C2_W3_Lab_3_Optimization_Using_Newtons_Method

January 14, 2025

1 Optimization Using Newton's Method

In this lab you will implement Newton's method optimizing some functions in one and two variables. You will also compare it with the gradient descent, experiencing advantages and disadvantages of each of the methods.

2 Table of Contents

- 1 - Function in One Variable
- 2 - Function in Two Variables

2.1 Packages

Run the following cell to load the packages you'll need.

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
```

1 - Function in One Variable

You will use Newton's method to optimize a function $f(x)$. Aiming to find a point, where the derivative equals to zero, you need to start from some initial point x_0, calculate first and second derivatives ($f'(x_0)$ and $f''(x_0)$) and step to the next point using the expression:

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)}, \tag{1}$$

Repeat the process iteratively. Number of iterations n is usually also a parameter.

Let's optimize function $f(x) = e^x - \log(x)$ (defined for $x > 0$) using Newton's method. To implement it in the code, define function $f(x) = e^x - \log(x)$, its first and second derivatives $f'(x) = e^x - \frac{1}{x}$, $f''(x) = e^x + \frac{1}{x^2}$:

[ ]:
```
def f_example_1(x):
    return np.exp(x) - np.log(x)

def dfdx_example_1(x):
    return np.exp(x) - 1/x
```

```
def d2fdx2_example_1(x):
    return np.exp(x) + 1/(x**2)

x_0 = 1.6
print(f"f({x_0}) = {f_example_1(x_0)}")
print(f"f'({x_0}) = {dfdx_example_1(x_0)}")
print(f"f''({x_0}) = {d2fdx2_example_1(x_0)}")
```

Plot the function to visualize the global minimum:

[ ]:
```
def plot_f(x_range, y_range, f, ox_position):
    x = np.linspace(*x_range, 100)
    fig, ax = plt.subplots(1,1,figsize=(8,4))

    ax.set_ylim(*y_range)
    ax.set_xlim(*x_range)
    ax.set_ylabel('$f\,(x)$')
    ax.set_xlabel('$x$')
    ax.spines['left'].set_position('zero')
    ax.spines['bottom'].set_position(('data', ox_position))
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')
    ax.autoscale(enable=False)

    pf = ax.plot(x, f(x), 'k')

    return fig, ax

plot_f([0.001, 2.5], [-0.3, 13], f_example_1, 0.0)
```

Implement Newton's method described above:

[ ]:
```
def newtons_method(dfdx, d2fdx2, x, num_iterations=100):
    for iteration in range(num_iterations):
        x = x - dfdx(x) / d2fdx2(x)
        print(x)
    return x
```

In addition to the first and second derivatives, there are two other parameters in this implementation: number of iterations `num_iterations`, initial point `x`. To optimize the function, set up the parameters and call the defined function gradient_descent:

[ ]:
```
num_iterations_example_1 = 25; x_initial = 1.6
newtons_example_1 = newtons_method(dfdx_example_1, d2fdx2_example_1, x_initial,␣
↪num_iterations_example_1)
```

```
print("Newton's method result: x_min =", newtons_example_1)
```

You can see that starting from the initial point $x_0 = 1.6$ Newton's method converges after 6 iterations. You could actually exit the loop when there is no significant change of x each step (or when first derivative is close to zero).

What if gradient descent was used starting from the same intial point?

[ ]:
```
def gradient_descent(dfdx, x, learning_rate=0.1, num_iterations=100):
    for iteration in range(num_iterations):
        x = x - learning_rate * dfdx(x)
        print(x)
    return x

num_iterations = 25; learning_rate = 0.1; x_initial = 1.6
# num_iterations = 25; learning_rate = 0.2; x_initial = 1.6
gd_example_1 = gradient_descent(dfdx_example_1, x_initial, learning_rate,␣
 ↪num_iterations)
print("Gradient descent result: x_min =", gd_example_1)
```

Gradient descent method has an extra parameter `learning_rate`. If you take it equal to `0.1` in this example, the method will start to converge after about 15 iterations (aiming for an accuracy of 4-5 decimal places). If you increase it to 0.2, gradient descent will converge within about 12 iterations, which is still slower than Newton's method.

So, those are disadvantages of gradient descent method in comparison with Newton's method: there is an extra parameter to control and it converges slower. However it has an advantage - in each step you do not need to calculate second derivative, which in more complicated cases is quite computationally expensive to find. So, one step of gradient descent method is easier to make than one step of Newton's method.

This is the reality of numerical optimization - convergency and actual result depend on the initial parameters. Also, there is no "perfect" algorithm - every method will have advantages and disadvantages.

2 - Function in Two Variables

In case of a function in two variables, Newton's method will require even more computations. Starting from the intial point (x_0, y_0), the step to the next point shoud be done using the expression:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} - H^{-1}(x_0, y_0) \nabla f(x_0, y_0), \tag{2}$$

where $H^{-1}(x_0, y_0)$ is an inverse of a Hessian matrix at point (x_0, y_0) and $\nabla f(x_0, y_0)$ is the gradient at that point.

Let's implement that in the code. Define the function $f(x, y)$ like in the videos, its gradient and Hessian:

$$f(x,y) = x^4 + 0.8y^4 + 4x^2 + 2y^2 - xy - 0.2x^2y, \tag{1}$$

$$\nabla f(x,y) = \begin{bmatrix} 4x^3 + 8x - y - 0.4xy \\ 3.2y^3 + 4y - x - 0.2x^2 \end{bmatrix}, \tag{2}$$

$$H(x,y) = \begin{bmatrix} 12x^2 + 8 - 0.4y & -1 - 0.4x \\ -1 - 0.4x & 9.6y^2 + 4 \end{bmatrix}. \tag{3}$$

[ ]:
```
def f_example_2(x, y):
    return x**4 + 0.8*y**4 + 4*x**2 + 2*y**2 - x*y -0.2*x**2*y

def grad_f_example_2(x, y):
    return np.array([[4*x**3 + 8*x - y - 0.4*x*y],
                     [3.2*y**3 +4*y - x - 0.2*x**2]])

def hessian_f_example_2(x, y):
    hessian_f = np.array([[12*x**2 + 8 - 0.4*y, -1 - 0.4*x],
                          [-1 - 0.4*x, 9.6*y**2 + 4]])
    return hessian_f

x_0, y_0 = 4, 4
print(f"f{x_0, y_0} = {f_example_2(x_0, y_0)}")
print(f"grad f{x_0, y_0} = \n{grad_f_example_2(x_0, y_0)}")
print(f"H{x_0, y_0} = \n{hessian_f_example_2(x_0, y_0)}")
```

Run the following cell to plot the function:

[ ]:
```
def plot_f_cont_and_surf(f):

    fig = plt.figure( figsize=(10,5))
    fig.canvas.toolbar_visible = False
    fig.canvas.header_visible = False
    fig.canvas.footer_visible = False
    fig.set_facecolor('#ffffff')
    gs = GridSpec(1, 2, figure=fig)
    axc = fig.add_subplot(gs[0, 0])
    axs = fig.add_subplot(gs[0, 1],  projection='3d')

    x_range = [-4, 5]
    y_range = [-4, 5]
    z_range = [0, 1200]
    x = np.linspace(*x_range, 100)
    y = np.linspace(*y_range, 100)
    X,Y = np.meshgrid(x,y)

    cont = axc.contour(X, Y, f(X, Y), cmap='terrain', levels=18, linewidths=2,␣
↪alpha=0.7)
```

```
    axc.set_xlabel('$x$')
    axc.set_ylabel('$y$')
    axc.set_xlim(*x_range)
    axc.set_ylim(*y_range)
    axc.set_aspect("equal")
    axc.autoscale(enable=False)

    surf = axs.plot_surface(X,Y, f(X,Y), cmap='terrain',
                    antialiased=True,cstride=1,rstride=1, alpha=0.69)
    axs.set_xlabel('$x$')
    axs.set_ylabel('$y$')
    axs.set_zlabel('$f$')
    axs.set_xlim(*x_range)
    axs.set_ylim(*y_range)
    axs.set_zlim(*z_range)
    axs.view_init(elev=20, azim=-100)
    axs.autoscale(enable=False)

    return fig, axc, axs

plot_f_cont_and_surf(f_example_2)
```

Newton's method (2) is implemented in the following function:

[ ]:
```
def newtons_method_2(f, grad_f, hessian_f, x_y, num_iterations=100):
    for iteration in range(num_iterations):
        x_y = x_y - np.matmul(np.linalg.inv(hessian_f(x_y[0,0], x_y[1,0])),␣
 ↪grad_f(x_y[0,0], x_y[1,0]))
        print(x_y.T)
    return x_y
```

Now run the following code to find the minimum:

[ ]:
```
num_iterations_example_2 = 25; x_y_initial = np.array([[4], [4]])
newtons_example_2 = newtons_method_2(f_example_2, grad_f_example_2,␣
 ↪hessian_f_example_2,
                                     x_y_initial,␣
 ↪num_iterations=num_iterations_example_2)
print("Newton's method result: x_min, y_min =", newtons_example_2.T)
```

In this example starting from the initial point $(4, 4)$ it will converge after about 9 iterations. Try to compare it with the gradient descent now:

[ ]:
```
def gradient_descent_2(grad_f, x_y, learning_rate=0.1, num_iterations=100):
    for iteration in range(num_iterations):
        x_y = x_y - learning_rate * grad_f(x_y[0,0], x_y[1,0])
        print(x_y.T)
    return x_y
```

```
num_iterations_2 = 300; learning_rate_2 = 0.02; x_y_initial = np.array([[4],␣
↪[4]])
# num_iterations_2 = 300; learning_rate_2 = 0.03; x_y_initial = np.array([[4],␣
↪[4]])
gd_example_2 = gradient_descent_2(grad_f_example_2, x_y_initial,␣
↪learning_rate_2, num_iterations_2)
print("Gradient descent result: x_min, y_min =", gd_example_2)
```

Obviously, gradient descent will converge much slower than Newton's method. And trying to increase learning rate, it might not even work at all. This illustrates again the disadvantages of gradient descent in comparison with Newton's method. However, note, that Newton's method required calculation of an inverted Hessian matrix, which is a very computationally expensive calculation to perform when you have, say, a thousand of parameters.

Well done on finishing this lab!

[ ]: