C2_W2_Lab_1_Optimization_Using_Gradient_Descent_in_One_Variable

January 11, 2025

1 Optimization Using Gradient Descent in One Variable

To understand how to optimize functions using gradient descent, start from simple examples - functions of one variable. In this lab, you will implement the gradient descent method for functions with single and multiple minima, experiment with the parameters and visualize the results. This will allow you to understand the advantages and disadvantages of the gradient descent method.

2 Table of Contents

2.1 Packages

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

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
# Some functions defined specifically for this notebook.
from w2_tools import plot_f, gradient_descent_one_variable, f_example_2,␣
 ↪dfdx_example_2
# Magic command to make matplotlib plots interactive.
%matplotlib widget
```

## 1 - Function with One Global Minimum

Function $f(x) = e^x - \log(x)$ (defined for $x > 0$) is a function of one variable which has only one **minimum point** (called **global minimum**). However, sometimes that minimum point cannot be found analytically - solving the equation $\frac{df}{dx} = 0$. It can be done using a gradient descent method.

To implement gradient descent, you need to start from some initial point x_0. Aiming to find a point, where the derivative equals zero, you want to move "down the hill". Calculate the derivative $\frac{df}{dx}(x_0)$ (called a **gradient**) and step to the next point using the expression:

$$x_1 = x_0 - \alpha \frac{df}{dx}(x_0), \tag{1}$$

where $\alpha > 0$ is a parameter called a **learning rate**. Repeat the process iteratively. The number of iterations n is usually also a parameter.

Subtracting $\frac{df}{dx}(x_0)$ you move "down the hill" against the increase of the function - toward the minimum point. So, $\frac{df}{dx}(x_0)$ generally defines the direction of movement. Parameter α serves as a scaling factor.

Now it's time to implement the gradient descent method and experiment with the parameters!

First, define function $f\left(x\right)=e^x - \log(x)$ and its derivative $\frac{df}{dx}\left(x\right)=e^x - \frac{1}{x}$:

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

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

Function $f\left(x\right)$ has one global minimum. Let's plot the function:

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

Gradient descent can be implemented in the following function:

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

Note that there are three parameters in this implementation: `num_iterations`, `learning_rate`, initial point `x_initial`. Model parameters for such methods as gradient descent are usually found experimentially. For now, just assume that you know the parameters that will work in this model - you will see the discussion of that later. To optimize the function, set up the parameters and call the defined function `gradient_descent`:

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

The code in following cell will help you to visualize and understand the gradient descent method deeper. After the end of the animation, you can click on the plot to choose a new initial point and investigate how the gradient descent method will be performed.

You can see that it works successfully here, bringing it to the global minimum point!

What if some of the parameters will be changed? Will the method always work? Uncomment the lines in the cell below and rerun the code to investigate what happens if other parameter values are chosen. Try to investigate and analyse the results. You can read some comments below.

Notes related to this animation: - Gradient descent is performed with some pauses between the iterations for visualization purposes. The actual implementation is much faster. - The animation stops when minimum point is reached with certain accuracy (it might be a smaller number of steps than `num_iterations`) - to avoid long runs of the code and for teaching purposes. - Please wait for the end of the animation before making any code changes or rerunning the cell. In case of any issues, you can try to restart the Kernel and rerun the notebook.

[ ]:
```
num_iterations = 25; learning_rate = 0.1; x_initial = 1.6
# num_iterations = 25; learning_rate = 0.3; x_initial = 1.6
# num_iterations = 25; learning_rate = 0.5; x_initial = 1.6
# num_iterations = 25; learning_rate = 0.04; x_initial = 1.6
# num_iterations = 75; learning_rate = 0.04; x_initial = 1.6
# num_iterations = 25; learning_rate = 0.1; x_initial = 0.05
# num_iterations = 25; learning_rate = 0.1; x_initial = 0.03
# num_iterations = 25; learning_rate = 0.1; x_initial = 0.02

gd_example_1 = gradient_descent_one_variable([0.001, 2.5], [-0.3, 13], f_example_1, dfdx_example_1,
                                    gradient_descent, num_iterations, learning_rate, x_initial, 0.0, [0.35, 9.5])
```

Comments related to the choice of the parameters in the animation above:

- Choosing `num_iterations = 25`, `learning_rate = 0.1`, `x_initial = 1.6` you get to the minimum point successfully. Even a little bit earlier - on the iteration 21, so for this choice of the learning rate and initial point, the number of iterations could have been taken less than `25` to save some computation time.
- Increasing the `learning_rate` to `0.3` you can see that the method converges even faster - you need less number of iterations. But note that the steps are larger and this may cause some problems.
- Increasing the `learning_rate` further to `0.5` the method doesn't converge anymore! You steped too far away from the minimum point. So, be careful - increasing `learning_rate` the method may converge significantly faster… or not converge at all.
- To be "safe", you may think, why not to decrease `learning_rate`?! Take it `0.04`, keeping the rest of the parameters the same. The model will not run enough number of iterations to converge!
- Increasing `num_iterations`, say to `75`, the model will converge but slowly. This would be more "expensive" computationally.
- What if you get back to the original parameters `num_iterations = 25`, `learning_rate = 0.1`, but choose some other `x_initial`, e.g. `0.05`? The function is steeper at that point, thus the gradient is larger in absolute value, and the first step is larger. But it will work - you will get to the minimum point.
- If you take `x_initial = 0.03` the function is even steeper, making the first step significantly larger. You are risking "missing" the minimum point.
- Taking `x_initial = 0.02` the method doesn't converge anymore…

This is a very simple example, but hopefully, it gives you an idea of how important is the choice of the initial parameters.

## 2 - Function with Multiple Minima

Now you can take a slightly more complicated example - a function in one variable, but with multiple minima. Such an example was shown in the videos, and you can plot the function with the following code:

[ ]:
```
plot_f([0.001, 2], [-6.3, 5], f_example_2, -6)
```

Function `f_example_2` and its derivative `dfdx_example_2` are pre-defined and uploaded into this notebook. At this stage, while you are mastering the optimization method, do not worry about the corresponding expressions, just concentrate on the gradient descent and the related parameters for now.

Use the following code to run gradient descent with the same `learning_rate` and `num_iterations`, but with a different starting point:

[ ]:
```
print("Gradient descent results")
print("Global minimum: x_min =", gradient_descent(dfdx_example_2, x=1.3,␣
 ↪learning_rate=0.005, num_iterations=35))
print("Local minimum: x_min =", gradient_descent(dfdx_example_2, x=0.25,␣
 ↪learning_rate=0.005, num_iterations=35))
```

The results are different. Both times the point did fall into one of the minima, but in the first run it was a global minimum, while in the second run it got "stuck" in a local one. To see the visualization of what is happening, run the code below. You can uncomment the lines to try different sets of parameters or click on the plot to choose the initial point (after the end of the animation).

[ ]:
```
num_iterations = 35; learning_rate = 0.005; x_initial = 1.3
# num_iterations = 35; learning_rate = 0.005; x_initial = 0.25
# num_iterations = 35; learning_rate = 0.01; x_initial = 1.3

gd_example_2 = gradient_descent_one_variable([0.001, 2], [-6.3, 5],␣
 ↪f_example_2, dfdx_example_2,
                                              gradient_descent, num_iterations,␣
 ↪learning_rate, x_initial, -6, [0.1, -0.5])
```

You can see that gradient descent method is robust - it allows you to optimize a function with a small number of calculations, but it has some drawbacks. The efficiency of the method depends a lot on the choice of the initial parameters, and it is a challenge in machine learning applications to choose the "right" set of parameters to train the model!

[ ]:
```

```