Master Degree in Artificial Intelligence
Statistical and Mathematical Methods for Artificial Intelligence

2023-2024

Homework 3: Optimization

Optimization via Gradient Descent

In this homework, we want to study methods to solve the general optimization problem where, given a function $f : \mathbb{R}^n \to \mathbb{R}$, we want to compute

$$x^* = \arg \min_{x \in \mathbb{R}^n} f(x) \tag{1}$$

In particular, we will consider the situation where $f(x)$ is at least differentiable, which implies that we can compute its gradient $\nabla f(x)$.

In this framework, one of the most common way to approach (1) is to use the Gradient Descent (GD) method, which is an iterative algorithm that, given an initial iterate $x_0 \in \mathbb{R}^n$ and a positive parameter called step size $\alpha_k > 0$ for each iteration, computes

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) \tag{2}$$

You are asked to implement the GD method (2) in Python and to test it with some remarkable functions.

- Write a script that implement the GD algorithm, with the following structure:

```
Input:
    f: the function f(x) we want to optimize.
       It is supposed to be a Python function, not an array.
    grad_f: the gradient of f(x). It is supposed to be a Python function, not an array.
    x0: an n-dimensional array which represents the initial iterate.
    kmax: an integer. The maximum possible number of iterations (to avoid infinite loops)
    tolf: small float. The relative tollerance of the algorithm.
          Convergence happens if ||grad_f(x_k)||_2 < tolf ||grad_f(x_0)||_2
    tolx: small float. The tollerance in the input domain.
          Convergence happens if ||x_{k} - x_{k-1}||_2 < tolx.
          Pay attention to to the first iterate.

Output:
    x: an array that contains the value of x_k FOR EACH iterate x_k (not only the latter).
    k: an integer. The number of iteration needed to converge. k < kmax.
    f_val: an array that contains the value of f(x_k) FOR EACH iterate x_k.
    grads: an array that contains the value of grad_f(x_k) FOR EACH iterate x_k.
    err: an array the contains the value of ||grad_f(x_k)||_2 FOR EACH iterate x_k.
```

  For the moment, consider a fixed value of $\alpha > 0$.

- On my website you can find a Python implementation of the backtracking algorithm for the automatic selection of the step size. That function works as follows:

```
Input:
    f: the function f(x) we want to optimize.
       It is supposed to be a Python function, not an array.
    grad_f: the gradient of f(x). It is supposed to be a Python function, not an array.
    x: an array. The actual iterate x_k for which you want to find the correct value for alpha.

Output:
    alpha: a float. The correct step size for the next iteration.
```

Modify the code for the GD method to let it be able to use the backtracking algorithm for the choice of the step size.

- Test the algorithm above on the following functions:
  1. $f : \mathbb{R}^2 \to \mathbb{R}$ such that
  $$f(x_1, x_2) = (x_1 - 3)^2 + (x_2 - 1)^2$$
  for which the true optimum is $x^* = (3, 1)^T$.
  2. $f : \mathbb{R}^2 \to \mathbb{R}$ such that
  $$f(x_1, x_2) = 10(x_1 - 1)^2 + (x_2 - 2)^2$$
  for which the true optimum is $x^* = (1, 2)^T$.
  3. $f : \mathbb{R}^n \to \mathbb{R}$ such that
  $$f(x) = \frac{1}{2}||Ax - b||_2^2$$
  where $A \in \mathbb{R}^{n \times n}$ is the Vandermonde matrix associated with the vector $v \in \mathbb{R}^n$ that contains n equispaced values in the interval $[0, 1]$, and $b \in \mathbb{R}^n$ is computed by first setting $x_{true} = (1, 1, \dots, 1)^T$ and then $b = Ax_{true}$. Try for different values of n (e.g. $n = 5, 10, 15, \dots$).
  4. $f : \mathbb{R}^n \to \mathbb{R}$ such that
  $$f(x) = \frac{1}{2}||Ax - b||_2^2 + \frac{\lambda}{2}||x||_2^2$$
  where A and b are the same of the exercise above, while λ is a fixed value in the interval $[0, 1]$. Try different values for λ.
  5. $f : \mathbb{R} \to \mathbb{R}$ such that
  $$f(x) = x^4 + x^3 - 2x^2 - 2x$$

- For each of the functions above, run the GD method with and without the backtracking, trying different values for the step size $\alpha > 0$ when you are not using backtracking. Observe the different behavior of GD.

- To help visualization, it is convenient to plot the error vector that contains the $||\nabla f(x_k)||_2$, to check that it goes to zero. Compare the convergence speed (in terms of the number of iterations k) in the different cases.

- For each of the points above, fix `x0`$= (0, 0, \dots, 0)^T$, `kmax` $= 100$, while choose your values for $tolf$ and $tolx$. It is recommended to also plot the error $||x_k - x^*||_2$ varying k when the true x^* is available.

- Only for the non-convex function defined in 5, plot it in the interval $[-3, 3]$ and test the convergence point of GD with different values of `x0` and different step-sizes. Observe when the convergence point is the global minimum and when it stops on a local minimum or maximum.

- *Hard (optional):* For the functions 1 and 2, plot the contour around the minimum and the path defined by the iterations (following the example seen during the lesson). See `plt.contour` to do that.

Optimization via Stochastic Gradient Descent

While working with Machine Learning (ML) you are usually given a dataset $\mathbb{D} = \{X, Y\}$ with

$$X = [x^1 x^2 \dots x^N] \in \mathbb{R}^{d \times N}$$

and

$$Y = [y^1 y^2 \dots y^N] \in \mathbb{R}^N$$

and a parametric function $f_w(x)$ where the vector w is usually referred to as the *weights* of the model. The *training* procedure can be written as

$$w^* = \arg\min_w \ell(w; \mathbb{D}) = \arg\min_w \sum_{i=1}^N \ell_i(w; x^i, y^i) \tag{3}$$

what is interesting in (3) from the optimization point of view, is that the objective function $\ell(w; \mathbb{D})$ is written as a sum of independent terms that are related to datapoints (we will see in the next lab why this formulation is so common).

Suppose we want to apply GD to (3). Given an initial vector $w_0 \in \mathbb{R}^n$, the iteration become

$$w_{k+1} = w_k - \alpha_k \nabla_w \ell(w_k \mathbb{D}) = w_k - \alpha_k \sum_{i=1}^N \nabla_w \ell_i(w_k; x^i, y^i)$$

Thus, to compute the iteration we need the gradient with respect to the weights of the objective functions, that can be computed by summing up the gradients of the independent functions $\ell_i(w; x^i, y^i)$.

Unfortunately, even if it is easy to compute the gradient for each of the $\ell_i(w; x^i, y^i)$, when the number of samples N is large (which is common in Machine Learning), the computation of the full gradient $\nabla_w \ell(w_k; \mathbb{D})$ is prohibitive. For this reason, in such optimization problems, instead of using a standard GD algorithm, it is better using the Stochastic Gradient Descent (SGD) method. That is a variant of the classical GD where, instead of computing $\nabla_w \ell(w; \mathbb{D}) = \sum_{i=1}^N \nabla_w \ell_i(w; x^i, y^i)$, the summation is reduced to a limited number of terms, called a *batch*. The idea is the following:

- Given a number N_{batch} (usually called `batch_size`), randomly extract a subdataset $\mathcal{M}$ with $|\mathcal{M}| = N_{batch}$ from $\mathbb{D}$.
- Approximate the true gradient $\nabla_w \ell(w; \mathbb{D}) = \sum_{i=1}^N \nabla_w \ell_i(w; x^i, y^i)$ with $\nabla_w \ell(w; \mathcal{M}) = \sum_{i \in \mathcal{M}} \nabla_w \ell_i(w; x^i, y^i)$.
- Compute one single iteration of the GD algorithm

$$w_{k+1} = w_k - \alpha_k \nabla_w \ell(w; \mathcal{M})$$

- Repeat until you have extracted the full dataset. Notice that the random sampling at each iteration is done without replacement.

Each iteration of the algorithm above is usually called *batch iteration*. When the whole dataset has been processed, we say that we completed an *epoch* of the SGD method. This algorithm should be repeated for e fixed number E of epochs to reach convergence.

Unfortunately, one of the biggest drawbacks of SGD with respect to GD, is that now we cannot check the convergence anymore (since we can't obviously compute the gradient of $\ell(w; \mathbb{D})$ to check its distance from zero) and we can't use the backtracking algorithm, for the same reason. As a consequence, the algorithm will stop ONLY after reaching the fixed number of epochs, and we must set a good value for the step size α_k by hand. Those problems are solved by recent algorithms like SGD with Momentum, Adam, AdaGrad, ...

- Write a Python script that implement the SGD algorithm, following the structure you already wrote for GD. That script should work as follows:

```
Input:
l: the function l(w; D) we want to optimize.
It is supposed to be a Python function, not an array.
grad_l: the gradient of l(w; D). It is supposed to be a Python function, not an array.
w0: an n-dimensional array which represents the initial iterate. By default, it
      should be randomly sampled.
data: a tuple (x, y) that contains the two arrays x and y, where x is the input data,
      y is the output data.
batch_size: an integer. The dimension of each batch. Should be a divisor of the number of data.
n_epochs: an integer. The number of epochs you want to reapeat the iterations.

Output:
w: an array that contains the value of w_k FOR EACH iterate w_k (not only the latter).
f_val: an array that contains the value of l(w_k; D)
       FOR EACH iterate w_k ONLY after each epoch.
grads: an array that contains the value of grad_l(w_k; D)
       FOR EACH iterate w_k ONLY after each epoch.
err: an array the contains the value of ||grad_l(w_k; D)||_2
     FOR EACH iterate w_k ONLY after each epoch.
```

- To test the script above, consider the MNIST dataset we used in the previous laboratories, and do the following:

  1. From the dataset, select only two digits. It would be great to let the user input the two digits to select.
  2. Do the same operation of the previous homework to obtain the training and test set from (X, Y), selecting the N_{train} you prefer.
  3. Implement a logistic regression classificator as described in the corresponding post on my website.

- Test the logistic regression classificator for different digits and different training set dimensions.

- The training procedure will end up with a set of optimal parameters w^*. Compare w^* when computed with Gradient Descent and Stochastic Gradient Descent, for different digits and different training set dimensions.

- Comment the obtained results (in terms of the accuracy of the learned classificator).

- *Hard (optional):* Try to implement the 3-digits logistic regression classificator and compare its accuracy with the accuracy of LDA and PCA classificators.