output form (html or pdf).

Note: it includes questions you may want to answer on paper instead of computer. You are welcome to do it but please include the result as an image into your final file.

Working together is fun and useful but you have to submit your own work. Discussing the solutions and problems with your classmates is all right but do not copy-paste their solution! Please list all your collaborators below:

1.

2. ...

1 Gradient Descent

1.1 The Algorithm

Gradient Descent (GD)[1] is a popular method to find maxima (and minima) of functions, widely used in various machine learning applications. Your task here is to write your own GD algorithm that can find the maximum of high-dimensional quadratic function.

Ensure you are familiar with GD. Consult the notes titled *machineLearning* (in canvas/files/readings).

The idea with GD is the following (see the notes for more details):

1. Start somewhere: pick an initial value of the parameter $\mathbf{x}^0 = (x_1^0, x_2^0, \ldots, x_n^0)$.
2. Compute the *gradient* of your objective function $f(\cdot)$ at $\mathbf{x}^0$. It shows the direction of the steepest ascent of the objective function.

   Compute gradient at $\mathbf{x}^0$, $\nabla f(\mathbf{x}^0)$.
3. Take a step of suitable length from $\mathbf{x}^0$ in the opposite direction to the gradient (direction of the steepest descent). This will land you in a new place we call $\mathbf{x}^1$. This is our new best bet for the location of maximum.
4. Are we in the correct place? There are several ways to check (these are called *stopping conditions*):
   (a) Gradient is very small. (Why does this indicate that maximum has been found?)
   (b) Function value does not decrease any more.
   (c) You also need the bail-out criterion: stop if the process has been repeated too many times.

[1] Gradient Ascent (GA) is just the "mirror image" of gradient ascent, essentially the same method.

5. If we are in correct place, we are done. $\mathbf{x}^1$ is our solution. If not, set $\mathbf{x}^0 \leftarrow \mathbf{x}^1$ and repeat from step 2.

1.2 Implement The 1D GD Algorithm

Now it's your turn!

1. Start slow and manually. Let's look at 1D case with $f(x) = x^2$.
   (a) What is the correct location of the minimum of this function?
   (b) What is the gradient vector of the function? What is it's dimension?
   (c) Now pick your starting value x^0 (don't pick $x^0 = 0$), and set the learning rate $R = 0.1$. Compute the gradient at x^0, $\nabla f(x^0)$.
   (d) Compute $x^1 = x^0 - R\nabla f(x^0)$.
   (e) Did we move closer to the minimum?
2. Now repeat the previous exercise on computer.
   (a) Choose at least one stopping criterion (e.g. $\|\nabla f(\mathbf{x})\| < \epsilon$), and the corresponding parameter ϵ. 10^{-6} is a good bet but for testing you may want a larger number, like 0.01. Also choose the bail-out number of iterations, N. For testing, something like 100 will do but for the "production run", you may need a value up to 100k. Here you may keep R rather large, 0.1 for instance, but further down you need much smaller values, perhaps 10^{-4}.
   (b) Define functions $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$.
   (c) Implement the above as an algorithm that at each step prints out the x value, the function's value $f(x)$ and gradient $\nabla f(x)$. At the end it should print the solution, and also how many iterations it took for the loop to converge (finish).
3. Experiment with different starting values, learning rates and stopping parameter values. Comment and explain your findings.

Congrats! You have implemented the 1D gradient descent! This wasn't too hard, right?

1.3 Implement The 2D Version

Now we get more serious and take 2D quadratic function. From now on everything will be done in matrix form because this scales–there is little difference between 2D and 200D problem if your code is written in that way. In matrix form the quadratic function can be expressed as

$$f(\mathbf{x}) = \mathbf{x}'A\mathbf{x} \tag{0.1}$$

where $\mathbf{x} = (x_1, x_2)'$ is a 2×1 matrix (column vector) and $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ is a 2×2 matrix. We only look at cases where A is symmetric and positive definite (i.e. all it's eigenvalues are positive).

1. What's the location of the true minimum of $f(\mathbf{x})$?
2. Program the function $f(\mathbf{x})$ as a python function that takes an argument $\mathbf{x}$ of a numpy vector, and returns a single number.

3. Compute and program the gradient vector of this function. It should take a numpy vector as argument $\mathbf{x}$ and return a numpy vector (gradient).

   This can be done in two ways: first, you can use the matrix calculus (see the notes, in appendix) and show that

   $$\frac{\partial}{\partial \mathbf{x}} \mathbf{x}' \mathsf{A} \mathbf{x} = 2\mathsf{A}\mathbf{x}.$$

   (compare with the 1D case: $\mathrm{d}(ax^2)/\mathrm{d}x = 2ax$.)

   However, if you don't know matrix calculus (I don't expect you to know it), you should do the following: a) manually multiply the elements in (0.1); b) take the derivatives of the result with respect to x_1 and x_2; and c) write these derivatives underneath each other as a vector. This is the gradient vector (see an example in the notes). You should get the same result as $2\mathsf{A}\mathbf{x}$.

   It is advantageous to use matrix operations, not looping over individual elements, in both f and gradient.

4. Now adapt the algorithm above for 2D case. It is rather similar to the 1D case if you do everything in matrix form.

5. Show that you get the correct solution if you pick $\mathbf{x}^0 = (2, -3)'$ and $\mathsf{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. (note: this is perhaps the simplest 2D quadratic function to handle).

   Note: you have to find a suitable learning rate. If it is too large, you overshoot (the algorithm jumps between positive and negative values in this case). If it is too small, you hardly see any improvement. Here $\mathsf{R} = 0.01$ or so should be a suitable bet. Try other values too.

6. Visualize your algorithm's work. You are welcome to do a 3D figure but here we expect you to stay in 2D.

   - plot the contours of the function. You have to create the grid matrices of x_1 and x_2 (see `np.meshgrid`), at each point of the grid, compute the function. Contours can be plotted with `plt.contour`. You want to use fixed 1:1 aspect ratio, otherwise the path does not look right. You can pretty much copy-paste your code of lab05 here.
   - mark all the intermittent points (the $\mathbf{x}^1$-s) on the plot and connect these with lines.
     Note: now you want to save, instead of print, these values in your code, or just create the contour plot earlier and then just add these on the plot.
     Note2: don't mark (and save) more than 100 or so points. The figure becomes incomprehensible and the algorithm terribly slow.
   - Explain what you see.

7. As before, experiment with a few different learning rates and see how does this influence the speed of convergence. Comment and explain your findings.

8. Finally, let's try if your code scales to 5D case without any modifications. Take

   $$\mathsf{A} = \begin{pmatrix} 11 & 4 & 7 & 10 & 13 \\ 4 & 17 & 10 & 13 & 16 \\ 7 & 10 & 23 & 16 & 19 \\ 10 & 13 & 16 & 29 & 22 \\ 13 & 16 & 19 & 22 & 35 \end{pmatrix} \tag{0.2}$$

   (generated as $\frac{1}{2}(\mathsf{A}+\mathsf{A}')+10\cdot\mathsf{I}_5$ where A is a 5×5 matrix of sequence $1\ldots25$, and I_5 is the corresponding unit matrix.) Take the initial value $\mathbf{x}^0 = \mathrm{c}(10, 20, 30, 40, 50)'$. Play with hyperparameters until you achieve convergence! Show and comment your results. In particular I'd like to see your comments comparing the easiness and speed of getting the 1D, 2D and 5D case to converge.

1.4 Condition Numbers

How easy it is to get your algorithm to converge also depends on how close or far is your matrix from being singular. This can be done using condition numbers. Let's stay with 2D case. Now let's return to 2D case. Take a simple singular matrix $\mathsf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$.

9. Show, on computer, that it is singular! Show it without error messages but using relevant linear algebra functions/properties instead.

Let's make it non-singular by adding a "certain amount" α of unit matrix to it: $\mathsf{C} = \mathsf{A} + \alpha \cdot \mathsf{I}_2$. Your task is to analyze and visualize the algorithm's work for three cases: $\alpha \in 100, 1, 0.01$.

Note: you may have to set different hyperparameters (R and stopping criteria) for different α-s, so the following does not work too well in a loop.

10. Run your algorithm for each of these three cases. For each α:
    (a) compute the corresponding C
    (b) print it's eigenvalues and condition number.
    (c) solve the problem using GD
    (d) report the hyperparameters, in particular learning rate
    (e) report the location of maximum and the number of iterations it took
    (f) visualize the process as you did above.
11. Comment on your results. How is the convergence speed related to the matrix condition number?

1.5 Loss function of linear regression

So far we played just with a toy function. Now let's do something more serious: implement linear regression using GD algorithm.

Linear regression aka *least squares*, hints that your task is to minimize a quadratic function: sum of squared deviations between the predicted and actual values. We keep the task easy here and use a 2D case: a model with just one explanatory variable and the intercept, i.e. just 2 parameters. You should use your code from above, and only change the function and gradient implementations.

Note that in case of linear regression we usually call the features as $\mathbf{x}$ and the parameters as $\boldsymbol{\beta}$. So we are optimizing over $\boldsymbol{\beta}$-s while we know all the $\mathbf{x}$-s. $\mathbf{x}$ in case of linear regression is something you know, there is nothing to optimize over. The $\boldsymbol{\beta}$-s are the unknowns.

Proceed as follows:

1. Create a smallish random vector of x-s (say, 100). Add to this vector a column of ones, so you get the design matrix X where the first column is ones, the second one are random numbers.
2. Create the random disturbance vector $\boldsymbol{\epsilon}$.
3. Pick the value of the parameter vector $\boldsymbol{\beta}^*$. $(1, 1)$ is a good choice but you can go for something else. This is the true value of your parameters.
4. Compute $\mathbf{y} = \mathsf{X}\boldsymbol{\beta}^* + \boldsymbol{\epsilon}$. Later we "forget" $\boldsymbol{\beta}^*$ and $\boldsymbol{\epsilon}$ and only use data about X and $\mathbf{y}$. We hardly ever know the true value, but we will estimate it.

5. Implement the loss function. It should compute the loss (a single number) as a function of $\boldsymbol{\beta}$. This obviously depends on both X and $\mathbf{y}$.

   Given argument $\boldsymbol{\beta}$, the loss function must a) compute predicted $\hat{\mathbf{y}} = \mathsf{X}\boldsymbol{\beta}$; b) compute the deviations $\boldsymbol{e} = \hat{\mathbf{y}} - \mathbf{y}$; and c) square and sum all these deviations, in matrix form $\mathsf{L} = \boldsymbol{e}'\boldsymbol{e}$.

6. Implement the gradient of the loss function as a function of $\boldsymbol{\beta}$. It depends in a similar fashion on X and $\mathbf{y}$. The notes gives the gradient of the linear regression in a matrix form (you can also use non-matrix form but that is noticeably slower and messier).

Now you are basically done. Use the algorithm you wrote above but ensure you don't overwrite your data (X and $\mathbf{y}$) there.

7. Use your GD algorithm to estimate the parameter values (you have to pick the initial value $\boldsymbol{\beta}^0$). If implemented well, you don't have to change your GD code at all. You may have to tinker with the hyperparameters again. Visualize the process exactly as above.

8. Did you get the estimates close to the true values?

1.6 Extra challenge (not graded)

So far we assumed the learning rate R is constant you fix before you start estimation. This is simple but a little too rigid approach as you have to find balance between overshooting and moving too slow.

An alternative approach is to make the learning rate adaptive. For instance, if you move in the correct direction (i.e. downhill for GD, uphill for GA), you increase R by a small percentage, for instance 10%, at each step. But if you overshoot, you decrease it by a large amount, for instance divide it by 4.

Does this lead to a faster convergence in the examples above?