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

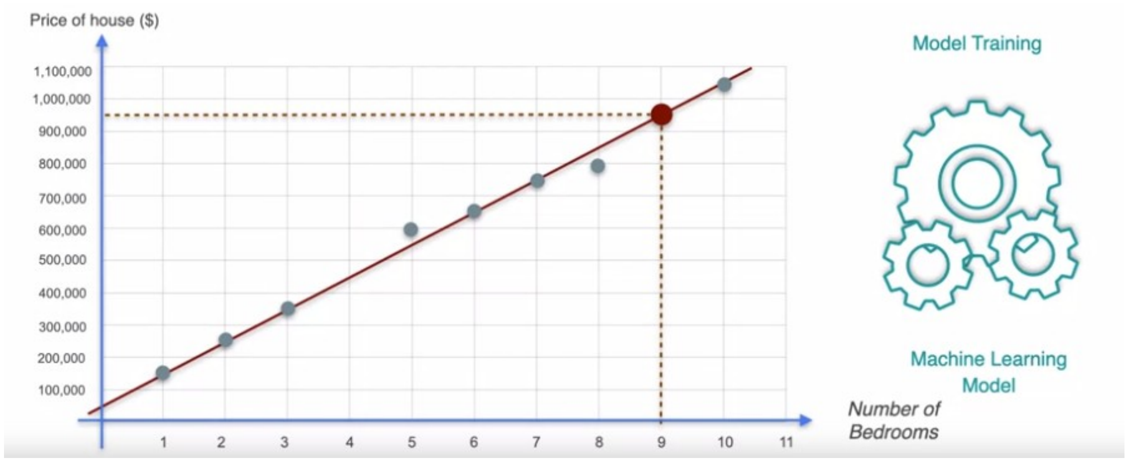
Week-1 of Course 2-Calculus for Machine Learning and Data Science

In this week I learnt

- A simple ML model
- Derivatives and Tangents/Slopes, maxima and minima
- Existence of the derivative
- Optimization

A simple ML model

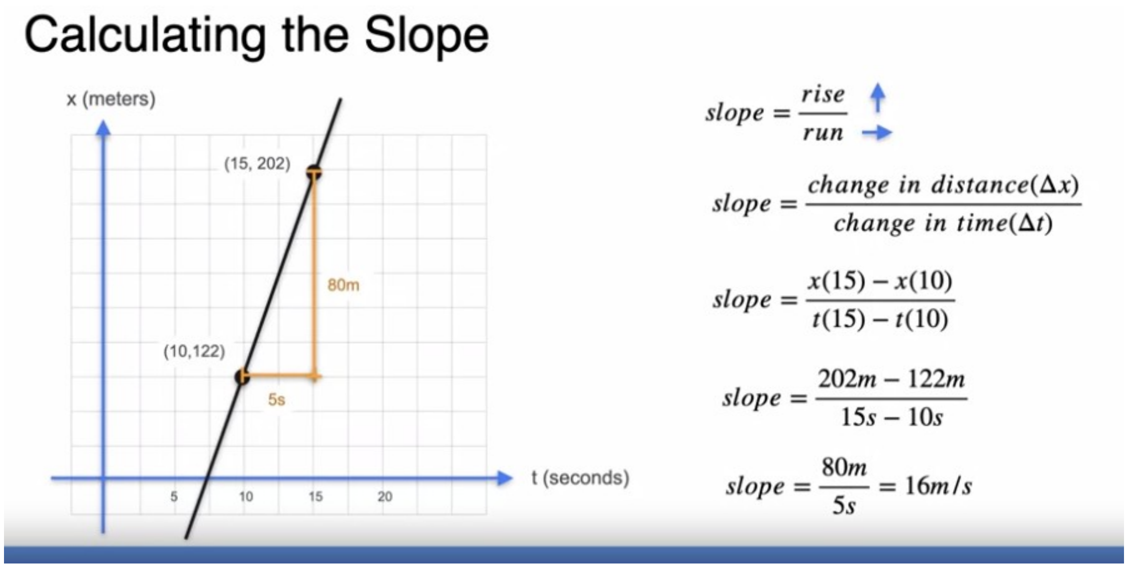
why are derivatives and calculus so important in machine learning? Derivatives are used to optimize functions in particular to maximize them and minimize them. That means finding the maximum value or the minimum value of a function. And this is very important in the machine learning. The reason is that when you want to find the model that fits your data in the best possible way you do this by calculating a loss function and minimizing it.Two of the most important loss functions in machine learning is The SQUARE LOSS and the LOG LOSS. Very important and very basic functions in mathematics, such as a constant function, linear functions, quadratic polynomial, exponential functions and logarithmic functions are important in machine learning. Below is one graph of no.of bedrooms vs price of the house and we have to predict the price of the house with 9 bedrooms to say.Lets build a machine learning model to help you predict the price of the house with nine bedrooms.The houses are here represented by dots.The machine learning model takes in all the input data of the house prices and begins a process called model training. Ml model acts as smart engine that always look out always looking for ways to optimize in order to produce the best possible result.In this case,a line that goes as close as possible to all the points will be our model which will represent our predicted price. when model training begins ,it starts with any random line. And the idea is that it tweaks the results in order to optimize the best possible prediction for the existing data points.And once training is done, the model is able to give you a good prediction for the house with nine bedrooms. So in this case the house with nine bedrooms is predicted to have a price of $950,000.This kind of problem is called LINEAR REGRESSION PROBLEM



Another model is classification problem that is used in sentiment analysis.The mathematics concepts used underhood almost all ML model are gradients, derivatives, optimization, loss and cost functions, grading descent and many, many more.

Derivatives and Tangents/Slopes, maxima and minima

A derivative is the instantaneous rate of change of a function. For example distance is a function and velocity is its derivative.velocity is calculated with the formula, distance over time. This is synonymous to the formula for calculating slope, which is rise over run. velocity is also represented by slope.The slope of the line that passes these two points is 16, which also translates to the velocity of the car between those two points is 16 meters per second



$$slope = \frac{rise}{run}$$

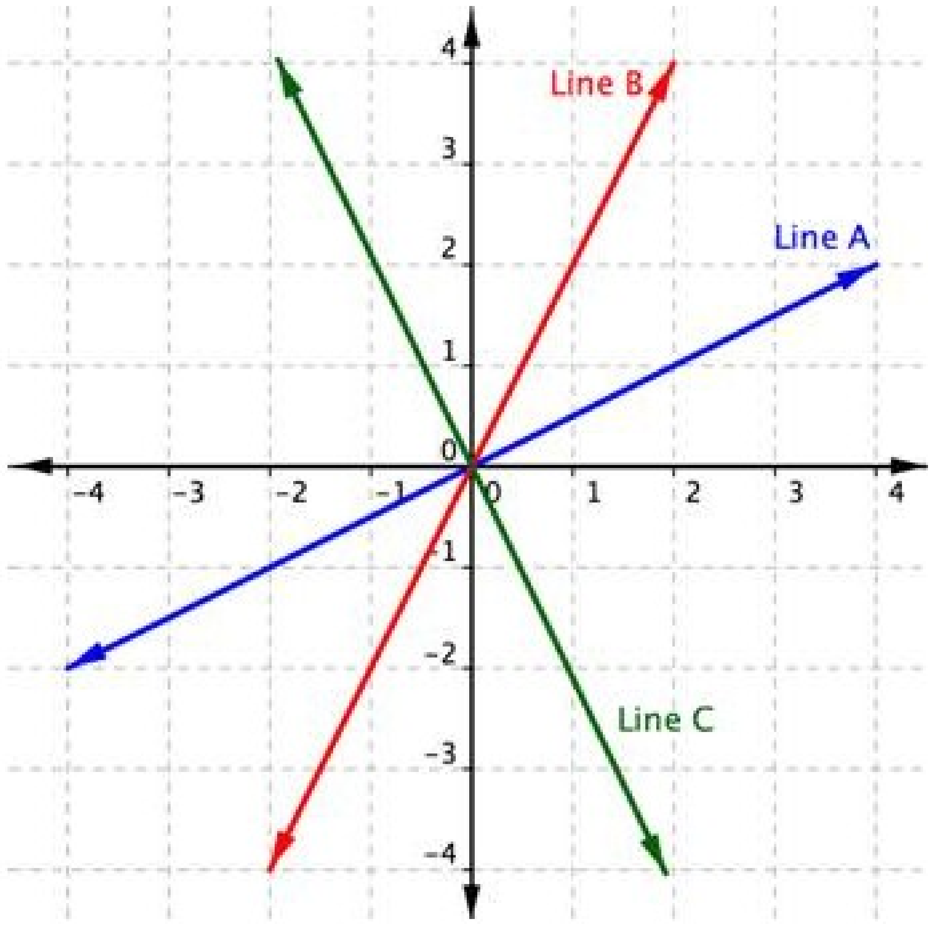
$$slope = \frac{change\ in\ distance(\Delta x)}{change\ in\ time(\Delta t)}$$

$$slope = \frac{x(15) - x(10)}{t(15) - t(10)}$$

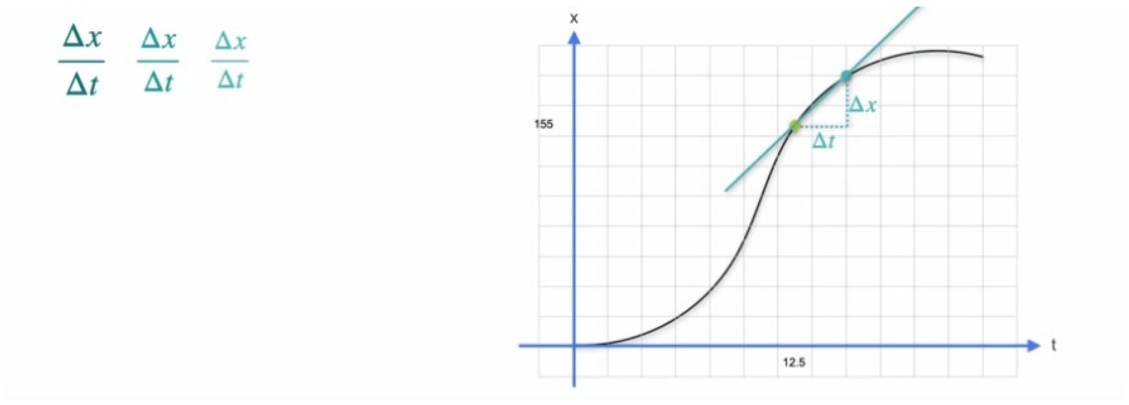
$$slope = \frac{202m - 122m}{15s - 10s}$$

$$slope = \frac{80m}{5s} = 16m/s$$

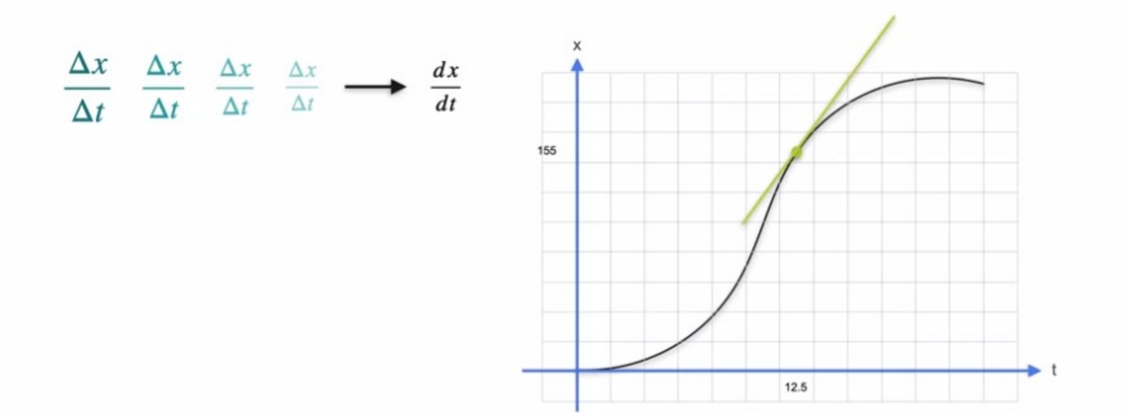
slope is used to describe the steepness and direction of lines.In below image line B is steeper than line A so line B has a greater slope than line A



So far we learned how to calculate the average velocity of an interval, but what is the instantaneous velocity at a point? In below image . Let's look at the point t equals 12.5 that you were looking at before. Calculating the instantaneous velocity here may be hard, but estimating it is possible. For example, let's take another point to the right of t equals 12.5 and calculate the average velocity over that interval. That is the slope of this line over here, which is exactly Delta x, the change in distance divided by Delta t, the change in time. But that is not the instantaneous velocity at t equals 12.5. However, you can get closer to it by making this interval smaller, for example, take now this point over here closer to t equals 12.5 and the average velocity of that interval is the new Delta x over the new Delta t, which is a slope of these new line. Now let's put the point even closer to 12.5 and even closer. Now imagine putting it so close that you can't even tell the distance between the two. You get the limit which is dx over dt and that is precisely the tangent line to the curve at t equals 12.5. This measure of how fast the distance is changing with respect to time is called the instantaneous rate of change and it is the slope of that tangent line.



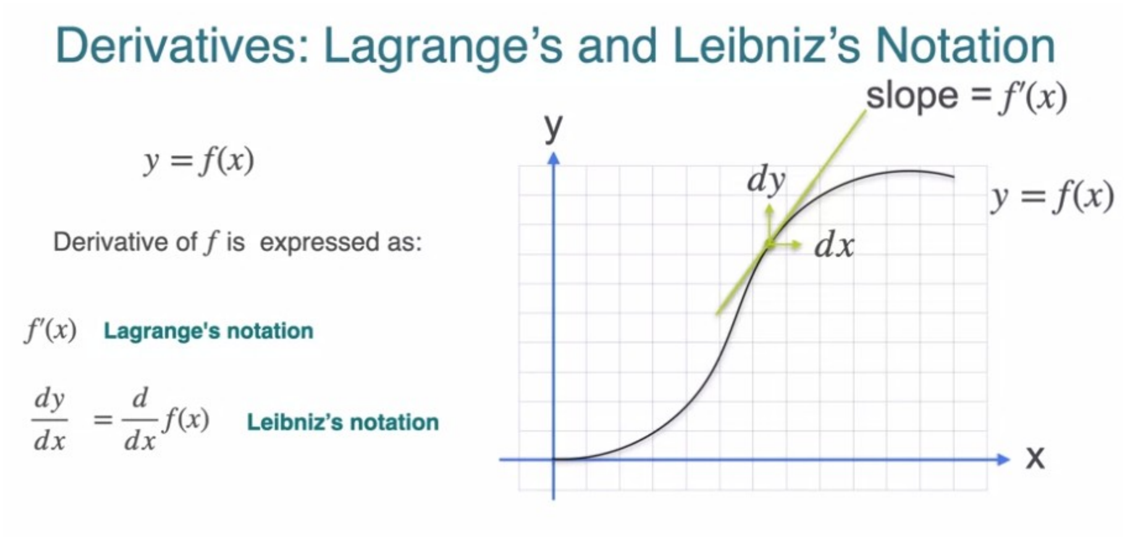
$\frac{\Delta x}{\Delta t}$ $\frac{\Delta x}{\Delta t}$ $\frac{\Delta x}{\Delta t}$
x
155
Δx
Δt
t
12.5



$\frac{\Delta x}{\Delta t}$ $\frac{\Delta x}{\Delta t}$ $\frac{\Delta x}{\Delta t}$ $\frac{\Delta x}{\Delta t}$ $\longrightarrow$ $\frac{dx}{dt}$
x
155
t
12.5

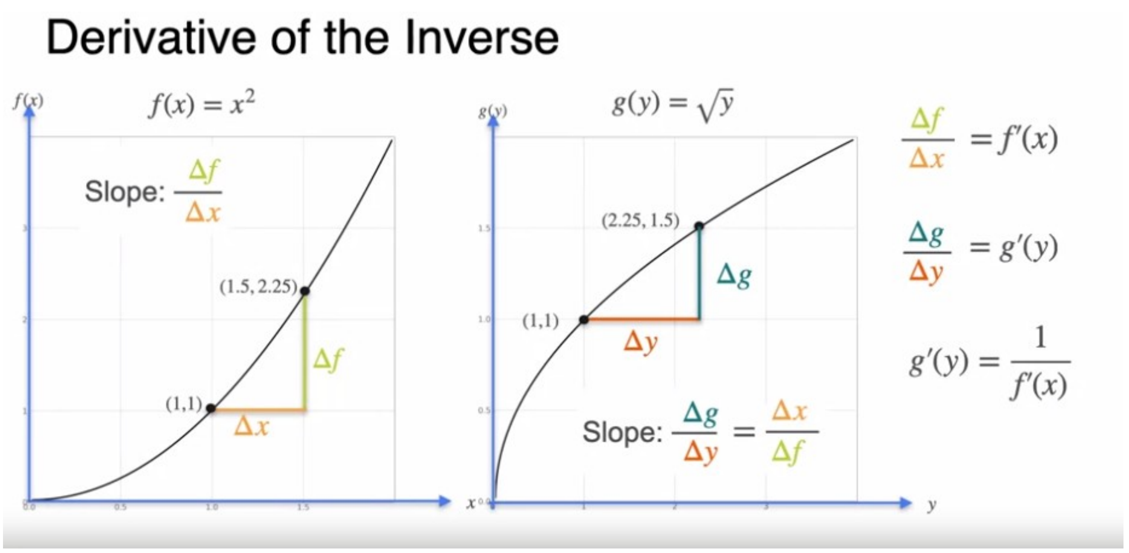
More generally, the instantaneous rate of change is a measure of how fast the relation between two variables is changing at any point.In other words, imagine that you move a tiny distance, dx in a tiny interval of time dt. This is the instantaneous rate of change. It's also called the derivative. The derivative of a function at a point is precisely the slope of the tangent at that particular point. ONe important thing we can think is that how we can compare how much the approximation of derivative $\Delta f/\Delta x$ differs from the exact derivative df/dx, depending on the size of Δx.

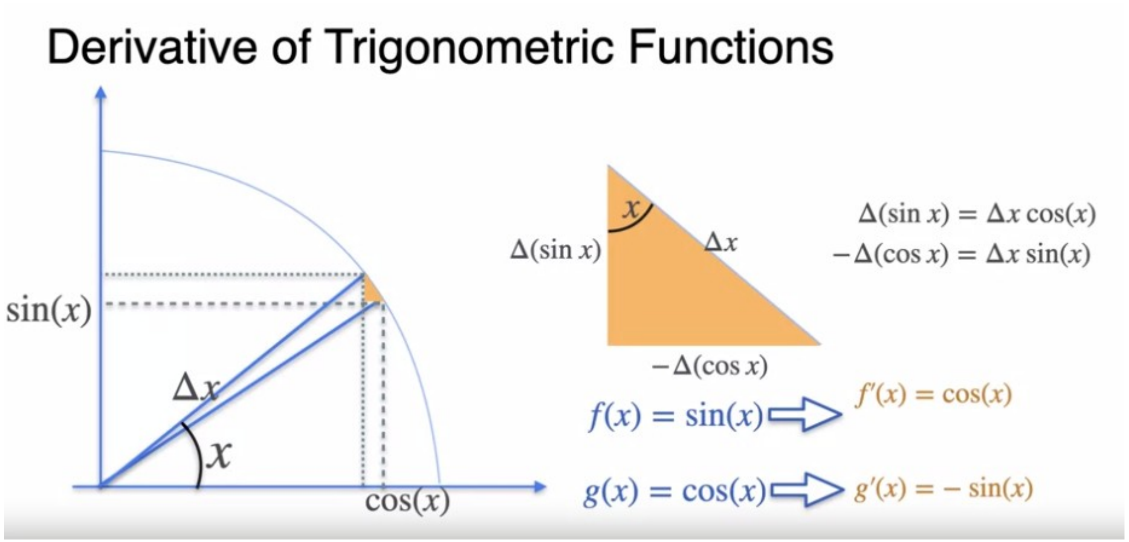
Horizontal line has a slope of zero because there is no rise in distance. At any of the points where the tangent is horizontal, therefore, the tangent has Slope 0 which means at those points no distance was covered.If you want to find the maximum or the minimum in a function, it occurs at one of the points where the derivative is zero. Or in other words, where the tangent line is horizontal.


Derivatives: Lagrange's and Leibniz's Notation
$y = f(x)$
Derivative of f is expressed as:
$f'(x)$ Lagrange's notation
$\frac{dy}{dx} = \frac{d}{dx}f(x)$ Leibniz's notation
slope $= f'(x)$
y
dy
dx
$y = f(x)$
x

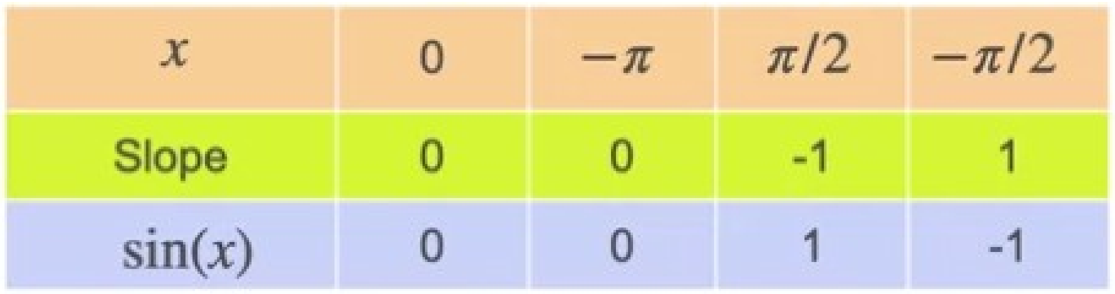
Note that dx/dt is called the infinitesimal changes in the x direction and in the t direction.

INVERSE FUNCTION DERIVATIVE: if your function does a certain thing, then the inverse function is the one that undoes that thing. So for example, if the function f, sends the number 3 to the number 5, then the inverse function sends the number 5 back to the number 3.


Derivative of the Inverse
$f(x) = x^2$
Slope: $\frac{\Delta f}{\Delta x}$
(1.5, 2.25)
Δf
(1,1)
Δx
$g(y) = \sqrt{y}$
(2.25, 1.5)
Δg
(1,1)
Δy
Slope: $\frac{\Delta g}{\Delta y} = \frac{\Delta x}{\Delta f}$
$\frac{\Delta f}{\Delta x} = f'(x)$
$\frac{\Delta g}{\Delta y} = g'(y)$
$g'(y) = \frac{1}{f'(x)}$

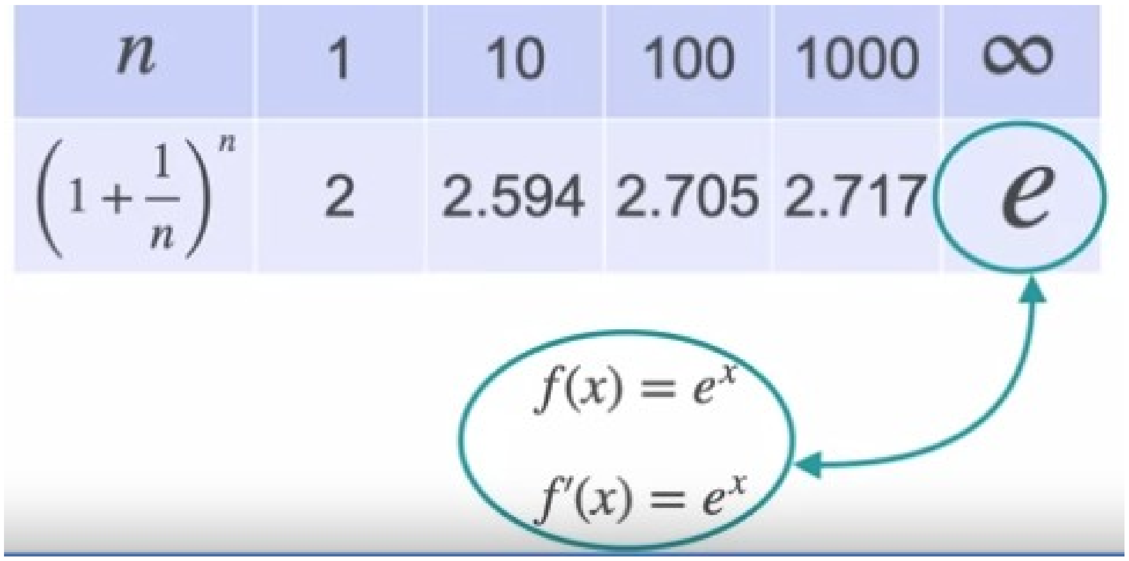


.

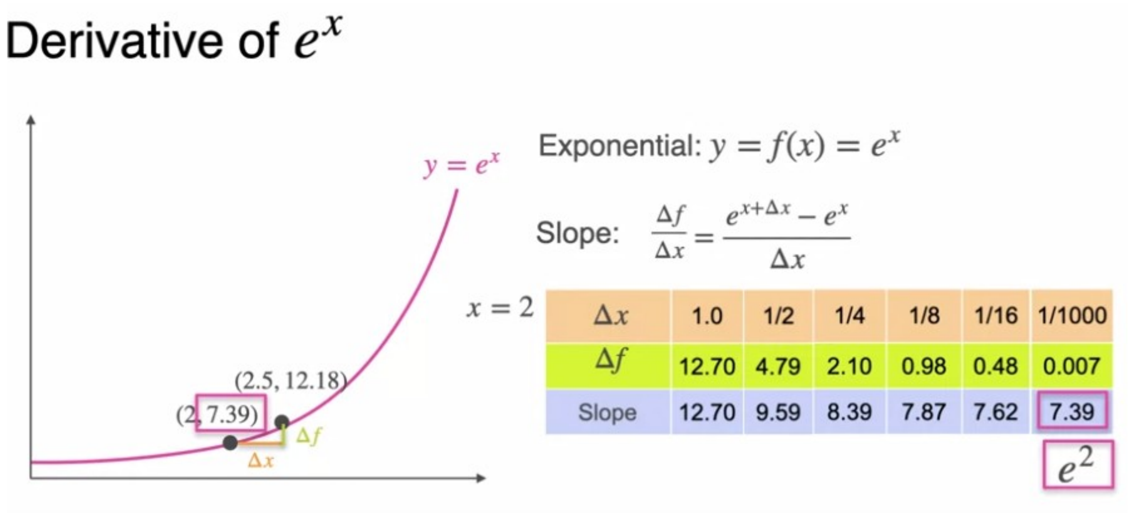


x	0	$-\pi$	$\pi/2$	$-\pi/2$
Slope	0	0	-1	1
$\sin(x)$	0	0	1	-1

EXPONENTIAL:Euler's number is quite a special number of mathematics. It appears in many branches and it can be defined in many ways. One of them is by the numerical value 2.71828182,also known as "e" or exponential etc. The decimal never terminates because a number is irrational, which means it cannot be expressed as a ratio between two integers.



n	1	10	100	1000	∞
$\left(1+\frac{1}{n}\right)^n$	2	2.594	2.705	2.717	e



Derivative of e^x

Exponential: $y = f(x) = e^x$

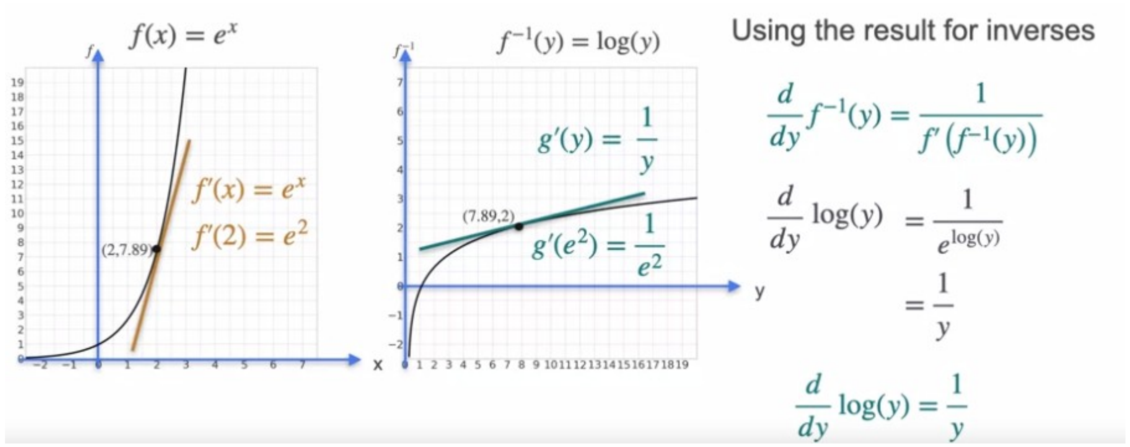
Slope: $\frac{\Delta f}{\Delta x} = \frac{e^{x+\Delta x} - e^x}{\Delta x}$

$x = 2$

Δx	1.0	1/2	1/4	1/8	1/16	1/1000
Δf	12.70	4.79	2.10	0.98	0.48	0.007
Slope	12.70	9.59	8.39	7.87	7.62	$\boxed{7.39}$

$\boxed{e^2}$

LOGARITHMIC: It is the inverse of exponential.



Using the result for inverses

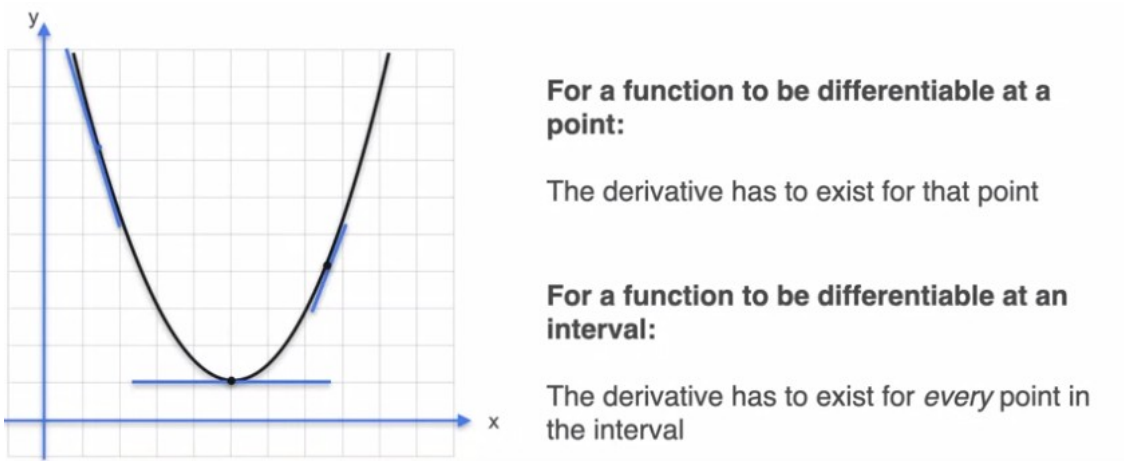
$$\frac{d}{dy} f^{-1}(y) = \frac{1}{f'\left(f^{-1}(y)\right)}$$

$$\frac{d}{dy} \log(y) = \frac{1}{e^{\log(y)}} = \frac{1}{y}$$

$$\frac{d}{dy} \log(y) = \frac{1}{y}$$

Existence of the derivative

So far we learned about some common functions and their derivatives. However, it's not always the case that you can find the derivative of a function at every point. For these functions where you cannot find the derivative at every point are called non differentiable functions.We know that to find the derivative of a function at a point, you draw a tangent line to the point on that line and find the slope to the tangent.

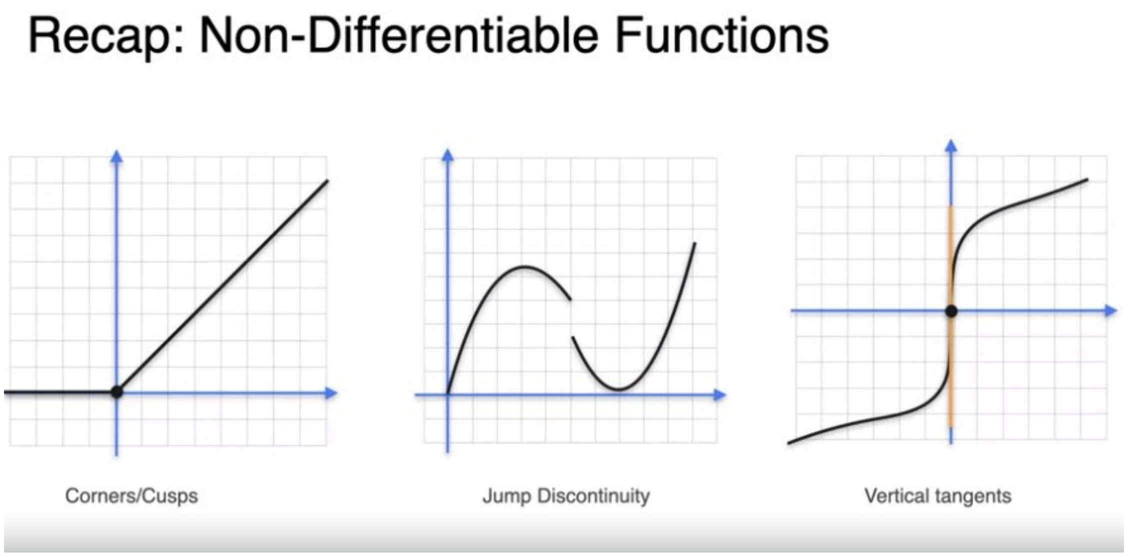


For a function to be differentiable at a point:

The derivative has to exist for that point

For a function to be differentiable at an interval:

The derivative has to exist for *every* point in the interval



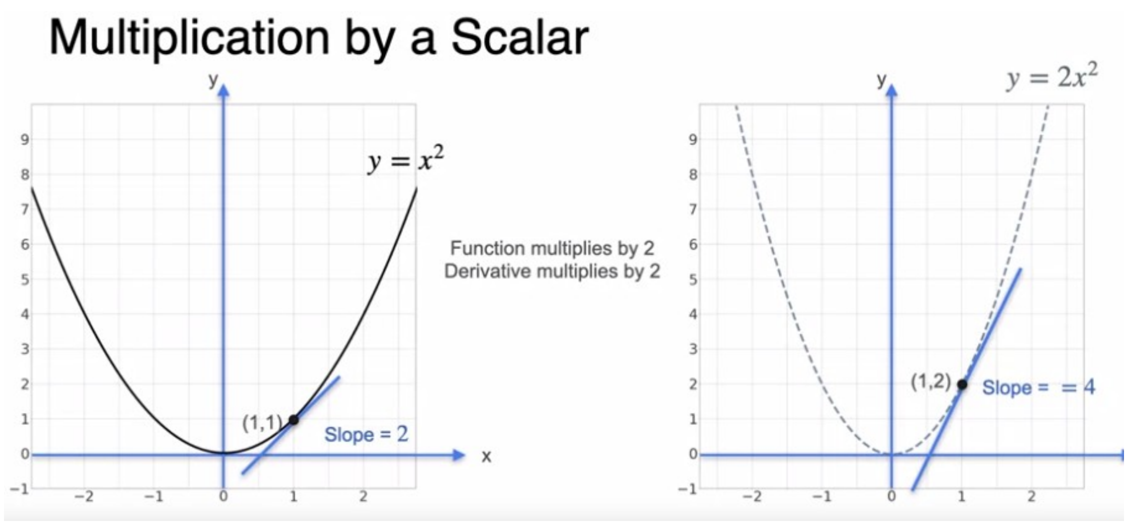
Recap: Non-Differentiable Functions

Corners/Cusps

Jump Discontinuity

Vertical tangents

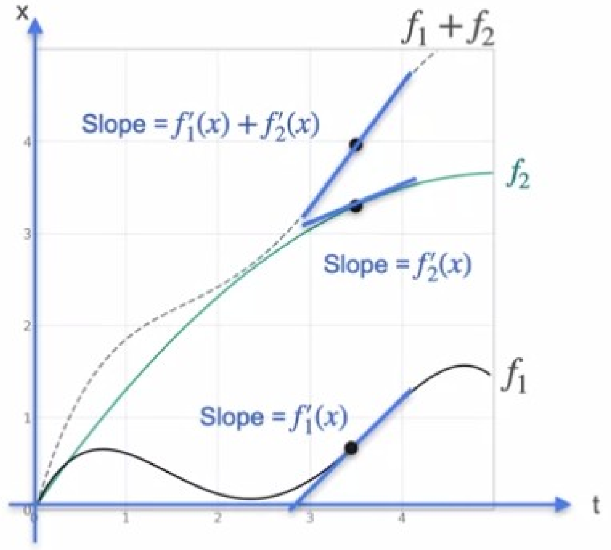
VERTICAL TANGENTS ALSO ARE NOT DIFFERENTIABLE because a vertical line do not have a well defined slope, it would be some number over 0, because the rise is any number, but the run is 0.



Multiplication by a Scalar

Function multiplies by 2
Derivative multiplies by 2

Sum Rule

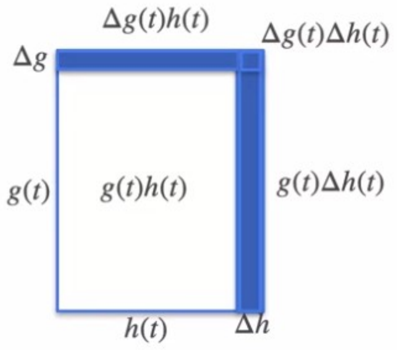


$$f = f_1 + f_2$$

$$\Downarrow$$

$$f' = f_1' + f_2'$$

Product Rule



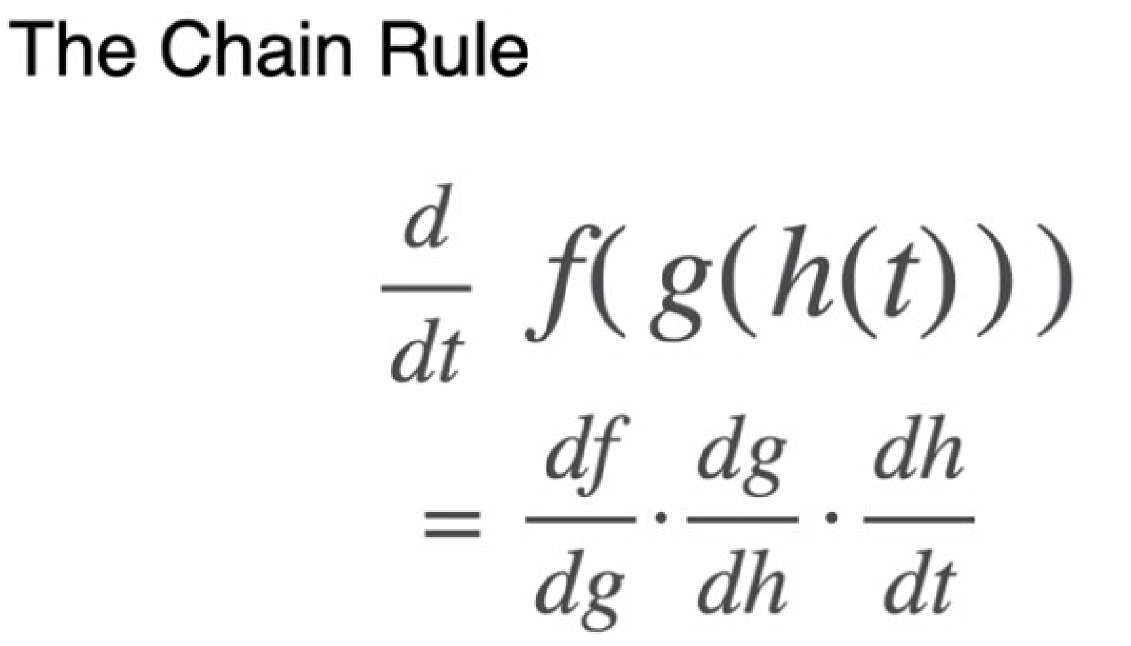
$$y = f(t) = g(t)h(t)$$

$$\overset{f'(t)}{\frac{\Delta f(t)}{\Delta t}} = \frac{\Delta g(t)h(t) + g(t)\Delta h(t) + \Delta g(t)\Delta h(t)}{\Delta t}$$

$$= \overset{g'(t)h(t)}{\frac{\Delta g(t)}{\Delta t}h(t)} + \overset{g(t)h'(t)}{g(t)\frac{\Delta h(t)}{\Delta t}} + \overset{0}{\frac{\Delta g(t)\Delta h(t)}{\Delta t}}$$

as $\Delta t \longrightarrow 0$

$$f'(t) = g'(t)h(t) + g(t)h'(t)$$

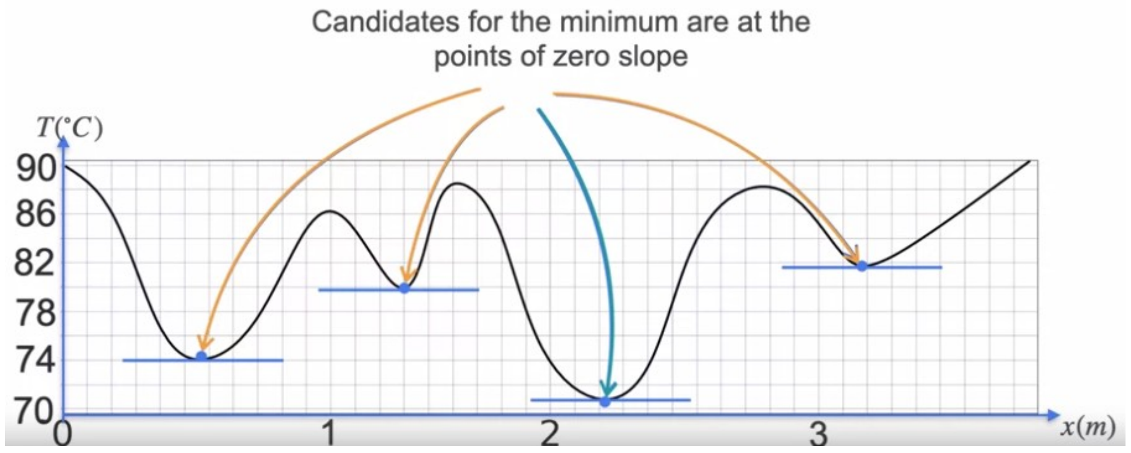


The Chain Rule

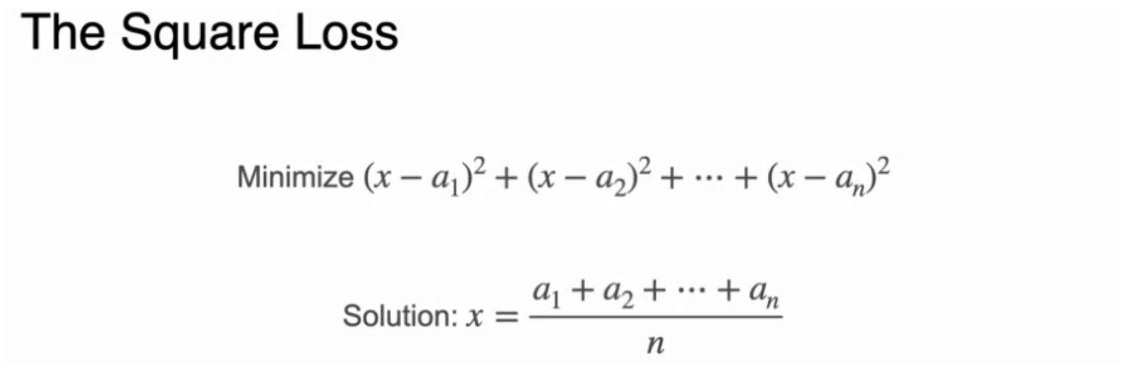
$$\frac{d}{dt} f(g(h(t))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dt}$$

Optimization

By now we have lots of tools to work with derivatives. But the question is, what are they useful for, other than calculating rates of change and more specifically, why are they useful in machine learning? Well, the main application here in machine learning of derivatives is that they are used for optimization. Optimization is when you want to find the maximum or the minimum value of a function. This is very important in machine learning because in machine learning, you want to find the model that best fits your dataset, and in order to find this model what you do is you calculate an error function that tells you how far are you from an ideal model. When you are able to minimize this error function then you have the best model.In general the slope at maximum and minimum is zero however thats not always true.There could be multiple minima which are called local minima .The candidates for the minimum are the points of zero slope and those are called local minima, and the absolute minimum is called the global minimum. Look the image below.



When you want to optimize a function whether maximizing or minimizing it and the function is differentiable at every point, then you know one thing, it's that the candidates for maximum and minimum are those points for which the derivative is zero.

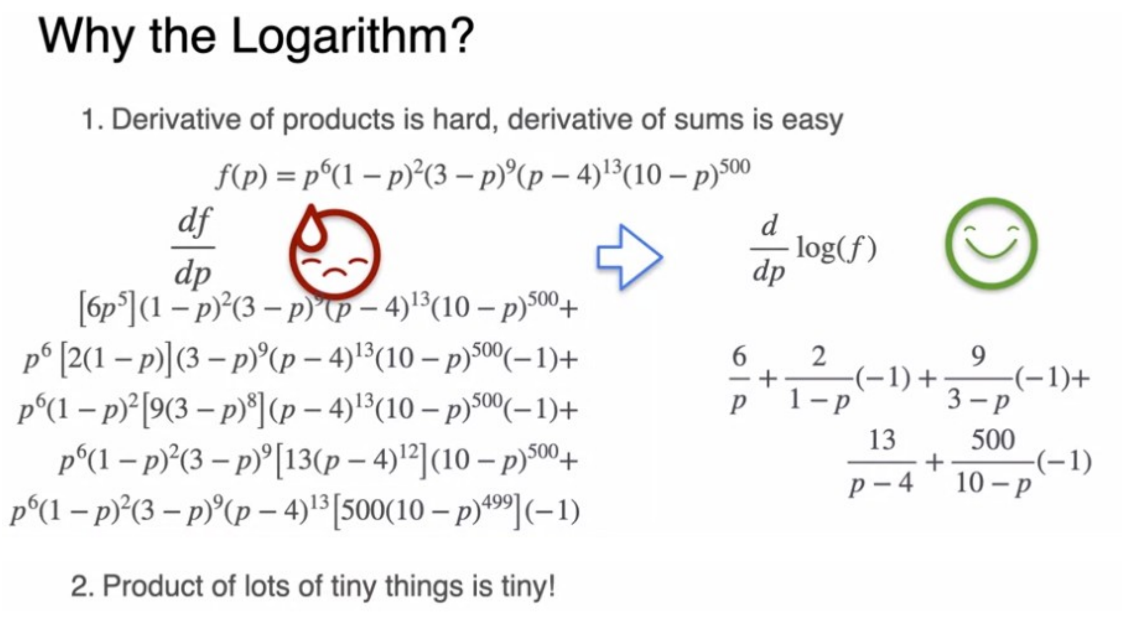


The Square Loss

$$\text{Minimize } (x-a_1)^2 + (x-a_2)^2 + \cdots + (x-a_n)^2$$

$$\text{Solution: } x = \frac{a_1 + a_2 + \cdots + a_n}{n}$$

Square loss is very important loss function in machine learning. Squared error is one of the most important functions in machine learning. It is used to train linear regression problems, and certain neural networks.Another widely used optimization method is LOG LOSS function.



Why the Logarithm?

1. Derivative of products is hard, derivative of sums is easy

$$f(p) = p^6(1-p)^2(3-p)^9(p-4)^{13}(10-p)^{500}$$

$$\frac{df}{dp}$$

$$\begin{aligned}
&[6p^5](1-p)^2(3-p)^9(p-4)^{13}(10-p)^{500} + \\
&p^6\,[2(1-p)](3-p)^9(p-4)^{13}(10-p)^{500}(-1) + \\
&p^6(1-p)^2[9(3-p)^8](p-4)^{13}(10-p)^{500}(-1) + \\
&p^6(1-p)^2(3-p)^9[13(p-4)^{12}](10-p)^{500} + \\
&p^6(1-p)^2(3-p)^9(p-4)^{13}[500(10-p)^{499}](-1)
\end{aligned}$$

$$\frac{d}{dp}\log(f)$$

$$\frac{6}{p} + \frac{2}{1-p}(-1) + \frac{9}{3-p}(-1) + \frac{13}{p-4} + \frac{500}{10-p}(-1)$$

2. Product of lots of tiny things is tiny!

Differentiation in Python: Symbolic, Numerical and Automatic

In this lab you explore which tools and libraries are available in Python to compute derivatives. You will perform symbolic differentiation with `SymPy` library, numerical with `NumPy` and automatic with `JAX` (based on `Autograd`). Comparing the speed of calculations, you will investigate the computational efficiency of those three methods.

Table of Contents

1 - Functions in Python

This is just a reminder how to define functions in Python. A simple function $f(x)=x^2$, it can be set up as:

```python
def f(x):
    return x**2

print(f(3))
```

```
9
```

You can easily find the derivative of this function analytically. You can set it up as a separate function:

```python
def dfdx(x):
    return 2*x

print(dfdx(3))
```

```
6
```

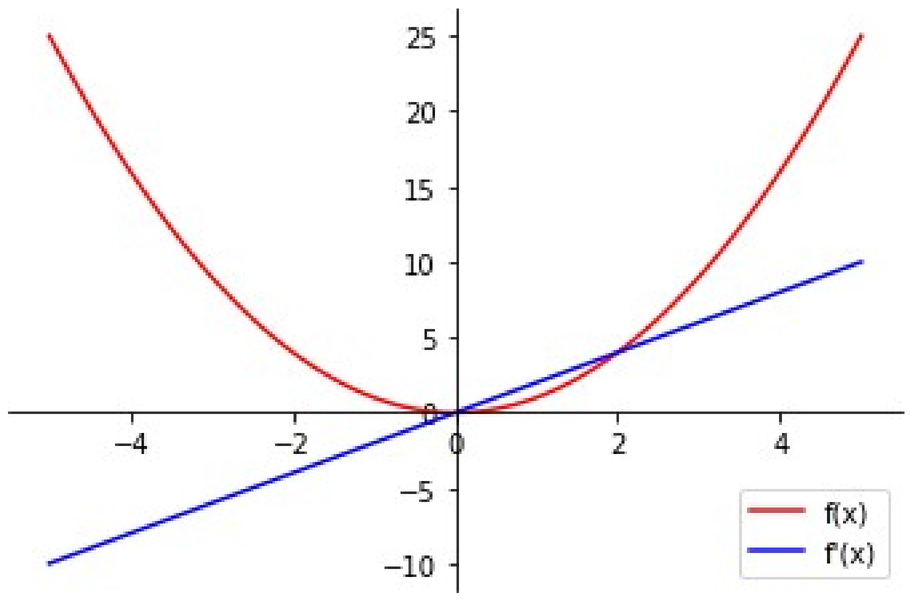
Since you have been working with the `NumPy` arrays, you can apply the function to each element of an array:

```python
import numpy as np

x_array = np.array([1, 2, 3])

print("x: \n", x_array)
print("f(x) = x**2: \n", f(x_array))
print("f'(x) = 2x: \n", dfdx(x_array))
```

```
x:
 [1 2 3]
f(x) = x**2:
 [1 4 9]
f'(x) = 2x:
 [2 4 6]
```

Now you can apply those functions `f` and `dfdx` to an array of a larger size. The following code will plot function and its derivative (you don't have to understand the details of the `plot_f1_and_f2` function at this stage):

```
import matplotlib.pyplot as plt

# Output of plotting commands is displayed inline within the Jupyter
notebook.
%matplotlib inline

def plot_f1_and_f2(f1, f2=None, x_min=-5, x_max=5, label1="f(x)",
label2="f'(x)"):
    x = np.linspace(x_min, x_max,100)

    # Setting the axes at the centre.
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
    ax.spines['left'].set_position('center')
    ax.spines['bottom'].set_position('zero')
    ax.spines['right'].set_color('none')
    ax.spines['top'].set_color('none')
    ax.xaxis.set_ticks_position('bottom')
    ax.yaxis.set_ticks_position('left')

    plt.plot(x, f1(x), 'r', label=label1)
    if not f2 is None:
        # If f2 is an array, it is passed as it is to be plotted as
unlinked points.
        # If f2 is a function, f2(x) needs to be passed to plot it.

        if isinstance(f2, np.ndarray):
            plt.plot(x, f2, 'bo', markersize=3, label=label2,)
        else:
            plt.plot(x, f2(x), 'b', label=label2)
    plt.legend()

    plt.show()

plot_f1_and_f2(f, dfdx)
```



In real life the functions are more complicated and it is not possible to calculate the derivatives analytically every time. Let's explore which tools and libraries are available in Python for the computation of derivatives without manual derivation.

2 - Symbolic Differentiation

Symbolic computation deals with the computation of mathematical objects that are represented exactly, not approximately (e.g. $\sqrt{2}$ will be written as it is, not as 1.41421356237). For differentiation it would mean that the output will be somehow similar to if you were computing derivatives by hand using rules (analytically). Thus, symbolic differentiation can produce exact derivatives.

2.1 - Introduction to Symbolic Computation with SymPy

Let's explore symbolic differentiation in Python with commonly used `SymPy` library.

If you want to compute the approximate decimal value of $\sqrt{18}$, you could normally do it in the following way:

```
import math
```

```
math.sqrt(18)
```

```
4.242640687119285
```

The output 4.242640687119285 is an approximate result. You may recall that $\sqrt{18}=\sqrt{9\cdot 2}=3\sqrt{2}$ and see that it is pretty much impossible to deduct it from the approximate result. But with the symbolic computation systems the roots are not approximated with a decimal number but rather only simplified, so the output is exact:

```
# This format of module import allows to use the sympy functions
without sympy. prefix.
from sympy import *

# This is actually sympy.sqrt function, but sympy. prefix is omitted.
sqrt(18)
```

```
3*sqrt(2)
```

Numerical evaluation of the result is available, and you can set number of the digits to show in the approximated output:

```
N(sqrt(18),8)
```

```
4.2426407
```

In SymPy variables are defined using **symbols**. In this particular library they need to be predefined (a list of them should be provided). Have a look in the cell below, how the symbolic expression, correspoinding to the mathematical expression $2x^2 - xy$, is defined:

```
# List of symbols.
x, y = symbols('x y')
# Definition of the expression.
expr = 2 * x**2 - x * y
expr
```

```
2*x**2 - x*y
```

Now you can perform various manipulations with this expression: add or subtract some terms, multiply by other expressions etc., just like if you were doing it by hands:

```
expr_manip = x * (expr + x * y + x**3)
expr_manip
```

```
x*(x**3 + 2*x**2)
```

You can also expand the expression:

```
expand(expr_manip)
```

```
x**4 + 2*x**3
```

Or factorise it:

```
factor(expr_manip)
```

```
x**3*(x + 2)
```

To substitute particular values for the variables in the expression, you can use the following code:

```
expr.evalf(subs={x:-1, y:2})
```

```
4.00000000000000
```

This can be used to evaluate a function $f(x)=x^2$:

```
f_symb = x ** 2
f_symb.evalf(subs={x:3})
```

```
9.00000000000000
```

You might be wondering now, is it possible to evaluate the symbolic functions for each element of the array? At the beginning of the lab you have defined a `NumPy` array `x_array`:

```
print(x_array)
```

```
[1 2 3]
```

Now try to evaluate function `f_symb` for each element of the array. You will get an error:

```
try:
    f_symb(x_array)
except TypeError as err:
    print(err)
```

```
'Pow' object is not callable
```

It is possible to evaluate the symbolic functions for each element of the array, but you need to make a function `NumPy`-friendly first:

```
from sympy.utilities.lambdify import lambdify

f_symb_numpy = lambdify(x, f_symb, 'numpy')
```

The following code should work now:

```
print("x: \n", x_array)
print("f(x) = x**2: \n", f_symb_numpy(x_array))
```

```
x: 
 [1 2 3]
f(x) = x**2: 
 [1 4 9]
```

`SymPy` has lots of great functions to manipulate expressions and perform various operations from calculus. More information about them can be found in the official documentation here.

2.2 - Symbolic Differentiation with SymPy

Let's try to find a derivative of a simple power function using `SymPy`:

```
diff(x**3,x)
```

```
3*x**2
```

Some standard functions can be used in the expression, and SymPy will apply required rules (sum, product, chain) to calculate the derivative:

```
dfdx_composed = diff(exp(-2*x) + 3*sin(3*x), x)
dfdx_composed
```

```
9*cos(3*x) - 2*exp(-2*x)
```

Now calculate the derivative of the function f_symb defined in 2.1 and make it NumPy-friendly:

```
dfdx_symb = diff(f_symb, x)
dfdx_symb_numpy = lambdify(x, dfdx_symb, 'numpy')
```

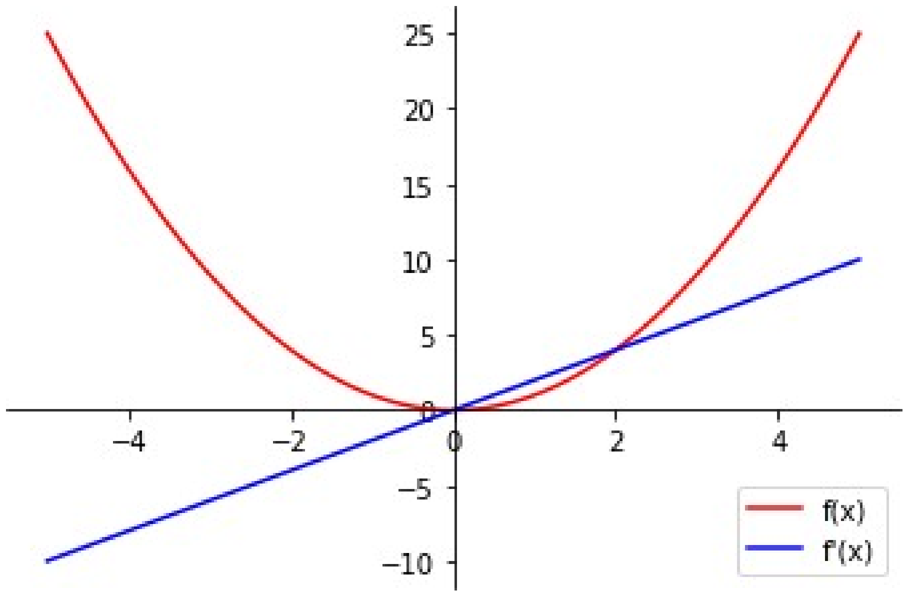
Evaluate function dfdx_symb_numpy for each element of the x_array:

```
print("x: \n", x_array)
print("f'(x) = 2x: \n", dfdx_symb_numpy(x_array))
```

```
x:
 [1 2 3]
f'(x) = 2x:
 [2 4 6]
```

You can apply symbolically defined functions to the arrays of larger size. The following code will plot function and its derivative, you can see that it works:

```
plot_f1_and_f2(f_symb_numpy, dfdx_symb_numpy)
```



2.3 - Limitations of Symbolic Differentiation

Symbolic Differentiation seems to be a great tool. But it also has some limitations. Sometimes the output expressions are too complicated and even not possible to evaluate. For example, find the derivative of the function

$$|x| = \begin{cases} x, \text{if } x > 0 \\ -x, \text{if } x < 0 \\ 0, \text{if } x = 0 \end{cases}$$

Analytically, its derivative is:

$$\frac{d}{dx}(|x|) = \begin{cases} 1, \text{if } x > 0 \\ -1, \text{if } x < 0 \\ \text{does not exist}, \text{if } x = 0 \end{cases}$$

Have a look the output from the symbolic differentiation:

```
dfdx_abs = diff(abs(x),x)
dfdx_abs
```

```
(re(x)*Derivative(re(x), x) + im(x)*Derivative(im(x), x))*sign(x)/x
```

Looks complicated, but it would not be a problem if it was possible to evaluate. But check, that for $x = -2$ instead of the derivative value -1 it outputs some unevaluated expression:

```
dfdx_abs.evalf(subs={x:-2})
```

```
-Subs(Derivative(re(x), x), x, -2)
```

And in the `NumPy` friendly version it also will give an error:

```
dfdx_abs_numpy = lambdify(x, dfdx_abs,'numpy')

try:
    dfdx_abs_numpy(np.array([1, -2, 0]))
except NameError as err:
    print(err)
```

```
name 'Derivative' is not defined
```

In fact, there are problems with the evaluation of the symbolic expressions wherever there is a "jump" in the derivative (e.g. function expressions are different for different intervals of x), like it happens with $\frac{d}{dx}(|x|)$.

Also, you can see in this example, that you can get a very complicated function as an output of symbolic computation. This is called **expression swell**, which results in unefficiently slow computations. You will see the example of that below after learning other differentiation libraries in Python.

3 - Numerical Differentiation

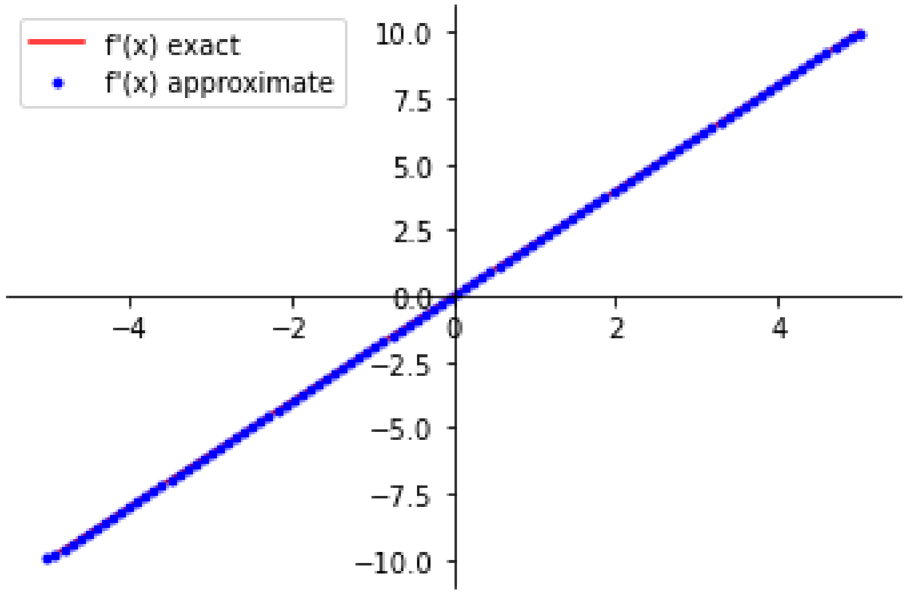
This method does not take into account the function expression. The only important thing is that the function can be evaluated in the nearby points x and $x+\Delta x$, where Δx is sufficiently small. Then $\frac{df}{dx}\approx\frac{f(x+\Delta x)-f(x)}{\Delta x}$, which can be called a **numerical approximation** of the derivative.

Based on that idea there are different approaches for the numerical approximations, which somehow vary in the computation speed and accuracy. However, for all of the methods the results are not accurate - there is a round off error. At this stage there is no need to go into details of various methods, it is enough to investigate one of the numerial differentiation functions, available in `NumPy` package.

3.1 - Numerical Differentiation with NumPy

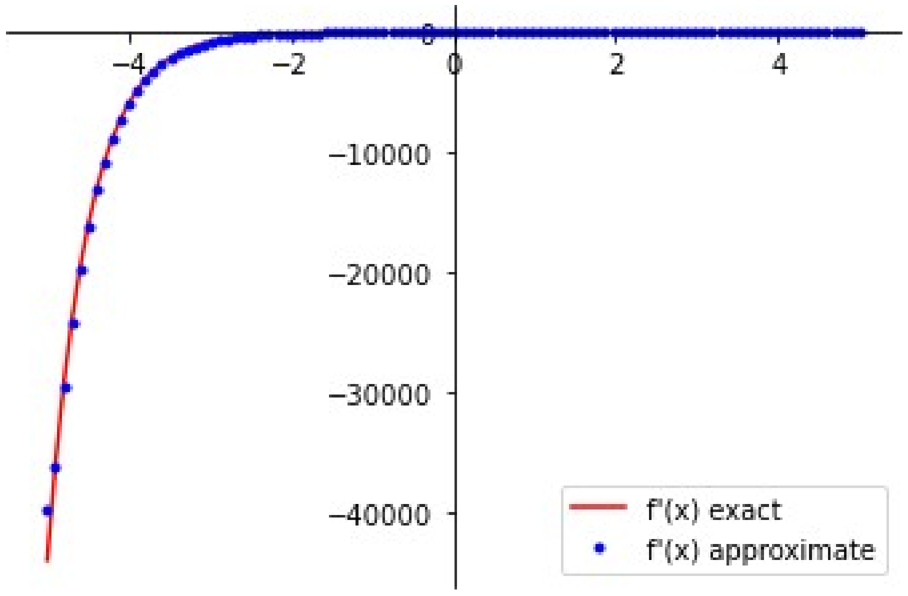
You can call function `np.gradient` to find the derivative of function $f(x)=x^2$ defined above. The first argument is an array of function values, the second defines the spacing Δx for the evaluation. Here pass it as an array of x values, the differences will be calculated automatically. You can find the documentation here.

```
x_array_2 = np.linspace(-5, 5, 100)
dfdx_numerical = np.gradient(f(x_array_2), x_array_2)

plot_f1_and_f2(dfdx_symb_numpy, dfdx_numerical, label1="f'(x) exact", label2="f'(x) approximate")
```



Try to do numerical differentiation for more complicated function:

```
def f_composed(x):
    return np.exp(-2*x) + 3*np.sin(3*x)

plot_f1_and_f2(lambdify(x, dfdx_composed, 'numpy'),
np.gradient(f_composed(x_array_2), x_array_2),
              label1="f'(x) exact", label2="f'(x) approximate")
```



The results are pretty impressive, keeping in mind that it does not matter at all how the function was calculated - only the final values of it!

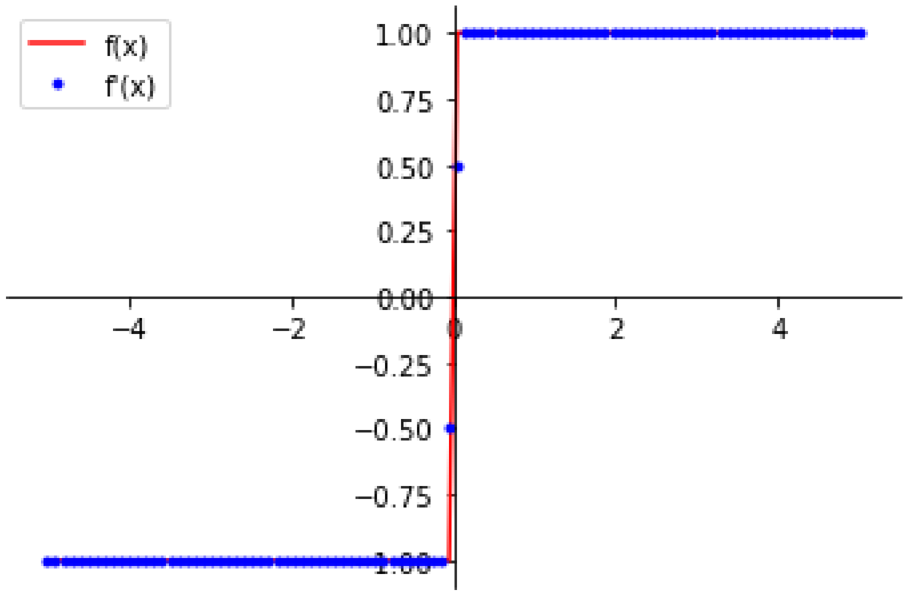
3.2 - Limitations of Numerical Differentiation

Obviously, the first downside of the numerical differentiation is that it is not exact. However, the accuracy of it is normally enough for machine learning applications. At this stage there is no need to evaluate errors of the numerical differentiation.

Another problem is similar to the one which appeared in the symbolic differentiation: it is inaccurate at the points where there are "jumps" of the derivative. Let's compare the exact derivative of the absolute value function and with numerical approximation:

```
def dfdx_abs(x):
    if x > 0:
        return 1
    else:
        if x < 0:
            return -1
        else:
            return None
```

```
plot_f1_and_f2(np.vectorize(dfdx_abs), np.gradient(abs(x_array_2),
x_array_2))
```



You can see that the results near the "jump" are 0.5 and -0.5, while they should be 1 and -1. These cases can give significant errors in the computations.

But the biggest problem with the numerical differentiation is slow speed. It requires function evalutation every time. In machine learning models there are hundreds of parameters and there are hundreds of derivatives to be calculated, performing full function evaluation every time slows down the computation process. You will see the example of it below.

4 - Automatic Differentiation

Automatic differentiation (autodiff) method breaks down the function into common functions (sin, cos, log, power functions, etc.), and constructs the computational graph consisting of the basic functions. Then the chain rule is used to compute the derivative at any node of the graph. It is the most commonly used approach in machine learning applications and neural networks, as the computational graph for the function and its derivatives can be built during the construction of the neural network, saving in future computations.

The main disadvantage of it is implementational difficulty. However, nowadays there are libraries that are convenient to use, such as MyGrad, Autograd and JAX. `Autograd` and `JAX` are the most commonly used in the frameworks to build neural networks. `JAX` brings together `Autograd` functionality for optimization problems, and `XLA` (Accelerated Linear Algebra) compiler for parallel computing.

The syntax of Autograd and JAX are slightly different. It would be overwhelming to cover both at this stage. In this notebook you will be performing automatic differentiation using one of them: JAX.

4.1 - Introduction to JAX

To begin with, load the required libraries. From jax package you need to load just a couple of functions for now (grad and vmap). Package jax.numpy is a wrapped NumPy, which pretty much replaces NumPy when JAX is used. It can be loaded as np as if it was an original NumPy in most of the cases. However, in this notebook you'll upload it as jnp to distinguish them for now.

```
from jax import grad, vmap
import jax.numpy as jnp
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_5394/3425913292.py in <module>
----> 1 from jax import grad, vmap
      2 import jax.numpy as jnp

ModuleNotFoundError: No module named 'jax'
```

Create a new jnp array and check its type.

```
x_array_jnp = jnp.array([1., 2., 3.])

print("Type of NumPy array:", type(x_array))
print("Type of JAX NumPy array:", type(x_array_jnp))
# Please ignore the warning message if it appears.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_5394/3298601773.py in <module>
----> 1 x_array_jnp = jnp.array([1., 2., 3.])
      2 
      3 print("Type of NumPy array:", type(x_array))
      4 print("Type of JAX NumPy array:", type(x_array_jnp))
      5 # Please ignore the warning message if it appears.

NameError: name 'jnp' is not defined
```

The same array can be created just converting previously defined `x_array = np.array([1, 2, 3])`, although in some cases JAX does not operate with integers, thus the values need to be converted to floats. You will see an example of it below.

```
x_array_jnp = jnp.array(x_array.astype('float32'))
print("JAX NumPy array:", x_array_jnp)
print("Type of JAX NumPy array:", type(x_array_jnp))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
/tmp/ipykernel_5394/3230805256.py in <module>
----> 1 x_array_jnp = jnp.array(x_array.astype('float32'))
      2 print("JAX NumPy array:", x_array_jnp)
      3 print("Type of JAX NumPy array:", type(x_array_jnp))

NameError: name 'jnp' is not defined
```

Note, that `jnp` array has a specific type `jaxlib.xla_extension.DeviceArray`. In most of the cases the same operators and functions are applicable to them as in the original `NumPy`, for example:

```
print(x_array_jnp * 2)
print(x_array_jnp[2])
```

But sometimes working with `jnp` arrays the approach needs to be changed. In the following code, trying to assign a new value to one of the elements, you will get an error:

```
try:
    x_array_jnp[2] = 4.0
except TypeError as err:
    print(err)
```

To assign a new value to an element in the `jnp` array you need to apply functions `.at[i]`, stating which element to update, and `.set(value)` to set a new value. These functions also operate **out-of-place**, the updated array is returned as a new array and the original array is not modified by the update.

```
y_array_jnp = x_array_jnp.at[2].set(4.0)
print(y_array_jnp)
```

Although, some of the `JAX` functions will work with arrays defined with `np` and `jnp`. In the following code you will get the same result in both lines:

```
print(jnp.log(x_array))
print(jnp.log(x_array_jnp))
```

This is probably confusing - which `NumPy` to use then? Usually when `JAX` is used, only `jax.numpy` gets imported as `np`, and used instead of the original one.

4.2 - Automatic Differentiation with JAX

Time to do automatic differentiation with JAX. The following code will calculate the derivative of the previously defined function $f(x)=x^2$ at the point $x=3$:

```
print("Function value at x = 3:", f(3.0))
print("Derivative value at x = 3:",grad(f)(3.0))
```

Very easy, right? Keep in mind, please, that this cannot be done using integers. The following code will output an error:

```
try:
    grad(f)(3)
except TypeError as err:
    print(err)
```

Try to apply the `grad` function to an array, calculating the derivative for each of its elements:

```
try:
    grad(f)(x_array_jnp)
except TypeError as err:
    print(err)
```

There is some broadcasting issue there. You don't need to get into more details of this at this stage, function `vmap` can be used here to solve the problem.

Note: Broadcasting is covered in the Course 1 of this Specialization "Linear Algebra". You can also review it in the documentation here.

```
dfdx_jax_vmap = vmap(grad(f))(x_array_jnp)
print(dfdx_jax_vmap)
```

Great, now `vmap(grad(f))` can be used to calculate the derivative of function `f` for arrays of larger size and you can plot the output:

```
plot_f1_and_f2(f, vmap(grad(f)))
```

In the following code you can comment/uncomment lines to visualize the common derivatives. All of them are found using JAX automatic differentiation. The results look pretty good!

```
def g(x):
#     return x**3
#     return 2*x**3 - 3*x**2 + 5
#     return 1/x
#     return jnp.exp(x)
#     return jnp.log(x)
#     return jnp.sin(x)
#     return jnp.cos(x)
    return jnp.abs(x)
#     return jnp.abs(x)+jnp.sin(x)*jnp.cos(x)
```

```
plot_f1_and_f2(g, vmap(grad(g)))
```

5 - Computational Efficiency of Symbolic, Numerical and Automatic Differentiation

In sections 2.3 and 3.2 low computational efficiency of symbolic and numerical differentiation was discussed. Now it is time to compare speed of calculations for each of three approaches. Try to find the derivative of the same simple function $f(x)=x^2$ multiple times, evaluating it for an array of a larger size, compare the results and time used:

```
import timeit, time

x_array_large = np.linspace(-5, 5, 1000000)

tic_symb = time.time()
res_symb = lambdify(x, diff(f(x),x),'numpy')(x_array_large)
toc_symb = time.time()
time_symb = 1000 * (toc_symb - tic_symb)  # Time in ms.

tic_numerical = time.time()
res_numerical = np.gradient(f(x_array_large),x_array_large)
toc_numerical = time.time()
time_numerical = 1000 * (toc_numerical - tic_numerical)

tic_jax = time.time()
res_jax = vmap(grad(f))(jnp.array(x_array_large.astype('float32')))
toc_jax = time.time()
time_jax = 1000 * (toc_jax - tic_jax)

print(f"Results\nSymbolic Differentiation:\n{res_symb}\n" +
      f"Numerical Differentiation:\n{res_numerical}\n" +
      f"Automatic Differentiation:\n{res_jax}")

print(f"\n\nTime\nSymbolic Differentiation:\n{time_symb} ms\n" +
      f"Numerical Differentiation:\n{time_numerical} ms\n" +
      f"Automatic Differentiation:\n{time_jax} ms")
```

The results are pretty much the same, but the time used is different. Numerical approach is obviously inefficient when differentiation needs to be performed many times, which happens a lot training machine learning models. Symbolic and automatic approach seem to be performing similarly for this simple example. But if the function becomes a little bit more complicated, symbolic computation will experiance significant expression swell and the calculations will slow down.

Note: Sometimes the execution time results may vary slightly, especially for automatic differentiation. You can run the code above a few time to see different outputs. That does not influence the conclusion that numerical differentiation is slower. `timeit` module can be used more efficiently to evaluate execution time of the codes, but that would unnecessary overcomplicate the codes here.

Try to define some polynomial function, which should not be that hard to differentiate, and compare the computation time for its differentiation symbolically and automatically:

```
def f_polynomial_simple(x):
    return 2*x**3 - 3*x**2 + 5

def f_polynomial(x):
    for i in range(3):
        x = f_polynomial_simple(x)
    return x

tic_polynomial_symb = time.time()
res_polynomial_symb = lambdify(x, diff(f_polynomial(x),x),'numpy')
(x_array_large)
toc_polynomial_symb = time.time()
time_polynomial_symb = 1000 * (toc_polynomial_symb -
tic_polynomial_symb)

tic_polynomial_jax = time.time()
res_polynomial_jax = vmap(grad(f_polynomial))
(jnp.array(x_array_large.astype('float32')))
toc_polynomial_jax = time.time()
time_polynomial_jax = 1000 * (toc_polynomial_jax - tic_polynomial_jax)

print(f"Results\nSymbolic Differentiation:\n{res_polynomial_symb}\n" +

      f"Automatic Differentiation:\n{res_polynomial_jax}")

print(f"\n\nTime\nSymbolic Differentiation:\n{time_polynomial_symb}
ms\n" +
      f"Automatic Differentiation:\n{time_polynomial_jax} ms")
```

Again, the results are similar, but automatic differentiation is times faster.

With the increase of function computation graph, the efficiency of automatic differentiation compared to other methods raises, because autodiff method uses chain rule!

Congratulations! Now you are equiped with Python tools to perform differentiation.