Trabajo 1: Programación

Patricia Córdoba Hidalgo

Índice

1. EJERCICIO SOBRE LA BÚSQUEDA ITERATIVA DE ÓPTIMOS

Este ejercicio está implementado en el fichero `ejercicio1.py`.

ACLARACIÓN: En el fichero `ejercicio1.py`, los apartados están numerados de la siguiente manera: *Apartado i = Ejercicio i-1*

1.1. Apartado 1

Este se implementó el algoritmo de *Gradiente Descendente* en los apartados siguientes, en las diferentes funciones donde era necesario programar el algoritmo. Un ejemplo es:

```
def gd(w, lr, grad_fun, fun, max_iters):
        for it in range(0, max_iters):
                w = w - lr*grad_fun(w)
        return w
```

Donde:

- w : punto inicial del que partimos para ejecutar el algoritmo
- lr : learning rate
- grad_fun: gradiente de la función de la que se quiere encontrar un mínimo
- fun: función de la que se quiere encontrar un mínimo
- max_iters: maximas iteraciones que se realizan del algoritmo

1.2. Apartado 2

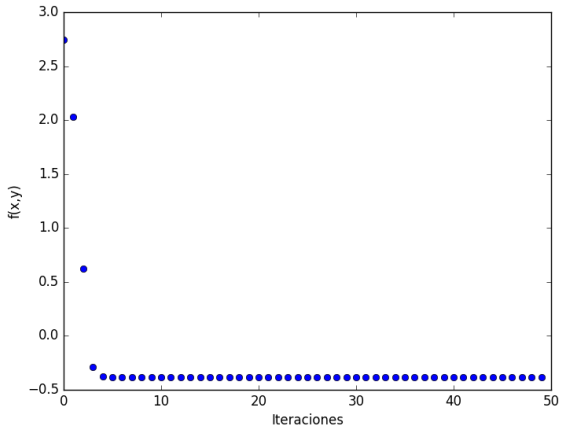
a) La expresión del gradiente de la función $E(u,v) = (ue^v - 2ve^{-u})^2$ es :

$$\nabla E(u,v)^T = (\frac{\partial E(u,v)}{\partial u}, \frac{\partial E(u,v)}{\partial v}) = (2(ue^v - 2ve^{-u})(e^v + 2ve^{-u}), 2(ue^v - 2ve^{-u})(ue^v - 2e^{-u}))$$

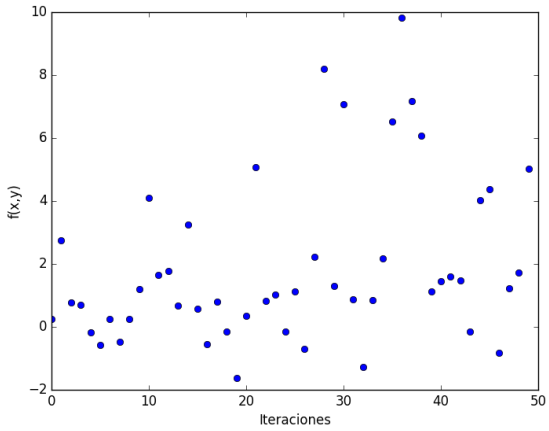
Las funciones que devuelven las derivadas parciales, devuelven estas funciones, no se usó cálculo simbólico para calcularlas.
b), c) La respuesta de estos apartados se obtienen ejecutando el código de dicho ejercicio. Son necesarias 10 iteraciones para obtener un valor de $f(w)$ menor que 10^{-14}, que se obtiene con $w = (0{,}04473629039778207 \;\; 0{,}023958714099141746)$.

1.3. Apartado 3

a) Usamos el algoritmo de *Gradiente Descendente* para minimizar la función $f(x,y) = (x-2)^2 + 2(y+2)^2 + 2sin(2\pi x)sin(2\pi y)$. Utilizando $w = (1,-1)$ como punto inicial y realizando 50 iteraciones, usamos el algoritmo con los siguientes valores de η:



(a) $\eta = 0.01$



(b) $\eta = 0.1$

Como podemos ver, con $\eta = 0{,}01$ el algoritmo rápidamente queda atrapado en un mínimo local, y una vez alcanzado el valor $f(w) \approx -0{,}3812494974381$, no podemos minimizar más la función.

Con $\eta = 0{,}1$, al ser un valor demasiado grande, los valores de f no siguen una dirección concreta, sino que van de un lado para otro. Esto origina que el último valor proporcionado por el algoritmo no nos dé el mínimo de la función, ya que da $f(w) \approx 5{,}030366822273669$, que es incluso mayor que la imagen del punto inicial. Sin embargo, con este valor de η no se ha quedado estancado en un mínimo local y se ha podido conseguir en medio un valor de $f(w) \approx -1{,}6170158813825848$, que es menor que el obtenido con $\eta = 0{,}01$.

Esto nos demuestra la gran dependencia del algoritmo de η, ya que partiendo del mismo punto inicial, al cambiar este valor los resultados son muy dispares. Lo ideal sería encontrar un valor de η tal que el algoritmo no quedase estancado en ese mínimo local, sino que alcanzara otro con una imagen menor y, una vez alcanzado este, poder seguir minimizando la función por ese camino, y no "saltar" otros valores mayores.
Desde mi punto de vista, es mejor usar $\eta = 0{,}01$, ya que aunque queda atascado en un mínimo local, devuelve un valor de f menor que el del punto inicial, ya que se mueve poco a poco hacia un punto con menor imagen.
b)
Usando $\eta = 0{,}01$, y con 50 iteraciones, se han obtenido los siguientes resultados al ejecutar el algoritmo utilizando diferentes puntos iniciales:

(x_0, y_0)	**w (mínimo)**	**f(w)**
$(2{,}1, -2{,}1)$	$(2{,}2438049693647883, \;\; -2{,}237925821486178)$	$-1{,}8200785415471563$
$(3{,}0, -3{,}0)$	$(2{,}7309356482481055, \;\; -2{,}7132791261667037)$	$-0{,}38124949743809955$
$(1{,}5, 1{,}5)$	$(1{,}7779244744891156, \;\; 1{,}032056872669696)$	$18{,}042078009957635$
$(1{,}0, -1{,}0)$	$(1{,}269064351751895, \;\; -1{,}2867208738332965)$	$-0{,}3812494974381$

Como podemos ver, con el mismo valor de η, obtenemos valores disitinto eligiendo puntos iniciales diferentes. Esto refleja la importancia del punto inicial en la ejecución del algoritmo, ya que en todos los casos se converge a un mínimo local diferente, y encontrar el mínimo entre ellos depende de escoger correctamente w_0.

1.4. Apartado 4

La dificultad de encontrar el mínimo reside en elegir tanto un buen punto inicial como un valor de η adecuado, ya que dependiendo de éstos el algoritmo puede devolverte hasta puntos que al evaluarlos se obtengan valores de la función más elevados que el valor en el punto inicial, no cumpliendo con el objetivo para el que fue diseñado. Otro problema que ocasiona no elegir estos parámetros correctamente es quedar atrapado en un mínimo local.

2. EJERCICIO SOBRE REGRESIÓN LINEAL

Este ejercicio está implementado en el fichero `ejercicio2.py`.

2.1. Apartado 1

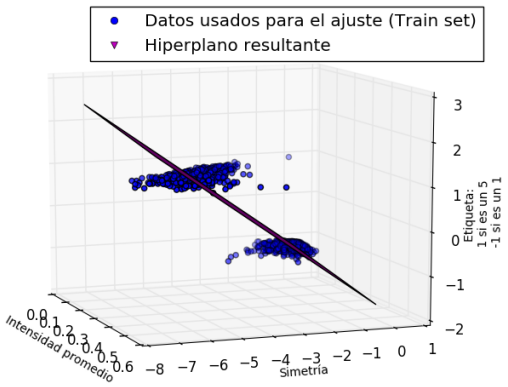
Aunque los datos que nos proporcionan se usaron en un ejemplo de un problema de clasificación, el problema que se nos plantea es de regresión. Nuestro objetivo no es separar los datos por un hiperplano, es calcular la función que mejor represente a la función desconocida que asigna las etiquetas. Los resultados obtenidos pueden luego ser interpretados para ajustarse a un problema de clasificación.

El modelo de regresión lineal usado es el siguiente:
Los datos son de la forma $x_i = (1, x1, x2)$, donde $x1$ es la intensidad promedio y $x2$ es la simetría, y sus etiquetas correspondientes son $y_i \in \{1, -1\}$. Usando el algoritmo de *Gradiente Descendente Estocástico* y *Matriz Pseudoinversa* se obtiene w, que son los pesos de la función h que aproxima a la función f desconocida que a cada x le asigna su etiqueta correspondiente. Tenemos pues que $h(x_i) = w_0 + w_1 x1 + w_2 x2$, con $w = (w_0, w_1, w_2)$.

Se implementaron los algoritmos de *Gradiente Descendente Estocástico* y *Matriz Pseudoinversa* siguiendo las pautas de teoría. Para el algoritmo de gradiente descendente se usaton los siguientes parámetros:

- $w_0 = (0, 0, 0)$, pues así lo indica el pseudocódigo visto en teoría.
- $\eta = 0{,}01$, porque es que menor error consigue empiricamente entre los que probé.
- Tamaño del minibatch = 64 elementos, pues en teoría se aconseja un valor entre 32 y 128.
- Iteraciones: 100

La función de pérdida usada es el Error Cuadrático Medio, es decir, el error es la media de los errores al cuadrado.
Las solución se pintó usando la librería `matplotlib` y para pintar la leyenda me documenté en la dirección:
`https://stackoverflow.com/questions/20505105/add-a-legend-in-a-3d-scatterplot-with-scatter-in-matplotlib`



(c) Ajuste con vector de características $(1, x_1, x_2)$

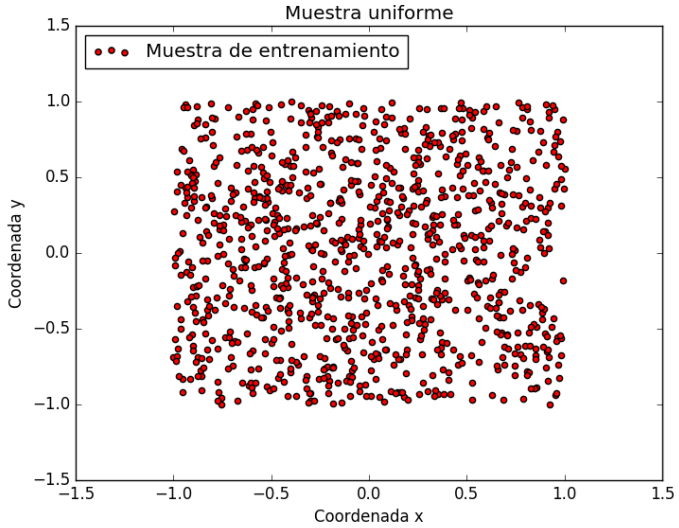
Los valores obtenidos (aproximados, ya que dependen del minibatch tomado aleatoriamente) de los errores son:

	SGD	**Matriz Pseudoinversa**
E_{in}	0,0858278567155219	0,07918658628900388
E_{out}	0,13503010648774788	0,13095383720052575

Podemos observar que el error es similar en ambos métodos. El error E_{out} es aproximadamente un 57 % mayor que el error E_{in}, luego el ajuste tampoco es muy bueno, depende mucho de los datos usados para éste.

2.2. Apartado 2

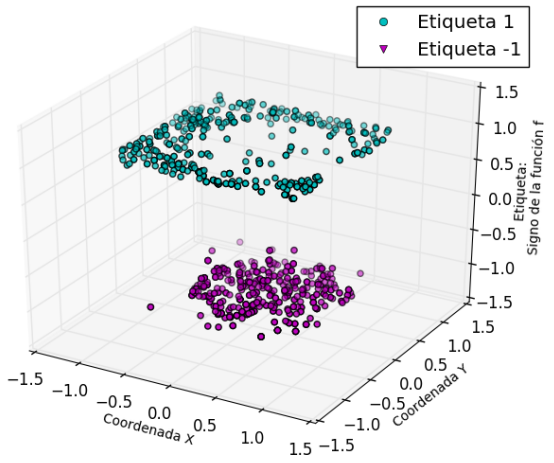
a) Con la función de `np.random.uniform()`, se extrajo una muestra uniforme del cuadrado $[-1, 1] \times [-1, 1]$ y se representaron los datos con la función `muestra_datos`.



(d) Muestra estraída

b) Definimos una función que a cada dato le asigne su etiqueta, `f`, y usando la función `asigno_etiqueta` a cada dato de la muestra le asigno su etiqueta correspondiente, generando ruido en el 10 % de los datos. Pintamos

las datos con una función con un esquema muy similar a la usada en el Apartado 1, `dibuja_gráfica`.

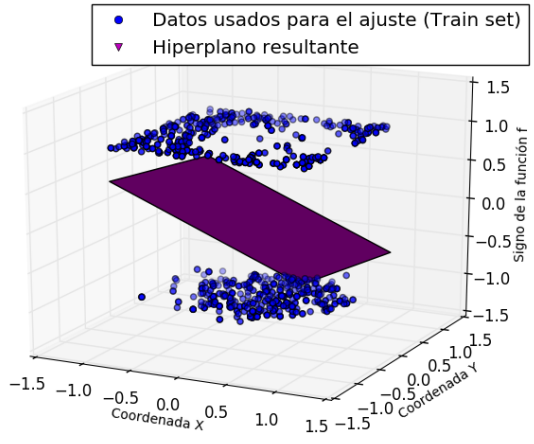


(e) Muestra con sus etiquetas

c) Como en el Apartado 1, buscamos la función que mejor aproxime a aquella que asigna las etiquetas. La buscamos entre funciones de la forma $h(x) = w_0 + w_1 x_1 + w_2 x_2 = w^T x$, donde $w = (w_0, w_1, w_2)$ y $x = (1, x_1, x_2)$.

Se usó la función que implementaba el algoritmo de *Gradiente Descendente Estocástico* creada en el Apartado 1. Se implementó la función `formato_datos` para expresar la muestra en el formato usado en el Apartado 1. Los pesos de w se imprimen por pantalla, y, como se escoge un minibatch de forma aleatoria, varían de una ejecución a otra. Uno de los pesos resultantes en una ejecución del ejercicio fue $w = (-0{,}02043103 \quad -0{,}50368271 \quad -0{,}10994177)$. El error es $E_{in} \approx 0{,}9341114116736479$ (suele estar entre 1 y 0,9)

Para facilitar la comprensión del ejercicio, se representó el resultado en una gráfica, aunque el enunciado no lo pidiese explícitamente. Esto se hizo con la función `grafica_plano`. (De hizo una nueva porque, aunque el funcionamiento es el mismo que la del Apartado 1, el nombre de los ejes cambia).



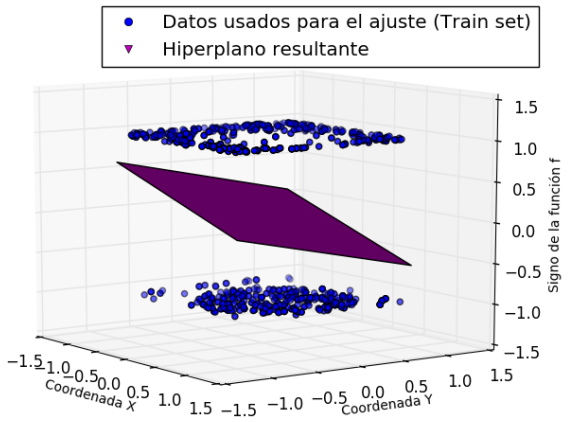
(f) Ajuste con vector de características $(1, x_1, x_2)$

d) Se creó la función `experimento` donde se repiten los pasos de los apartados **a) - c)** en un bucle. Además, esta función genera datos de prueba para poder estimar la bondad del ajuste calculando el error E_{out} acumulado en todas las iteraciones.

Esta función podría solo devolver E_{in} y E_{out} para resolver el ejercicio, pero para comprobar que se hace corretamente la regresión, decidí representar lo resultados obtenidos en la última iteración del experimento. Es por esto que la función también devuelve:

- x: Muestra usada para el ajuste en la última iteración.
- im_datos: Etiquetas de la muestra.
- w: Pesos de la función lineal que estima la f.

Se representó el resultado usando la función `grafica_plano`.



(g) Ajuste

Los errores promedio son:
$E_{in} \approx 0{,}9208064558549589$
$E_{out} \approx 0{,}9487533042593158$

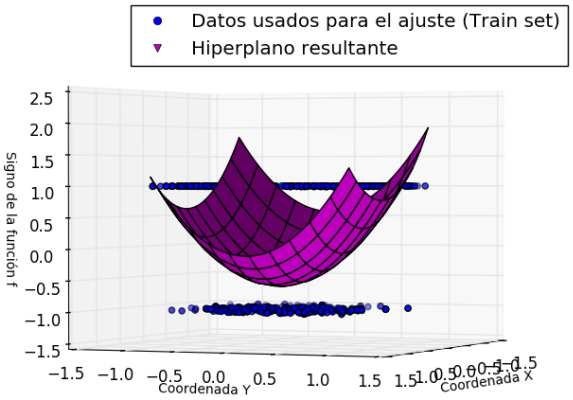
e) El ajuste con este modelo lineal no es el ideal. Los errores medios son altos, ya que, como podemos ver graficamente, no se ajusta nada bien a los datos. Al saber nosotros que la función que asigna etiquetas corresponde gráficamente a pertenecer o no a cierta circunferencia, lo ideal para ajustarlo es un paraboloide, no un hiperplano.

Repetición del mismo experimento usando características no lineales

Creamos la función `experimento_2` equivalente a la función del apartado **d)**, que englobaba todos los apartados anteriores. Ahora el vector de datos es $(1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$, y para representarlo se implementó la función `ajuste_datos`, que dado un vector de listas de dos componentes, $[x_1, x_2]$, te crea otro con vectores de seis componentes: $[1, x_1, x_2, x_1x_2, x_1^2, x_2^2]$. La función que buscamos con este modelo de regresión es de la forma $h(w) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2$.

En la función `experimento_2` se llama a la función `ajuste_datos` en vez de a la función `formato_datos`, el resto es análogo a la función `experimento`.

Al igual que en el apartado **d)**, se representaron los resultados obtenidos en la última iteración del experimento para la comprobación y mejor entendimiento del resultado usando la función `grafica_parabola`. Como ahora no es un plano, sino una parábola lo que representamos, la cuadrícula necesaria para representar la superficie necesita muchas más divisiones (para ver mejor la variabilidad de los valores de la función sobre los puntos de la cuadrícula), por eso ahora se usó la función `meshgrid`, por lo demás, es equivalente a la función `grafica_plano`.



(h) Ajuste

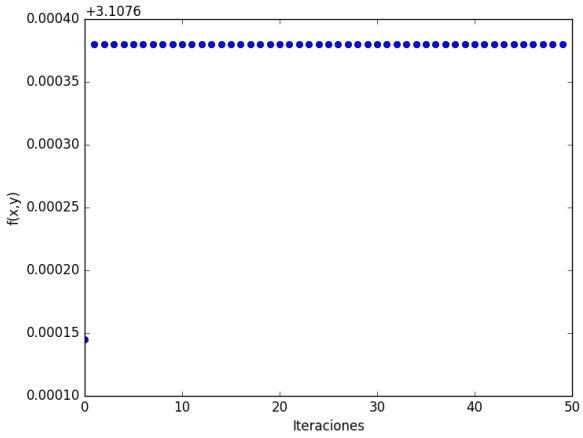
El valor de w que se usó en esta representación es $w = [-0{,}58334763 \quad -0{,}34050249 \quad 0{,}07684784 \quad -0{,}03595671 \quad 0,85958344 \quad 1,32624242]$.

Los errores promedio son:
$E_{in} \approx 0{,}5927647770981717$
$E_{out} \approx 0{,}6058953967302595$

Son menores que en la regresión usando características lineales, por lo cual este ajuste es mucho más adecuado. Como ya se comentó en el apartado **e)**, pintando los datos (representando la nube de puntos) se ve claramente que un paraboliode representa mucho mejor la asignación de etiquetas.

3. BONUS

Se implementó en el fichero `ejercicio3.py` el algoritmo del método de Newton y se le aplicó a la función $f(x, y) = (x - 2)^2 + 2(y + 2)^2 + 2sin(2\pi x)sin(2\pi y)$. Las derivadas parciales de esta función, necesarias para definir la matriz Hessiana, se calcularon a mano y se implementaron funciones que las devuelven.
La gráfica con los valores que toma la función en las 50 iteraciones es:

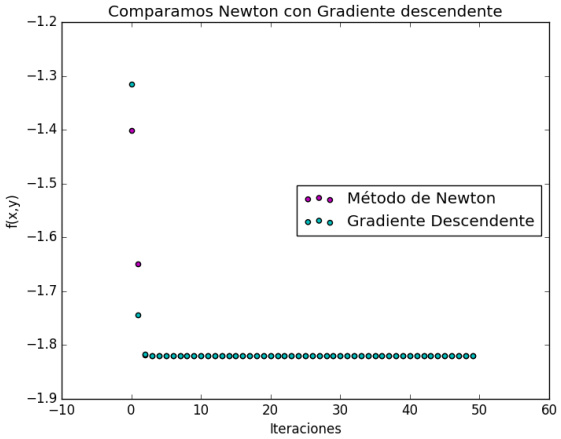


(i) $(1, -1)$

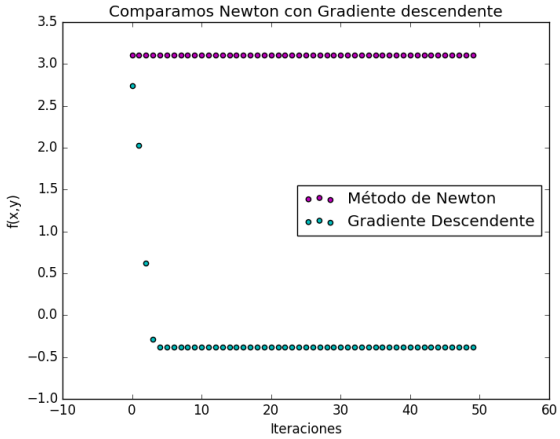
Los resultados partiendo de los puntos del Apartado 3 son:

(x_0, y_0)	**w (mínimo)**	**f(w)**
(2,1, −2,1)	(1,7561950306352119, −1,7620741785138223)	−1,8200785415471565
(3,0, −3,0)	(3,05397555493864, −3,028461459660191)	3,1079800610352026
(1,5, 1,5)	(1,7048309830043302, 0,9731715834776464)	18,08874203270784
(1,0, −1,0)	(0,9460244450613605, −0,9715385403398094)	3,1079800610352013

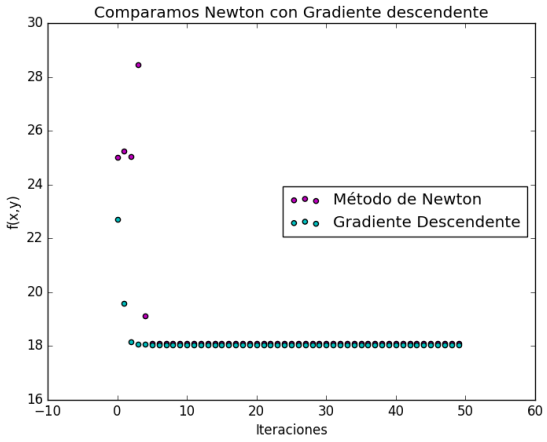
En los siguientes gráficos comparamos el desempeño de los algoritmos *Gradiente Descendente* y *Método de Newton* en esos cuatro puntos:



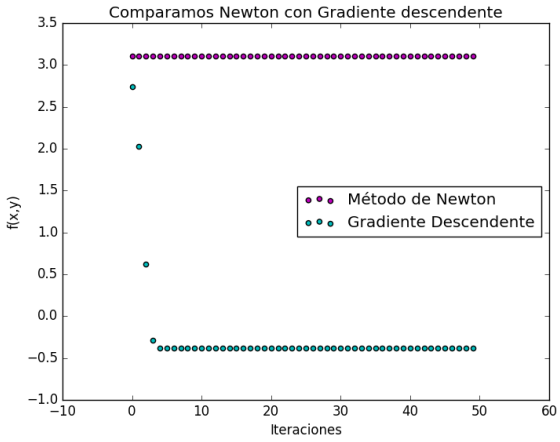
(j) $(2{,}1, -2{,}1)$



(k) $(3{,}0, -3{,}0)$



(l) $(1{,}5, -1{,}5)$



(m) $(1{,}0, -1{,}0)$

El *Método de Newton* debe usarse cuando sepamos que estamos cerca de un mínimo, ya que su desarrollo se basa en aproximar la función por una parábola (ya que cerca de un mínimo adopta una forma parecida a una) y avanzar hacia un punto donde la deribada se anule. El problema entonces lo encontramos en dos diferentes motivos:
El primero es que no hay garantías de que empecemos cerca de un mínimo, por lo que aproximar la función por una parábola no es del todo correcto. Si la función es una recta, por ejemplo, esa aproximación no tendría sentido. El segundo problema es que nos aproximamos a un cero de la derivada, luego ese punto será un punto crítico, no sabemos si será un mínimo o no, puede ser también un máximo o un punto de silla.

Por eso, aunque este método nos ahorre el estimar un *learning rate* adecuado, necesita de conocimientos previos que a priori no tenemos, ya que sólo es razonable usarlo cuando tenemos la certeza de que estamos cerca de un mínimo. En nuestro caso, como podemos ver en las gráficas, los resultados obtenidos con el *Método de Newton* son iguales o peores que usando el *Gradiente Descendente*, ya que lo que buscábamos eran mínimos y el *Método de Newton* a veces tiende a máximos o puntos de silla de la función.