IDNM 680: HOMEWORK 4

JAMES DELLA-GIUSTINA

Task 1

Using the gradient descent method, develop a python code to find the minimum of the functions:

- $f(x) = x^4 - 3x^3 + 2$
- $f(x, y) = 3(x-2)^2 + 5(y+3)^2$

Solution 1. Gradient descent involves finding the minimum of a function f. Let the vector of partial derivatives of f with respect to each component of x be defined as:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]^T$$

The gradient of a vector is the direction of steepest ascent, and taking the negative of the gradient allows us to approach some minima (if a function is convex, i.e. it is uniform continuous or Lipschitz, then any minima is a global minimum). But sometimes, we cannot find exactly what value of x yields $\nabla f = 0$. Therefore, we employ an iterative approach, by first choosing a random x_0 and evaluating $\nabla f(x_0)$. With this in hand, we find:

$$x_1 = x_0 - \eta f(x_0)$$

for a small learning rate $\eta \in (0, 1)$ usually chosen ~ 0.001 (η may also be initially larger and decrease with each iteration). In general, we have;

$$x_i = x_{i-1} - \eta f(x_{i-1}) \tag{1}$$

We can stop iterating and be safe in assuming that we have reached a minimum when $\|\eta \nabla f(x_i)\|$ falls below some specified threshold.

We first use the method of line search to decrease our learning rate to converge to a minima. In this method, we begin with a learning rate η, with parameters $\alpha \in (0, 1)$ and $\beta \in (0, 0.5)$. We calculate perform the calculation in Equation (1), while also calculating $f(x_i)$ and $f(x_{i-1})$. If $f(x_i) \leq f(x_{i-1}) - \alpha\eta \|\nabla f(x_i)\|$, then we update our learning rate $\eta = \beta \cdot \eta$. The following `Python` code minimizes the function $f(x) = x^4 - 3x^3 + 2$.

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA
# Function to optimize (change this to whatever function you want to minimize)
def function(x):
    return x**4-3*x**3+2

# Line search function to find optimal learning rate
def line_search(x, grad, lr, iterations, alpha, beta, function):
    for i in range(iterations):
        cost = function(x - lr*grad)
        prev_cost = function(x)
        if cost <= prev_cost - alpha*lr*LA.norm(grad):
            return lr * beta
        else:
            return lr
    return lr

# Gradient descent with line search
```

Date: Spring 2023.

```
def gradient_descent(x0, lr, iterations, threshold, function):
    history = [x0]
    x = x0
    step_count = 0
    beta = 0.1
    alpha = 0.5
    for i in range(iterations):
        grad = 4*x**3-9*x**2 # derivative of the function
        lr = line_search(x, grad, lr, iterations, alpha, beta, function)
        x = x - lr*grad
        history.append(x)
        step_count += 1
        if lr * LA.norm(grad) < threshold: # stopping condition
            break
    print("Number of steps taken:", step_count)
    return history


# Define initial parameters
x0 = np.random.uniform(-20,20) # starting point
lr = .001
iterations = 1000 # number of iterations
threshold = 1e-12 # threshold for stopping condition

# Run gradient descent with line search
history = gradient_descent(x0, lr, iterations, threshold, function)

# Convert the history list to a numpy array
history = np.array(history)

# Plot the function and the history of gradient descent
x = np.linspace(-50,50, 1000)
y = function(x)
plt.plot(x, y, label='Function')
plt.plot(history, function(history), 'ro-', label='Gradient descent')
plt.legend()
plt.show()
```

with convergence in 11 steps and illustrated by Figure 1.

For our function of two variables $f(x,y) = 3(x-2)^2 + 5(y+3)^2$, we may use the exact same code but modify it to take into account that we now have two variables to update and two components of our gradient $\nabla f(x,y)$.

```
# Function to optimize
def function(x, y):
    return 3*(x-2)**2 + 5*(y+3)**2

# Line search function to find optimal learning rate
def line_search(x, y, grad_x, grad_y, lr, function, alpha, beta, max_iterations=100,
    threshold=1e-12):
    for i in range(max_iterations):
        cost = function(x - lr*grad_x, y - lr*grad_y)
        prev_cost = function(x, y)
        if cost <= prev_cost - threshold*lr*LA.norm([grad_x, grad_y]):
            return lr * beta
        else:
```

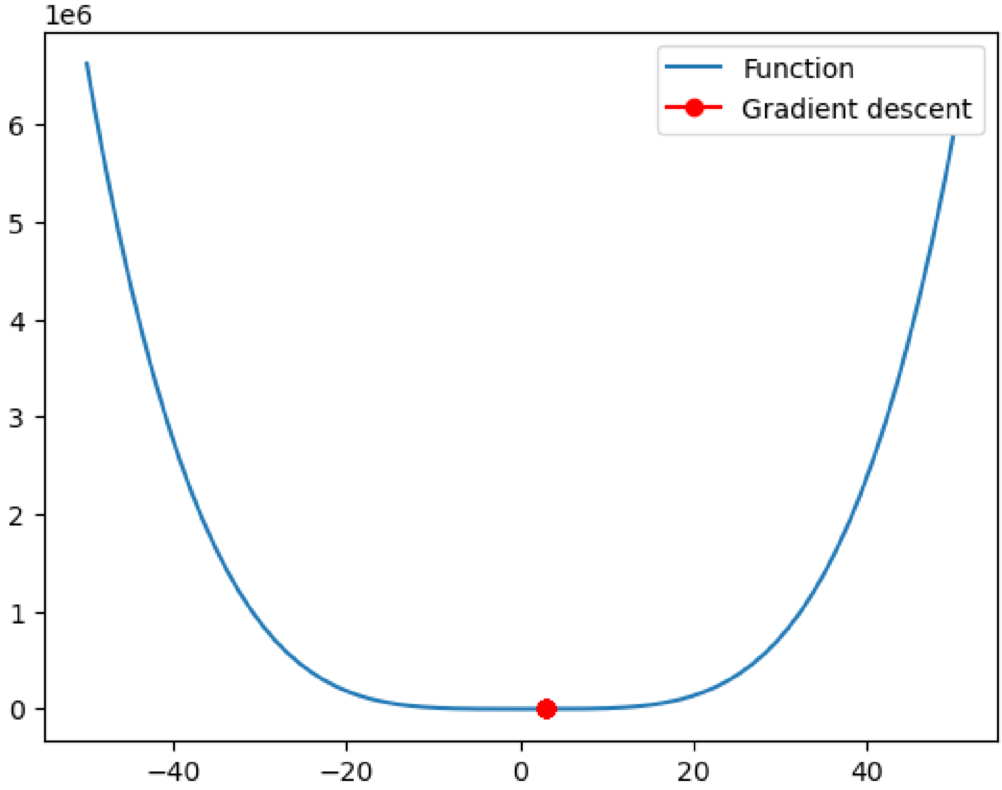

FIGURE 1. Minimizing the function $f(x) = x^4 - 3x^3 + 2$ using gradient descent and updating our learning rate through the backtrack line search method.

```
            return lr
    return lr

# Gradient descent with line search
def gradient_descent(x0, y0, lr, iterations, threshold, function):
    history_x = [x0]
    history_y = [y0]
    x = x0
    y = y0
    step_count = 0
    alpha = .1
    beta = .5
    for i in range(iterations):
        grad_x = 6*(x-2)
        grad_y = 10*(y+3)
        lr = line_search(x, y, grad_x, grad_y, lr, function, alpha, beta)
        x = x - lr*grad_x
        y = y - lr*grad_y
        history_x.append(x)
        history_y.append(y)
        step_count += 1
        if lr * np.linalg.norm([grad_x, grad_y]) < threshold: # stopping condition
            break
    print("Number of steps taken:", step_count)
    return history_x, history_y

# Define initial parameters
x0 = np.random.uniform(-10,10) # starting point for x
y0 = np.random.uniform(-10,10) # starting point for y
```

```
lr = 0.1 # initial learning rate
iterations = 1000 # number of iterations
threshold = 1e-12 # threshold for stopping condition

# Run gradient descent with line search
history_x, history_y = gradient_descent(x0, y0, lr, iterations, threshold, function)

# Convert the history lists to numpy arrays
history_x = np.array(history_x)
history_y = np.array(history_y)

# Generate a grid of (x,y) points for plotting the function surface
x = np.linspace(-10, 10, 100)
y = np.linspace(-10, 10, 100)
X, Y = np.meshgrid(x, y)
Z = function(X, Y)

# Plot the function surface and the history of gradient descent
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X, Y, Z, cmap='inferno',alpha = .5)
ax.plot(history_x, history_y, function(history_x, history_y), 'ro-', label='Gradient
    descent')
ax.plot(history_x, history_y, np.zeros_like(history_x), 'ro-', label='History of
    gradient descent')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x,y)')
plt.legend()
plt.show()
```

This converges in 44 steps and produces Figure 2.

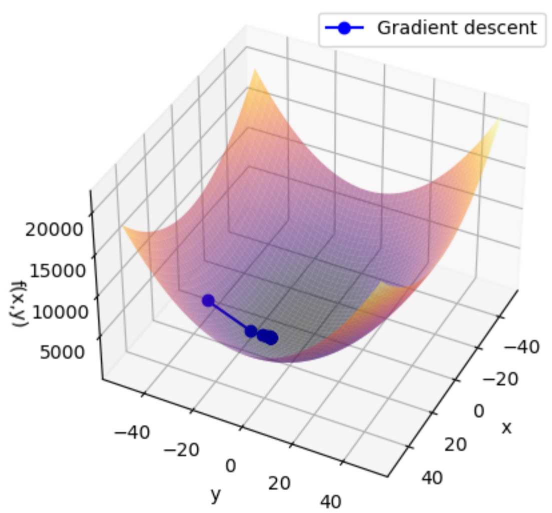

FIGURE 2. Minimizing the function $f(x, y) = 3(x-2)^2 + 5(y+3)^2$ using gradient descent and updating our learning rate through the backtrack line search method.

Task 2

We will have a webcam input. How could we compute a beauty score? How to learn what is beauty? Follow the link below to create the dataset and find the range of the rating (min and max).
`https://scientific-python.readthedocs.io/en/latest/notebooks_rst/6_Machine_Learning/04_Exercices/02_Practical_Work/02_CNN_1_Beauty.htmlproblem-statement`

Solution 2. This code was executed in GoogleColab. We first download and unzip the data set SCUT-FBP5500 dataset `https://github.com/HCIILAB/SCUT-FBP5500-Database-Release` by the Human Computer Intelligent Interaction Lab of South China University of Technology.

```
#%%capture
!pip install livelossplot
!pip uninstall gdown -y && pip install gdown
![ ! -d 'datasets' ] && echo 'datasets dir not found, will create' && mkdir datasets
![ ! -d 'logs' ] && echo 'logs dir not found, will create' && mkdir logs
![ ! -d 'drive/My Drive/Colab Notebooks/models' ] && echo 'models dir not found, will
    create' && mkdir 'drive/My Drive/Colab Notebooks/models'
![ ! -d './datasets/SCUT-FBP5500_v2' ] && echo './datasets/SCUT-FBP5500_v2 not found,
    will download & unzip it' && gdown 1w0TorBfTIqbquQVd6k3h_77ypnrvfGwf -O ./
   datasets/ && unzip -n -q -d './datasets/' './datasets/SCUT-FBP5500_v2.1.zip' &&
   rm -f datasets/SCUT-FBP5500_v2.1.zip
```

In order to easily access the file system in GoogleColab, path variables are defined:

```
import os

dataset_path = os.path.relpath("datasets/SCUT-FBP5500_v2")
txt_file_path = os.path.join(dataset_path, "train_test_files", "All_labels.txt")
images_path = os.path.join(dataset_path, "Images")

assert os.path.exists(dataset_path)
assert os.path.exists(txt_file_path)
assert os.path.exists(images_path)

print("dataset_path:", dataset_path)
print("txt_file_path:", txt_file_path)
print("images_path:", images_path)
```

We then use a Pandas data frame object to read in the `All_labels.txt` CSV where a space ‘ ’ serves as a delimiter between the two columns ‘filename’ and ‘rating’. Pandas data frame objects have many useful built in functions described by the comments in the code:

```
import pandas as pd

df = pd.read_csv('/content/datasets/SCUT-FBP5500_v2/train_test_files/All_labels.txt',
                 sep=' ',
                 names=["filename", "rating"])

# Check the 5th first row
df.head()

# Describe the dataset
df.describe()

# Rates histogram
df.rating.hist(bins=30, density=1)
```

which gives output:

```
    filename  rating
0  CF437.jpg  2.883333
1  AM1384.jpg  2.466667
2  AM1234.jpg  2.150000
3  AM1774.jpg  3.750000
4  CF215.jpg  3.033333

       rating
count  5500.000000
mean  2.990891
std  0.688112
min  1.016667
25%  2.500000
50%  2.833333
75%  3.533333
max  4.750000
```

and produces Figure 3. The range of ratings up to two significant digits is [1.02,4.75] and the average rating is 2.9.

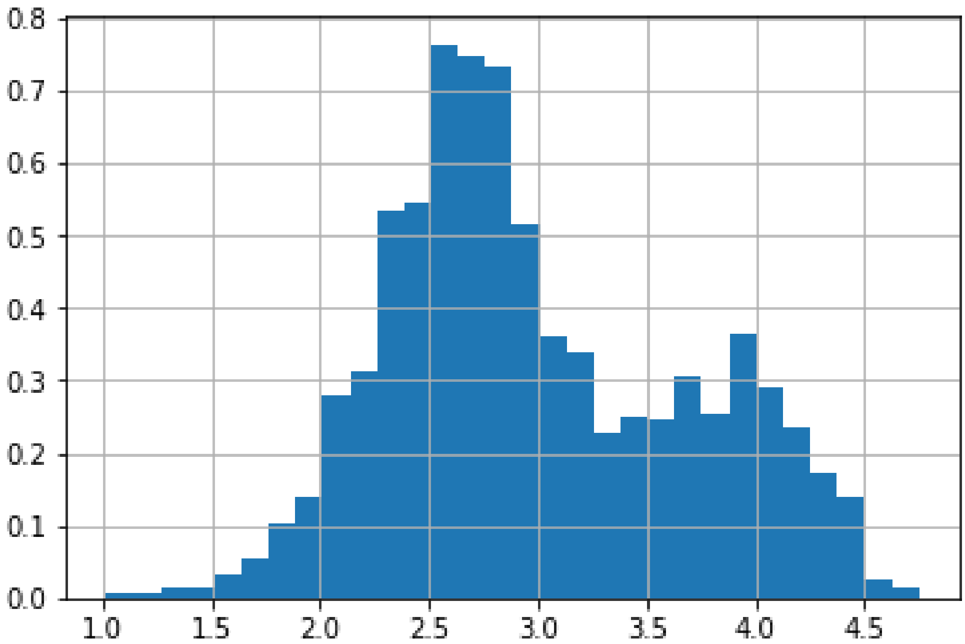

FIGURE 3. Distribution of ratings and frequency.

TASK 3

Use the gradient descent method to find the minimum of $f(x, y, z) = -(x-1)^2+3(y-2)^3+4(z+3)^2$. Dynamically show the step-by-step optimization process.

Solution 3. Rather than trying to start immediately coding this, let us first do perform some analysis on our function.

Claim: The function $f(x, y, z) = -(x-1)^2 + 3(y-2)^3 + 4(z+3)^2$ has domain $\mathbb{R}^3$ and co-domain $\mathbb{R}$.

Proof. To show this, we only need to examine the term $(y-2)^3$ which is $\mathcal{O}(y^3)$ (in fact the whole function is $\mathcal{O}(y^3)$). For all $y \in \mathbb{R}$, $y^3 \in \mathbb{R}$, and therefore the co-domain of $f(x, y, z)$ is $\mathbb{R}$. As $|y| \to \infty$, this term dominates and causes the function to become unbounded. Therefore, there exists no global minima for the function. $\square$

We now provide an alternative proof which will work in general to determine whether or nor an arbitrary function is convex and therefore has a global minima.

Proof. The Hessian matrix of a function is given as

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Definition 1. An $n \times n$ symmetric real matrix M is said to be positive semi-definite iff $\mathbf{x}^\top M \mathbf{x} \geq 0$ for all $\mathbf{x}$ in $\mathbb{R}^n$. Formally,

$$M \text{is positive semi-definite} \iff \mathbf{x}^\top M \mathbf{x} \geq 0 \text{ for all } \mathbf{x} \in \mathbb{R}^n$$

If the Hessian matrix of a function is semi-positive definite, then the function is convex. The Hessian of our function is given as:

$$\begin{bmatrix} -2 & 0 & 0 \\ 0 & 18(y-2) & 0 \\ 0 & 0 & 8 \end{bmatrix}$$

Now take the vector $\mathbf{x}^\top = e_1 = (1, 0, 0)$. Then $\mathbf{x}^\top H \mathbf{x}$

$$[1 \quad 0 \quad 0] \begin{bmatrix} -2 & 0 & 0 \\ 0 & 18(y-2) & 0 \\ 0 & 0 & 8 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = -2 \not\geq 0$$

Therefore the function is non-convex and consequentially there does not exist any global minima. However, we can now talk about finding a local minimum in terms of eigenvalues. H may not be positive definite for any vector $\mathbf{x} \in \mathbb{R}^3$, but it may have a local minimum if all eigenvalues of H are positive. Since our Hessian H is diagonal, then all eigenvalues lie on the main diagonal and so $\lambda_i = \{-2, 18(y-2), 8\}$ for $i \in [3]$. This implies that for $\forall y > 2$, H is semi-positive definite. □

I employed the same code I used for Problem (1); however, knowing that our function is non-convex and that H is semi-positve definite for $\forall y \geq 2$, I chose to initialize y from a normal distribution of numbers from $[2, 100]$. Although this did not gurantee that y fell below two, it handled the overflow errors that would have resulted for a different intial range.

```python
import numpy as np
from numpy import linalg as LA
# Function to optimize
def function(x, y, z):
    return (-x+1)**2 + (3*y-6)**3 + (4*z+12)**2

# Line search function to find optimal learning rate
def line_search(x, y, z, grad_x, grad_y, grad_z, lr, function, alpha, beta,
    max_iterations=1000, threshold=1e-12):
    for i in range(max_iterations):
        cost = function(x - lr*grad_x, y - lr*grad_y, z - lr*grad_z)
        prev_cost = function(x, y, z)
        if cost <= prev_cost - threshold*lr*LA.norm([grad_x, grad_y, grad_z]):
            return lr * beta
```

```
        else:
            return lr
    return lr

# Gradient descent with line search
def gradient_descent(x0, y0, z0, lr, iterations, threshold, function):
    history_x = [x0]
    history_y = [y0]
    history_z = [z0]
    x = x0
    y = y0
    z = z0
    step_count = 0
    alpha = .01
    beta = .05
    for i in range(iterations):
        grad_x = np.float128(-2*(x-1))
        grad_y = np.float128(9*(y-2)**2)
        grad_z = np.float128(8*(z+3))
        lr = line_search(x, y, z, grad_x, grad_y, grad_z, lr, function, alpha, beta)
        x = x - lr*grad_x
        y = y - lr*grad_y
        z = z - lr*grad_z
        history_x.append(x)
        history_y.append(y)
        history_z.append(z)
        step_count += 1
        if lr * LA.norm([grad_x, grad_y, grad_z]) < threshold: # stopping condition
            break
    print("Number of steps taken:", step_count)
    print("Minima found at:(", x,',',y,',',z,') with f(x,y,z)=',function(x,y,z))
    return history_x, history_y, history_z

# Define initial parameters
x0 = np.random.uniform(0, 50) # starting point for x
y0 = np.random.uniform(5,100) # starting point for y
z0 = np.random.uniform(-50,50) # starting point for z
lr = 0.1 # initial learning rate
iterations = 1000 # number of iterations
threshold = 1e-15 # threshold for stopping condition

# Run gradient descent with line search
history_x, history_y, history_z = gradient_descent(x0, y0, z0, lr, iterations,
    threshold, function)
```

Which produced output

```
Number of steps taken: 16
Minima found at:( 26.783149920160626152 , -613.32636881404249407 , -1.4514259468715789799 )
with f(x,y,z)= -6290429437.343050923
```