



UNIVERSIDAD
DE GRANADA



Práctica 1: Búsqueda Iterativa de Óptimos y Regresión Lineal

Aprendizaje Automático

Marzo, 2022

Autor:

Lugli, Valentino Glauco · YB0819879

Índice

0 Notas sobre la implementación	2
1 Búsqueda Iterativa de Óptimos	3
1.1 Algoritmo de Gradiente Descendente	3
1.2 Experimentos con la función $E(u, v)$	3
1.2.1 Cálculo del gradiente de $E(u, v)$	3
1.2.2 Utilizando Gradiente Descendente para Minimizar la Función	4
1.3 Experimentos con la función $f(x, y)$	6
1.3.1 Cálculo del Gradiente de $f(x, y)$	6
1.3.2 Variando la Tasa de Aprendizaje al Minimizar la Función	6
1.3.3 Variando el Punto Inicial	8
1.4 Conclusiones sobre Gradiente Descendente respecto a la dificultad de encontrar mínimos globales	11
2 Regresión Lineal	11
2.1 Experimento con Datos de Dígitos Manuscritos	11
2.1.1 Algoritmo de la Pseudoinversa o <i>Normal Equation</i>	12
2.1.2 Gradiente Descendente Estocástica con Minibatches	12
2.1.3 Ajustando los datos con Pseudoinversa y Gradiente Descendente Estocástica	13
2.2 Experimentos con Características no Lineales	17
2.2.1 Visualización de los datos y ajuste a los mismos	17
2.2.2 Múltiples ejecuciones del experimento	19
3 Bonus: Método de Newton	21
3.1 Implementación del algoritmo y cálculo de Hessiana	21
3.2 Variando la Tasa de Aprendizaje al Minimizar la Función	21
3.3 Variando el Punto Inicial	22
Referencias	26

0. Notas sobre la implementación

- La práctica fue realizada completamente en Python haciendo uso del IDE “Spyder”, este IDE permite tener “celdas” de código que se pueden ejecutar independientemente del código que tienen antes o después, es decir, es como una celda de código de un Colab Notebook: no es necesario ejecutar todo el programa entero, se puede ir paso a paso.
- Al igual que un Colab Notebook, se han de ejecutar las celdas en orden la primera vez que se carga el fichero para tener las funciones y variables de celdas anteriores en memoria. Una vez realizado esto se pueden ejecutar las celdas en cualquier orden.
- Las celdas se delimitan con un comentario de la forma “`# %%`” y pueden ejecutarse haciendo `Ctrl+Enter` en una celda resaltada o bien haciendo clic en el ícono en la Figura 1 que está a la derecha del ícono de “Play” que ejecuta el código entero secuencialmente.
- Las celdas están organizadas de manera que la primera celda abarca todas las funciones implementadas, y luego existe una celda por cada ejercicio, de esta manera solamente es necesario ejecutar la celda inicial, denominada la celda 0 y la celda del ejercicio que se desee ejecutar, el cual se encuentra apropiadamente identificada en el código.
- También se muestra donde comienzan y terminan las funciones de cada ejercicio para facilitar la legibilidad de la práctica.



Figura 1: Ícono para ejecutar la celda seleccionada

1. Búsqueda Iterativa de Óptimos

1.1. Algoritmo de Gradiente Descendente

Para la realización de los experimentos se ha desarrollado la función denominada `gradient_descent` (`w_ini`, `lr`, `grad_fun`, `fun`, `epsilon`, `max_iters`) que toma como parámetros los pesos iniciales, la tasa de aprendizaje, el gradiente de la función a minimizar, la función a minimizar, el error que se desea obtener y las iteraciones máximas para la minimización e internamente implementa el algoritmo de Gradiente Descendente.

Este algoritmo tiene como base la expresión 1, que toma una función diferenciable \mathcal{F} y valores w_i contenidos dentro del vector W y los minimiza por medio de la resta de dichos valores con la gradiente de \mathcal{F} .

Matemáticamente, el gradiente de una función es un vector que indica la dirección donde ocurre el cambio más fuerte de sus valores, por lo tanto, apunta a regiones que aumentan los valores de la función, entonces, si se desea es minimizar los valores, debe de restarse: ahora se apunta adonde ocurre el cambio más fuerte y también donde se reducen los valores de la función.

Este proceso se repite las veces necesarias para que los valores contenidos en W obtengan un valor mínimo en \mathcal{F} .

El valor η es la tasa de aprendizaje, controla esencialmente el valor que se resta a W y por lo tanto controla que tanto cambia en cada iteración, que tanto se avanza.

$$W' = W - \eta \times \frac{\partial \mathcal{F}(W)}{\partial W} \quad (1)$$

A nivel de código, la expresión 1 se puede transformar fácilmente a un equivalente, donde \mathcal{F} es `fun`, $\frac{\partial \mathcal{F}(W)}{\partial W}$ es `grad_fun`, η es `lr` y W es un vector de Numpy `w`, con valores iniciales `w_ini`.

Esta expresión se encuentra dentro de un bucle, donde en cada iteración se realiza la actualización de los valores W hasta que se ha llegado a un valor mínimo o se ha excedido el número de iteraciones del mismo.

El algoritmo, una vez finaliza ya sea de cualquier manera; devuelve los pesos finales `w` obtenidos en la última iteración, junto el valor de esa iteración y una traza de los valores `w` anteriores y la evaluación de dichos valores en la función `fun`.

Se realizó de esta manera para poder pasar con comodidad diferentes funciones a minimizar.

1.2. Experimentos con la función $E(u, v)$

1.2.1. Cálculo del gradiente de $E(u, v)$

Se tiene la función 2 a minimizar por medio de gradiente descendente.

$$E(u, v) = u \cdot v \cdot e^{(-u^2-v^2)^2} \quad (2)$$

Para realizar esto, se debe de obtener la gradiente; esto no es más un vector que contiene las derivadas parciales de la función, es decir $\frac{\partial E}{\partial u}$ y $\frac{\partial E}{\partial v}$ en el caso actual.

Esto puede realizarse analíticamente con la regla de la cadena; las derivadas pueden observarse en 3 y 4 respectivamente.

$$\frac{\partial E}{\partial u} = -2 \cdot u \cdot (2 \cdot u^2 - 1) \cdot v^2 \cdot e^{-2 \cdot (u^2 + v^2)} \quad (3)$$

$$\frac{\partial E}{\partial v} = -2 \cdot u^2 \cdot v \cdot (2 \cdot v^2 - 1) \cdot e^{-2 \cdot (u^2 + v^2)} \quad (4)$$

Por lo tanto, $\nabla E(u, v)$ tiene la forma de la expresión 5.

$$\nabla E(u, v) = \begin{bmatrix} -2 \cdot u \cdot (2 \cdot u^2 - 1) \cdot v^2 \cdot e^{-2 \cdot (u^2 + v^2)} \\ -2 \cdot u^2 \cdot v \cdot (2 \cdot v^2 - 1) \cdot e^{-2 \cdot (u^2 + v^2)} \end{bmatrix} \quad (5)$$

Esto se encuentra representado en código en las funciones `E(u, v)` para la función original, `dEu(u, v)`, `dEv(u, v)` para las derivadas parciales y `gradE(u, v)` contiene el gradiente, esta última internamente representa dicho gradiente como un vector de Numpy equivalente, que devuelve los valores de las derivadas parciales dado un `u` y `v` de entrada.

1.2.2. Utilizando Gradiente Descendente para Minimizar la Función

Considerando el punto $(u, v) = (0,5; -0,5)$ se realizaron experimentos para obtener un mínimo en la función $E(u, v)$.

Esto se realizó por medio una ejecución de la función `gradient_descent()` previamente definida, a la que se le pasan por parámetro: el valor inicial de los pesos, en este caso $(0,5, 0,5)$ como `w_ini`, la tasa de aprendizaje $\eta = 0,1$ como `lr`, por defecto se tienen 1×10^{10} iteraciones máximas y se desea obtener unas coordenadas con un valor inferior o igual a 10^{-8} , que sería el parámetro `epsilon` de la función.

Una vez finalizada la ejecución se obtuvieron las coordenadas que se presentan en 6 en 25117 iteraciones.

$$(0,010000842574554563, -0,010000842574554563) \quad (6)$$

Se puede comprobar que, efectivamente, estos valores evaluados en $E(u, v)$ dan como resultado el valor $9,9994 \times 10^{-9}$, por lo tanto se puede ver que se ha llegado a un valor mínimo del `epsilon` pedido, y también un mínimo de dicha función por medio del algoritmo de Gradiente Descendente.

De manera visual, es posible confirmar lo dicho anteriormente en la Figura 2, se observa como los pesos calculados van yendo directamente al mínimo más cercano de la función $E(u, v)$.

Esto confirma que, en efecto, restar el gradiente hace que el vector apunte adonde se reducen los valores de la función de la manera más fuerte: el punto inicial se encuentra casi al tope de una “montaña” de la función, y toma el camino más directo, que en este caso es ir recto hacia “abajo”, para llegar al “valle” de la función.

Evolución de los valores w en $E(u, v)$

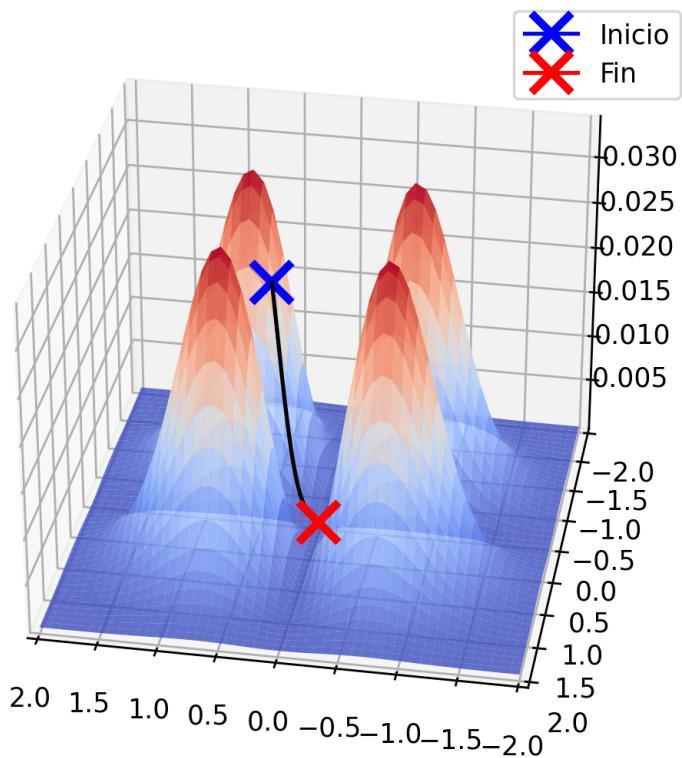


Figura 2: Visualización de la evolución de los pesos calculados por $\nabla E(u, v)$ en la función $E(u, v)$.

1.3. Experimentos con la función $f(x, y)$

1.3.1. Cálculo del Gradiente de $f(x, y)$

Dada la nueva función 7 a minimizar por Gradiente Descendente.

$$f(x, y) = x^2 + 2y^2 + 2\sin(2\pi x)\sin(\pi y) \quad (7)$$

Se obtiene su gradiente, nuevamente tomando las derivadas parciales haciendo uso de la regla de la cadena y juntándolas en un vector como puede verse en la expresión 8.

$$\nabla f(x, y) = \begin{bmatrix} 2 \cdot (2 \cdot \pi \cdot \cos(2\pi x) \sin(\pi y) + x) \\ 2 \cdot \pi \sin(2\pi x) \cos(\pi y) + 4 \cdot y \end{bmatrix} \quad (8)$$

Al igual que la función $E(u, v)$, la función $f(x, y)$ se encuentra codificada dentro del subprograma `F(x, y)`, sus derivadas parciales en `dFx(x, y)`, `dFy(x, y)` respectivamente y el gradiente en sí en `gradF(x, y)`.

1.3.2. Variando la Tasa de Aprendizaje al Minimizar la Función

Se realizó un experimento que consistió en, dado el punto de inicio $(1, -1)$, minimizar la función $f(x, y)$ en un máximo de 50 iteraciones y variando la tasa de aprendizaje con los valores $\eta = 0,01$ y $\eta = 0,1$.

Una vez definidas las variables pertinentes, se realizan dos llamadas a la función `gradient_descent()`, ambas poseen los mismos parámetros salvo la tasa de aprendizaje.

Una vez finalizada la ejecución, se obtuvieron los resultados que se encuentran en el Cuadro 1, lo primero a observar es que, si bien todos los parámetros son idénticos exceptuando la tasa de aprendizaje, se obtuvieron dos resultados significativamente diferentes: con $\eta = 0,01$ se llegó a un valor mucho menor que con $\eta = 0,1$.

η	x	y	$f(x, y)$
0.01	-1.2178	0.4134	-0.0623
0.1	-1.147	-0.273	2.6720

Cuadro 1: Resultado de minimización por GD de $f(x, y)$ con $\eta = 0,01$ y $\eta = 0,1$

Para obtener más información de la evolución de la minimización, se graficó como se minimiza la función a lo largo de las 50 iteraciones, esto puede observarse en la Figura 3.

En primer lugar, se puede observar como con $\eta = 0,01$, la función siempre decremente su valor, y en efecto converge en alrededor de las 10 iteraciones, quedándose en ese valor hasta el final, mientras que con $\eta = 0,1$, el valor de la función oscila, llegando a valores más altos y más bajos que su contra parte, si bien también tiene una tendencia decreciente comenzando por la iteración 15, no termina de converger en ningún punto, el gradiente hace que los valores estén “dando vueltas” por la función, y justo cuando terminan las 50 iteraciones resulta que se ha posado en un valor mayor de los que ha encontrado antes y también mayor que lo obtenido con la otra tasa de aprendizaje.

Evolución de $f(x, y)$ por iteración dependiendo de η

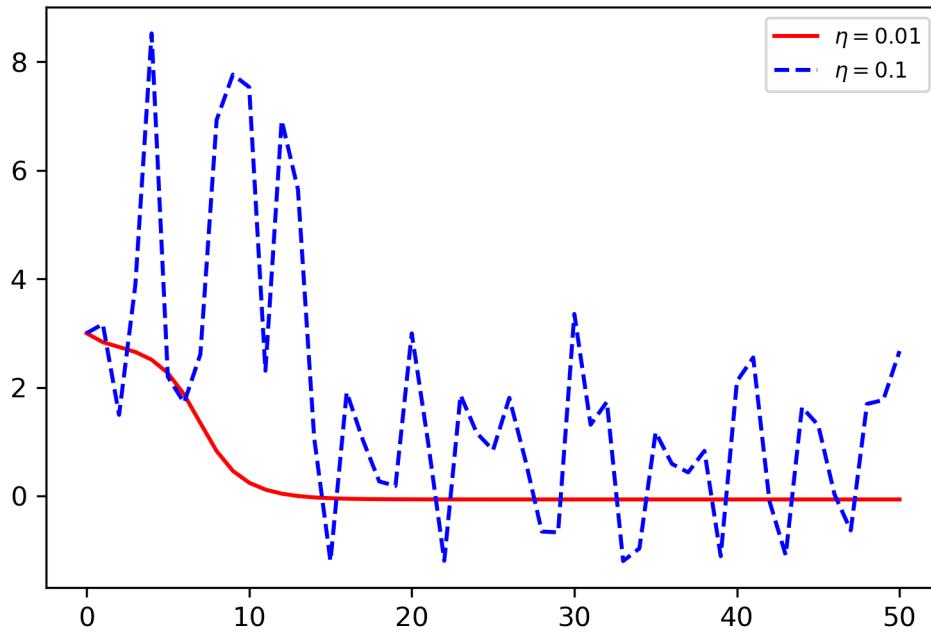


Figura 3: Comparación entre $\eta = 0,01$ y $\eta = 0,1$ y como desciende el valor de la función $f(x, y)$ por cada iteración.

Esto por un lado, confirma lo mencionado anteriormente y lo que se sabe teóricamente sobre la utilidad de la tasa de aprendizaje η : este valor indica que tan rápido se desea que se realice la minimización de los valores, puede verse como la “velocidad” o la rapidez con la que se minimiza la función y por lo tanto es lógico que en la expresión – vista anteriormente en 1 – esto multiplique el valor del gradiente, permite afinar que tanto se avanza por la función.

Dicho esto, no sigue quedando evidente *porqué* fluctúan de tal manera los valores con $\eta = 0,1$ y no con $\eta = 0,01$: de forma teórica se puede suponer que, dependiendo de la topología de la función a minimizar, al realizar el gradiente con una tasa de aprendizaje en particular, la dirección donde apunta el gradiente efectivamente es donde hay valores mínimos, pero estos se encuentran una distancia menor de la que puede obtenerse con la tasa de aprendizaje fijada, esto hace los valores no lleguen a ese mínimo, sino que pasan por encima del mismo y se calculan en otra región de la función que se encuentre en la dirección que apunta el gradiente, también puede suceder que si lleguen a un mínimo pero, si se itera nuevamente, el nuevo cálculo puede hacer que los puntos vuelvan a salirse del mismo.

Esta suposición puede confirmarse con la Figura 4. Puede observarse como con $\eta = 0,01$, el camino que traza el gradiente va directamente al mínimo más cercano, como los saltos son pequeños, no ocurre que los puntos “pasen por encima” el mínimo y una vez que llega al mínimo como los pasos son pequeños, no sucede que se salga de dicho mínimo, por otro lado, con $\eta = 0,1$, ocurre lo contrario y lo que se supuso en el párrafo anterior: los pasos que se dan son muy grandes para la función $f(x, y)$ y por lo tanto, los valores siempre pasan por encima del mínimo sin llegar a converger en él, y en el caso de que lleguen a un mínimo, al recalcular la siguiente iteración, los valores se salen nuevamente hacia otra región de la función; de esta manera también se explica

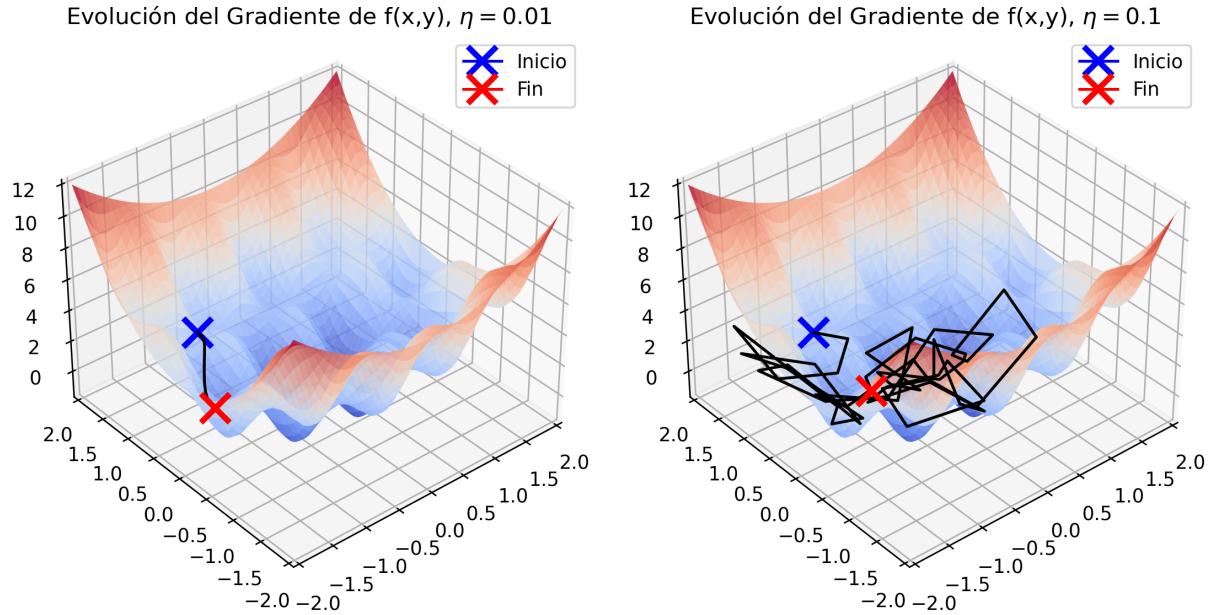


Figura 4: Comparación visual entre $\eta = 0,01$ y $\eta = 0,1$ y como afecta el avance del $\nabla f(x,y)$.

porqué en la gráfica 3 el valor de la función era tan errático con $\eta = 0,1$.

Por otro lado, el problema opuesto también hay que considerarlo: Si se elige un valor muy pequeño de η , puede que entonces, al dar pasos tan pequeños, los valores se queden atrapados en mínimos locales subóptimos y no se logren obtener mejores mínimos para las funciones, o bien que se tarde mucho en converger a un mínimo.

Entonces, este valor debe de ser ajustado a la función con la que se esté trabajando, pues dependiendo de él puede que, si se está “avanzando” muy lentamente, el algoritmo quede atrapado en un mínimo subóptimo o tarde muchas iteraciones en converger; también puede suceder lo mismo con una tasa muy alta como se ha podido comprobar, pues si es una función muy ruidosa, el gradiente calculará el siguiente paso pasando por encima de esos mínimos.

1.3.3. Variando el Punto Inicial

Se realizaron experimentos variando el punto inicial de minimización de la función $f(x,y)$, haciendo uso de los parámetros del experimento anterior, es decir, utilizando un η de 0.01 y 0.1 y haciendo uso de 50 iteraciones máximas.

Los puntos a evaluar son: $(-0,5; -0,5)$, $(1; 1)$, $(2,1; -2,1)$, $(-3; 3)$ y $(-2; 2)$.

Para realizar este experimento, al igual que el anterior se realizaron distintas ejecuciones de la función `gradient_descent()` variando en un bucle anidado, los puntos de inicio y la tasa de aprendizaje.

Una vez finalizadas las ejecuciones, se pueden observar los valores mínimos obtenidos en $f(x,y)$ junto con los valores de x y de y en el Cuadro 2.

De los valores obtenidos, nuevamente se confirma que la tasa de aprendizaje tiene un efecto profundo en los valores finales, y ahora también se puede añadir otro parámetro que se ha notado es

x_{INI}, y_{INI}	η	x_{FIN}, y_{FIN}	$f(x_{FIN}, y_{FIN})$
-0.5, -0.5	0.01	-0.7306, -0.4144	-1.0366
	0.1	-1.5152, -0.2142	2.2685
1, -1	0.01	0.7308, 0.4144	-1.0366
	0.1	-1.4236, -0.2664	2.8544
2.1, -2.1	0.01	1.6651, -1.1728	4.6338
	0.1	0.5090, 0.6340	0.9598
-3, 3	0.01	-2.1888, 0.5868	3.6942
	0.1	1.1960, -0.2105	0.3607
-2, 2	0.01	-1.6643, 1.1713	4.6337
	0.1	0.5547, 0.3094	-0.0578

Cuadro 2: Comparativa de los puntos de inicio, haciendo uso de las dos tasas de aprendizaje, los valores de salida y su valor en la función

también muy importante para el algoritmo: El punto inicial de la minimización.

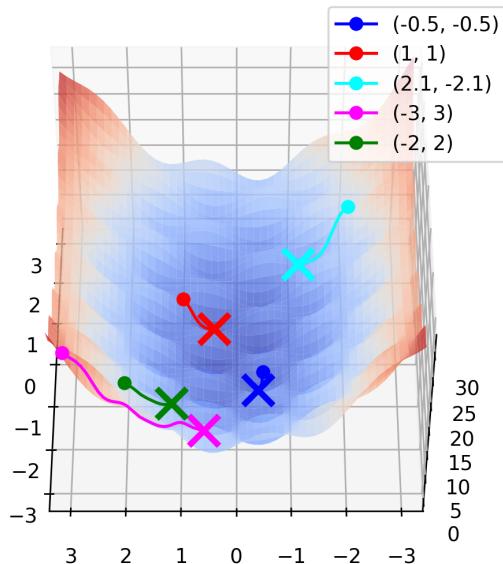
Se puede observar que, los valores de $f(x, y)$ se hacen más y más negativos a medida que el punto inicial se acerca también a 0 con la tasa de aprendizaje más baja, por ejemplo, se puede notar que los puntos con menor valor obtenido han sido $(-0.5, -0.5)$ y $(1, -1)$, y a partir de allí se puede ver como mientras más alejado se encuentre el punto, se obtienen valores más grandes con $\eta = 0.01$, mientras que con los valores de $\eta = 0.01$ se reducen a valores similares pero en valores cercanos al 0, no se minimiza bien; esto confirma que el valor inicial del gradiente tiene también importancia junto con la tasa de aprendizaje, esta idea se puede ver perfectamente en las Figuras 5 y 6.

En dichas figuras, se pueden observar los efectos que tiene tanto el punto de inicio, ya que por ejemplo, se puede ver como el valor que comenzó en $(-3, 3)$ tiene más camino por recorrer hacia los valores mínimos con la misma tasa de aprendizaje y por lo tanto, no logra llegar más que al mínimo que tiene más cerca a excepción de cuando tiene una tasa de aprendizaje mayor, en ese caso logra llegar a un mínimo de mejor calidad, lo mismo sucede con el resto de puntos que se encuentran más alejados del “valle”; en contraste con el punto $(-0.5, -0.5)$ el cual comienza muy cerca de un mínimo y básicamente se estanca en él sin moverse mucho de lugar, y con una tasa de aprendizaje grande, sucede que se sale de ese mínimo y empieza a oscilar por la función.

Cabe notar que esto sucede de esta manera, es decir, que los valores más cercanos a cero minimicen más la función se debe a la misma forma de esta función en particular, que tiene una topología análoga a un valle con cráteres; si la función tuviera una forma distinta, naturalmente se obtendrían valores diferentes, por ejemplo, podría darse el caso de que valores más alejados del cero minimicen más la función, en todo caso, puede generalizarse y concluir que los valores de inicio para la minimización son importantes, pues dependiendo de los mismos se podrá llegar a un mínimo de buena calidad o bien se llegará en menor número de iteraciones al mismo.

Por otro lado vale la pena comentar que nuevamente se puede ver el efecto de la tasa de aprendizaje tiene sobre la minimización de la función, si bien en la Figura 5 se puede ver un avance suave y

Diferentes puntos de inicio para $f(x, y)$, $\eta = 0.01$



Ídem, vista aérea

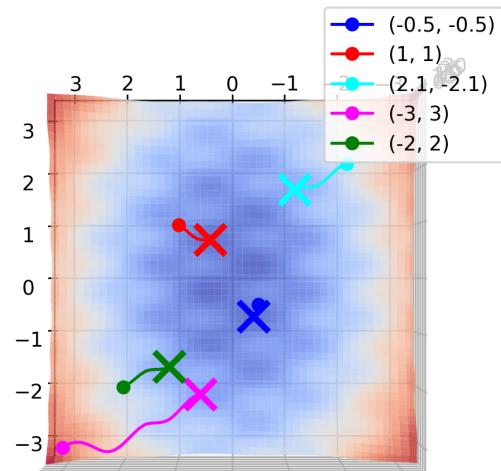
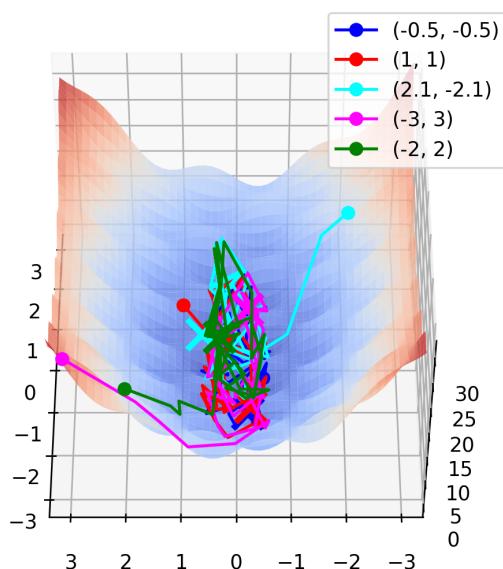


Figura 5: Visualización de los caminos que toma la minimización dependiendo del punto de inicio.

Diferentes puntos de inicio para $f(x, y)$, $\eta = 0.1$



Ídem, vista aérea

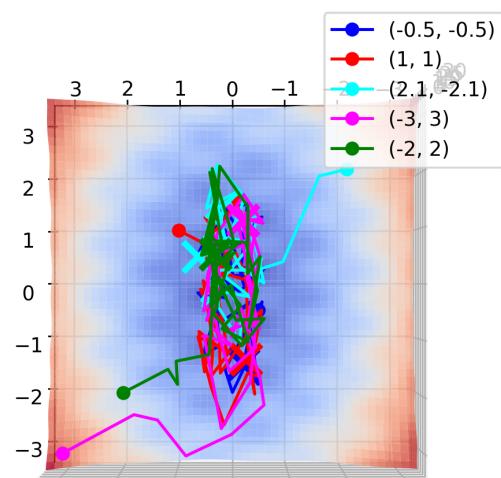


Figura 6: Ídem a la Figura 5, diferente η .

directo a los mínimos, en 6 puede verse que todos los puntos de inicio logran llegar al valle, pero tienen los pasos tan agigantados que no logran “aterrizar” en un mínimo y quedarse en él, por lo tanto se puede ver como dan círculos al rededor del mínimo global de la función.

1.4. Conclusiones sobre Gradiente Descendente respecto a la dificultad de encontrar mínimos globales

De los experimentos realizados, se han notado dos dificultades clave para poder encontrar los mínimos globales: La tasa de aprendizaje y los puntos iniciales donde se comienza a minimizar.

Como se ha comentado anteriormente, la tasa de aprendizaje afecta la posibilidad de encontrar óptimos ya que el valor de la tasa marca que tanta distancia se cubre en la función al calcular los nuevos puntos, entonces, dependiendo de la función a minimizar, si se selecciona un valor muy grande, el algoritmo estará oscilando alrededor de un mínimo; o bien, si el tamaño es muy pequeño, se tomarán muchas más iteraciones para llegar a ese mínimo, o bien el algoritmo convergerá a un mínimo subóptimo.

Esto, también depende del punto inicial, pues el algoritmo no “conoce” la topología de la función entera, la información que posee es solamente del punto dónde se encuentra, cual es la dirección de mayor cambio *en ese punto*, es un conocimiento local, por lo tanto, aunque se tuviese una tasa de aprendizaje perfecta para la función, si se localiza el punto inicial en una región de la función que posee cerca un mínimo subóptimo, el algoritmo convergerá a ese punto, aún existiendo un mínimo global en otra región de la función: simplemente, no puede saber que existe, porque solo tiene información local de la función.

Por lo tanto, y en vista de los experimentos realizados, lo que se concluye es que para mitigar dentro de lo posible estos dos problemas, se deben de realizar múltiples ejecuciones del algoritmo sobre una función, combinando distintos valores la tasa de aprendizaje y los puntos iniciales para poder cubrir más regiones de la función e ir almacenando que configuraciones obtienen las mejores minimizaciones.

2. Regresión Lineal

2.1. Experimento con Datos de Dígitos Manuscritos

Se realizaron experimentos de ajustar un modelo lineal para poder clasificar imágenes de los dígitos manuscritos 1 y 5, haciendo uso de las características de nivel medio de gris del dígito y la simetría que posee el mismo.

El vector de características que se tiene es $(1, x_1, x_2)$, con el primer término x_0 siendo 1, pues un término de sesgo o bias para permitir al modelo poder ajustar la recta para que no necesariamente pase por el eje y en cero, x_1 son los valores de intensidad promedio y x_2 es la simetría, por lo que la ecuación que tendrá que ajustar el modelo es 9, es decir, se desea encontrar el valor de los w_i que divida de la mejor manera posible los datos y por lo tanto, permita identificar de la mejor manera posible si un dígito pertenece a una clase o a otra; para ello el vector y de etiquetas se ha dispuesto de manera que las características del dígito 5 sean etiquetadas con el valor 1 y las del dígito 1 con el valor -1.

$$y = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 \quad (9)$$

Se obtuvo dicho modelo de dos maneras: por un lado se utilizó el algoritmo de la Pseudoinversa, también conocido como la “*Normal Equation*” para obtener en un solo paso los pesos óptimos junto con el algoritmo de Gradiente Descendente Estocástica haciendo uso de Minibatches para obtenerlos de manera iterativa.

2.1.1. Algoritmo de la Pseudoinversa o Normal Equation

La implementación del algoritmo se encuentra dentro de la función `pseudoinverse(x, y)` que toma el parámetro X que es una matriz con las características de cada dato y el parámetro y que serían las etiquetas asociadas a dichos datos de X , tiene como valor de retorno un vector w con los pesos con los valores óptimos del ajuste lineal para los datos y las etiquetas proporcionadas.

La expresión que calcula los pesos en un solo paso está definida como $w = X^\dagger y$, para poder obtenerla de los datos de entrada es necesario de utilizar la Descomposición de Valores Singulares de dicha matriz de características: $SVD(X) = U \cdot \Sigma \cdot V^T$, por teoría se sabe que $X^\dagger = (X^T X)^{-1} X^T$ por lo tanto, se puede desarrollar de la siguiente manera:

$$X^\dagger = (X^T \cdot X)^{-1} \cdot X^T \quad (10)$$

$$X^\dagger = (V \cdot \Sigma \cdot U^T \cdot U \cdot \Sigma \cdot V^T)^{-1} \cdot X^T \quad (11)$$

$$X^\dagger = (V \cdot \Sigma \cdot \Sigma \cdot V^T)^{-1} \cdot X^T \quad (12)$$

$$X^\dagger = (V \cdot \Sigma \cdot \Sigma \cdot V^T)^{-1} \cdot V \cdot \Sigma \cdot U^T \quad (13)$$

$$X^\dagger = V \cdot \Sigma^{-1} \cdot \Sigma^{-1} \cdot \Sigma \cdot U^T \quad (14)$$

$$X^\dagger = V \cdot \Sigma^\dagger \cdot U^T \quad (15)$$

La expresión 15 obtenida se codificó directamente, haciendo uso de la función `np.linalg.svd()` para obtener la descomposición inicial; luego de tener la pseudoinversa no queda más que multiplicar sus valores con y para obtener directamente los pesos deseados.

2.1.2. Gradiente Descendente Estocástica con Minibatches

La implementación del algoritmo se encuentra en una nueva función `sgd(x, y, wIni, lr, batchSize, epsilon, maxIters)`, que tiene por parámetros la matriz de características de los datos de entrenamiento, las etiquetas de cada dato de entrenamiento, los pesos iniciales, la tasa de entrenamiento, el tamaño de los lotes de entrenamiento, el error mínimo y un parámetro para las iteraciones.

El algoritmo amplía la implementación de Gradiente Descendente escrita en `gradient_descent()`, pues esta anterior función es más genérica y diseñada para minimizar funciones dado un punto inicial, ahora se mantendrá fija la función a minimizar, y lo que va a variar serán los datos que se pasan a dicha función, que en este caso son los vectores de características junto con sus etiquetas.

La función que se utilizará es la función de Mínimos Cuadrados, denominada E_{IN} definida en 16, por lo tanto y siguiendo la nomenclatura de 1, tendrá ahora que $\mathcal{F} = E_{IN}$ para el nuevo algoritmo de Gradiente Descendente Estocástico.

$$E_{IN}(w) = \frac{1}{N} \sum_{n=1}^N (w_n \cdot x_n - y_n)^2 \quad (16)$$

De la misma manera, ahora la gradiente $\frac{\partial \mathcal{F}(W)}{\partial W} = \nabla E_{IN}(W)$ queda definida en 17, donde x_{nj} es el componente j -ésimo de la entrada n .

$$\nabla E_{IN}(W) = \frac{2}{N} \sum_{n=1}^N x_{nj} \cdot (w_n \cdot x_n - y_n) \quad (17)$$

Está claro que las ecuaciones 16 y 17 pueden convertirse fácilmente en operaciones matriciales y en efecto, se encuentran implementadas de dicha forma en `Err(x, y, w)` y `gradSGD(w, x, y)` respectivamente.

Ahora, otra diferencia con la implementación original de Gradiente Descendente es el hecho de que se trata de la variante estocástica con minibatches: Ahora, para calcular el gradiente no se utilizarán todos los datos, solamente un pedacito de los mismos, un “lote” de ellos que será seleccionado estocásticamente.

Dentro del código se hicieron los cambios necesarios para poder, dado un valor al que se desea fijar la longitud de dichos lotes, se obtengan tantos lotes se necesiten para cubrir los datos y luego, se reparten los datos en los lotes de forma aleatoria.

Una vez que se recorren todos los datos, lo cual se denomina una época, se vuelven a barajar de distinta manera; en todo momento manteniendo la correspondencia de un dato de entrenamiento con su etiqueta.

Cabe notar que el número de iteraciones se cuenta como cada vez que se calcula un gradiente, es decir, no es el número de épocas.

El resto del código es similar a la implementación original, cuando se ha llegado al número de iteraciones pedidas o bien se ha llegado a un umbral de error, el algoritmo se detiene y se regresan los valores de los pesos y también el número de iteraciones.

2.1.3. Ajustando los datos con Pseudoinversa y Gradiente Descendente Estocástica

Una vez implementadas las funciones, se procedió a la carga de los datos, esto se realizó utilizando la función provista `readData()` dos veces, una para cargar los datos de entrenamiento y otra para los datos de prueba, o test.

Ya que se está realizando un problema de clasificación, se recuerda que la clase 1 es el dígito manuscrito 5, mientras que -1 es el dígito manuscrito 1, como se está trabajando con 2 parámetros lineales, el vector de características queda de la forma $(1, x_1, x_2)$ con x_1 siendo el nivel de simetría y x_2 el nivel medio de gris.

	w_0	w_1	w_2
Pseudoinversa	-1.1159	-1.2486	-0.4975
SGD	-1.108	-0.1301	-0.419

Cuadro 3: Pseudoinversa vs. Gradiente Descendente Estocástica: Pesos obtenidos

	Algoritmo	
	Pseudoinversa	SGD
Datos de Entrenamiento		
Error E_{IN}	0.0792	0.0919
Accuracy	99.4875 %	99.4875 %
Datos de Test		
Error E_{OUT}	0.131	0.1413
Accuracy	98.3491 %	98.3491 %

Cuadro 4: Pseudoinversa vs. Gradiente Descendente Estocástica: Ajuste a los datos

Una vez realizada la carga de los datos, se obtuvieron los pesos w_i que ajustan la ecuación 9 en un solo paso llamando a la función `pseudoinverse()`, y de igual forma, se obtuvieron otros pesos invocando la función `sgd()` con 200 iteraciones como máximo, $\eta = 0,01$, parar si se obtiene un error de 0, pesos iniciales $(0, 0, 0)$, y un tamaño de lotes de 32.

Se realizó de esta forma para tener una primera idea de cómo se ajusta el algoritmo a los datos, sabiendo que, en caso de obtener resultados de poca calidad, se puede empezar a variar tanto el η como los valores iniciales y, en último caso, las iteraciones máximas pues por la complejidad de los datos, 200 iteraciones deberían ser más que suficientes; respecto al tamaño de lotes, se recomendó que se utilizase un tamaño de lotes de 32.

Una vez se completó la ejecución de ambas funciones se obtuvieron los pesos dispuestos en el Cuadro 3. Si bien los valores son muy similares, se aplicaron dos métricas en las que comparar los algoritmos, primero se utilizó el Error, es decir, el E_{IN} que se ha mencionado previamente, y también se introduce la métrica de Accuracy, es decir, el número de predicciones correctas entre el número total de predicciones, se asume que una predicción es correcta si posee el mismo signo que la etiqueta, con indiferencia del valor exacto que se obtuvo; esta funcionalidad se encuentra en la función `acc()`.

Los resultados de las métricas sobre los pesos obtenidos por los algoritmos se pueden observar en el Cuadro 4; se puede ver que en los mismos tanto la Pseudoinversa como SGD obtienen los mismos valores de accuracy para Test y Train, además de valores muy cercanos en el propio error, por un lado, esto confirma que ambos algoritmos están minimizando correctamente el error de mínimos cuadrados sobre los datos, indicando que la Pseudoinversa si es verdaderamente capaz de obtener los pesos óptimos para modelos lineales y también por otro lado indicando que de igual manera el Gradiente Descendente Estocástico es igual de capaz de llegar a prácticamente los mismos valores.

De manera más visual, se puede detallar cómo ambos algoritmos están dividiendo los datos en las Figuras 7 y 8; si bien las rectas no son exactamente las mismas, los valores que han obtenido en sus pesos son igual de capaces de dividir básicamente de la misma manera los datos, esto nuevamente

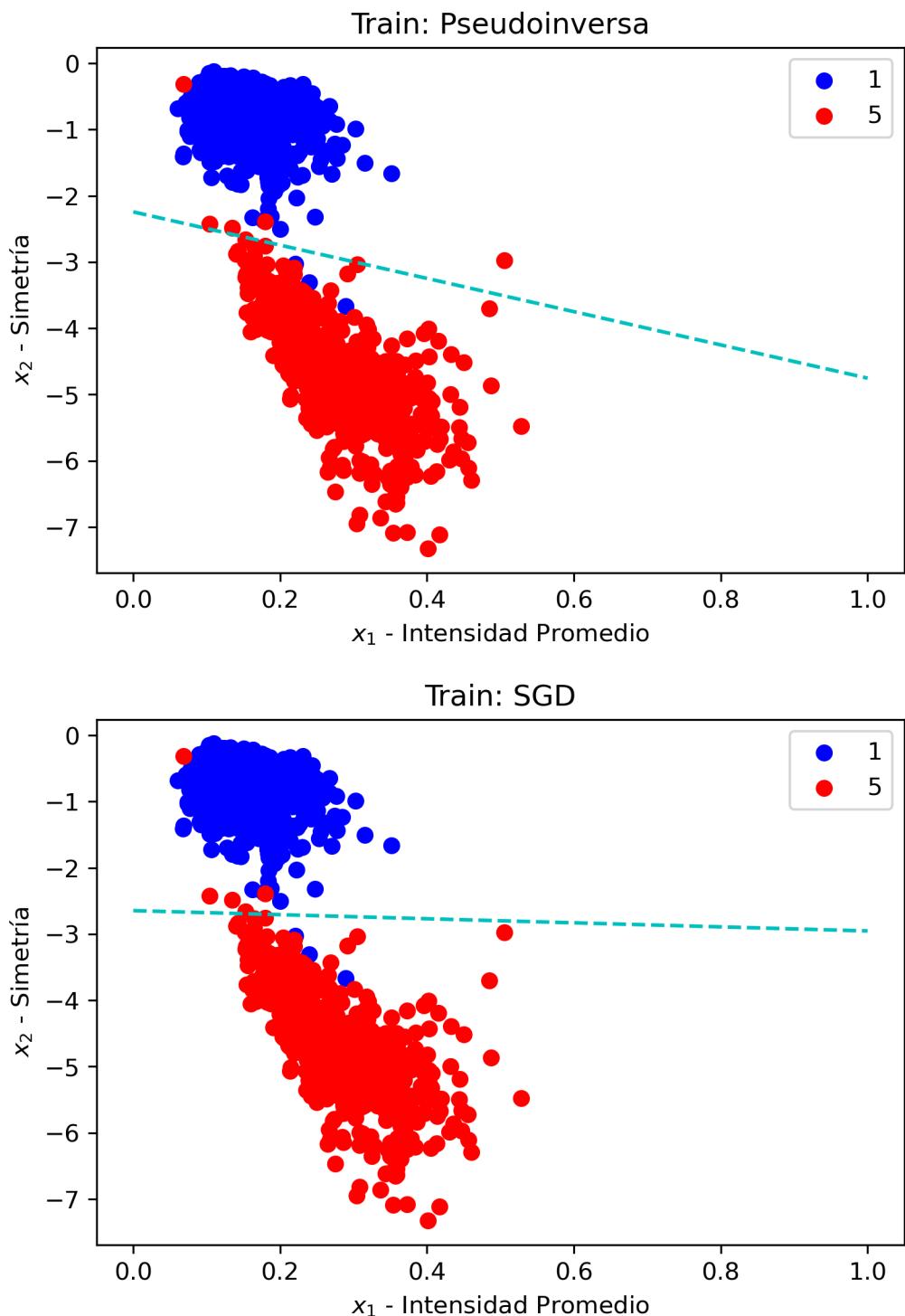


Figura 7: Pseudoinversa vs. Gradiente Descendente Estocástica: Ajuste a los datos de entrenamiento

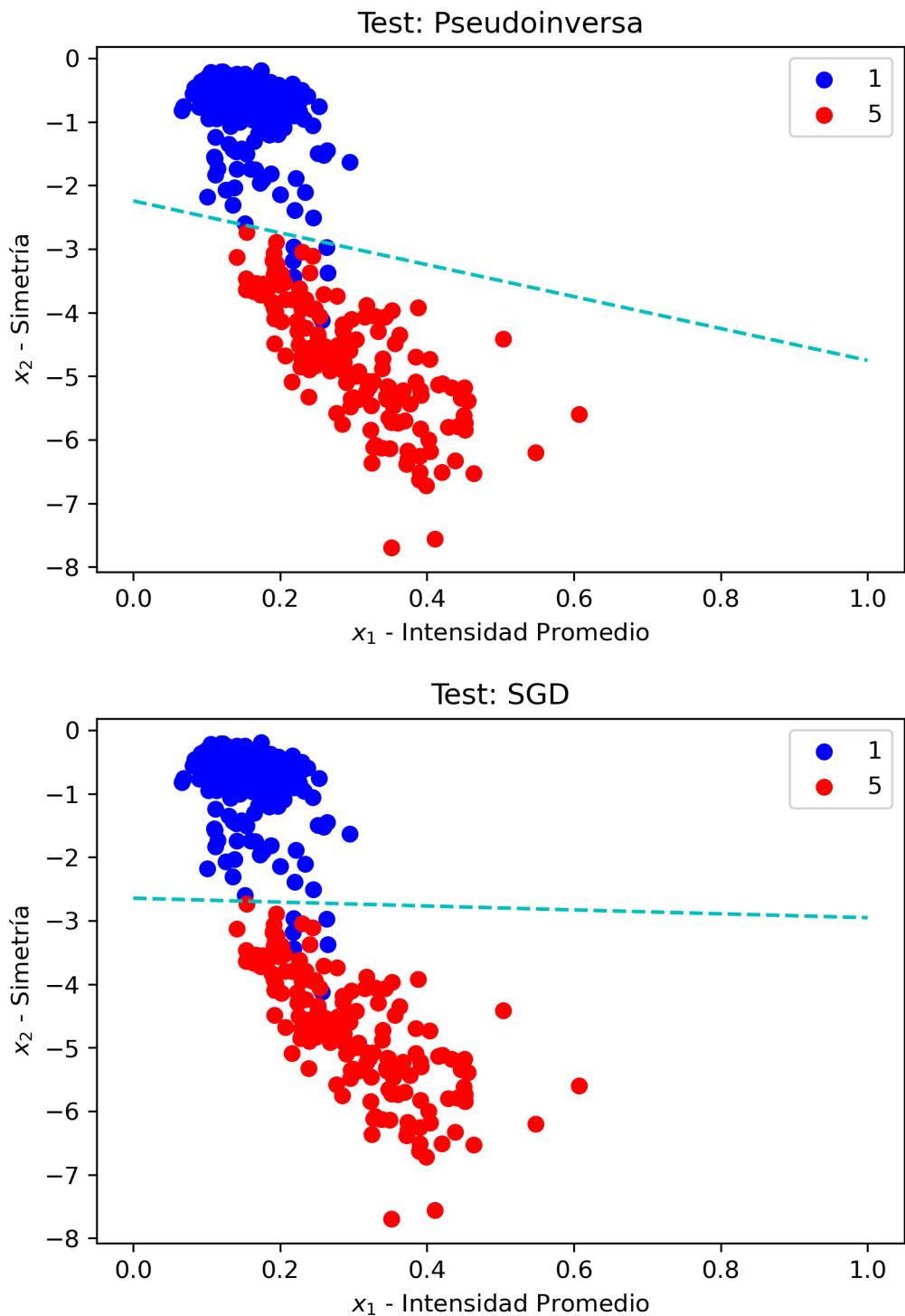


Figura 8: Pseudoinversa vs. Gradiente Descendente Estocástica: Ajuste a los datos de prueba

afirmando que son ambos algoritmos igual de capaces de ajustar modelos lineales.

Si bien es válido recalcar que se sabe que el uso de la pseudoinversa está limitado a problemas sencillos y con pocos datos pues el coste computacional de realizar la inversión de una matriz es mucho mayor que el coste que tiene el algoritmo de Gradiente Descendente, además de esto, la pseudoinversa está también limitada en solo poder realizar regresión lineal, no puede extrapolarse a otros modelos.

2.2. Experimentos con Características no Lineales

2.2.1. Visualización de los datos y ajuste a los mismos

Para la realización de este experimento, el cual tiene como objetivo analizar como se transforman los errores dentro y fuera de la muestra al aumentar la complejidad del modelo lineal, se realizaron los siguientes pasos:

Primero se generó una muestra de 1000 datos de entrenamiento por medio de la llamada a la función ya provista `simula_unif()` con los parámetros indicados para generar 1000 puntos en 2 dimensiones sobre un cuadrado $\mathcal{X} = [-1, 1] \times [-1, 1]$.

Luego, haciendo uso de la función $f(x_1, x_2) = sign((x_1 - 0,2)^2 + x_2^2 - 0,6)$ que está implementada dentro de `f(x1, x2)` y la cual asigna la etiqueta de -1 a valores que se encuentran en un óvalo unitario con centro (0,0, 0,25) y 1 a valores que están fuera de ese óvalo.

Luego, se le asignaron las 1000 etiquetas a los 1000 datos con la función `wrapper` denominada `getTags(x)` que es un bucle sobre la función `f()` y rellena el vector de etiquetas dependiendo de los datos de entrada, el cual luego retorna.

Luego, se le añadió ruido al vector de etiquetas, para ello se implementó la función `addNoise(y, percent)` que dado el vector de etiquetas y un valor entre 0 y 1, internamente se invierten las etiquetas de manera proporcional pero aleatoria; en el caso del experimento se introdujo un 10 % de ruido a los datos, es decir que al 10 % de las etiquetas 1, se cambiaron a -1 y al 10 % de las etiquetas de -1 se cambiaron a 1 de forma aleatoria y por lo tanto eso equivale a un cambio de 10 % en el total de los datos.

Se pueden visualizar las etiquetas con ruido introducido asociadas a cada dato en la Figura 9.

Usando el vector de características lineales ya conocido para problemas de dos características, es decir, $(1, x_1, x_2)$ se estimaron los pesos por medio de una llamada a `sgd()`, se utilizó previamente una función auxiliar `addBias()` que añade la columna del bias o sesgo a los datos, pues no la poseen por defecto y es necesaria para el ajuste de los mismos.

Adicionalmente a esto, se generó un vector de características no lineales definido como $\phi_2(x) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$ partiendo de las mismas características lineales; se encuentra implementado en la función auxiliar `phi()` que internamente realiza los cálculos pertinentes y los añade vector de características, luego se añade también el bias. También se utiliza otra llamada a `sgd()` para obtener los pesos.

En ambos casos, se utilizó un valor de $\eta = 0,01$, los pesos inicializados a cero, y 200 iteraciones máximas. Puesto que, de momento utilizando estos valores por defecto, se han obtenido resultados

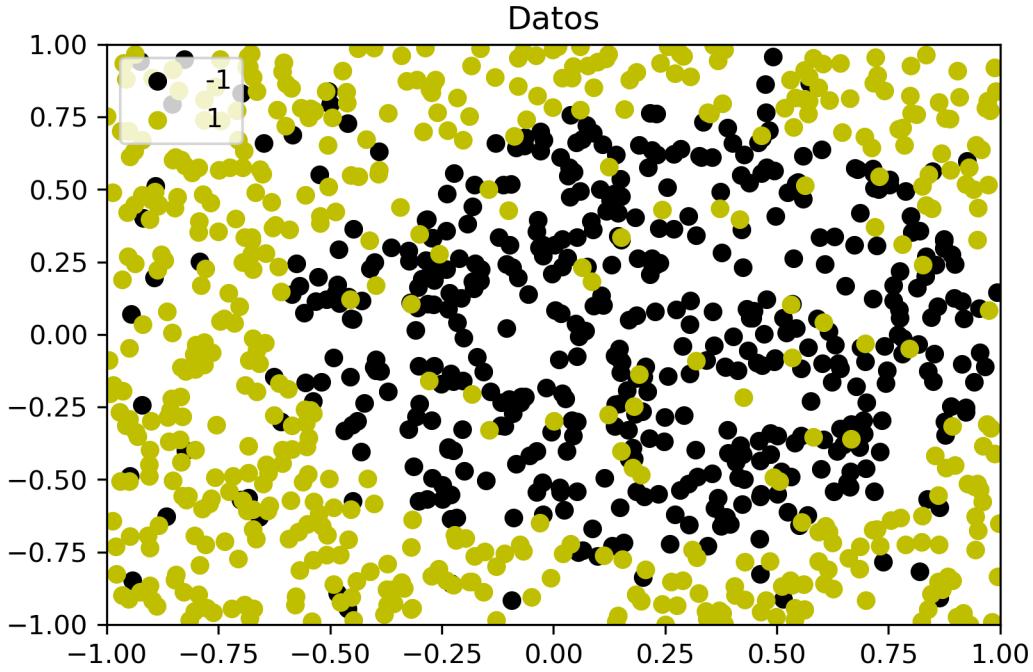


Figura 9: Visualización del mapeo de las etiquetas a los datos generados por `simula_unif()`, tener en cuenta que esta función genera un conjunto diferente de datos en cada ejecución por lo que ejecuciones del código tendrán una distribución de etiquetas ligeramente diferente.

favorables, y ya se ha visto que en la mayoría de los casos, utilizar un η grande produce malos resultados.

	w_0	w_1	w_2	w_3	w_4	w_5
Lineales	0.0343	-0.35486	-0.0435	-	-	-
No Lineales	-0.1733	-0.351	-0.0421	0.0258	0.3283	0.5048

Cuadro 5: Características Lineales vs No Lineales: Pesos obtenidos

	Características	
	Lineales	No Lineales
E_{IN}	0.9283	0.7505
Accuracy	60.7 %	78.9 %

Cuadro 6: Características Lineales vs No Lineales: Valores de ajuste obtenidos en entrenamiento

Una vez realizadas las ejecuciones del algoritmo, se obtuvieron los pesos indicados en 5, y cuyas métricas sobre los datos de entrenamiento y test están dispuestos en el Cuadro 6, se puede apreciar claramente que el ajuste realizado por las características solamente lineales es bastante pobre, pues obtener un 60 % de *accuracy* indica que poco se ha podido ajustar el modelo; recordando que en esta clase de problemas de clasificación binaria, el peor resultado que se puede obtener es 50 % de *accuracy*, pues esto indica que el modelo está eligiendo al azar las etiquetas para cada dato.

Por otro lado, el modelo con características no lineales ha dado un resultado mucho mejor, con una

	Características	
	Lineales	No Lineales
Entrenamiento		
E_{IN} medio	0.9284	0.7536
Accuracy medio	59.22 %	77.93 %
Test		
E_{OUT} medio	0.934	0.7588
Accuracy medio	58.85 %	77.47 %

Cuadro 7: Características Lineales vs No Lineales: Ajuste promedio de 1000 ejecuciones a diferentes datos de entrenamiento y test.

Accuracy de casi 80 % de etiquetas acertadas correctamente, también esto resulta en un valor de error en entrenamiento de 0.7 en comparación con el 0.9 de las características lineales.

De una manera más clara, se ha graficado nuevamente sobre los datos de qué manera el modelo los divide, en este caso se puede apreciar en la Figura 10 como el ajuste solamente lineal no logra en absoluto poder ajustarse a los datos, esto debido a que básicamente es una recta y los datos poseen características no lineales, en este caso, que se encuentran distribuidos en una forma ovalada o circular; por lo tanto no es de sorprender que el ajuste del modelo con las características no lineales, al tener esa potencia añadida de la no linealidad es capaz de dibujar una frontera elíptica, aunque no del todo perfecta, pero es un buen precedente que indica que la manera que hay que proseguir para este tipo de datos es hacer uso de características no lineales.

2.2.2. Múltiples ejecuciones del experimento

Los resultados del experimento anterior resultan prometedores, pero al ser una ejecución única, también es necesario considerar que podría haber sido un *outlier*, que haya sido mera suerte que las características no lineales posean una ventaja sobre las lineales para estos datos, entonces, debido a que la propia función generadora de datos los produce de manera estocástica, ahora se procedió a repetir el experimento anterior 1000 veces.

Adicionalmente, ahora también se evaluará la calidad del ajuste no solamente en los datos de entrenamiento, sino también en los datos de test, en este caso significa que se generarán otros 1000 datos que servirán como el conjunto de test.

El código es análogo al del experimento anterior, con las modificaciones pertinentes para poder almacenar los valores de evaluación; una vez realizado el experimento, los datos obtenidos se pueden apreciar en el Cuadro 7.

A la vista de los experimentos realizados, se puede concluir con seguridad que el modelo más adecuado para estos datos es aquél que posee características no lineales, ya que al visualizar los datos se obtuvo una idea de cómo está distribuido el problema; se observa que unas características únicamente lineales, por ser exactamente eso, lineales, no son capaces de ajustarse a dichos datos.

Queda confirmado lo visto en teoría referente a la posibilidad de aplicar la no linearidad a las características y seguir estando en un modelo lineal, pues lo que se está cambiando no es el modelo en sí, sino el espacio de los datos, se están transformando los datos de manera que el problema

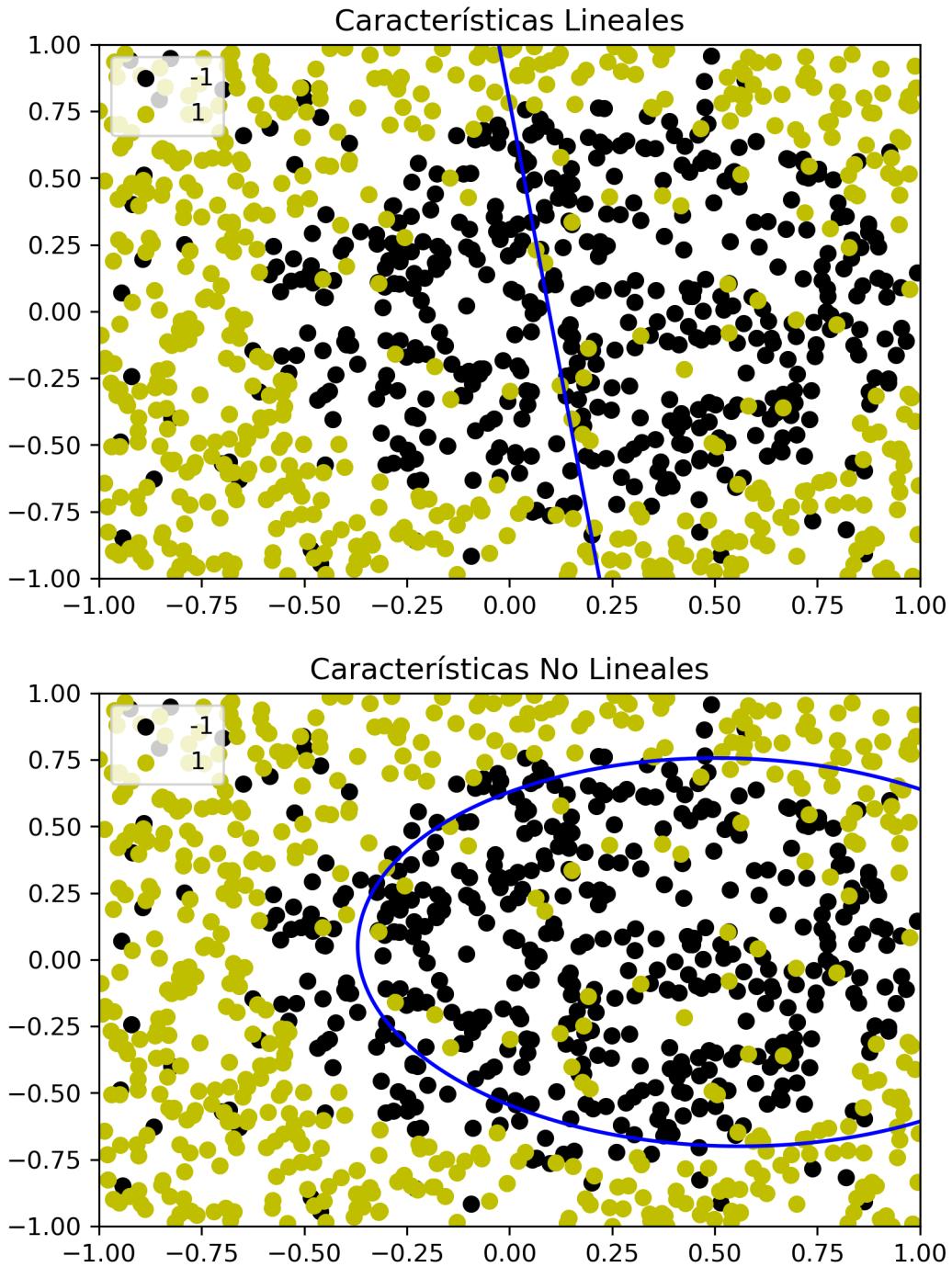


Figura 10: Características Lineales vs No Lineales: Ajuste a los datos de entrenamiento

pueda volverse lineal luego de dicha transformación, es decir, realizar una especie de mapeo de un espacio de características no lineal a uno que sí lo es y allí es donde se aplica el algoritmo; pues puede apreciarse que efectivamente los pesos se están ajustando a los datos sin tener que haber realizado ninguna modificación al algoritmo de Gradiente Descendente Estocástico, solamente se han tenido que aumentar los datos.

3. Bonus: Método de Newton

3.1. Implementación del algoritmo y cálculo de Hessiana

Para la realización de esta sección bonus se implementó una función nueva denominada `newton(w_ini, lr, grad_fun, fun, epsilon, max_iters)` que comparte los mismos parámetros que la función `gradient_descent()` definida previamente, técnicamente se ha implementado la variante conocida como *Relaxed Newton-Raphson Method* al poder fijar la tasa de aprendizaje diferente de 1.

El método de Newton es en esencia una adición al algoritmo de Gradiente Descendente; donde la expresión original hace uso únicamente de la gradiente de la función a minimizar, este método añade a este cálculo las segundas derivadas, esto es, se le añade la matriz Hessiana de la función a minimizar, por lo tanto, partiendo de la expresión original de 1, se obtiene ahora la expresión 18.

$$W' = W - \eta \times -\mathcal{H}_{\mathcal{F}}(W)^{-1} \cdot \nabla \mathcal{F}(W) \quad (18)$$

Por lo tanto, dentro de la función `newton()` se ha modificado el bucle principal, originario de la función `gradient_descent()` para reflejar los cambios comentados.

Para el experimento, se reutilizará la función $\mathcal{F} = f(x, y)$ que se encuentra definida en 7, se posee ya calculado el $\nabla f(x, y)$ en 8; ahora lo único necesario es la matriz Hessiana, que puede obtenerse analíticamente derivando nuevamente las primeras derivadas, esto produce la matriz $\mathcal{H}_{f(x,y)}$ que se puede ver en 19.

$$\mathcal{H}_{f(x,y)}(W) = \begin{bmatrix} 2 - 8\pi^2 \sin^2(2\pi x) \sin^2(\pi y) & 4\pi^2 \cos(2\pi x) \cos(\pi y) \\ 4\pi^2 \cos(2\pi x) \cos(\pi y) & 4 - 2\pi^2 \sin(2\pi x) \sin(\pi y) \end{bmatrix} \quad (19)$$

Esta matriz se encuentra implementada en la función `hessianF(x, y)` de forma análoga.

Una vez hecho esto, se repiten los experimentos de la subsecciones 1.3.2 y 1.3.3 ahora comparando `gradient_descent()` con `newton()`.

3.2. Variando la Tasa de Aprendizaje al Minimizar la Función

El experimento se repite con los mismos parámetros que en 1.3.2, es decir, punto inicial $(1, -1)$, variando el η entre 0.01 y 0.1 con el límite de iteraciones a 50.

Una vez finaliza el algoritmo de Newton, se obtienen los resultados dispuestos en el Cuadro 8.

η	Métrica	Algoritmo	
		GD	Newton
0.01	x_{FIN}, y_{FIN}	-1,2178, 0,4134	-0,9614, 0,9814
	$f(x_{FIN}, y_{FIN})$	-0.0623	2.8787
0.1	x_{FIN}, y_{FIN}	-1,147, -0,273	-0,8963, 0,9429
	$f(x_{FIN}, y_{FIN})$	2.6720	2.7979

Cuadro 8: Comparativa de valores obtenidos luego de ejecutar ambos algoritmos con mismos parámetros.

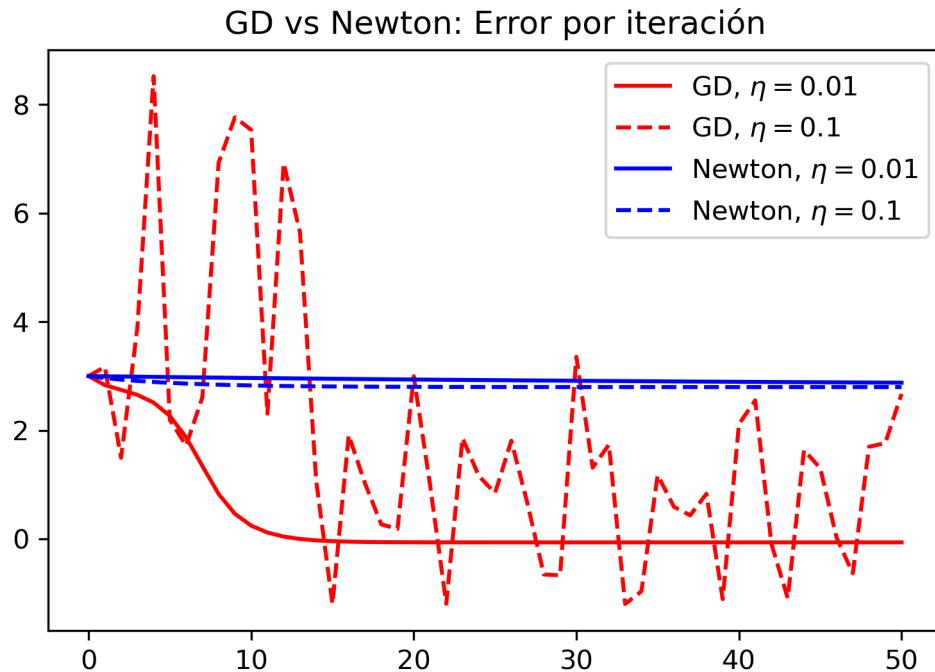


Figura 11: Comparativa entre la minimización que se obtuvo originalmente con gradiente descendente y los valores que obtiene Newton para distintos valores de η .

Se puede observar que el método de Newton está obteniendo valores subóptimos, a diferencia del algoritmo de Gradiente Descendente, esto se puede evidenciar más con la Figura 11, dónde puede verse la drástica diferencia que existe en la minimización por el método de Newton implementado y el Gradiente Descendente.

Debido a investigaciones realizadas, se ha descubierto que el método de Newton es muy propenso a quedarse atrapado en puntos de inflexión y también en puntos silla de las funciones si resulta ser que el punto dónde se comienza a iterar queda cerca del mismo; por lo tanto no es descabellado pensar que justamente eso es lo que está ocurriendo en el punto evaluado, pues el valor se mantiene prácticamente constante por toda las iteraciones. [1]

3.3. Variando el Punto Inicial

Se repite el experimento 1.3.3, nuevamente se tienen los puntos de inicio $(-0.5; -0.5)$, $(1; 1)$, $(2,1; -2,1)$, $(-3; 3)$ y $(-2; 2)$.

Una vez se realiza la ejecución del bucle, se obtienen los siguientes puntos finales, dispuestos en el Cuadro 9.

x_{INI}, y_{INI}	η	Métrica	Algoritmo	
			GD	Newton
-0.5, -0.5	0.01	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	-0,7306, -0,4144 -1.0366	-0,6430, -0,4579 -0.7185
	0.1	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	-1,5152, -0,2142 2.2685	-1,2851, -0,6355 4.2366
1, 1	0.01	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	0,7308, 0,4144 -1.0366	1,043, 1,023 3.1424
	0.1	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	-1,4236, -0,2664 2.8544	1,1252, 1,0812 3.2468
2.1, -2.1	0.01	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	1,6651, -1,1728 4.6338	2,5286, -2,7189 21.4546
	0.1	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	0,5090, 0,6340 0.9598	4,2023, 15,316 485.2281
-3, 3	0.01	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	-2,1888, 0,5868 3.6942	-2,8712, 2,9226 25.6755
	0.1	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	1,1960, -0,2105 0.3607	-2,1559, 1,4892 10.743
-2, 2	0.01	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	-1,6643, 1,1713 4.6337	-2,0914, 2,0531 12.6245
	0.1	X_{FIN}, Y_{FIN} $f(X_{FIN}, Y_{FIN})$	0,5547, 0,3094 -0.0578	-2,1458, 1,9547 12.4713

Cuadro 9: Comparativa entre la minimización que se obtiene entre Gradiente Descendente y Newton dependiendo de los puntos iniciales.

Se pueden observar que los resultados que se obtienen con el método de Newton dejan mucho que desear, en general los valores tienden a crecer en vez de decrecer, esto confirma otra desventaja que posee el método y es que si la Hessiana no es positiva definida entonces lo que se está realizando es ir en la dirección contraria[2], y entonces se está maximizando la función, también puede suceder que la derivada se esté volviendo muy pequeña y por lo tanto, el valor resultante sea entonces muy grande, y también evite llegar a un mínimo.

Se puede visualizar de mejor forma la evolución del valor de la función en las Figuras 12, dónde se ha dispuesto de esta forma la evolución del algoritmo, pues intentar dibujarlo en 3D no resultaba útil a la hora de evaluar lo que sucede con los valores.

Estos gráficos resultan útiles pues confirman lo anteriormente dicho, se puede ver claramente como los valores de la función divergen para Newton en el punto $(-0.5, -0.5)$ y $\eta = 0.1$, esto se repite en $(2.1, -2.1)$ y $\eta = 0.1$; en estos casos es claro que la Hessiana no es positiva definida y por lo tanto la dirección que está marcando el gradiente es una de maximización de valores.

En otros puntos, se puede observar que sí ha ocurrido una reducción de los valores de la función, pero no logra converger en valores mejores que los obtenidos por la Gradiente Descendente, esto

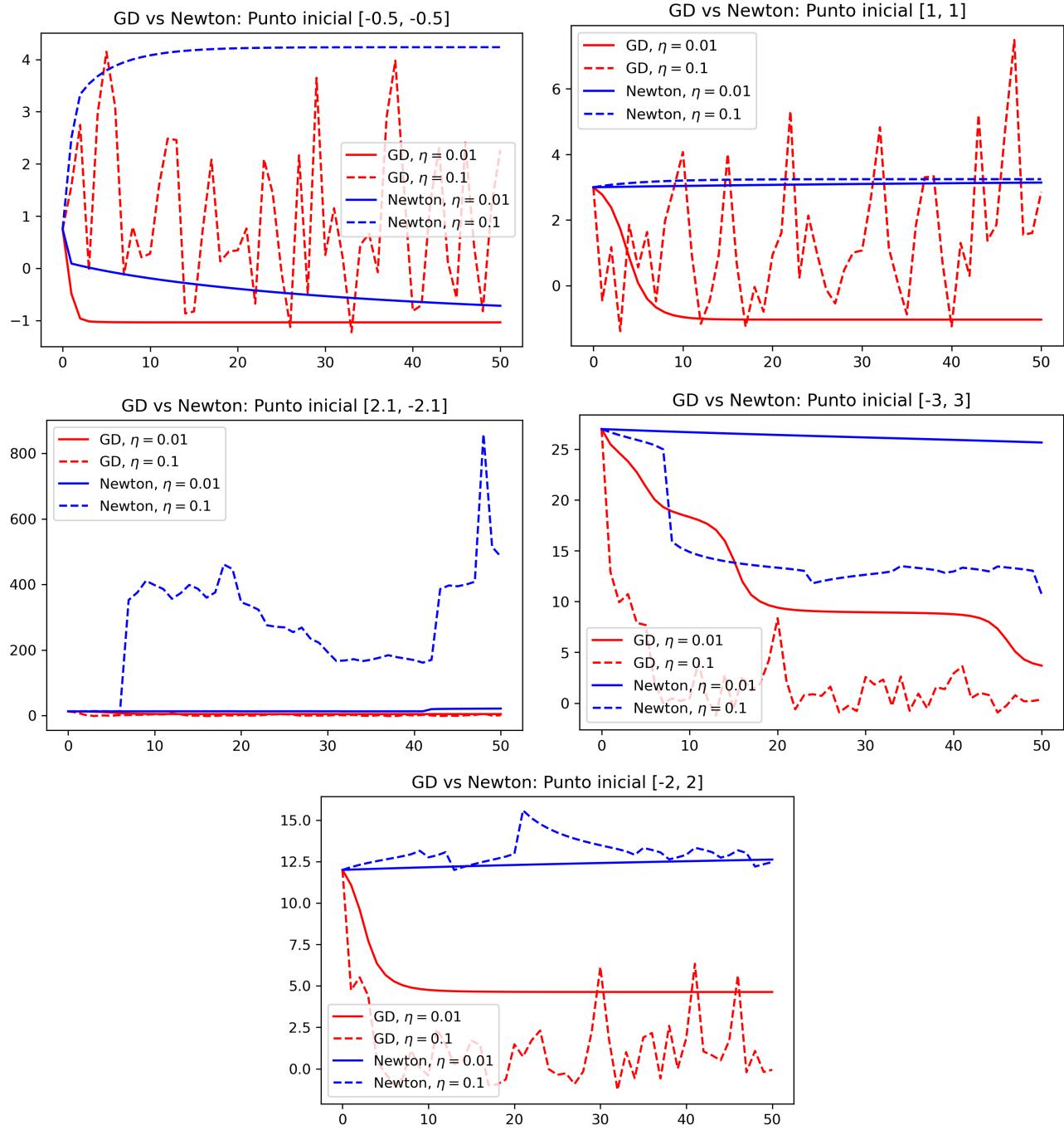


Figura 12: Gráficas mostrando para cada punto de inicio, la minimización de la función $f(x, y)$

puede deberse a lo mencionado anteriormente; este método es muy sensible a otros puntos de la función que no son el mínimo, como un punto silla o un punto de inflexión, por lo tanto, al saber que esta función posee bastantes “cráteres”, por lo que se puede decir que es una función muy ruidosa, esto evita que el método logre converger en un valor de calidad.

Si bien, se sabe que en las condiciones correctas el método de Newton llega en menos iteraciones al mínimo, por los resultados obtenidos, se concluye que las mejores situaciones para hacer uso de tal método es si se conoce de antemano la función a minimizar, o bien, se puede utilizar como un paso posterior a una minimización por el Gradiente Descendente, puesto que si el punto inicial se encuentra cerca de un mínimo local, en ese caso se puede estar más seguro que el método de Newton podrá converger en él rápidamente; además, también se sabe que es computacionalmente más complejo puesto que debe de realizar la inversión de la matriz Hessiana además de estar limitado a funciones que posean dichas segundas derivadas, cosa que con Gradiente Descendente no sucede.

Referencias

- [1] A. (<https://stats.stackexchange.com/users/36041/aksakal>), *Why is Newton's method not widely used in machine learning?* Cross Validated, URL:<https://stats.stackexchange.com/q/301728> (version: 2017-09-06). eprint: <https://stats.stackexchange.com/q/301728>. dirección: <https://stats.stackexchange.com/q/301728>.
- [2] M. (<https://math.stackexchange.com/users/210201/maniam>), *Newton optimization algorithm with non-positive definite Hessian*, Mathematics Stack Exchange, URL:<https://math.stackexchange.com/q/1132912> (version: 2015-02-04). eprint: <https://math.stackexchange.com/q/1132912>. dirección: <https://math.stackexchange.com/q/1132912>.