Submission

March 24, 2023

[1]:
```python
# Set up the imports
%matplotlib ipympl
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import numpy as np
from numpy import cos, sin, pi, exp
from IPython.display import HTML
```

1 Problem 1

- Implement a function that takes the following inputs, and finds the minimum using gradient descent
  - Function definition (one variable)
  - Derivative (also a function definition)
  - Starting point
  - Learning rate

1.1 Model

[2]:
```python
class gradientDecent():
    def __init__(self, func, dfunc, strtpoint, lr):
        self.func= func
        self.dfunc= dfunc
        self.strtpoint= strtpoint
        self.lr= lr

    def bestx_ret(self):
        xbest= self.strtpoint
        mem=[xbest]
        memy=[self.func(xbest)]
        for i in range(100):
            x= xbest- self.dfunc(xbest)*self.lr
            xbest=x
            mem.append(xbest)
            memy.append(self.func(xbest))
        mem= np.array(mem)
```

```
        memy= np.array(memy)
        return xbest, mem, memy
```

Required Functions and derivative of it

[3]:
```
def cfunc(x):
    return x**4 - 3*x**2 + 1*x

def cfuncd(x):
    return 4*x**3 -6*x + 1
```

[8]:
```
g= gradientDecent(cfunc,cfuncd,2.5,0.01)
res= g.bestx_ret()
print("Best value: ", res[0])
```

```
Best value:  1.130921239372324
```

[9]:
```
xbase= np.linspace(-3,3,100)
ybase= cfunc(xbase)
```

[6]:
```
plt.ioff()

fig, ax= plt.subplots()
ax.plot(xbase,ybase)

lngood, = ax.plot([],[],'go',markersize=10)
lnall, =ax.plot([],[],'ro')
allx=[]
ally=[]
def onestep(frame):
    x= res[1][frame]
    y= res[2][frame]
    allx.append(x)
    ally.append(y)
    lnall.set_data(allx, ally)
    lngood.set_data(x,y)

ani= FuncAnimation(fig, onestep, frames=range(100), interval=100, repeat= False)
HTML(ani.to_jshtml())
```

[6]: `<IPython.core.display.HTML object>`

[7]:
```
plt.close()
```

2 Problem 2

Repeat the above, but with 2 or more variables (you will be tested with different functions with different numbers of variables depending on what you have implemented)

2.1 Model

---

[11]:
```
class GeneralGradientDecent_v2():
    def __init__(self, func,alpha, epochs,*args, **extra):
        self.no_of_args = len(args)
        self.func       = func
        self.alpha      = alpha
        self.epochs     = epochs
        self.args       = args
        self.dfunc_exist= False
        if 'p0' in extra.keys():            #here if starting point is not␣
↪there then by default it will start from origin
            self.sol_start= extra['p0']
        else:
            self.sol_start= [0]*self.no_of_args
        if 'dfunc' in extra.keys():     #if partial derivatives of the function␣
↪is given then it will be stored in dfunc
            self.dfunc= extra['dfunc']
            self.dfunc_exist= True

    #if the dfunc is not given then it will simply find the derivation of a␣
↪function using the standard formulae
    def Partial_Diffentiation(self, func, term, sol):
        h= 1e-4
        reqarray= [0]*self.no_of_args
        reqarray[term]= h
        reqarray=       np.array(reqarray)
        in1=            sol+ reqarray
        df=             (func(*in1)- func(*sol))/h      #simply the formulae␣
↪df= (f(x1+h,x2,..)- f(x1,x2,...))/h
        return df

    #Here is the main optimization code
    def optimize(self):
        sol_best= self.sol_start.copy() #initialising the starting point
        history= []     #i am keeping a record of the values i got in the␣
↪iterations
        history.append(sol_best.copy())
        if self.dfunc_exist:        #if we have given the derivative of the␣
↪functions
            for iter in range(self.epochs):
                temp= sol_best
```

```
                for term in range(self.no_of_args):
                    df= self.dfunc[term]
                    temp[term]-= self.alpha*df(*sol_best)
                sol_best=temp
                history.append(temp.copy())
            history= np.array(history)
        else:   #else it will find the derivative of the functions, manually
            for iter in range(self.epochs):
                temp= sol_best
                for term in range(self.no_of_args):
                    df= self.Partial_Diffentiation(
                                        self.func,
                                        term,
                                        temp
                                        )
                    temp[term]-= self.alpha*df
                sol_best=temp
                history.append(temp.copy())
            history= np.array(history)

        return sol_best,history
```

2.2 Problem 1 - 1-D simple polynomial

The gradient is not specified. You can write the function for gradient on your own. The range within which to search for minimum is [-5, 5].

[12]:
```
#problem 1
def f1(x):
    return x ** 2 + 3 * x + 8
```

here i am making the range of inputs

[13]:
```
xbase= np.linspace(-5,5, 100)
ybase= f1(xbase)
```

using the class GeneralGradientDecent_v2 and giving required inputs.

[14]:
```
problem1_model= GeneralGradientDecent_v2(
                                        f1,     #the function
                                        0.05,   #learning rate
                                        10000,  #epochs
                                        xbase,  #range on which the function␣
 ↪will be made
                                        p0=[4]  #initial value
                                    )
```

[15]:
```
sol, his= problem1_model.optimize()      #the class will return the local minima
                                          #and the history of the sol_best its␣
 ↪been through
```

[16]:
```
print("The best solution here is: ", *sol)
```

```
The best solution here is:  -1.5000499999957597
```

2.3 Plot 2d

[ ]:
```
plt.ioff()

fig, ax= plt.subplots()
ax.plot(xbase,ybase)

lngood, = ax.plot([],[],'go',markersize=10)
lnall, =ax.plot([],[],'ro')
allx=[]
ally=[]
def onestep(frame):
    x= his[frame]
    y= f1(x)
    allx.append(x)
    ally.append(y)
    lnall.set_data(allx, ally)
    lngood.set_data(x,y)

ani= FuncAnimation(fig, onestep, frames=9000, interval=100, repeat= False)
HTML(ani.to_jshtml())
```

[ ]:
```
plt.close()
```

2.4 Problem 2 - 2-D polynomial

Functions for derivatives, as well as the range of values within which to search for the minimum, are given.

[17]:
```
xlim3 =  [-10, 10]
ylim3 =  [-10, 10]
xbase= np.linspace(*xlim3, 1000)
ybase= np.linspace(*ylim3, 1000)
def f3(x, y):
    return x**4 - 16*x**3 + 96*x**2 - 256*x + y**2 - 4*y + 262

def df3_dx(x, y):
    return 4*x**3 - 48*x**2 + 192*x - 256

def df3_dy(x, y):
```

```
    return 2*y - 4
```

[18]:
```
model= GeneralGradientDecent_v2(
                                    f3,                            #the function
                                    0.05,
                                    10000,
                                    xbase,
                                    ybase,
                                    p0=[1.5,2],
                                    dfunc= [df3_dx, df3_dy]     #here since we␣
 ↪have the dfunc we can input the dfuncs
                                )
```

[19]:
```
sol, his= model.optimize()
```

[20]:
```
print("Here the best solution is:", sol)
```

```
Here the best solution is: [4.015802723063649, 2.0]
```

[ ]:
```
X, Y = np.meshgrid(xbase, ybase)
Z = f3(X, Y)

bestcost = 100000
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis', alpha=0.7 ,␣
 ↪edgecolor='none')
ax.set_title('3-D Polynomial')
#ax.set_xlim3d([-10, 10])
ax.set_xlabel('X')
#ax.set_ylim3d([-10, 10])
ax.set_ylabel('Y')
ax.set_zlabel('Z')
x_all, y_all, z_all = [], [], []

lnall,  = ax.plot([], [], [], 'yo')
lngood, = ax.plot([], [], [], 'go', markersize=10)

def onestepderiv(frame):
    global his
    x_all.append(his[frame][0])
    y_all.append(his[frame][1])
    z_all.append(f3(*his[frame]))
    lngood.set_data_3d(*his[frame], f3(*his[frame]))
    lnall.set_data_3d(x_all, y_all, z_all)
    return lngood,
```

```
anime = FuncAnimation(fig, onestepderiv, frames=9500, interval=1000,␣
↪repeat=False)
plt.show()
```

[ ]:
```
plt.close()
```

2.5 Problem 3 - 2-D function

Derivatives and limits given.

[22]:
```
xlim4 = [-pi, pi]
ylim4= [-pi,pi]
xbase= np.linspace(*xlim4, 1000)
ybase= np.linspace(*ylim4, 1000)
def f4(x,y):
    return exp(-(x - y)**2)*sin(y)

def f4_dx(x, y):
    return -2*exp(-(x - y)**2)*sin(y)*(x - y)

def f4_dy(x, y):
    return exp(-(x - y)**2)*cos(y) + 2*exp(-(x - y)**2)*sin(y)*(x - y)
```

making the model,

[23]:
```
model= GeneralGradientDecent_v2(
                                    f4,
                                    0.05,
                                    10000,
                                    xbase,
                                    ybase,
                                    dfunc= [f4_dx, f4_dy]
                                )
```

[24]:
```
sol, his= model.optimize()  #finding the solution
```

[25]:
```
print("The best solution is: ",sol)
```

```
The best solution is:  [-1.5707963267948912, -1.5707963267948923]
```

[ ]:
```
X, Y = np.meshgrid(xbase, ybase)
Z = f4(X, Y)
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis', alpha=0.7 ,␣
↪edgecolor='none')
ax.set_title('3-D Polynomial')
```

```python
ax.set_xlabel('X')

ax.set_ylabel('Y')
ax.set_zlabel('Z')
x_all, y_all, z_all = [], [], []

lnall,  = ax.plot([], [], [], 'yo')
lngood, = ax.plot([], [], [], 'go', markersize=10)

def onestepderiv(frame):
    global his
    x_all.append(his[frame][0])
    y_all.append(his[frame][1])
    z_all.append(f3(*his[frame]))
    lngood.set_data_3d(*his[frame], f3(*his[frame]))
    lnall.set_data_3d(x_all, y_all, z_all)
    #print(bestx, besty, f3(bestx, besty))
    return lngood,

anime = FuncAnimation(fig, onestepderiv, frames=9500, interval=100,␣
 ↪repeat=False)
plt.show()
```

[ ]:
```python
plt.close()
```

2.6 Problem 4 - 1-D trigonometric

Derivative not given. Optimization range [0, 2*pi]

[26]:
```python
xlim= [0, 2*pi]
xbase= np.linspace(*xlim, 100)
def f5(x):
    return cos(x)**4 - sin(x)**3 - 4*sin(x)**2 + cos(x) + 1

ybase = f5(xbase)
```

[27]:
```python
model= GeneralGradientDecent_v2(
                                    f5,
                                    0.01,
                                    10000,
                                    xbase,
                                )
```

[28]:
```python
sol, his= model.optimize()
print("The best solution is :", sol)
```

```
The best solution is : [1.6616108121059847]
```

[ ]:
```python
plt.ioff()
fig, ax= plt.subplots()
ax.plot(xbase,ybase)

lngood, = ax.plot([],[],'go',markersize=10)
lnall, =ax.plot([],[],'ro')
allx=[]
ally=[]
def onestep(frame):
    x= his[frame]
    y= f5(x)
    allx.append(x)
    ally.append(y)
    lnall.set_data(allx, ally)
    lngood.set_data(x,y)

ani= FuncAnimation(fig, onestep, frames=9000, interval=100, repeat= False)
HTML(ani.to_jshtml())
```

[ ]:
```python
plt.close()
```

[29]:
```python
model2= GeneralGradientDecent_v2(
                                    f5,
                                    0.01,
                                    10000,
                                    xbase,
                                    p0=[6]
                                )
```

[30]:
```python
sol1, his1= model2.optimize()
print("The best solution for starting point as x=6 is:", sol1)
```

```
The best solution for starting point as x=6 is: [4.518962831886393]
```

[ ]:
```python
plt.ioff()
fig, ax= plt.subplots()
ax.plot(xbase,ybase)

lngood, = ax.plot([],[],'go',markersize=10)
lnall, =ax.plot([],[],'ro')
allx=[]
ally=[]
def onestep(frame):
    x= his1[frame]
    y= f5(x)
    allx.append(x)
    ally.append(y)
    lnall.set_data(allx, ally)
```

```
    lngood.set_data(x,y)

ani= FuncAnimation(fig, onestep, frames=9000, interval=100, repeat= False)
HTML(ani.to_jshtml())
```

[ ]:
```
plt.close()
```

[ ]:
```
print("Here we can see that the better solution is at x= ", sol,"and f5(x)= "␣
 ↪,f5(sol))
```

[ ]:
```

```