Alejandro Borrego Megías DGIIM

Práctica 1 Aprendizaje Automático

Ejercicio 1

Búsqueda Iterativa de óptimos

1. Implementar el algoritmo de gradiente descendente

El algoritmo implementado es el siguiente, que puede verse en el fichero template_trabajo1.py

```
def gradient_descent(w,eta,num_iterations, error):
    #
    # gradiente descendente
    #
    iterations=0
    Err=1000.0

    while Err>error and iterations<num_iterations:
       partial_derivative=gradE(w[0],w[1])
       w=w - eta*partial_derivative
       iterations=iterations + 1
       Err=E(w[0],w[1])

    return w, iterations
```

Este algoritmo está hecho específicamente para la función E(u,v) del apartado siguiente pero se puede generalizar fácilmente para cualquier otra función. Los valores que se pasan como argumento son w (vector que contendrá las coordenadas del mínimo de la función, inicializado a [1 1 ... 1]), eta (la tasa de aprendizaje), num_iterations (contiene las iteraciones máximas que hará el algoritmo) y error (contiene el error a alcanzar).

La idea del algoritmo es, en primer lugar inicializamos las iteraciones a 0 y establecemos un error base de 1000 para ir mejorándolo en el bucle while.

Una vez hecho esto entramos en el bucle principal del algoritmo, dicho bucle parará si se verifica alguna de las dos condiciones: O bien porque llegamos al número de iteraciones máximas (el valor num_iterations) o bien porque el error cometido (Err) es menor que el establecido por la variable "error" que pasamos como argumento a la función.

En cada iteración el algoritmo calcula el gradiente de la función a minimizar, lo evalúa en el vector w y esto nos daría el vector que nos apunta hacia el "máximo" de la función. Pues tras esto actualizamos el punto w desplazándolo en la dirección opuesta al gradiente (en dirección a un mínimo local o global) con un paso del tamaño de eta.

Tras esto evaluamos la función en el nuevo punto, lo que nos dará el error del nuevo punto obtenido (si todo va bien, debería ser menor que el error de la iteración anterior, aunque luego veremos que esto no siempre es así), y si este error es mayor que el que queríamos obtener y aún quedan iteraciones por hacer se repite el mismo procedimiento.

Como podemos ver, el correcto funcionamiento del algoritmo depende entre otras cosas del eta establecido y del punto inicial establecido, hechos que veremos mejor en los ejercicios siguientes.

2. Considerar la función E(u, v) = $\left(e^{v-2}u^3 - 2v^2e^{-u}\right)^2$**. Usar gradiente descendente para encontrar un mínimo de esta función, comenzando desde el punto** $(u, v) = (1, 1)$ **y usando una tasa de aprendizaje** $\eta = 0.1$

a) Calcular analíticamente y mostrar la expresión del gradiente de la función E(u, v)

1. La función E(u,v)=$\left(e^{v-2}u^3 - 2v^2e^{-u}\right)^2$

```
def E(u,v):
    return  (u**3*np.e**(v-2)-2*v**2*np.e**(-u))**2
```

2. Las derivadas parciales con respecto a u y v, que serían $\frac{\partial}{\partial u}E(u, v) = 2\left(e^{v-2}u^3 - 2v^2e^{-u}\right)\left(2v^2e^{-u} + 3e^{v-2}u^2\right)$ y $\frac{\partial}{\partial v}E(u, v) = 2\left(u^3e^{v-2} - 4e^{-u}v\right)\left(u^3e^{v-2} - 2e^{-u}v^2\right)$

```
#Derivada parcial de E con respecto a u
def dEu(u,v):
    return 2*(np.e**(v-2)*u**3-2*v**2*np.e**(-u))*(2*v**2*np.e**(-u)+3*np.e**(v-2)*u**2)

#Derivada parcial de E con respecto a v
def dEv(u,v):
    return 2*(u**3*np.e**(v-2)-4*np.e**(-u)*v)*(u**3*np.e**(v-2)-2*np.e**(-u)*v**2)
```

3. Finalmente la funcion gradiente de E, que nos calcula el vector $(\frac{\partial}{\partial u}E(u, v), \frac{\partial}{\partial v}E(u, v))$

```
#Gradiente de E
def gradE(u,v):
    return np.array([dEu(u,v), dEv(u,v)])
```

b)¿Cuántas iteraciones tarda el algoritmo en obtener por primera vez un valor de E(u,v) inferior a 10^(-14) ?.

c)¿En qué coordenadas (u,v) se alcanzó por primera vez un valor igual o menor a 10^-14 en el apartado anterior?

Una vez tenemos todo esto, como el enunciado nos especifica los valores $\eta = 0.1$ y punto inicial $(u, v) = (1, 1)$ tenemos ya todos los ingredientes necesarios para ejecutar el algoritmo de gradiente descendente, solo nos quedaría especificar el

error a conseguir y las iteraciones máximas, yo las he establecido en los valores que venían en el template:

```
eta = 0.1
maxIter = 10000000000
error2get = 1e-14 #Error a alcanzar
initial_point = np.array([1.0,1.0])
w, it = gradient_descent(initial_point, eta,maxIter, error2get)
```

Con estos valores los resultados obtenidos por el algoritmo de gradiente descendente serían:

```
Funcion a minimizar: E(u,v)=(u^3*e^(v-2)-2*v^2*e^(-u))^2
Gradiente: [2*(e^(v-2)*u^3-2*v^2*e^(-u))*(2*v^2*e^(-u)+3*e^(v-2)*u^2), 2*(u^3*e^(v-2)-4*e^(-
Numero de iteraciones:  10
Coordenadas obtenidas: ( 1.1572888496465497 ,  0.9108383657484797 )
```

Es decir, el algoritmo ha tardado 10 iteraciones en llegar a un valor de la función inferior al establecido y el primer punto donde se alcanza un valor de la función menor o igual al error establecido serían las coordenadas obtenidas.

Finalmente representamos los resultados obtenidos en un gráfico usando el código de ejemplo que venía en el template:

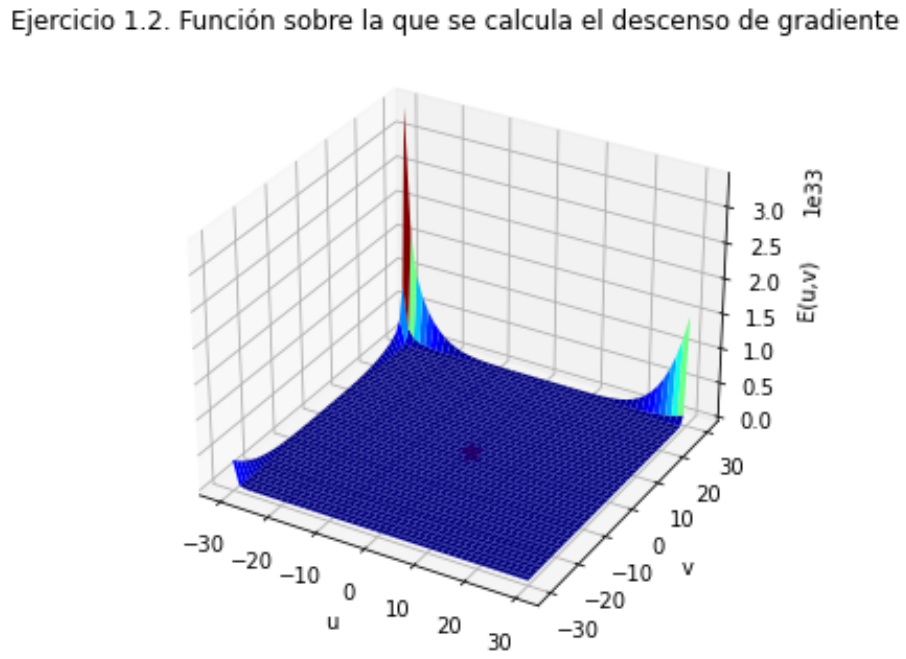

Figure 1: Ejercicio 1.2

Dicho código es:

```
x = np.linspace(-30, 30, 50)
y = np.linspace(-30, 30, 50)
X, Y = np.meshgrid(x, y)
Z = E(X, Y) #E_w([X, Y])
fig = plt.figure()
ax = Axes3D(fig)
surf = ax.plot_surface(X, Y, Z, edgecolor='none', rstride=1,
                       cstride=1, cmap='jet')
min_point = np.array([w[0],w[1]])
min_point_ = min_point[:, np.newaxis]
ax.plot(min_point_[0], min_point_[1], E(min_point_[0], min_point_[1]), 'r*', markersize=10)
ax.set(title='Ejercicio 1.2. Función sobre la que se calcula el descenso de gradiente')
ax.set_xlabel('u')
ax.set_ylabel('v')
ax.set_zlabel('E(u,v)')
plt.show()
```

En resumen, generamos un vector x e y de 50 puntos equidistantes entre -30 y 30, los juntamos en una rejilla usando la función meshgrid, evaluamos los valores de la rejilla en la función E(u,v) y la representamos en 3 Dimensiones usando la función plot_surface.

Finalmente representamos el mínimo en el mismo gráfico usando la función plot, donde le pasamos la coordenada u, v, el valor de la función en ese punto y queremos que se represente como una estrella roja, por eso usamos 'r*' y de tamaño 10.

3. Considerar ahora la función $f(x,y) = 2\sin(2\pi x)\sin(2\pi y) + 2(y-2)^2 + (x+2)^2$

Igual que antes calculamos las derivadas parciales y el gradiente, e implementamos el gradiente descendente de acuerdo a la nueva función (es idéntico al del apartado anterior salvo que la función y el gradiente cambian y que no se especifica error)

1. La función $f(x,y) = 2\sin(2\pi y)\sin(2\pi x) + (x+2)^2 + 2(y-2)^2$

```
def f(x,y):
    return  (x+2)**2 + 2*(y-2)**2 + 2*np.sin(2*np.pi*x)*np.sin(2*np.pi*y)
```

2. Las derivadas parciales con respecto a x e y, que serían $\frac{\partial}{\partial x}f(x,y) = 4\pi\sin(2\pi y)\cos(2\pi x) + 2(x+2)$ y $\frac{\partial}{\partial y}f(x,y) = 4\pi\sin(2\pi x)\cos(2\pi y) + 4(y-2)$

```
#Derivada parcial de f con respecto a x
def dfx(x,y):
    return 4*np.pi*np.sin(2*np.pi*y)*np.cos(2*np.pi*x)+2*(x+2)
```

```
#Derivada parcial de f con respecto a y
def dfy(x,y):
    return 4*np.pi*np.sin(2*np.pi*x)*np.cos(2*np.pi*y)+4*(y-2)
```

3. La funcion gradiente de $f(x, y)$, que nos calcula el vector $(\frac{\partial}{\partial x}f(x, y), \frac{\partial}{\partial y}f(x, y))$

```
def gradf(x,y):
    return np.array([dfx(x,y), dfy(x,y)])
```

4. Finalmente el algoritmo del gradiente descendente.

```
def gradient_descent2(w,eta,num_iterations):
    #
    # gradiente descendente
    #
    iterations=0
    vector_puntos=np.array([[w[0],w[1]]])
    while iterations<num_iterations:
       h_x=f(w[0],w[1])
       partial_derivative=gradf(w[0],w[1])
       w=w -(eta*np.transpose(partial_derivative))
       iterations=iterations + 1
       Err=f(w[0],w[1])
       vector_puntos=np.append(vector_puntos, [[w[0],w[1]]], axis=0)


    return w, iterations,vector_puntos
```

como comentario al gradiente descendente, en este caso, también devuelvo el vector de puntos que he ido generando en cada iteración para poder hacer las gráficas del apartado siguiente.

a) Usar gradiente descendente para minimizar esta función. Usar como punto inicial $(x_0 = -1, y_0 = 1)$**, tasa de aprendizaje** $eta = 0.01$ **y un máximo de 50 iteraciones. Repetir el experimento pero usando** $\eta = 0.1$**, comentar las diferencias y su dependencia de** η**.**

Tras las 50 iteraciones los resultados obtenidos para $\eta = 0.01$ y el punto inicial $(x_0 = -1, y_0 = 1)$ son:

```
Numero de iteraciones:  50
Coordenadas obtenidas con eta=0.01 : ( -1.269064351751895 ,  1.2867208738332965 )
valor obtenido:  -0.3812494974381
```

y usando $\eta = 0.1$ los valores obtenidos por el algoritmo son:

```
Funcion a minimizar: f(x,y)=(x+2)^2 + 2*(y-2)^2 + 2*sin(2*pi*x)*sin(2*pi*y)
Gradiente: [4*pi*sin(2*pi*y)*cos(2*pi*x)+2*(x+2), 4*pi*sin(2*pi*x)*cos(2*pi*y)+4*(y-2)]
Numero de iteraciones:  50
Coordenadas obtenidas con eta=0.1: ( -2.8537959548927576 ,  1.9803903507510756 )
```

```
valor obtenido:  0.5343830643374345
```

Aparentemente, podemos pensar que el learning rate no ha influido mucho, pues en realidad, la distancia al 0 obtenida por los dos algoritmos ha sido muy similar, en cambio, si generamos un gráfico que nos muestre como descendía el valor de la función en cada iteración del gradiente descendente vemos lo siguiente:

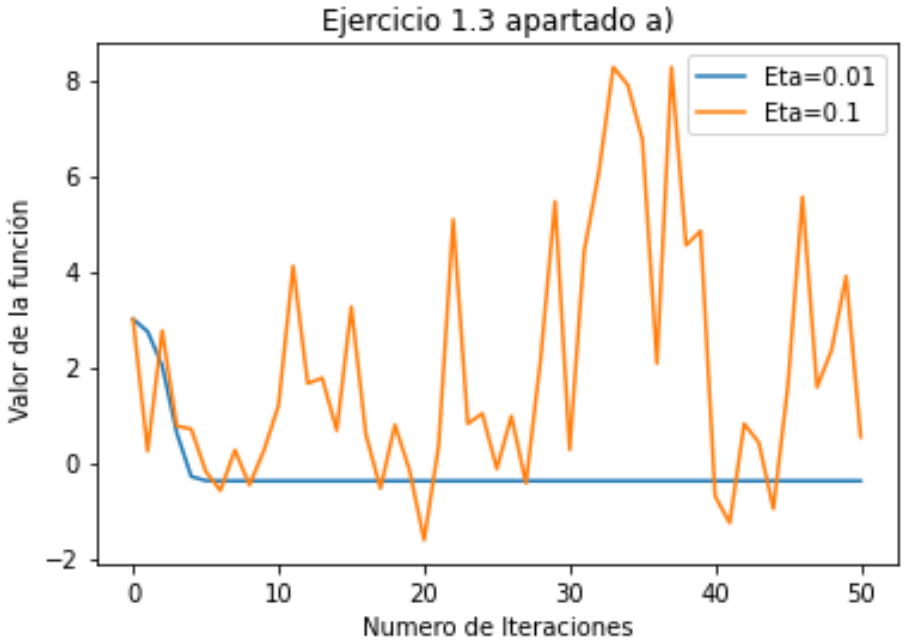

Figure 2: Ejercicio 1.3 apartado a)

Y ahora si que podemos ver una gran diferencia entre la elección de un η u otro, y es que si el eta es demasiado grande (como ocurre con el 0.1) puede ser que en cada iteración el paso que demos sea excesivamente grande y se puede dar el caso de que "saltamos" por encima del mínimo, y en cierto modo podemos quedarnos oscilando en torno al mínimo de la función. Por eso observamos en la gráfica valores tan dispares para la función $f(x, y)$ en cada iteración.

En cambio, con el valor 0.01, al ser los pasos en cada iteracion menores, se asegura una mejor convergencia al mínimo, y una vez alcanzado dicho mínimo, la función permanece constante (pues el gradiente es prácticamente 0 y en cada iteración los valores de los pesos no se modifican a penas).

b) Obtener el valor mínimo y los valores de las variables (x, y) en donde se alcanzan cuando el punto de inicio se fija en: $(-0.5, -0.5), (1, 1), (2.1, -2.1), (-3, 3), (-2, 2)$. Generar una tabla con los valores obtenidos. Comentar la dependencia del punto inicial.

En este caso los valores obtenidos son estos:

```
Puntos iniciales (x,y)= ( -0.5 , -0.5 )
Numero de iteraciones:  50
Coordenadas obtenidas: ( -0.7934994705090673 ,  -0.12596575869895063 )
valor obtenido:  9.125146662901855

--------------------------

Puntos iniciales (x,y)= ( 1 , 1 )
Numero de iteraciones:  50
Coordenadas obtenidas: ( 0.6774387808772109 ,  1.290469126542778 )
valor obtenido:  6.4375695988659185

--------------------------

Puntos iniciales (x,y)= ( 2.1 , -2.1 )
Numero de iteraciones:  50
Coordenadas obtenidas: ( 0.14880582855887767 ,  -0.09606770499224294 )
valor obtenido:  12.490971442685037

--------------------------

Puntos iniciales (x,y)= ( -3 , 3 )
Numero de iteraciones:  50
Coordenadas obtenidas: ( -2.7309356482481055 ,  2.7132791261667037 )
valor obtenido:  -0.38124949743809955

--------------------------

Puntos iniciales (x,y)= ( -2 , 2 )
Numero de iteraciones:  50
Coordenadas obtenidas: ( -2.0 ,  2.0 )
valor obtenido:  -4.799231304517944e-31
```

Y para entender un poco mejor lo que ocurre he realizado el siguiente gráfico:

Como podemos observar la elección del punto inicial es clave para encontrar un mínimo local u otro. Por ejemplo empezando en el $(1, 1)$ el algoritmo se queda en el punto $(0.6774387808772109, 1.290469126542778)$, que es un mínimo local de la función, pero no un mínimo global, y lo mismo ocurre con los demás puntos, y es que si representamos la función en 3 dimensiones:

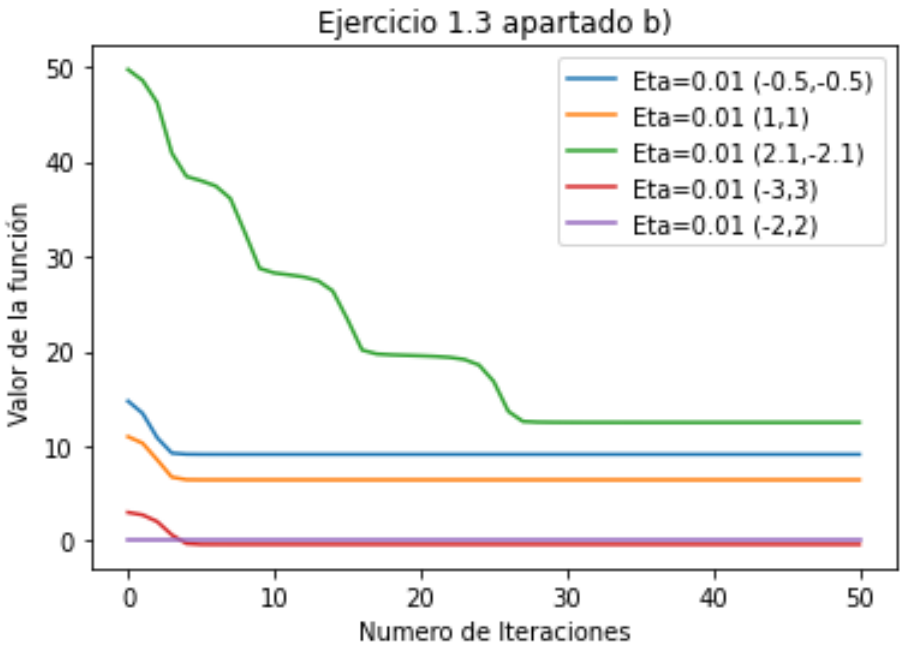

Figure 3: Ejercicio 1.3 b)

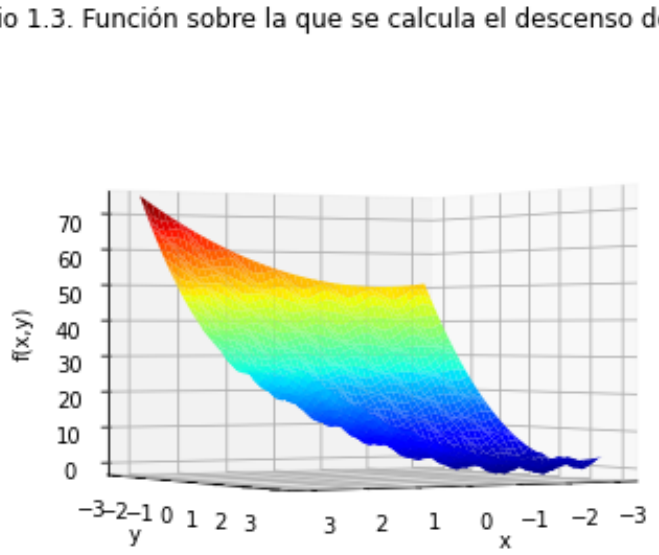

Figure 4: Ejercicio1.3 b) 2

Podemos ver que no es una superficie lisa, sino que tiene como "hoyos" donde nuestro algoritmo puede quedarse "atrapado" dependiendo del valor que tomemos como punto inicial. Cabe destacar el caso del punto inicial (2.1, -2.1) pues en este caso, el algoritmo como podemos observar va sorteando distintos hoyos donde parece que va a quedarse atrapado, hasta llegar a uno donde se queda atrapado y ya permanece constante.

Como comentario sobre la forma en que se han creado las distintas gráficas, para las gráficas que son en 2 dimensiones con las iteraciones y el valor de la función he usado este código:

```
imagenes=[]
for i in vector_puntos:
    imagenes.append(f(i[0],i[1]))

iteraciones=np.arange(it+1)
plt.plot(iteraciones, imagenes, label='etiqueta')
```

Donde he ido generando para cada elemento del vector de puntos devuelto por el gradiente descendente su correspondiente imagen y luego con la función plot he representado en un gráfico las iteraciones y las imágenes.

Finalmente añadiendo plt.show() mostraba en el mismo gráfico todas las funciones

juntas.

por otro lado, para la última gráfica he usado el mismo código del ejercicio 1.2.

4.¿Cuál sería su conclusión sobre la verdadera dificultad de encontrar el mínimo global de una función arbitraria?

Por todo lo visto en apartados anteriores, la dificultad para encontrar el mínimo global de una función arbitraria, a mi modo de entender, reside en la correcta elección del punto inicial donde lanzar el algoritmo y de el η adecuado, pues como vimos en el ejercicio 1.2 un η demasiado grande puede hacer que nuestro algoritmo no converga al mínimo, y como vimos en el ejercicio 1.3, con un η adecuado, si no tomamos un buen punto de partida, nuestro algoritmo puede no converger al mínimo global y quedar atrapado en mínimos locales. Por lo tanto a mi parecer, dada una función arbitraria $f(x, y)$ sería una buena práctica, tomar un conjunto arbitrario de puntos iniciales y un conjunto arbitrario de posibles valores de η y llevar a cabo un estudio de con qué punto inicial y con qué valor de η se alcanza el "mejor" mínimo de la función.

Ejercicio 2

Regresión lineal

Este ejercicio ajusta modelos de regresión a vectores de características extraidos de imágenes de digitos manuscritos. En particular se extraen dos características concretas que miden: el valor medio del nivel de gris y la simetría del número respecto de su eje vertical. Solo se seleccionarán para este ejercicio las imágenes de los números 1 y 5.

Ejercicio 2.1

Estimar un modelo de regresión lineal a partir de los datos proporcionados por los vectores de características (Intensidad promedio, Simetría) usando tanto el algoritmo de la pseudo-inversa como el Gradiente Descendente Estocático (SGD). Las etiquetasa serán (-1,1), una por cada vector de cada uno de los números. Pintar las soluciones obtenidas junto con los datos usados en el ajuste. Valorar predicciones usando Ein y Eout (para E out calcular las predicciones usando los datos del fichero de test)