

# Aprendizaje Automático

-  
**2020/2021**

## Practica 1

1. **Búsqueda iterativa de óptimos**
  - 1.1. Implementación del algoritmo de gradiente descendente
  - 1.2. Estudio de la función  $E(u,v)$ 
    - 1.2.1. *Calculo analítico del gradiente de la función  $E(u,v)$*
    - 1.2.2. *Aplicando el gradiente descendente sobre  $E(u,v)$*
  - 1.3. Estudio de la función  $F(x,y)$ 
    - 1.3.1. *Gradiente descendente con distintos valores de learning rate*
    - 1.3.2. *Gradiente descendente con distintos puntos de inicio*
  - 1.4. Dificultad de encontrar el mínimo global de una función arbitraria
2. **Regresión lineal**
  - 2.1. Implementación del Error Cuadrático Medio
  - 2.2. Implementación de SGD
    - 2.2.1. *Aplicación de SGD sobre dataset*
  - 2.3. Implementación de la Pseudo-Inversa
    - 2.3.1. *Aplicación de la Pseudo-Inversa sobre dataset*
  - 2.4. SGD vs Pseudo-Inversa
  - 2.5. Experimento sobre características lineales
    - 2.5.1. *Cálculo de errores medios sobre distintos dataset*
  - 2.6. Experimento sobre características no lineales
    - 2.6.1. *Cálculo de errores medios sobre distintos dataset*
3. **Método de newton**
  - 3.1. Implementación del método de newton
    - 3.1.1. *Método de newton con distintos learning rate*
    - 3.1.2. *Método de newton con distintos puntos de inicio*

## 1.-Búsqueda iterativa de óptimos

### 1.1.-Implementación del algoritmo de gradiente descendente

A la función del gradiente descendente se le pasan como parámetros:

- El punto de inicio, donde va a comenzar la búsqueda del mínimo
- El número máximo de iteraciones
- El error que se busca alcanzar
- El learning rate o tasa de aprendizaje

Como se puede observar a continuación lo que hacemos es iterar *numMaxIter* veces o hasta que encontremos un valor de  $E(u,v)$  el cual sea menor o igual que el umbral que le pongamos. En cada iteración lo que hacemos es calcular el gradiente en el punto  $w$ , esto nos indicará hacia donde esta la máxima pendiente en ese punto para  $E(u,v)$ , pero como lo que estamos buscando es la máxima inclinación descendente pues lo que haremos es cambiar el signo al gradiente que hemos calculado anteriormente y multiplicarlo por el *learning rate*. Con esto obtendremos el incremento a aplicar a  $w$  en cada iteración.

Lo que se traduce en la siguiente modificación de  $w$ :

$$w_j = w_j - \text{learningRate} \frac{dE_{in}(w)}{dw_j}$$

Más concretamente para el problema de buscar el mínimo de  $E(u,v)$ :

$$w_u = w_u - \text{learningRate} \frac{dE(u,v)}{du}$$

$$w_v = w_v - \text{learningRate} \frac{dE(u,v)}{dv}$$

```
def gradient_descent(initial_point, numMaxIter, errorToGet, learning_rate):  
    w = initial_point  
    errorActual = E(w[0],w[1])  
    iterations=0  
    while (iterations < numMaxIter and errorToGet < errorActual):  
        incrementoW = - learning_rate * gradE(w[0],w[1])  
        w = w + incrementoW          #Modificamos el vector de los pesos  
        errorActual = E(w[0],w[1]) #Actualizamos el error actual  
        Iterations += 1  
    return w, iterations
```

### 1.2.-Estudio de la función $E(u,v)$

Estando la función  $E(u,v)$  definida como:

$$E(u, v) = (u^3 e^{(v-2)} - 2v^2 e^{-u})^2$$

### 1.2.1.-Cálculo analítico del gradiente de la función $E(u,v)$

El gradiente para este ejercicio concretamente está definido de la siguiente forma

$$\frac{dE_{in}(w)}{dw_j} = \frac{dE(u,v)}{du}, \frac{dE(u,v)}{dv}$$

Por lo que debemos calcular la derivada parcial respecto a  $u$  y respecto a  $v$

Derivada parcial de  $E(u,v)$  respecto a  $u$  donde aplicamos la regla de la cadena

$$\frac{dE(u,v)}{du} = 2(u^3 e^{(v-2)} - 2v^2 e^{-u}) \frac{d}{du} (u^3 e^{(v-2)} - 2v^2 e^{-u}) = 2(u^3 e^{(v-2)} - 2v^2 e^{-u}) (3u^2 e^{(v-2)} + 2v^2 e^{-u})$$

Calculamos la derivada parcial de  $E(u,v)$  respecto a  $v$  donde aplicamos la regla de la cadena

$$\frac{dE(u,v)}{dv} = 2(u^3 e^{(v-2)} - 2v^2 e^{-u}) \frac{d}{dv} (u^3 e^{(v-2)} - 2v^2 e^{-u}) = 2(u^3 e^{(v-2)} - 2v^2 e^{-u}) (u^3 e^{(v-2)} - 4ve^{-u})$$

### 1.2.2-Aplicando el gradiente descendente sobre $E(u,v)$

Con los siguientes parámetros:

- Punto de inicio = [1.0, 1.0]
- Learning rate = 0.1

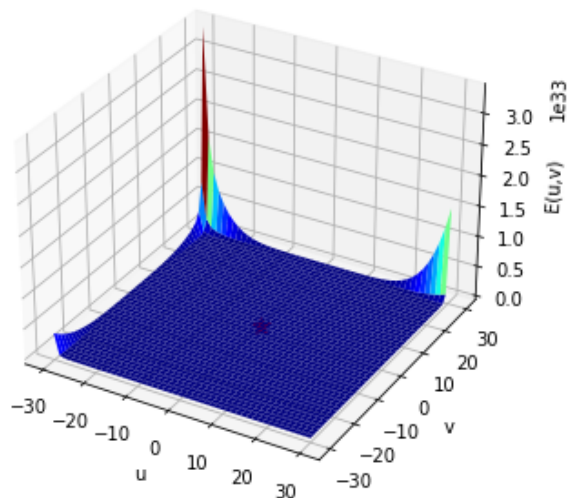
Al cabo de 10 iteraciones obtenemos un valor de  $E(u,v)$  inferior a  $10^{-14}$  en el punto (1.157289, 0.910838)

Captura de la ejecución:

```
Gradiente descendente sobre E(u,v)
Punto inicial: [ 1.0 , 1.0 ]
Learning rate: 0.1
Numero de iteraciones: 10
Coordenadas obtenidas: (1.1572888496465497, 0.9108383657484797)
Valor obtenido: 3.1139605842768533e-15
```

Punto mínimo obtenido gráficamente:

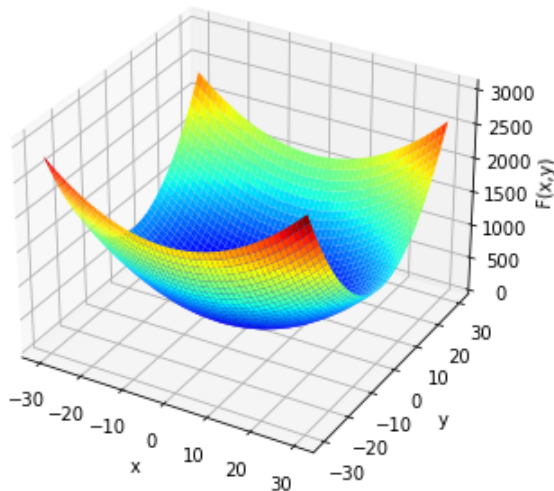
Ejercicio 1.2. Función sobre la que se calcula el descenso de gradiente



### 1.3.-Estudio de la función F(x,y)

$$F(x, y) = (x + 2)^2 + 2(y - 2)^2 + 2\sin(2\pi x)\sin(2\pi y)$$

Función F(x,y)



Vamos a aplicar el gradiente descendente sobre F(x,y) con el objetivo de minimizar la misma, por lo que debemos calcular sus derivadas parciales respecto a x e y

Derivada parcial de F(x,y) respecto a x

$$\frac{dF(x,y)}{dx} = 2(2\pi\cos(2\pi x)\sin(2\pi y) + x + 2)$$

Derivada parcial de F(x,y) respecto a y

$$\frac{dF(x,y)}{dy} = 4(\pi\sin(2\pi x)\cos(2\pi y) + y - 2)$$

Gradiente F(x,y)

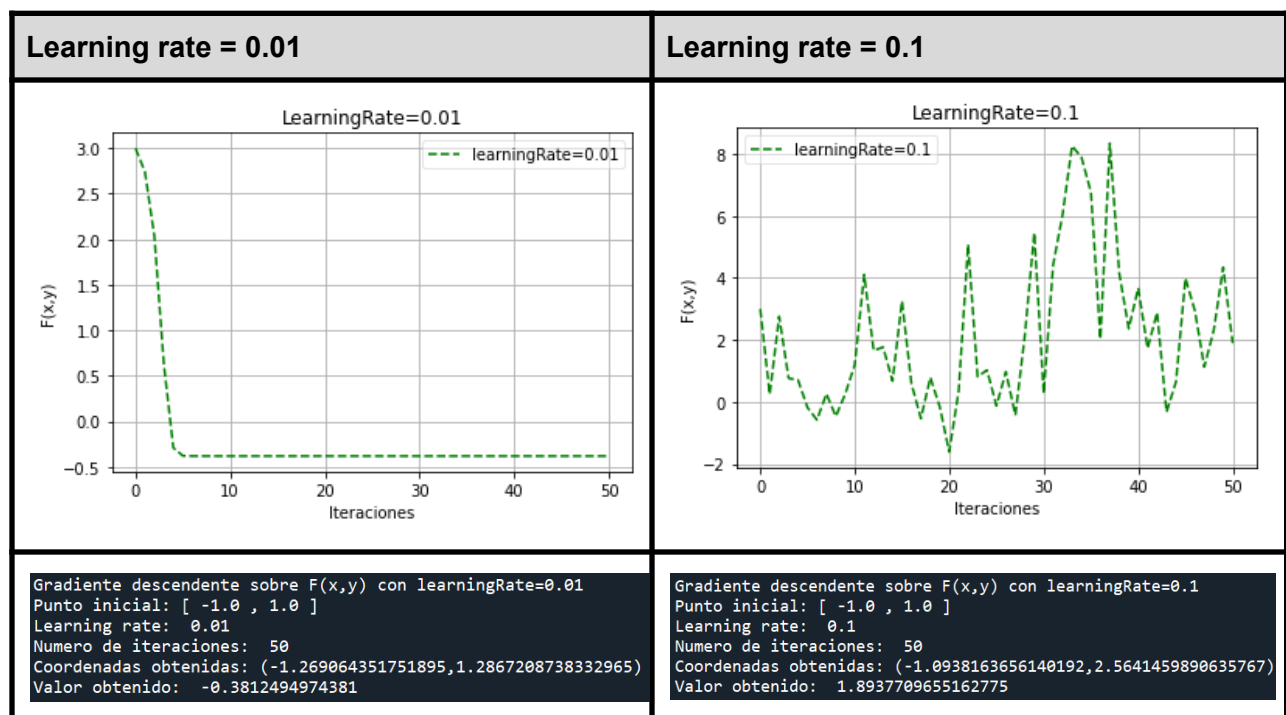
$$\text{Gradiente}F(x, y) = \frac{dF(x,y)}{dx}, \frac{dF(x,y)}{dy}$$

#### 1.3.1-. Gradiente descendente con distintos valores de learning rate

Para el cálculo del gradiente hacemos lo mismo que para el de E(u,v) solo que ahora vamos guardando los valores de f(x,y) que obtenemos en cada iteración para posteriormente ver como va descendiendo el error con distintos parámetros:

```
def gradient_descent(w, numMaxIter, errorToGet, learning_rate):  
    valoresF = np.ndarray((numMaxIter+1))  
    valoresF[0] = F(w[0],w[1])  
    iterations=0  
    while (iterations < numMaxIter and errorToGet<F(w[0],w[1])):  
        incrementoW = - learning_rate * gradF(w[0],w[1])  
        w = w + incrementoW  
        iterations+=1  
        valoresF[iterations] = F(w[0],w[1])  
    return w, iterations, valoresF
```

Como se puede observar a continuación con un learning rate de 0.01 nos vamos acercando al mínimo sin problema ya que los incrementos que realizamos a W son pequeños y no vamos saltando sin control sobre el espacio de búsqueda, y al cabo de unas pocas iteraciones ya hemos alcanzado un mínimo, pero cuando el learning rate es 0.1 los incrementos que aplicamos a W son bastante más grandes y no llegamos a converger en el mínimo, he incluso aunque lleguemos al mínimo o estemos muy cerca como puede ser en la iteración 20 al tener un learning rate tan alto, el algoritmo nos saca del mínimo y va hacia un valor mayor, es indiferente cuántas veces iteremos, no convergerá al mínimo



Tener un learning rate alto nos puede ayudar a salir de mínimos locales, ya que por ejemplo si  $F(x,y)$  tuviera un mínimo local en -0.39, el que tiene un learning rate de 0.01 probablemente se va a quedar estancado en ese mínimo local, pero si tuviera un learning rate un poco más alto

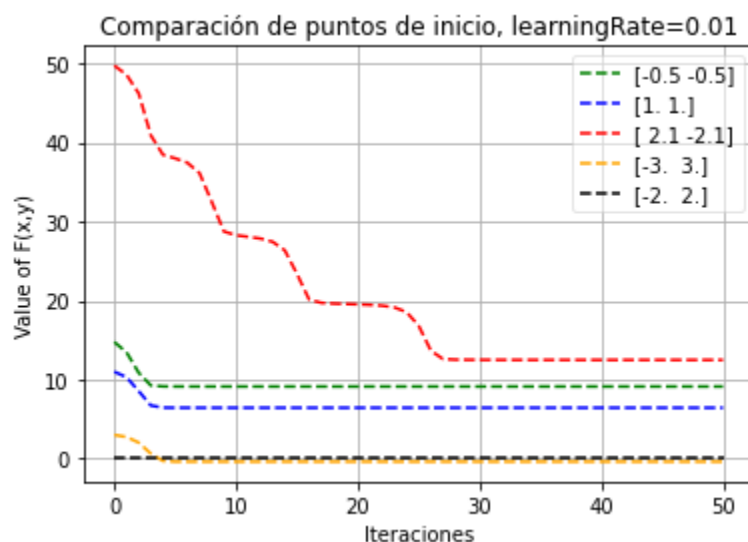
quizá podría salir del mínimo local y buscar un mínimo global. Consiste en encontrar un equilibrio entre un learning rate lo suficientemente grande para que nos permita salir de los mínimos locales, pero lo suficientemente pequeño como para que no vaya saltando por el espacio de búsqueda.

### 1.3.2.-Gradiente descendente con distintos puntos de inicio

Tratando de encontrar el mínimo global de  $f(x,y)$ , con distintos valores iniciales, he obtenido los siguientes resultados con un learning rate = 0.01

Punto inicial(x,y)	Valor minimo	Donde Valor min(x,y)
[-0.5 -0.5]	9.125146662901855	x= -0.7934994705090673 y= -0.12596575869895063
[1. 1.]	6.4375695988659185	x= 0.6774387808772109 y= 1.290469126542778
[ 2.1 -2.1]	12.490971442685035	x= 0.1488058285588777 y= -0.096067704992243
[-3. 3.]	-0.38124949743809955	x= -2.7309356482481055 y= 2.7132791261667037
[-2. 2.]	-4.799231304517944e-31	x= -2.0 y= 2.0

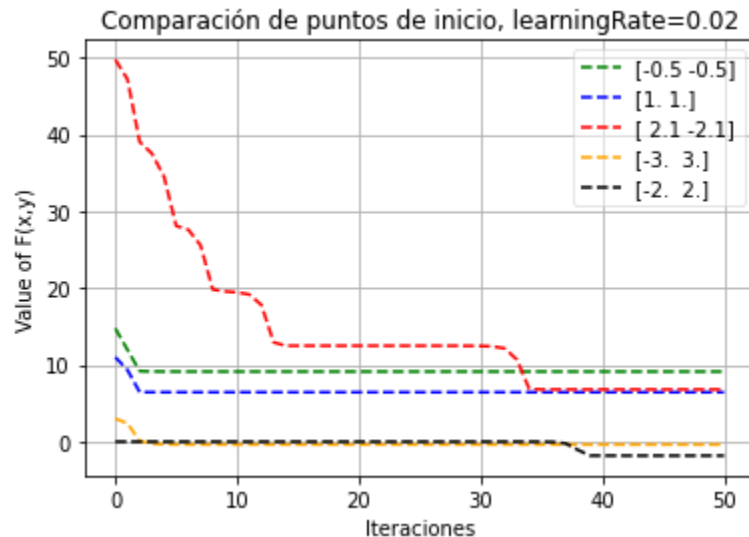
Podemos observar como dependiendo del punto en el que iniciemos la búsqueda del mínimo alcanzamos unos mínimos locales u otros, no conseguimos el mínimo global en todos los casos debido a que cada ejecución del algoritmo empieza en un punto distinto por lo que toman caminos distintos para llegar al mínimo, esto implica que no todas las ejecuciones se van a encontrar con los mismos mínimos locales.



Observando la imagen anterior podemos distinguir cada uno de los 5 mínimos locales en los que ha quedado “estancado” el gradiente descendente para cada punto de inicio..

He realizado otra ejecución para respaldar la afirmación que he hecho anteriormente acerca de la importancia de ajustar la tasa de aprendizaje a un valor que nos ayude a salir de los mínimos locales pero que no salte por todo el espacio de búsqueda sin control.

En esta ejecución he utilizado un learning rate=0.02



Podemos observar como el gradiente descendente que comenzó en el punto [2.1,-2.1] ha conseguido salir del mínimo local en el que estaba, y pasar de  $f(x,y)=12.49097$  a  $f(x,y)=6.794547$ . Así como antes podríamos creer que el mínimo global estaba en -0.38125 ahora el gradiente descendente que comenzó en el punto [-2,2] ha conseguido escapar del mínimo local en el que se quedaba con learningRate=0.01 y encontrar un valor menor,  $f(x,y)=-1.820078$ .

#### 1.4. Dificultad de encontrar el mínimo global de una función arbitraria

Más o menos ya he ido comentando en los apartados anteriores los problemas de encontrar el mínimo global de una función arbitraria. Se puede decir, que la dificultad yace en la existencia de mínimos locales, en los que el algoritmo del descenso del gradiente podría quedar “estancado”, ya que daría un paso para salir de este, pero el descenso del gradiente alrededor del mínimo local, lo volverá a meter en este. Lo que nos permitirá salir de estos mínimos locales es el learning rate, ya que lo que hacemos al aumentarlo es que los pasos que da el algoritmo sobre el espacio de búsqueda sean más grandes, por lo que si se encuentra en un mínimo local podría salir de este, o incluso saltarlo sin caer en el. Pero también hay que tener cuidado con no aumentar mucho el learning rate ya que entonces lo que vamos a hacer es obtener una búsqueda sin control sobre todo el espacio de búsqueda, sin converger en ningún momento. En la búsqueda del mínimo global también va a influir el punto de inicio que escojamos, ya que no es lo mismo comenzar la búsqueda casi al lado del óptimo global, o en un punto en el cual haya un descenso constante hacia el mínimo global, sin mínimos locales de por medio, que en un

punto en el que tengas que pasar por una gran cantidad de mínimos locales en los que puedas quedar estancado. Por lo que como conclusión podemos decir que el learning rate y el punto de inicio son valores decisivos para la búsqueda del mínimo global, y estos dependen de la función en la que estemos, no hay de forma general un valor mejor o peor que otro, hay que estudiar cada función de forma individual.

## 2.-Regresión lineal

### 2.1.-Implementación del Error Cuadrático Medio

Para calcular el error primero calculamos un vector de errores, que contiene los errores que obtenemos en cada instancia de X, al cuadrado

Una vez que tenemos el vector de errores, nos basta con hacer la media de todas las componentes del vector de errores para tener el error creado por W en un conjunto de instancias X

$$\text{Vector Errores} = (XW - Y)^2$$
$$\text{Error} = \frac{1}{N} \sum_{i=0}^{N-1} \text{VectorErrores}[i] \quad N = \text{Número de instancias}$$

```
def Err(x,y,w):  
    sumatoria = np.sum((np.matmul(x,w)-y)**2,axis=0)  
    return sumatoria/y.shape[0]
```

#### 2.1.1.-Implementacion de SGD

En el gradiente descendente estocástico vamos a tener que calcular la derivada parcial respecto a cada componente de w para poder obtener el gradiente en un punto.

Esto se puede representar de forma matricial de la siguiente manera:

$$\text{Gradiente en } W = \frac{2}{N} * (X^T (XW - Y)) \quad N = \text{Número de instancias}$$

```
def gradErr(x,y,w):  
    return (2/y.shape[0])*np.matmul(x.T,np.matmul(x,w)-y)
```

Por lo que ya para actualizar los pesos calculamos el gradiente en w para un mini-batch, lo multiplicamos por el learning rate y lo restamos a los pesos antiguos

$$W = W - \text{learningRate} * \frac{2}{N} * (X^T (XW - Y))$$

Comentando el código un poco por encima...

He creado una función que me crea mini-batches de un tamaño dado(En el caso de que no sea exacta la división, el último mini-batch tendrá menos elementos).



Como se puede observar a dividimos índices en bloques de 32 y los vamos devolviendo con yield (Info sobre como dividir una lista en python utilizando yield <https://www.geeksforgeeks.org/break-list-chunks-size-n-python/>) en vez de con return, esto permite que se pueda volver a esta función y proseguir en el estado en el que estaba, es decir no perder la iteración del bucle en este caso y como se puede observar si la llamamos y las convertimos en una lista, obtenemos una lista de arrays de tamaño 32, excepto el último que puede ser menor o igual que 32

```
def getBachs(indices,tamanoBachs=32):  
    for i in range(0, indices.shape[0], tamanoBachs):  
        yield indices[i:i+tamanoBachs]
```

Como vemos a continuación iteramos numMaxIter y en cada iteración:

- Generamos los índices del dataset
- Barajamos los índices del dataset
- Dividimos dichos índices en mini-bachs
- Para cada uno de los mini-bachs:
  - Calculamos el gradiente en W para dicho mini-bach o subconjunto de datos
  - Restamos a W el gradiente obtenido anteriormente multiplicado por el learning rate

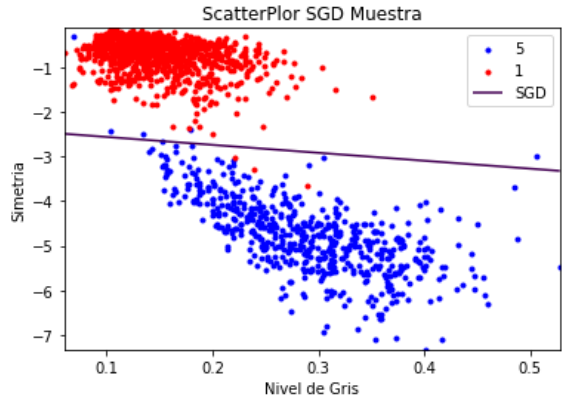
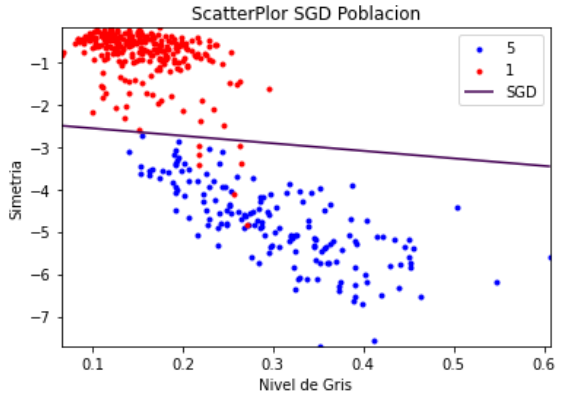
```
def sgd(x, y, w, numMaxIter, learning_rate, tamanoBachs=32):  
    iterations=0  
    while(iterations<numMaxIter):  
        indices = np.arange(y.shape[0])  
        np.random.shuffle(indices)  
        mis_bachs = list(getBachs(indices,tamanoBachs))  
        for mi_bach in mis_bachs:  
            incrementoW=-learning_rate * gradErr(x[mi_bach:],y[mi_bach:],w)  
            w = w + incrementoW  
        iterations+=1  
    return w
```

### 2.2.1 Aplicación de SGD sobre dataset

A Continuación vamos a aplicar el SGD sobre el dataset proporcionado con los siguientes parámetros:

- Learning rate = 0.01
- *núm Iteraciones = 1000 (En cada iteración barajo y divido en mini bachs e itero sobre todos los bachs)*
  - *He utilizado 1000 iteraciones por qué haciendo pruebas veía que si aumentaba las iteraciones, el error seguía descendiendo, no se estancaba en ningún mínimo local, a partir de 1000 iteraciones podemos observar cierta mejora, pero muy pequeña, y siempre > que la pseudoinversa, por lo que me ha parecido razonable dejarlo en 1000, por ejemplo con 100000 iteraciones he conseguido Ein: 0.079207606 que como veremos después es > que la pseudo inversa, y aun asi lleva un tiempo desproporcionado*
- *Tamaño de los mini bachs = 32 instancias*

he obtenido los siguiente resultados:

	En la muestra	En la población
Error	0.07962249498239364	0.13291300951599333
Scatter Plot		
W	[-1.16130004 -0.86544478 -0.48723478]	

Para dibujar  $w$  utilizamos la función `np.meshgrid(x,y)` la cual hace el producto cartesiano entre  $X$  e  $Y$ , es decir si  $X=[0,1]$  e  $Y=[0,1]$  obtenemos  $X=[0,0,1,1]$ ,  $Y=[0,1,0,1]$

A Continuación aplicamos la función  $F = w[0] + w[1]*X + w[2]*Y$

Y con `plt.contour(X,Y,F,[0])` dibujamos el primer contorno generado por  $F$

(Info sobre como utilizar `plt.contour` <https://www.geeksforgeeks.org/contour-plot-using-matplotlib-python/>)

## 2.3.-Implementación de la Pseudo-Inversa

Para obtener la pseudo inversa realizamos la siguiente operación matricial:

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

`np.linalg.inv(X)` Calcula la inversa de una matriz

```
def pseudoinverse(x):
    pseudoinversa = np.matmul(np.linalg.inv(np.matmul(X.T,X)),X.T)
    return pseudoinversa
```

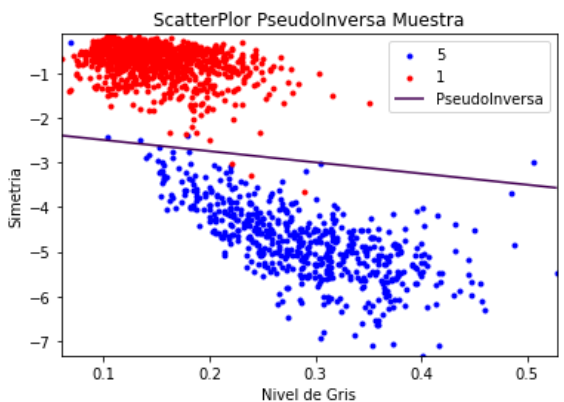
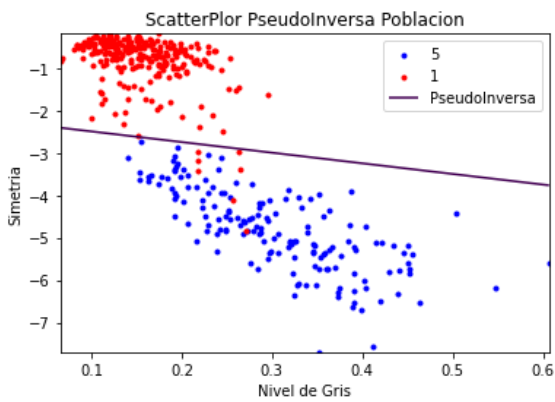
Una vez tenemos la pseudo inversa para obtener los pesos( $w$ ) que ajustan el modelo simplemente tenemos que hacer la siguiente operación matricial:

$$W = X^{\dagger} Y$$

```
def Regress_Lin(x,y):
    pseudoinversa = pseudoinverse(x)
    w = np.matmul(pseudoinversa,y)
    return w
```

### 2.3.1-.Aplicación de la Pseudo-Inversa sobre dataset

A Continuación aplicando la pseudo inversa sobre el dataset proporcionado hemos obtenido los siguientes resultados.

	En la muestra	En la población
Error	0.07918658628900395	0.1309538372005258
Scatter Plot		
W	[-1.11588016 -1.24859546 -0.49753165]	

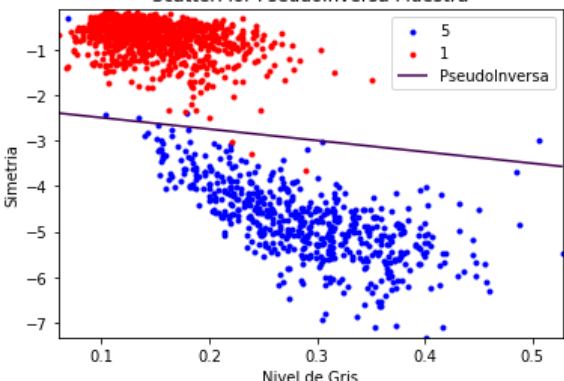
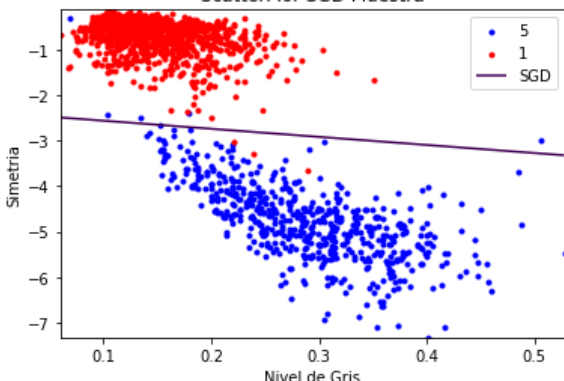
### 2.4-.SGD vs Pseudo-Inversa

Se puede observar que ambos modelos han conseguido una separación bastante buena de los datos, así como bastante similar. Aunque la Pseudo-inversa ha conseguido obtener una solución ligeramente mejor, tanto en la muestra como en la población. Entonces si la Pseudo-inversa consigue un mejor ajuste, ¿Porque usamos también el gradiente descendente estocástico?

La Pseudo-inversa hace uso de todo el dataset completo, y al ser una función cerrada nos da el menor error posible para un dataset, al contrario que el SGD que en cada momento utiliza un subconjunto del dataset(mini-batch), y es un algoritmo iterativo, por lo que aunque alcance soluciones muy buenas no siempre obtiene la mejor.

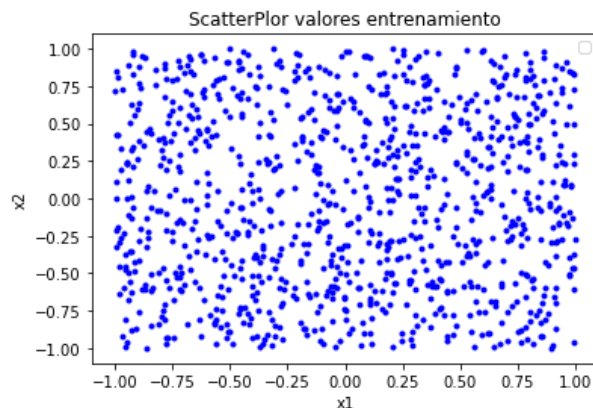
Pero aunque la Pseudo-inversa tenga una formula cerrada, esto no quiere decir que sea un cálculo rápido, cuando la cantidad de datos es muy grande la Pseudo-inversa es muy costosa de calcular, es más rápido, aplicar SGD, así como también hay casos en los que no se puede aplicar la Pseudo-inversa ya que como hemos visto antes en la fórmula hay que realizar la inversa de una matriz y una matriz sólo tiene inversa cuando el rango de la matriz coincide con el orden de la misma, por lo que el rango de  $X^T X$  tiene que ser igual a su orden.

	Pseudo-inversa	SGD
--	----------------	-----

$E_{in}$	0.07918658628900395	0.07962249498239364
$E_{out}$	0.1309538372005258	0.13291300951599333
		

## 2.5-. Experimento sobre características lineales

A Continuación creo un dataset de 1000 instancias con las características  $x_1$  y  $x_2$ , las cuales toman valores entre -1 y 1.



Para obtener las etiquetas del dataset anterior aplicamos la función:

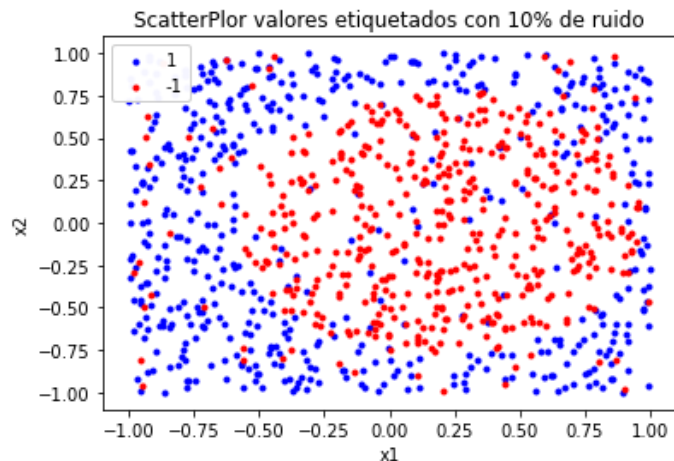
$$f(x_1, x_2) = \text{sign}((x_1 - 0.2)^2 + (x_2)^2 - 0.6)$$

$\text{Sign}(x) = \text{Devuelve el signo de } x$

Y aplicamos un 10% de ruido a la muestra, para lo cual generé los índices de las etiquetas, los barajo y calculo el número de etiquetas a modificar, e itero cambiandole el signo al número correspondiente de etiquetas

```
def getNoisyLabels(y, porcentaje=10):
    noisy_y = np.copy(y)
    indices = np.arange(y.shape[0])
    np.random.shuffle(indices)
    a_modificar = (porcentaje*y.shape[0])//100
    for i in range(a_modificar):
```

```
noisy_y[indices[i]] = y[indices[i]]*-1  
return noisy_y
```



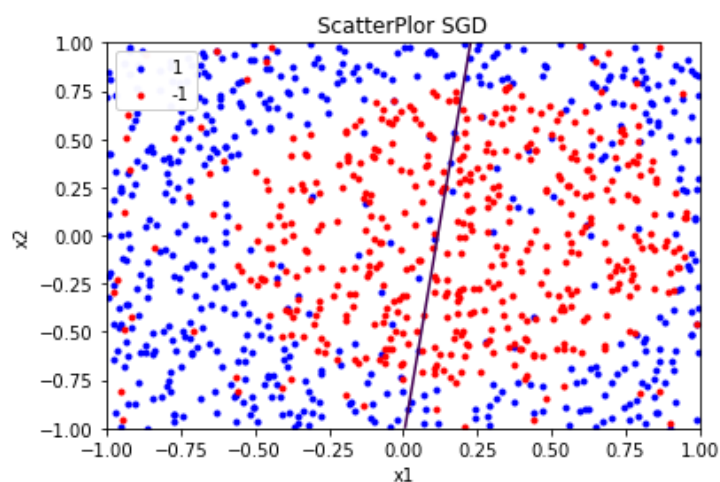
Una vez tenemos el dataset y sus respectivas etiquetas, añadimos al principio una columna de unos a las características, y aplicamos el algoritmo del gradiente descendente estocástico sobre el conjunto de datos anterior, con:

- Learning rate = 0.01
- Número de iteraciones = 50
- Tamaño de los batches = 32

Obtenemos:

```
Bondad del resultado para grad. descendente estocastico:  
Ein: 0.9285502494412031  
W: [ 0.05345664 -0.45687058 0.05053331]
```

Gráficamente la función ajustada queda tal que así:



### 2.5.1.-Cálculo de errores medios sobre distintos dataset

Vamos a calcular el error medio en la muestra y en la población sobre 1000 datasets distintos, para esto:

1. Generamos un conjunto de datos de entrenamiento, 1000 instancias, tanto las características como las etiquetas con un 10% de ruido.
2. Aplicamos el algoritmo de gradiente descendente estocástico
3. Calculamos el error que produce nuestro modelo en la muestra
4. Generamos otro conjunto de datos de test, 1000 instancias, tanto las características como las etiquetas con un 10% de ruido
5. Calculamos el error sobre la población (El dataset antes generado)

Tras hacer esto 1000 veces y haber calculado la media del error en la muestra y en la población obtenemos lo siguiente:

```
Errores medios tras 1000 iteraciones:  
Ein medio: 0.9273473475274606  
Eout medio: 0.932523048891284
```

Como podemos observar, por muchos dataset que le pasamos el modelo no es capaz de reducir el error significativamente, esto es debido a que estamos intentando ajustar un modelo lineal a un dataset no lineal, es decir las distintas clases no son linealmente separables. Por lo que no vamos a conseguir soluciones con una mejora significativa mientras no aumentemos la complejidad del modelo, como hacemos en el apartado anterior

## 2.6.-Experimento sobre características no lineales

Una vez hayamos vuelto a crear el conjunto de datos con sus respectivas etiquetas con un 10% de ruido vamos a pasar a ajustar un modelo de características no lineales.

A continuación vamos a crear la no linealidad, cada instancia del dataset va a quedar de la siguiente manera:

$$x = 1, x_1, x_2, x_1 x_2, x_1^2, x_2^2 \quad x \in X$$

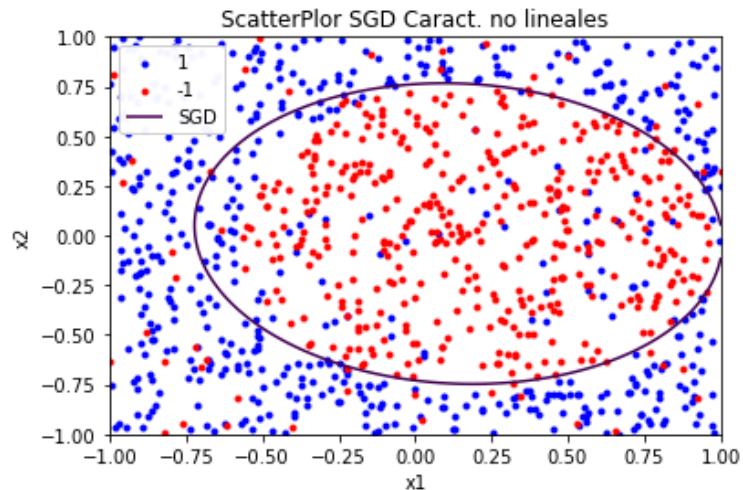
Esto lo he hecho añadiendo columnas al dataset original,  $[x_1, x_2]$  con la función

```
dataSetModificado = np.append(X, Y, axis=1)
```

Si aplicamos el gradiente descendente estocástico sobre las características anteriores, con un learning rate de 0.01 y mini-batches de 32 elementos obtenemos:

```
Bondad del resultado para grad. descendente estocastico sobre  
caracteristicas no lineales:  
Ein: 0.6285138775983187  
W: [-1.03588158 -0.40706804 -0.06086786 0.17566938 1.43249129 1.8634715 ]
```

Donde W representa la siguiente función:



### 2.6.1.-Cálculo de errores medios sobre distintos dataset

Si ahora repetimos lo anterior 1000 veces como lo hemos hecho para la combinación lineal obtenemos :

```
Errores medios tras 1000 iteraciones:  
Ein medio: 0.5806660944672062  
Eout medio: 0.5860404464192567
```

Como podemos observar hemos reducido el error que obtenemos al ver un solo dataset, y mejorado, drásticamente respecto al conjunto de características lineales, por lo que cuando las distintas clases no son linealmente separables, obtendremos una mejor solución si utilizamos un vector de características no lineal, que uno lineal.

## 3.-Método de newton

### 3.1.-Implementación del método de newton

A Continuación voy a explicar como he implementado el método de Newton para la búsqueda iterativa de óptimos.

Para aplicar el método de Newton sobre  $f(x,y)$  necesitamos las segundas derivadas parciales, las cuales son las siguientes:

$$\frac{d^2F(x,y)}{dx^2} = 2 - 8\pi^2 \sin(2\pi x) \sin(2\pi y)$$

$$\frac{d^2F(x,y)}{dx dy} = 8\pi^2 \cos(2\pi x) \cos(2\pi y)$$

$$\frac{d^2F(x,y)}{dy dx} = 8\pi^2 \cos(2\pi x) \cos(2\pi y)$$

$$\frac{d^2F(x,y)}{dy^2} = 4 - 8\pi^2 \sin(2\pi x) \sin(2\pi y)$$

Una vez tenemos las derivadas parciales, creamos la matriz hessiana:

$$H = \begin{bmatrix} \frac{d^2F(x,y)}{dx^2} & \frac{d^2F(x,y)}{dxdy} \\ \frac{d^2F(x,y)}{dydx} & \frac{d^2F(x,y)}{dy^2} \end{bmatrix}$$

```
def Hessian(x,y):  
    hessiana = np.ndarray(shape=(2,2),dtype="float64")  
    hessiana[0,0] = dFxx(x,y)  
    hessiana[0,1] = dFxy(x,y)  
    hessiana[1,0] = dFyx(x,y)  
    hessiana[1,1] = dFyy(x,y)  
    return hessiana
```

El incremento a W sería el siguiente:

$$w_j = w_j - learningRate * H(w)^{-1} \frac{dE_{in}(w)}{dw_j}$$

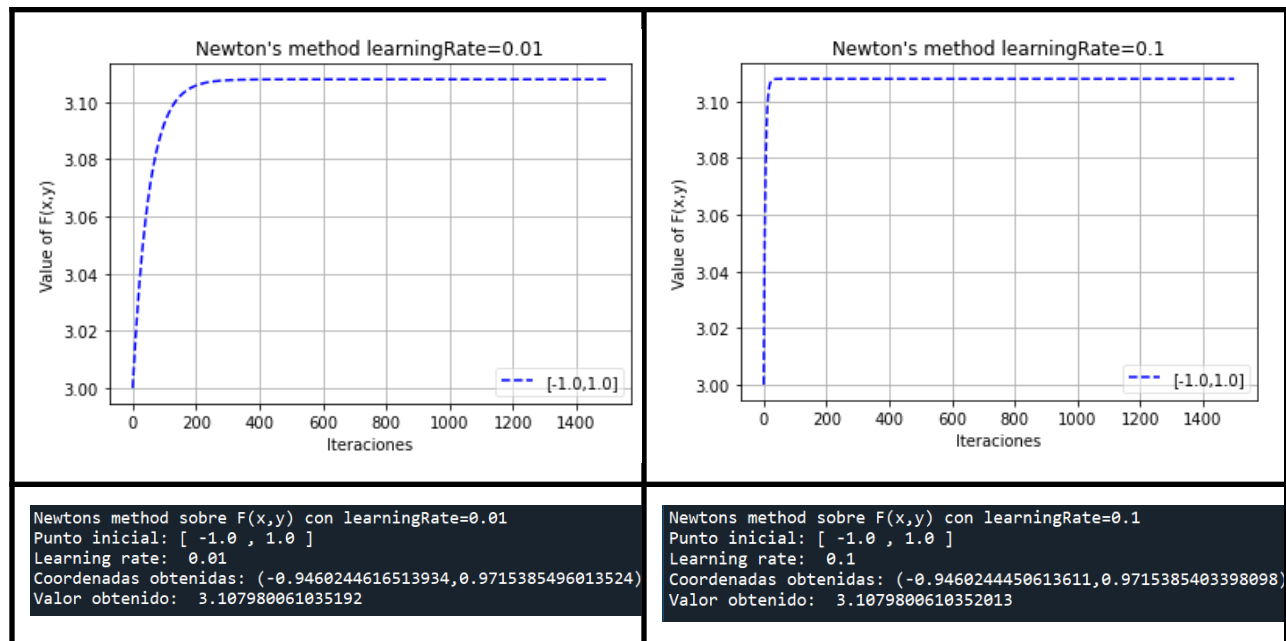
```
def NewtonsMethod(w, numMaxIter, learning_rate):  
    iterations = 0  
    valoresF = np.ndarray((numMaxIter+1))  
    valoresF[0] = F(w[0],w[1])  
    while(iterations<numMaxIter):  
        a = np.linalg.inv(Hessian(w[0],w[1]))  
        b = gradF(w[0],w[1])  
        incrementoW = -learning_rate*np.matmul(a,b)  
        w = w + incrementoW  
        iterations+=1  
        valoresF[iterations] = F(w[0],w[1])  
    return w, valoresF
```

### 3.1.1-. Método de newton con distintos learning rate

Partiendo del punto inicial [-1.0,1.0] aplicamos el método de newton y obtenemos los siguientes resultados, para distintos learning rates

Learning rate = 0.01	Learning rate = 0.1
----------------------	---------------------





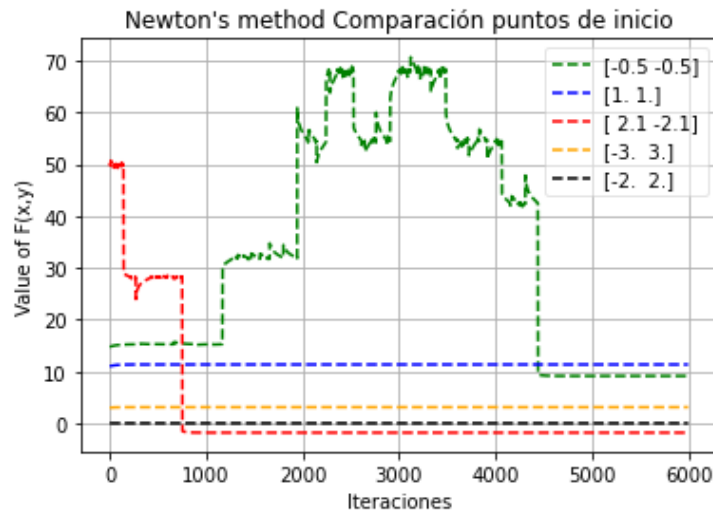
Podemos observar que en vez de hallar un mínimo, se ha ido a un máximo local, esto es por que el método de newton lo que hace es encontrar un  $f'(w)=0$ , sin importar si es un mínimo o un máximo, es por esto que se suele aplicar una vez que con gradiente descendente estamos muy cerca del mínimo global y el método de newton nos da el mínimo global exacto.

Como vemos en este caso si aumentamos el learning rate lo único que hace es llegar al óptimo local más rápidamente, pero no ocurre como con el Gradiente descendente que exploraba todo el espacio de búsqueda sin control.

### 3.1.2-. Método de newton con distintos puntos de inicio

Aplicando el método de newton sobre de  $f(x,y)$ , con distintos puntos iniciales, he obtenido los siguientes resultados con un learning rate = 0.01

Punto inicial(x,y)	Valor minimo	Donde Valor min(x,y)
[-0.5 -0.5]	9.125146662901864	x=-0.7934994549961764 y= -0.12596577566955575
[1. 1.]	11.344755748360594	x= 1.0673726386671512 y= 0.9100709200235662
[ 2.1 -2.1]	-1.8200785415471565	x= -2.2438049693647666 y= 2.2379258214861557
[-3. 3.]	3.1079800610352026	x= -3.053975554938618 y= 3.0284614596601687
[-2. 2.]	-4.799231304517944e-31	x= -2.0 y= 2.0



El método de newton se suele usar para obtener las raíces de una función de forma iterativa, por lo que si aplicamos el método de newton sobre la derivada de una función obtendremos los óptimos, obtendremos un óptimo bajo las siguientes condiciones:

- Dado un cierto intervalo de trabajo  $[a,b]$ , dentro del mismo debe cumplirse que  $f'(a) \cdot f'(b) < 0$ , es decir que en el intervalo de trabajo haya un punto donde la derivada pasa por 0
- Dentro del intervalo de trabajo  $[a,b]$ ,  $f''(x)$  debe ser diferente de cero, es decir no puede haber un punto de inflexión dentro del intervalo en  $f'(x)$
- La función  $f'(x)$  dentro del intervalo de trabajo  $[a,b]$ , debe ser cóncava, hacia arriba o hacia abajo
- Se debe asegurar que la tangente a  $f'(x)$  en el extremo del intervalo  $[a,b]$ , intersecta al eje x dentro del intervalo  $[a,b]$ . Esto quiere decir que si en algún momento  $f'(x)$  tiene una pendiente casi nula, su tangente en ese punto se podría ir fuera de intervalo.

(Condiciones del metodo de newton <https://estadistica-dma.ulpgc.es/FCC/05-3-Raices-de-Ecuaciones-2.html> )

Como reflexión entre la diferencia del gradiente descendente y el método de newton podríamos decir que el método newton es una forma eficaz de encontrar el óptimo más cercano a un punto, ya que vemos como converge de forma muy rápido, el problema es que encuentra óptimos, no únicamente mínimos por lo que si estamos en un punto y hay un máximo más cerca que un mínimo, este se dirigirá al máximo, como nos ha pasado con el punto de inicio  $[-1,1]$ .

El gradiente descendente, es más progresivo, va buscando únicamente mínimos, y no tiene unas condiciones tan estrictas, ya que no hay que cumplir ninguna de las condiciones anteriores, únicamente debe ser una función derivable para que podamos aplicar el descenso de gradiente y obtener un mínimo, local o global, pero siempre lo obtendremos.

Por lo que como he dicho antes, el método de newton se suele aplicar, una vez hemos aplicado el gradiente descendente y tenemos la certeza de que estamos cerca de un buen minimo, entonces aplicamos el método de newton y este nos da el punto exacto en el que se encuentra el mínimo que buscamos.