Mathematics for Machine Learning and Data Science Specialization offered by DeepLearning.AI on COURSERA

Week-2 of Course 2-Calculus for Machine Learning and Data Science

In this week I learnt to:

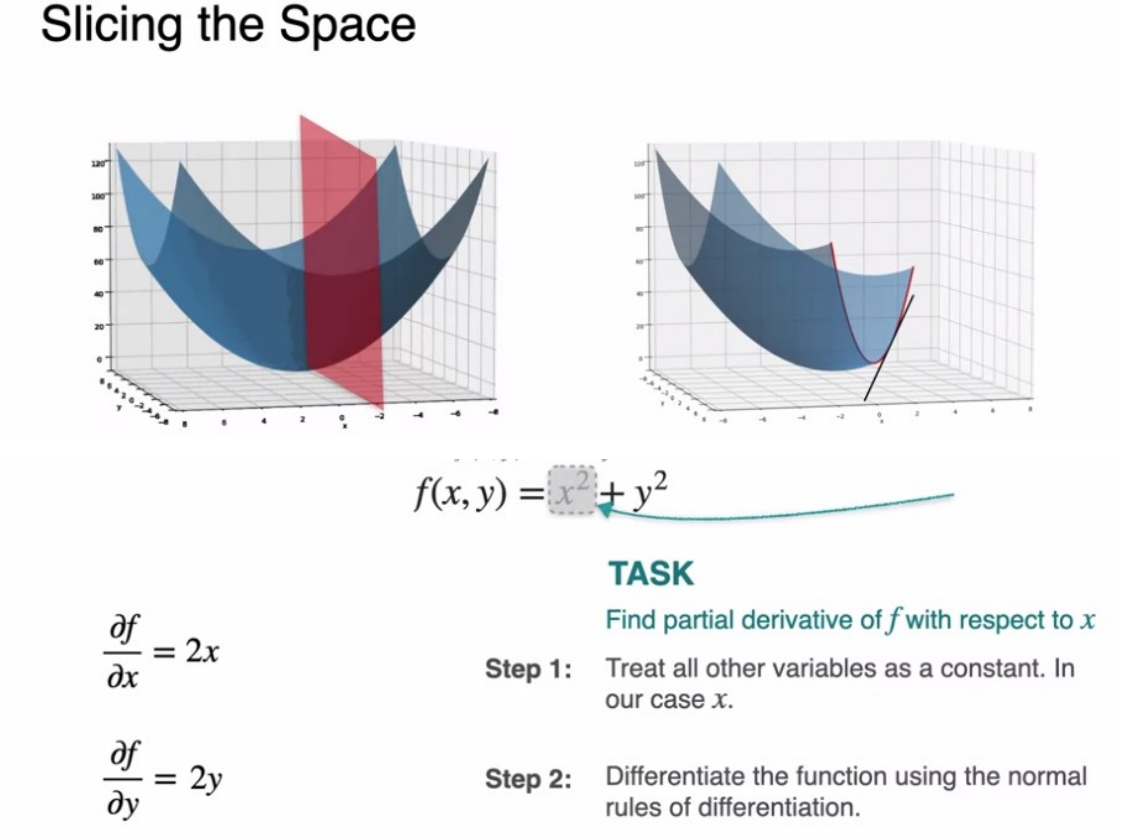
- Perform gradient descent in neural networks with different activation and cost functions
- Visually interpret differentiation of different types of functions commonly used in machine learning
- Approximately optimize different types of functions commonly used in machine learning using first-order (gradient descent) and second-order (Newton's method) iterative methods
- Analytically optimize different types of functions commonly used in machine learning using properties of derivatives and gradients.

Table of Contents

Tangent planes and Partial Derivatives

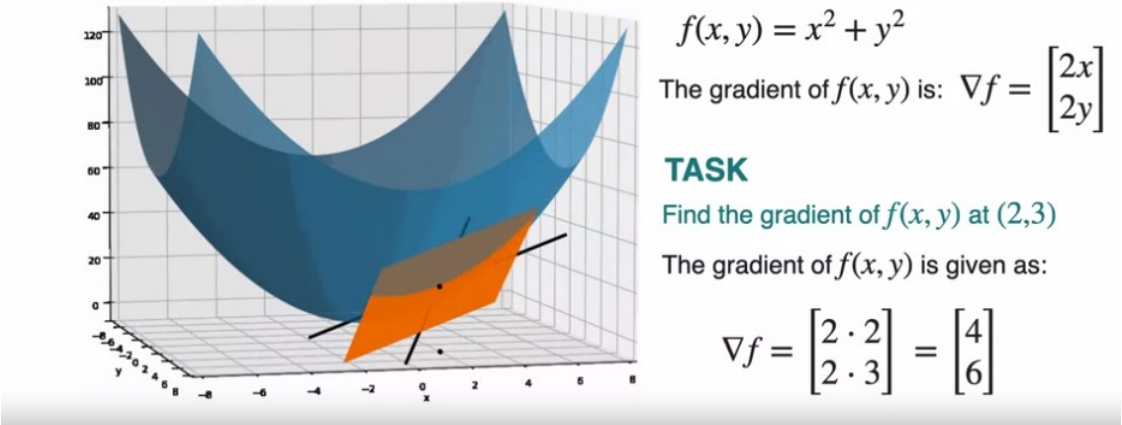
In Week 1, we learn everything about functions with one variable. Now we're going to learn everything about functions with two or more variables.For example things like a tangent line in one-dimension will generalize to a tangent plane in two dimensions. This can also be used in optimization in the exact same way, and it also applies to machine learning. In the same way. However, optimizing functions in two or more variables can get very complicated even for a computer. We're going to introduce methods that will speed this up. One method that we'll learn is called gradient descent.

Imagine that you have this function of two variables plotted in 3D space and that you cut it with a plane like this, what do you get?.And in the cut you can see this red parabola with a tangent over there, that's the partial derivative. Now you can also cut it in another direction, get also a red parabola and that red parable is a function of one variable.

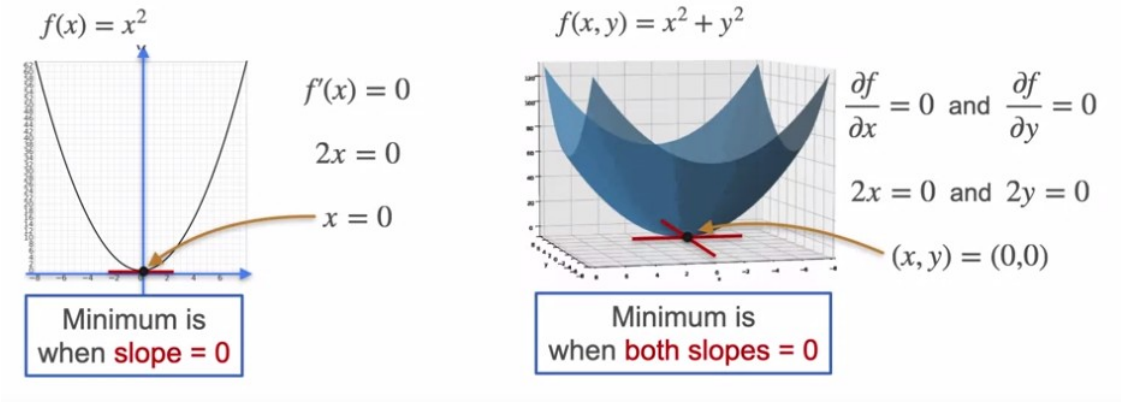


Gradients and maxima/minima

All the partial derivatives can be condensed into a vector known as the gradient.Now that you know what a partial derivative is, you pretty much know what a gradient is.Recall that if you have a function in two variables, you can slice it in two ways, treating y as a constant and treating x as a constant. Each one gives you a different partial derivatives. The first one gives you the partial derivative df/dx, which is 2x, and the second one gives you the partial derivative df/dy, which is 2y. So the gradient is simply the vector containing these two partial derivatives, so basically the vector 2x, 2y.



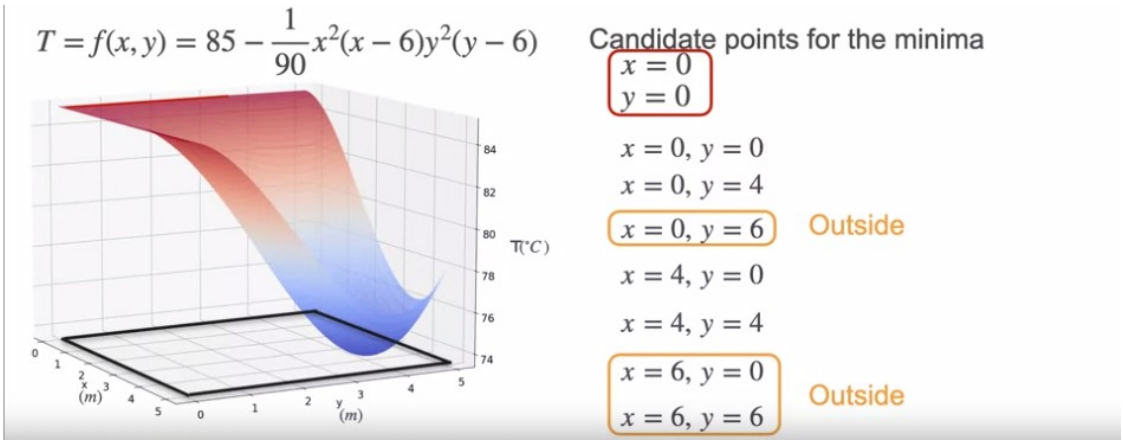
The gradient is pretty useful to minimize functions of two or more variables in the same way as the derivative was useful to minimize functions of one variable.Lets see the example in below image.To find minimum and maimum value we will equate the derivative and partial derivative with 0 and the value we get for x or y will be our point where slope is zero or minimum.



To find the minimums and maximums, all we have to do is set all the partial derivatives to 0 and solve that system of equations.

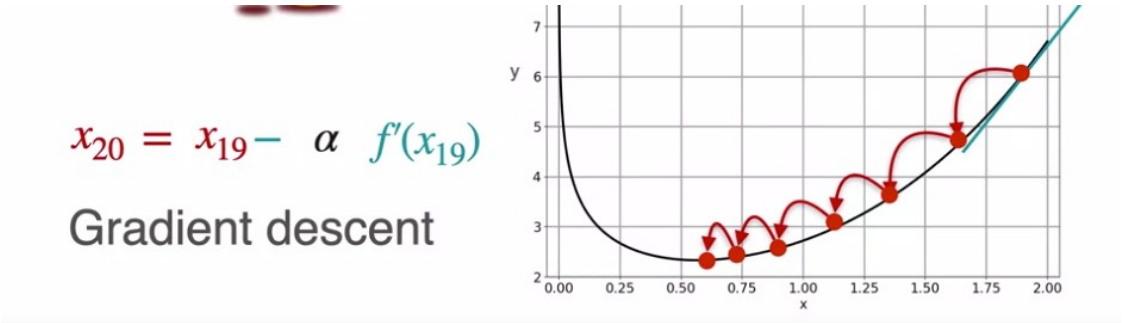
Optimization using Gradient Descent in one variable

Now that we've learned how derivatives work in more than one dimension,lets see how to use them in optimization.In the example below inmage there is graph of temperature of a sauna room where I want to find the coolest point since i started feeling hot now.The red color represents the hot points and blue color the cold points.We have been given with an equation of temperature.We need to find the candidates for minimum points to find the coldest region.Let's say that you're standing somewhere near hottest points here and you're going to basically move around a little bit and see where does it get colder. Let's continue doing that and moving in some random steps and checking out those steps which one took us to a colder place and continue doing this.It's believable that if you do this for a while, you will be able to take steps that take you to the coldest place in the room.Another way to look at it is if you were to take the tangent plane to the temperature function, this tangent plane is parallel to the floor because the two partial derivatives are both zero, and these are given by dT/dx and dT/dy, where T is the temperature function and x and y are the two coordinates in the sauna.After solving the derivative we will equate the equation with zero and we get following candidates for minima.Now, for x equals 0 or y equals 0, those are all maximum because as you can see, all these points give 85 as the temperature, so those are not the minimum. The only minimum is this one over here, x equals 4, y equals 4, which is the point we were looking for. For this point, the temperature is 73.6, so this is the minimum. As you can see, just like we did for one variable, you set the derivative equal to zero, that gives you a bunch of candidates and then you check them individually. Over here, we simply said both derivatives equal to zero and get a lot of candidates and then check them individually.



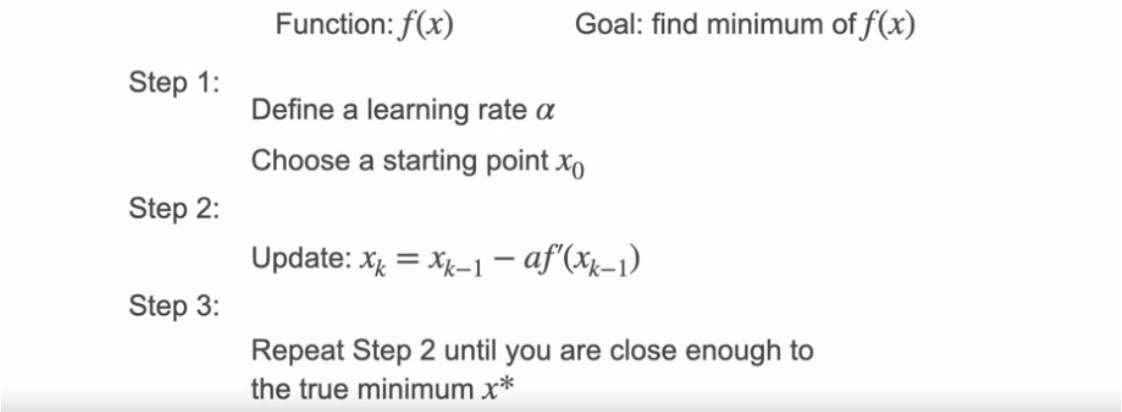
Learning Rate

So far we did see how to find minimum of function using derivative and taking baby steps on the graph.The concept of taking baby steps is whole lot maths behind it and you need to be very careful while taking the steps. if you are at a point where the slope is negative, you need to add a little bit to the coordinate of the point. Whereas if you're in a place where the slope is positive, then you need to subtract a little bit to the coordinate of the point. In short, if you want a new point that is closer to the minimum, then that should be the old point minus the slope. Why? Because if you're to the left of the new minimum, the slope is negative. You're subtracting a negative number and moving to the right. If you were to the right of the minimum, then you subtract a positive number, the slope, and move to the left. You have to be mindful at a very steep part of the curve, because the function is steep, then the derivative is very large so you should not make long jump as it can be pretty chaotic because because you may actually miss the minimum and go pretty far away and get lost. We want to take small steps and to be more secure in our path.The learning rate ensures that the steps we are performing are small enough so the algorithm can converge to the minimum.How do we take into account a small step? Well, you simply modify the formula and put a small number, multiplying the slope, for example, 0.01. You can take any small number you'd like, and that's called the learning rate and is denoted Alpha. There's a whole science in picking good learning rates. Take a look at the graph n equation below and try to understand.Here the α is the learning rate.



In short You start with a function f of x and the goal is to find the minimum of f of x. First you define a learning rate and you choose a starting point, and then you do the updating step, which is X_k equals X_k minus 1 minus Alpha times the derivative at X_k minus 1.

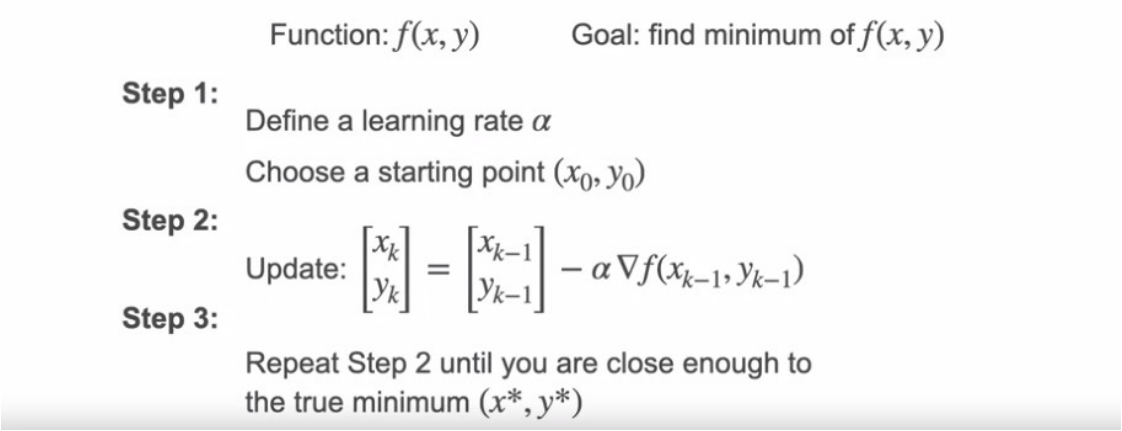
Then you repeat this step until you're close enough to the true minimum. You can tell that you're close enough to the true minimum when your steps don't really change that much.You never need to solve for the derivative zero. You only do know the derivative and then apply it in the algorithm when you're taking the updating step.



NOTE: Learning rate is a big deal in machine learning and finding a good learning rate can be pretty hard. It can also be pretty influential to how well your model does. Let's say you have a learning rate that's too large, then you may miss the minimum and never be able to find it because your steps are just too big. What if the learning rate is too small? Well then you may take forever to actually reach the minimum or perhaps never reach it. What you want is a learning rate that is just right. However, finding a learning rate that is just right can be pretty hard. It's actually a research problem to find a good learning rate. There are a lot of really good methods that change the learning rate based on how the problem is doing, but there's no definite method to find a good learning rate.here's another problem that gradient descent may have. Let's say that you have this function over here, so the minimum is actually this one. However, let's say that you start running your gradient descent algorithm over here, well, it's never going to take you to that minimum because it takes you towards a local minimum, to point and looks like a minimum, but it's not. How do you overcome this problem? There's actually no secure way to overcome it. But a way to get pretty good results is to run the gradient descent algorithm many times with many different starting points. Once you run it with this, chances are one of them will get you to the minimum or at least to a pretty good point.

Optimization using Gradient Descent in two variable

Earlier we have have seen an okay way to take you closer to the coldest place in the sauna. However, there was a lot of random steps in that algorithm. We're going to do it more exactly and more mathematically just like we did before with one variable, we're going to define gradient sending two variables. It's very similar. You start with an initial position, except now the initial position has two coordinates, x_0 and y_0. Before you used to draw the tangent or the derivative but now we're in two variables, so we have two tangents, this is exactly the same as gradient descent in one variable, except instead of the derivative, we simply take the gradient.



just like gradient descent in one variable grade decent, two or more variables still has the same drawbacks. For example you want to get to one global minimum, but you may accidentally get into some other local minimums

Below is the lab practice

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.The way to overcome this, at least with high probability, is to start at very different places.

Table of Contents

Packages

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

```
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:

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

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!

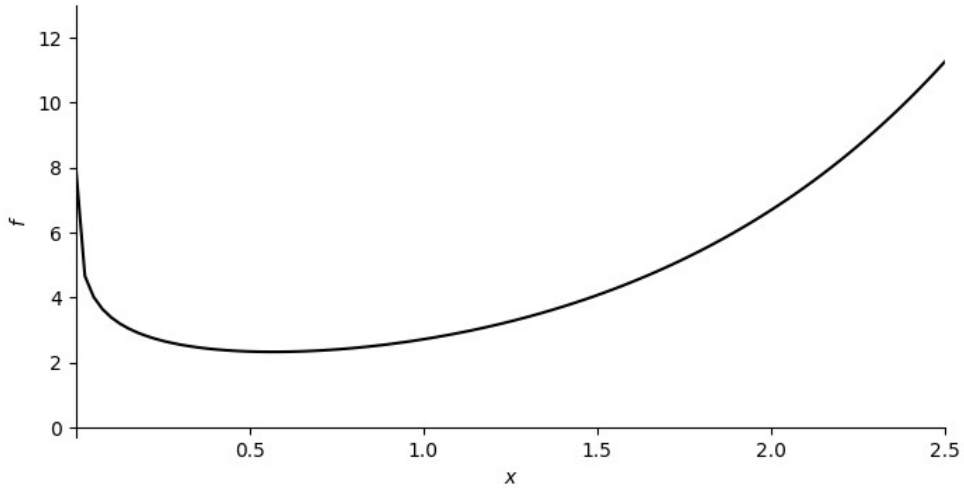
First, define function $f(x) = e^x - \log(x)$ and its derivative $\frac{df}{dx}(x) = e^x - \frac{1}{x}$:

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

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

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

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

```
(<Figure size 800x400 with 1 Axes>, <AxesSubplot: xlabel='$x$',
ylabel='$f$'>)
```



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))
```

```
Gradient descent result: x_min = 0.5671434156768685
```

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.

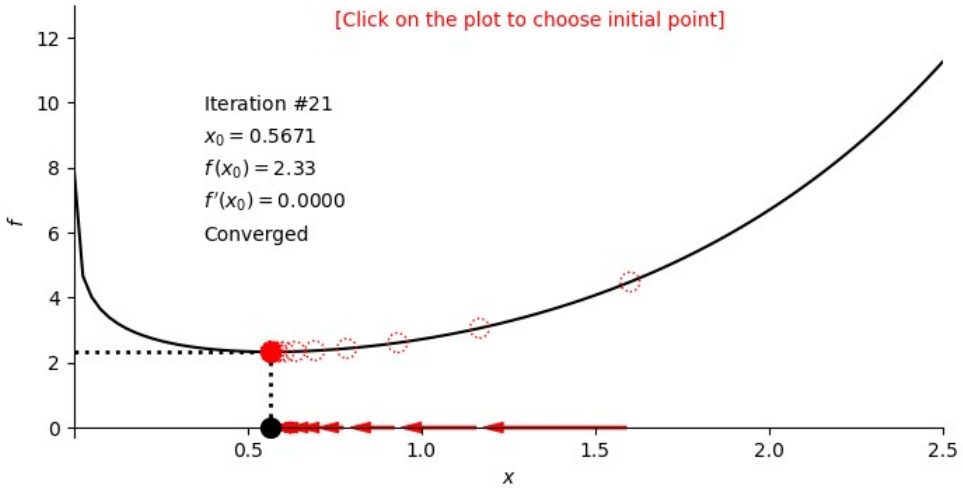
You can see that it works successfully here, bringing it to the global minimum point!

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...

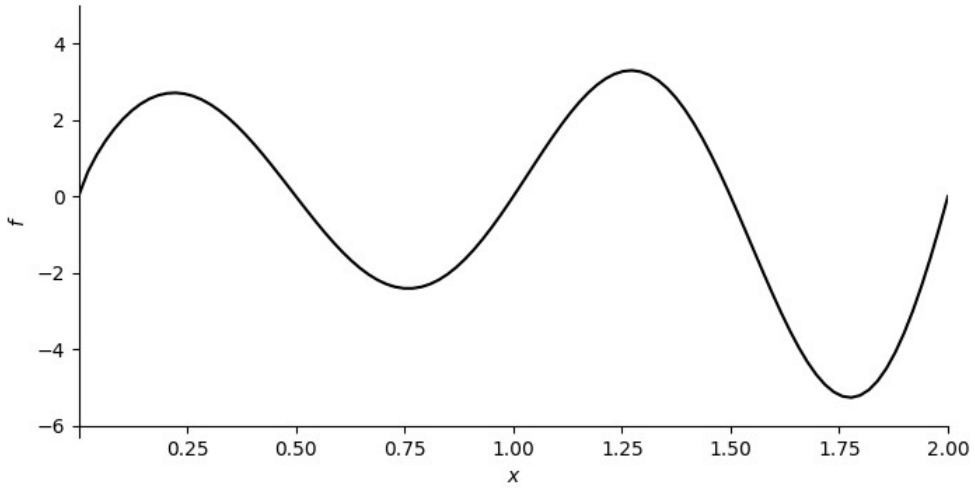
This is a very simple example, but hopefully, it gives you an idea of how important is the choice of the initial parameters.

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)
```

```
(<Figure size 800x400 with 1 Axes>, <AxesSubplot: xlabel='$x$',
ylabel='$f$'>)
```



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.

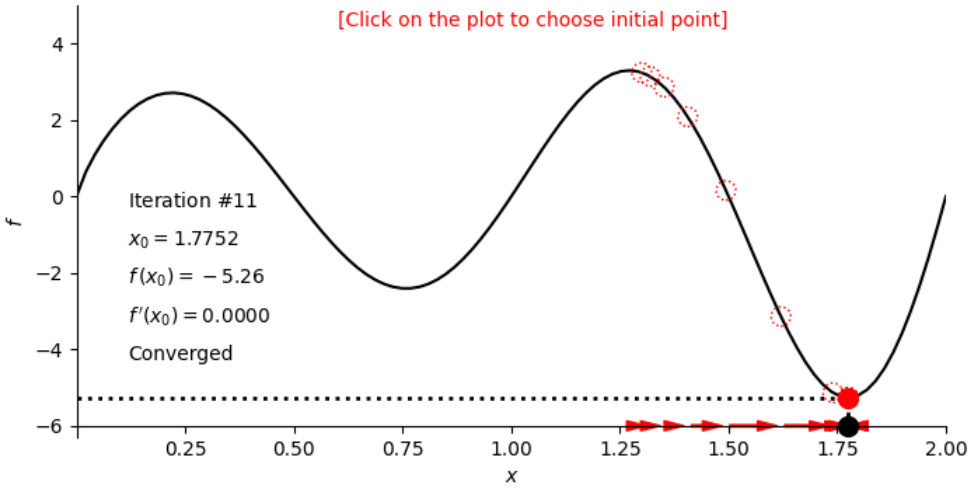
Use the following code to run gradient descent with the same `learning_rate` and `num_iterations`, but with a different starting point:

```
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))
```

```
Gradient descent results
Global minimum: x_min = 1.7751686214270586
Local minimum: x_min = 0.7585728671820583
```

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!

Optimization Using Gradient Descent in Two Variables

In this lab, you will implement and visualize the gradient descent method optimizing some functions in two variables. You will have a chance to experiment with the initial parameters, and investigate the results and limitations of the method.

Table of Contents

Packages

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

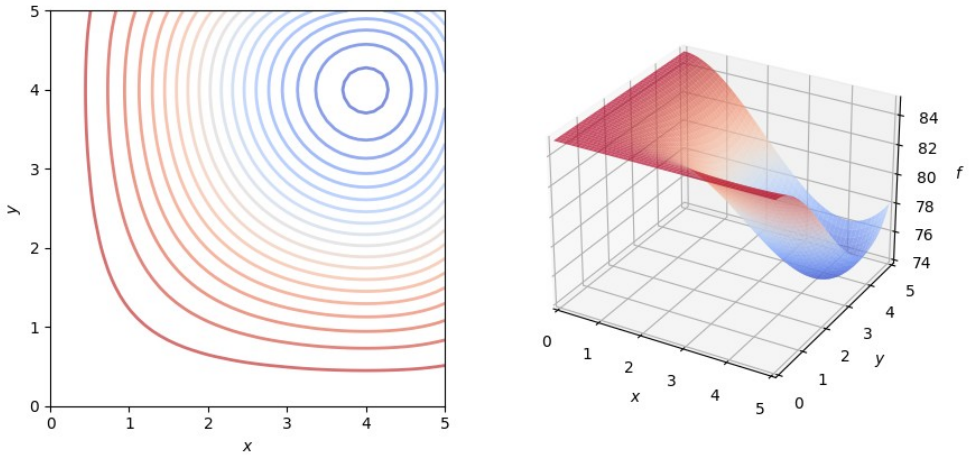
```
import numpy as np
import matplotlib.pyplot as plt
# Some functions defined specifically for this notebook.
from w2_tools import (plot_f_cont_and_surf,
gradient_descent_two_variables,
                      f_example_3, dfdx_example_3, dfdy_example_3,
                      f_example_4, dfdx_example_4, dfdy_example_4)
# Magic command to make matplotlib plots interactive.
%matplotlib widget
```

1 - Function with One Global Minimum

Let's explore a simple example of a function in two variables $f(x, y)$ with one global minimum. Such a function was discussed in the videos, it is predefined and uploaded into this notebook as `f_example_3` with its partial derivatives `dfdx_example_3` and `dfdy_example_3`. At this stage, you do not need to worry about the exact expression for that function and its partial derivatives, so you can focus on the implementation of gradient descent and the choice of the related parameters. Run the following cell to plot the function.

```
plot_f_cont_and_surf([0, 5], [0, 5], [74, 85], f_example_3,
cmap='coolwarm', view={'azim':-60,'elev':28})
```

```
(<Figure size 1000x500 with 2 Axes>,
 <AxesSubplot: xlabel='$x$', ylabel='$y$'>,
 <Axes3DSubplot: xlabel='$x$', ylabel='$y$', zlabel='$f$'>)
```



To find the minimum, you can implement gradient descent starting from the initial point (x_0, y_0) and making steps iteration by iteration using the following equations:

$$x_1 = x_0 - \alpha \frac{\partial f}{\partial x}(x_0, y_0),$$

$$y_1 = y_0 - \alpha \frac{\partial f}{\partial y}(x_0, y_0),$$

where $\alpha > 0$ is a learning rate. Number of iterations is also a parameter. The method is implemented with the following code:

```
def gradient_descent(dfdx, dfdy, x, y, learning_rate = 0.1,
num_iterations = 100):
```

```
    for iteration in range(num_iterations):
        x, y = x - learning_rate * dfdx(x, y), y - learning_rate *
dfdy(x, y)
    return x, y
```

Now to optimize the function, set up the parameters `num_iterations`, `learning_rate`, `x_initial`, `y_initial` and run gradient descent:

```
num_iterations = 30; learning_rate = 0.25; x_initial = 0.5; y_initial
= 0.6
print("Gradient descent result: x_min, y_min =",
      gradient_descent(dfdx_example_3, dfdy_example_3, x_initial,
y_initial, learning_rate, num_iterations))
```

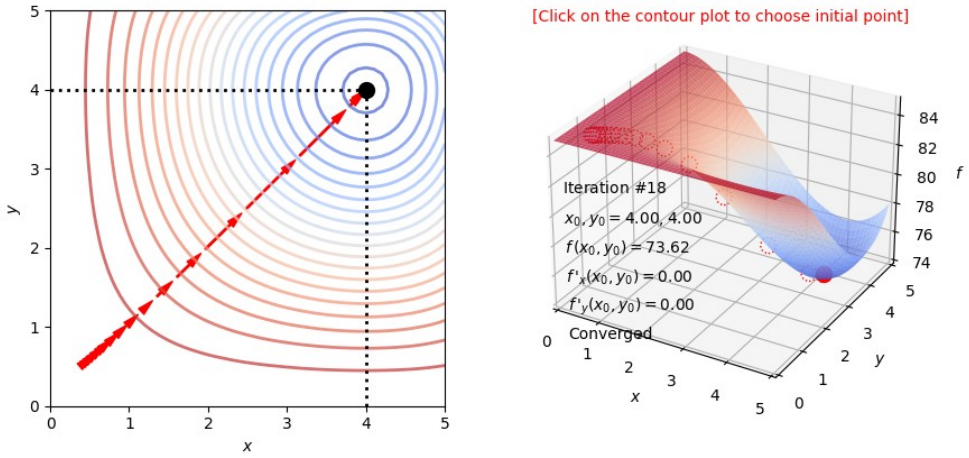
```
Gradient descent result: x_min, y_min = (4.0, 4.0)
```

You can see the visualization running the following code. Note that gradient descent in two variables performs steps on the plane, in a direction opposite to the gradient vector $\begin{bmatrix} \frac{\partial f}{\partial x}(x_0, y_0) \\ \frac{\partial f}{\partial y}(x_0, y_0) \end{bmatrix}$ with the learning rate α as a scaling factor.

By uncommenting different lines you can experiment with various sets of the parameter values and corresponding results. At the end of the animation, you can also click on the contour plot to choose the initial point and restart the animation automatically.

Run a few experiments and try to explain what is actually happening in each of the cases.

```
num_iterations = 20; learning_rate = 0.25; x_initial = 0.5; y_initial
= 0.6
# num_iterations = 20; learning_rate = 0.5; x_initial = 0.5; y_initial
= 0.6
# num_iterations = 20; learning_rate = 0.15; x_initial = 0.5;
y_initial = 0.6
# num_iterations = 20; learning_rate = 0.15; x_initial = 3.5;
y_initial = 3.6

gd_example_3 = gradient_descent_two_variables([0, 5], [0, 5], [74,
85],
                                              f_example_3,
dfdx_example_3, dfdy_example_3,
                                              gradient_descent,
num_iterations, learning_rate,
                                              x_initial, y_initial,
                                              [0.1, 0.1, 81.5], 2, [4,
1, 171],
                                              cmap='coolwarm',
view={'azim':-60,'elev':28})
```

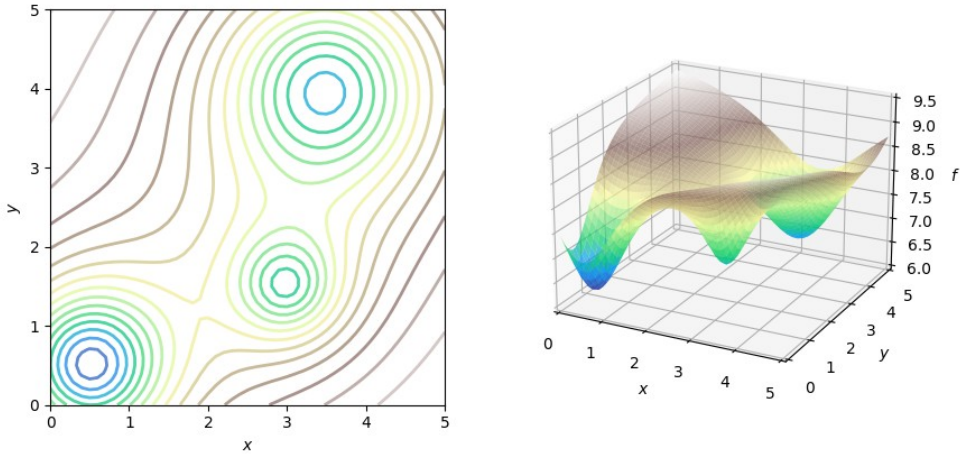


2 - Function with Multiple Minima

Let's investigate a more complicated case of a function, which was also shown in the videos:

```
plot_f_cont_and_surf([0, 5], [0, 5], [6, 9.5], f_example_4,
cmap='terrain', view={'azim':-63,'elev':21})
```

```
(<Figure size 1000x500 with 2 Axes>,
 <AxesSubplot: xlabel='$x$', ylabel='$y$'>,
 <Axes3DSubplot: xlabel='$x$', ylabel='$y$', zlabel='$f$'>)
```



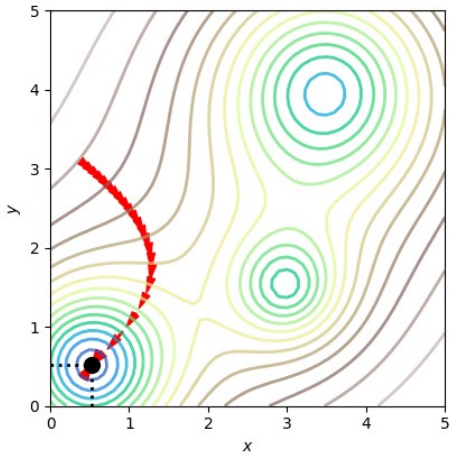
You can find its global minimum point by using gradient descent with the following parameters:

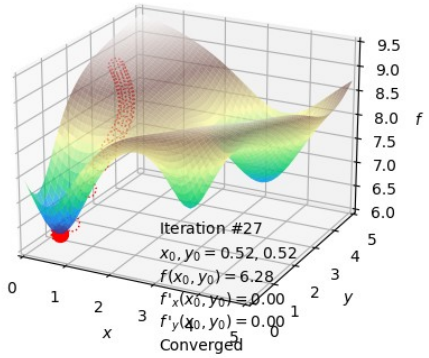
```
num_iterations = 100; learning_rate = 0.2; x_initial = 0.5; y_initial
= 3

print("Gradient descent result: x_min, y_min =",
      gradient_descent(dfdx_example_4, dfdy_example_4, x_initial,
y_initial, learning_rate, num_iterations))
```

```
Gradient descent result: x_min, y_min = (0.5230322579358745,
0.5169891562802605)
```

However, the shape of the surface is much more complicated and not every initial point will bring you to the global minimum of this surface. Use the following code to explore various sets of parameters and the results of gradient descent.

```
# Converges to the global minimum point.
num_iterations = 30; learning_rate = 0.2; x_initial = 0.5; y_initial =
3
# Converges to a local minimum point.
# num_iterations = 20; learning_rate = 0.2; x_initial = 2; y_initial =
3
# Converges to another local minimum point.
# num_iterations = 20; learning_rate = 0.2; x_initial = 4; y_initial =
0.5

gd_example_4 = gradient_descent_two_variables([0, 5], [0, 5], [6,
9.5],
                                              f_example_4,
dfdx_example_4, dfdy_example_4,
                                              gradient_descent,
num_iterations, learning_rate,
                                              x_initial, y_initial,
                                              [2, 2, 6], 0.5, [2, 1,
63],
                                              cmap='terrain',
view={'azim':-63,'elev':21})
```


5
4
3
2
1
0
y
0
1
2
3
4
5
x


9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
f
Iteration #27
$x_0, y_0 = 0.52, 0.52$
$f(x_0, y_0) = 6.28$
$f'_x(x_0, y_0) = 0.00$
$f'_y(x_0, y_0) = 0.00$
Converged
0
1
2
3
4
5
x
y