6.867: Machine Learning, Homework 1

1. Implement Gradient Descent

1.1 Implementation the gradient Descent

We implemented a function `gradientDescent` that takes the parameters:

- A scalar function $f : \mathbb{R}^n \mapsto \mathbb{R}$.
- The gradient of f $\nabla f : \mathbb{R}^n \mapsto \mathbb{R}^n$.
- An initial guess $x_0 \in \mathbb{R}^n$.
- A step size $s > 0$.
- A convergence threshold $\epsilon > 0$.

`gradientDescent` returns an array of $n+1$ numbers, corresponding to the n coordinates of the current x_t, and the value of $f(x_t)$, for each of the steps t before convergence of the algorithm.

This choice of output lets us monitor the rate of convergence of the algorithm (which zones lead to slower or faster convergence), which will be useful when we compare different methods (see section 1.5). It also allows us to plot the path followed by the algorithm (see figures 1to 4). In all the figures that follow, we plot an initial guess (red dot) and a final result after gradient descent (blue dot). The blue line corresponds to the path followed by the gradient descent algorithm.

1.2 Results and Impact of the choice of parameters

We chose to benchmark our gradient descent on a variety of functions, designed to test all the potential cases that we might encounter. Because of plotting constraints, we focus the figures to function of two parameters: $f : \mathbb{R}^2 \mapsto \mathbb{R}$.

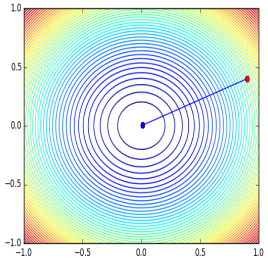

Figure 1. $f : (x, y) \to 10x^2 + 10y^2$, $x_0 = (4, 4)$, $s = 0.1$ and $\epsilon = 0.01$

In simple cases such as a quadratic bowl (figure 1), the convergence does not strongly depend on the choice of the parameters, and converges quite fast (here in 11 steps). Furthermore, there is no need to specify very small s and ϵ to ensure convergence to the global maximum.

1.2.1 Impact of the initial guess

In the case of a very non convex function such as in figure 2 however, the choice of x_0 becomes important, and we see that even a slight perturbation in the original guess can lead to a totaly different result. Another issue with highly non-convex functions is that the gradient descent may not even converge to a local minimum. For instance in figure 2, for $x_0 = (1.5, 1.5)$, the gradient at x_0 is 0, therefore the algorithm is stuck at x_0.

In some cases (See figure 3), the function does not have a minimum. This leads to the algorithm never converging, and therefore we need to stop it after a certain number of iterations. As we can see in figure 3, for some initial values of x_0, the algorithm converges to the saddle point $(0, 0)$ even without starting there.

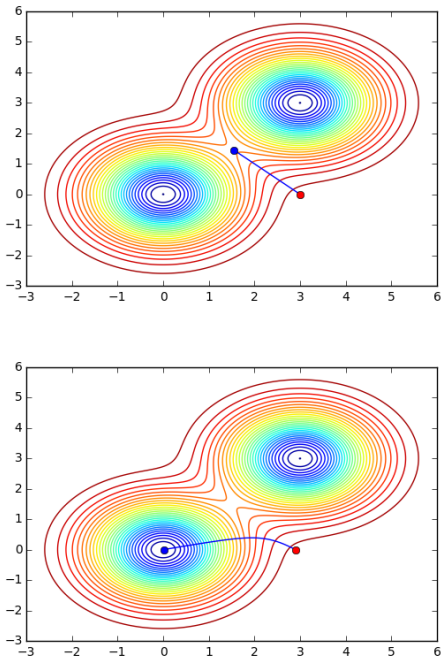

Figure 2. $f : (x, y) \to -e^{-\frac{x^2+y^2}{2}} - e^{-\frac{(x-3)^2+(y-3)^2}{2}}$,$x_0 = (4, 3.9)$ (top figure) and $x_0 = (4, 4)$(bottom figure), $s = 0.1$ and $\epsilon = 10^{-3}$.

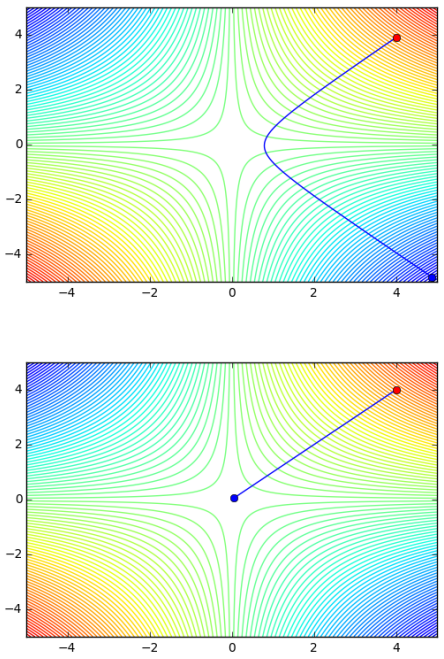

Figure 3. $f : (x, y) \to xy$, $x_0 = (3, 0)$ (top figure) and $x_0 = (2.9, 0)$(bottom figure), $s = 0.3$ and $\epsilon = 10^{-4}$.

1.2.2 Impact of the step size

In figure 4, we show the impact of the step size on the convergence of the algorithm, even in the case of a very simple function. For s small enough, the algorithm converges but when we increase s, the gradient descent starts cycling, and for even higher values of s, it diverges completely.

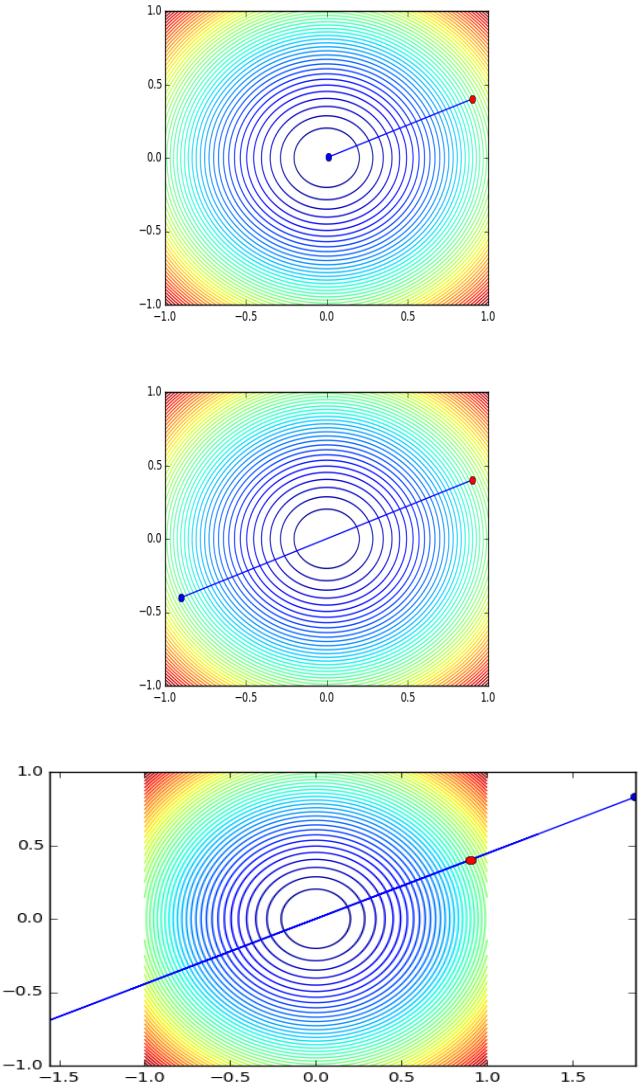

Figure 4. $f : (x, y) \to 10x^2 + 10y^2$, $x_0 = (4, 4)$, top to bottom: $s = 0.01, 0.1, 0.11$ and $\epsilon = 0.01$.

1.3 Impact of the convergence criterion

We also noticed that the computational time seems to grow rapidly when $\epsilon \to 0$. However, reasonably small values for ϵ are needed in order to ensure that the algorithm does not get stuck at a saddle point (in the interest of concision, we did not plot these results, and we refer the reader to our figures 2 and 3 for examples of saddle points).

1.4 Numerical Gradient

We implemented a function `numericalGradient` that takes the parameters:

- A scalar function $f : \mathbb{R}^n \mapsto \mathbb{R}$.
- An value $x \in \mathbb{R}^n$ at which we wish to compute the gradient
- Finite difference size $\eta > 0$ (taken as small as possible).

`gradientDescent` returns an array of n numbers, corresponding to gradient of f at x. To do that, we use the fact that $(\nabla f(x))_i = \frac{\partial f}{\partial x_i}|_x = \lim_{\eta \to 0} \frac{f(x+\eta e_i)-f(x)}{\eta}$.

Let us denote $(\widetilde{\nabla_\eta f})(x)$ the numerical gradient given by our algorithms. For $\eta = 0.5$, and $x = (1, 2)$, this gives us the following gradients:

f	$(\nabla f)(x)$	$(\widetilde{\nabla_\eta f})(x)$
$(x, y) \to x^2 + y^2$	$(2, 4)$	$(2.5, 4.5)$
$(x, y) \to xy$	$(2, 1)$	$(2.0, 1.0)$
$(x, y) \to -e^{-\frac{x^2+y^2}{2}}$	$\approx (0.082, 0.164)$	$(0.076, 0.111)$

We notice that decreasing η to $\eta' = 0.01$ increases the precision of the results. Another way to increase the precision is to take the middle of the finite differences: $\frac{\partial f}{\partial x_i}|_x \approx \frac{f(x+\eta e_i/2)-f(x-\eta e_i/2)}{\eta}$.

f	$(\nabla f)(x)$	$(\widetilde{\nabla_{\eta'} f})(x)$
$(x, y) \to x^2 + y^2$	$(2, 4)$	$(2.01, 4.01)$
$(x, y) \to xy$	$(2, 1)$	$(2.0, 1.0)$
$(x, y) \to -e^{-\frac{x^2+y^2}{2}}$	$\approx (0.082, 0.164)$	$(0.082, 0.163)$

1.5 Comparison with existing optimization methods

We use the `scipy.optimize.fmin_bfgs` method to benchmark our algorithm performance. The performance measure is the number of calls to the function. We consider three cases corresponding to figure 1, figure 2 (top case) and figure 3 (bottom case).

Case	`gradientDescent`	`fmin_bfgs`
1	57	28
2	21	3
3	41	2

We see that in most cases, the scipy algorithm performs better than our `gradientDescent`. This is likely to be due to a better choice of step sizes (compared to our fixed steps, which can be highly inefficient). However, similarly to our solver, it sometimes does not converge to the global minimum.

2. Linear Basis Function Regression

2.1 Maximum Likelihood weight vector

Our procedure for estimating the weight vector is in two steps. First a function `designMatrix` computes the design matrix

$$\Phi = \left(X[i]^j\right)_{0 \le i \le N-1, 0 \le j \le M}.$$

`designMatrix` takes as input the vector X containing N scalar observations, and the order M of the polynomial model. A second function `regressionFit` takes as inputs the matrix Φ output by `designMatrix`, as well as a vector of N scalar outputs Y. The w vector is then computed using the relation:

$$w = (\Phi^T \Phi)^{-1} \Phi^T Y$$

2.2 Sum of Squared Errors

Using the notation for all $i \in [0, N-1]$, $Y_{pred}[i] = \sum_{j=0}^{M} w_j X[i]^j$, we wrote a function `sse` that leveraged the identity:

$$SSE = \sum_{i=0}^{N-1} (Y_{pred}[i] - Y[i])^2.$$

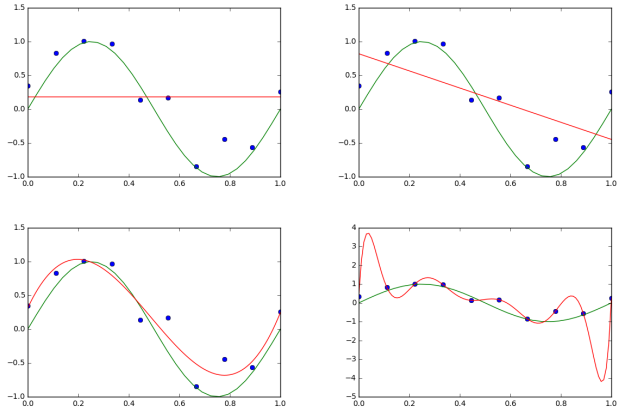

Figure 5. Figures obtained using the optimal w from `regressionFit`

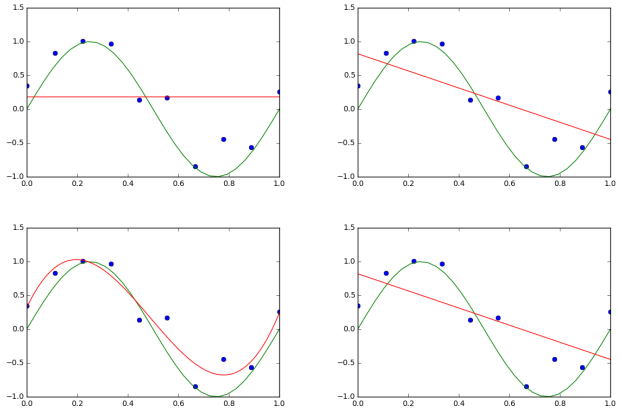

Figure 6. Figures obtained using the w computed using from `gradientDescent` to minimize the SSE. $w = 0 \in \mathbb{R}^1 0$, $s = 0.04$ and $\epsilon = 10^{-7}$.

And similarly, we wrote a function `gradsse` based on the identity:

$$\frac{\partial SSE}{\partial w_j} = -2 \sum_{i=0}^{N-1} X[i]^j (Y_{pred}[i] - Y[i]).$$

We verified the value of the gradient of the SSE function using the `numericalGradient`function obtained in section 1.4. We did our test with $M = 3$, for a variety of values for w. We used `numericalGradient` with $\eta = 0.01$. As we see in the following table, the numerical approximation of $(\nabla SSE)(w)$ and the closed form solution yield the same results with precision 0.1.

w	$(\nabla SSE)(w)$	$(\nabla_{num} SSE)(w)$
(2,2,2,2)	(81.46, 55.25, 43.89, 37.02)	(81.56, 55.29, 43.91 , 37.04)
(1,2,0,-1)	(30.72, 20.12, 15.23, 12.24)	(30.82, 20.15, 15.25, 12.25)
(5,2,-4,0)	(88.13, 42.57, 28.77, 21.77)	(88.23, 42.61, 28.79, 21.79)

2.3 Gradient Descent to optimize SSE

For $M = 0, 1, 3$ the gradient descent algorithm converges for any initial point we tried, for a step size small enough. However, for $M = 9$, and "reasonable" x_0, the algorithm does not converge towards the global minimum (as can be seen in figure 6) but rather to an interpolation that looks very similar to the case $M = 3$. The optimal case can be obtained by starting close to the solution $w \approx (0.34, 2.3{*}10^2, -5.3{*}10^3, 4.8{*}10^4, -2.3{*}10^5, 6.4{*}10^5, -1.1{*}10^6, 1.0 * 10^6, -5.6 * 10^5, 1.3 * 10^5)$.

This shows that the optimal interpolation polynomial is only one of the many local minima of the SSE function.

We next compare the number of calls to SSE needed by `gradientDescent` and `scipy.optimize.fmin_bfgs` to achieve a precision of $\epsilon = 10^{-9}$.

M	iterations `GD`	iterations `bfgs`	error `GD`	error `bgfs`
0	14	3	3.77	3.77
1	232	5	2.13	2.13
3	45906	15	0.35	0.35
9	$\geq 300,000$	205	0.34	0.079

2.4 Sin basis functions

If we use sin basis functions, the optimal function is always very close to the original sin function. However, if we didn't know the original data was generated by a sin function, this could prevent us from reaching the optimal function. For instance all linear combinations of sin functions will have value 0 at 0. Therefore if our original functions was non-zero at the origin, it can never be obtained with sin functions.

3. Ridge Regression

4. Generalizations