Tarea Módulo Optimización

Cristian Pinoleo - Paz Esparza - Nayadeth Aguirre - Andrés Lagos

Pregunta 1

1. Implemente el algoritmo del descenso del gradiente (GD).

   Antes de poder implementar el algoritmo es necesario programar la función asignada y su gradiente, estas se almacenan en las funciones "*funcion*" y "*gradiente*" respectivamente.
   Luego, el algoritmo se programa en la función *"gradientdescendant",* la cual tiene como parámetros de entrada:

   - **w:** vector inicial, de tamaño 6.
   - **alpha:** tamaño del paso, en este caso 0.01 .
   - **tol:** tolerancia mínima para decir que un punto es optimal.
   - **imax:** cantidad máxima de iteraciones.
   - **n:** cantidad de variables.
   - **a:** parámetro dado por la tarea, usado para calcular el gradiente.
   - **f11:** décimo primer término de la función f, que se utiliza para calcular el gradiente.

   Algoritmo:

```
def gradientdescendant(w,imax,tol,alpha,n,a,f11):
    i=0;
    error=np.linalg.norm(gradiente(w,n,a,f11))
    if(error<tol):
        print("El valor ingresado es el minimo")
        return w
    else:
        while (i<imax) and (error>tol):
            w = w - alpha*gradiente(w,n,a,f11)
            i = i+1
            error=np.linalg.norm(gradiente(w,n,a,f11))
        print('El punto final es', w)
        print('En la iteración: ', i)
        print('con error: ', error)
        return w
```

   El algoritmo seguirá buscando el punto óptimo hasta que se alcance la iteración máxima o el error sea menor a la tolerancia, donde el error es la norma del gradiente evaluado en el punto.

2. Utilice su implementación del GD para encontrar un punto optimal de la primera función asignada a su grupo.

   Para esto llamamos la función anterior, con los parámetros imax=1000, tol=0.00001, alpha=0.01 w0=(0,0,0,0,0,0), se obtiene el siguiente resultado:

```
El punto final es [1.99995874e-01 3.39290947e-06 8.55647127e-06 1.40475335e-05
 2.01161078e-05 1.11437705e-05]
En la iteración:  533
con error:  9.975755825377246e-06
```

   Se encuentra el óptimo con 533 iteraciones.

3. Compare los resultados utilizando distintos puntos iniciales, y estime cuántas iteraciones necesita el algoritmo para mejorar su resultado en un factor de 0.1.

Se resumen los resultados para 4 puntos diferentes, donde se encontraron 4 mínimos distintos en *"It"* iteraciones. Además la cantidad de iteraciones necesarias para alcanzar una mejora de un factor de 0.1 se encuentra en la columna 4.

W Inicial	It.	Punto Final	It. para mejorar 0.1
(0,0,0,0,0,0)	533	(2e-01, 3.4e-06, 8.6e-06, 1.4e-05, 2e-05, 1.11e-05)	69
(1,1,1,1,1,1)	598	(0.2, 1, 1, 1, 1, 1)	138
(10,100,300,7,-20 0.00002)	664	(2e-01, -4.7e+09, -6.85e+18, -4.30e+05, -2e+02, 4.95e-05)	262
(-2,-0.1,3,2,-2,0)	650	(2e-01, -10e-02, 3, 2, -2, 1.6e-05)	188

Pregunta 2

1. Implemente el algoritmo de descenso del gradiente estocástico (SGD).
2. Utilice su implementación de GD y de SGD para encontrar un punto optimal de la segunda función asignada a su grupo.
3. Compare las soluciones obtenidas por cada algoritmo, y las diferencias en tiempo de ejecución de los mismos.

1. Se implementó la función SGD (Stochastic Gradient Descent), la cual ejecuta el algoritmo del descenso de gradiente estocástico para la función indicada por el inciso.
   Previo a esta implementación, se debió definir el gradiente de las funciones $f_i(w)$ para poder utilizarlo en el algoritmo, por tanto de definen la funciones "*f2*", "*gradf2*", "*gradF2*":
   f2:

```
def f2(w,k):
    t = 5/(k+1)
    return ((w[0] + t*w[1] - e**t)**2 + (w[2] + w[3]*np.sin(t) - np.cos(t))**2)
```

   La cual es la segunda función indicada para el grupo 3. Tiene como parámetros w y k, el primero indica el vector de ω, y el k indica el índice de la función $f_k(\omega)$, que también permite calcular t_k.

   gradf2:

```
def gradf2(w,k): #cuadrado
    t = 5/(k+1)
    f_1 = 4*f2(w,k)*(w[0]+t*w[1]-e**t)
    f_2 = 4*t*f2(w,k)*(w[0]+t*w[1]-e**t)
    f_3 = 4*f2(w,k)*(w[2]+w[3]*np.sin(t)-np.cos(t))
    f_4 = 4*np.sin(t)*f2(w,k)*(w[2]+w[3]*np.sin(t)-np.cos(t))
    return np.array([f_1,f_2,f_3,f_4])
```

   La cual es el gradiente del cuadrado de la función previa. Tiene como parámetros w y k, al igual que en la función anterior, w es el vector ω y k el índice de $f_k(\omega)$.

   gradF2:

```
def gradF2(w,m): #cuadrado
    F_1 = sum(4*f2(w,k)*(w[0]+(5/(k+1))*w[1]/5+np.exp(5/(k+1))) for k in range(1,m+1))
    F_2 = sum(4*(5/(k+1))*f2(w,k)*(w[0]+k*w[1]/5+np.exp(5/(k+1))) for k in range(1,m+1))
    F_3 = sum(4*f2(w,k)*(w[2]+w[3]*np.sin(5/(k+1))-np.cos(5/(k+1))) for k in range(1,m+1))
    F_4 = sum(4*np.sin(5/(k+1))*f2(w,k)*(w[2]+w[3]*np.sin(5/(k+1))-np.cos(5/(k+1))) for k in range(1,m+1))
    return np.array([F_1,F_2,F_3,F_4])
```

   La cual es el gradiente de la suma de los cuadrados de la primera función. Tiene como parámetros w y m, al igual que en la función anterior, w es el vector ω y aunque m es la m entregada en el inciso, el cual se deja modificable de manera arbitraria por problemas de numéricos.

Cabe mencionar que para ambas funciones anteriores, el valor de t_k fue modificado de $t_k = k/5$ a $t_k = 5/(k+1)$, se trabaja con $(k+1)$, con el fin de evitar problemas de división por cero.
Sin embargo, con estas funciones surgieron errores de número muy grandes, por tanto se decidió como grupo omitir el cuadrado de $(f_i(\omega))^2$, tanto para el descenso de gradiente simple como el estocástico.
Así, las nuevas funciones quedaron:

```
def gradF22(w,m): #sin cuadrado
    F_1 = sum(2*(w[0]+(5/(k+1))*w[1]/5+np.exp((5/(k+1)))) for k in range(0,m))
    F_2 = sum(2*(5/(k+1))*(w[0]+(5/(k+1))*w[1]/5+np.exp((5/(k+1)))) for k in range(0,m))
    F_3 = sum(2*(w[2]+w[3]*np.sin((5/(k+1)))-np.cos((5/(k+1)))) for k in range(0,m))
    F_4 = sum(2*np.sin((5/(k+1)))*(w[2]+w[3]*np.sin((5/(k+1)))-np.cos((5/(k+1)))) for k in range(0,m))
    return np.array([F_1,F_2,F_3,F_4])
```

y

```
def gradf22(w,k): #sin cuadrado
    t = 5/(k+1)
    f_1 = 2*(w[0]+t*w[1]-e**t)
    f_2 = 2*t*(w[0]+t*w[1]-e**t)
    f_3 = 2*(w[2]+w[3]*np.sin(t)-np.cos(t))
    f_4 = 2*np.sin(t)*(w[2]+w[3]*np.sin(t)-np.cos(t))
    return np.array([f_1,f_2,f_3,f_4])
```

.

El algoritmo de descenso de gradiente, GD (gradient descent) se define como:

```
def GD(w,m,imax,tol,lr,f): #parametro f para especificar el gradiente usado
    i = 0
    err = np.linalg.norm(f(w,m))
    if err < tol:
        print("El punto ya es mínimo")
    else:
        while i < imax and tol < err:
            i = i+1
            w = w - lr*f(w,m)
            err = np.linalg.norm(f(w,m))
        print("Punto Final",w)
        print("Iteración",i)
        print("Error",err)
        return w
```

Esta función ejecuta el algoritmo de descenso de gradiente estocástico. Tiene los siguientes parámetros:

- w: el vector de valores a optimizar ω
- m: el valor de m (encontramos como grupo que m = 10000, era muy grande, por tanto lo dejamos libre a modificar en nuestras funciones).

- imax: el número máximo de iteraciones.
- tol: la tolerancia de error para el algoritmo.
- lr (learning rate): tamaño del paso.
- f: el parámetro que especifica el gradiente utilizado (ya sea el de cuadrados o sin cuadrados)

Este es el mismo algoritmo que en la pregunta anterior, salvo el parámetro del gradiente.
El algoritmo del descenso de gradiente estocástico, SGD (stochastic gradient descent), se define como:

```
def SGD(w,m,imax,tol,lr,f): #parametro f para especificar el gradiente usado
    i = 0
    err=np.inf
    while i < imax and  tol < err :

        k = np.random.randint(0,m-1)
        err = np.linalg.norm(f(w,k))
        w = w - lr*f(w,k)
        i = i+1
    print("Punto Final",w)
    print("Iteración",i)
    print("Error",err)
    return w
```

El cual tiene los mismos parámetros del algoritmo anterior.
Este algoritmo actúa de manera similar al definido en la primera pregunta en donde busca un punto óptimo hasta que se alcance el número máximo de iteraciones o se obtenga un error menor a la tolerancia, sin embargo este hace uso de los gradientes individuales de las $f_k(\omega)$, escogidos al azar, para poder alcanzar el óptimo.

A continuación, se trabajan las preguntas 2 y 3 juntas.

2. Utilice su implementación de GD y de SGD para encontrar un punto optimal de la segunda función asignada a su grupo.
3. Compare las soluciones obtenidas por cada algoritmo, y las diferencias en tiempo de ejecución de los mismos.

Dadas las condiciones a continuación, se ejecutaron los algoritmos para las los gradientes definidos sin cuadrados.
Para el SGD se obtuvo:

```
#pregunta 2
w = np.array([0,0,0,0])
m = 500
imax = 10000
lr = 0.00001
tol = 0.0001
SGD(w,m,imax,tol,lr,f = gradf22)
```

✓ 0.3s

```
Punto Final [0.2341251  0.24711835 0.17889051 0.00756589]
Iteración 10000
Error 2.2601136674432185
array([0.2341251 , 0.24711835, 0.17889051, 0.00756589])
```

Para el GS simple, se obtuvo:

```
#pregunta 2
w = np.array([0,0,0,0])
m = 500
imax = 10000
lr = 0.00001
tol = 0.0001
GD(w,m,imax,lr,tol, f = gradF22)
```

✓ 1m 50.8s

```
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
array([-3.58660733e-02, -9.92746917e+01,  1.00871050e+00, -4.58795151e-01])
```

Podemos observar que si bien el SGD tiene un tiempo de ejecución menor al GD, este tiene un nivel de convergencia mucho más lento, dado que tras múltiples iteraciones, el GD convergió al mismo punto, y el SGD a distintos puntos con valores cercanos, esto se refleja además en el error entregado por cada algoritmo. Esto no indica que los algoritmos no puedan converger al mismo punto, simplemente que el SGD, por razones de aleatoriedad no logra acercarse al punto óptimo obtenido por el GD, implicando que requiere de muchas más iteraciones de las especificadas.
Además se podrían considerar problemas de la convexidad de la función $F(\omega)$ sin los cuadrados, lo que a una convergencia distinta a la del SGD, de todas formas, no es posible comprobar el caso, dado que la función F con los cuadrados, genera problemas con los números que entrega, ya que estos son muy grandes, y luego de unas pocas iteraciones, no es posible no es posible seguir obteniendo valores numéricos propios y simplemente entrega variables como inf, -inf, nan, etc.

Evaluando los tiempos de ejecución:
El algoritmo SGD entrega lo siguiente:

```
%timeit SGD(w,m,imax,tol,lr,f = gradf22)
✓ 2.6s
Punto Final [0.24928332 0.30977476 0.17854909 0.0074535 ]
Iteración 10000
Error 2.2489040875661455
Punto Final [0.22786509 0.20596849 0.17887928 0.00745889]
Iteración 10000
Error 2.266345163687663
Punto Final [0.25536536 0.36278199 0.17882429 0.00781538]
Iteración 10000
Error 2.240442333630838
Punto Final [0.26480411 0.40829983 0.17881664 0.00755955]
Iteración 10000
Error 2.2272857177333956
Punto Final [0.23839093 0.26277294 0.17892677 0.00749435]
Iteración 10000
Error 2.253051917720807
Punto Final [0.23739706 0.26093612 0.1789979  0.00753861]
Iteración 10000
Error 2.2688585951388345
Punto Final [0.26037315 0.39834917 0.17830661 0.00739594]
Iteración 10000
Error 2.2452027453202597
Punto Final [0.27316673 0.44270549 0.17830425 0.00782035]
Iteración 10000
Error 2.202086613313421
321 ms ± 6.43 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

El algoritmo GD entrega:

```
%timeit GD(w,m,imax,lr,tol, f = gradF22)
✓ 15m 13.9s

Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
Punto Final [-3.58660733e-02 -9.92746917e+01  1.00871050e+00 -4.58795151e-01]
Iteración 10000
Error 0.0002841184203864806
1min 54s ± 3.86 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Podemos observar que el SGD tiene un tiempo de ejecución mucho más rápido que el GD convencional. Concluímos que el SGD puede ser una herramienta más eficiente que el GD si la aleatoriedad lo permite. Sin embargo, siempre existe la posibilidad de que podría desviarse del camino del gradiente principal por el al gradiente aproximado escogido, y en este sentido, es menos confiable que el tradicional.