# SCICOMP302 Algorithms and Data Structures

## Project 2 - Investigation of the Gradient Descent Optimization Algorithm

*This is all my own work. I have not knowingly allowed others to copy my work. This work has not been submitted for assessment in any other context.*


Author:

Joanikij Chulev

Professor:

Dr. Tamas Vinko


November, 2023

## *Abstract*

*Gradient descent is a widely used optimization algorithm that iteratively moves in the direction of the negative gradient of a function until it reaches a minimum. The choice of learning rate, which controls the step size at each iteration, is a key hyperparameter in Gradient Descent. This experiment investigates the effect of different learning rates on the convergence time of Gradient Descent. The experiment was conducted using three different mathematical functions. For each function, the experiment was run with five different learning rates. The average convergence time was calculated for each combination of function and learning rate. The experiment was conducted with both code that implements numerical differentiation and symbolic differentiation to have a comparison of these results. Results were interesting, since no statistical difference was detected for the convergence rates of these functions for both code cases.*

# Contents



**Experimental platform**

***Hardware technology platform:***

OS Name: Microsoft Windows 11 Pro

Windows version: 22H2

System Type: x64-based PC

Processor: Intel(R) Core (TM) i7-1065G7 CPU @ 1.30GHz, 1498 Mhz, 4 Core(s), 8 Logical Proc.

Installed Physical Memory (RAM): 4.00 GB

GPU: Integrated Intel GPU (Intel Graphics)

***Software technology platform:***

Spyder version: 5.4.3 (conda)

Python version: 3.9.17 64-bit

Qt version: 5.15.2

PyQt5 version: 5.15.7

Operating System: Windows 11

# 1 Introduction

A gradient in mathematics is a directional derivative that indicates the direction of the greatest rate of change of a function [1]. It is a vector that points in the direction of the greatest increase in the function's value, and its magnitude is equal to the rate of change in the function's value at that point. Gradients are used in many different fields, including physics, engineering, and machine learning [1]. The gradient of a function f(x) is denoted as ∇f(x) or grad(f(x)) and is represented as a vector.

For a function f(x) of multiple variables x = ($x_1$, $x_2$, ...), the gradient ∇f(x) is a vector with n components, where each component represents the partial derivative of the function with respect to the corresponding input variable. It is defined as:

$$\nabla f(x) = (\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_n)$$

In this expression, ∂f/∂x represents the partial derivative of the function f with respect to the n-th input variable x. For a function with a single variable, the gradient reduces to a scalar value, which is the derivative of the function with respect to that variable. If you have a function f(x, y) and you want to calculate its gradient at a point (a, b) you would compute the partial derivatives of f(x,y) with respect to x and y evaluated at that point. We can see an example of a gradient in Figure 1.

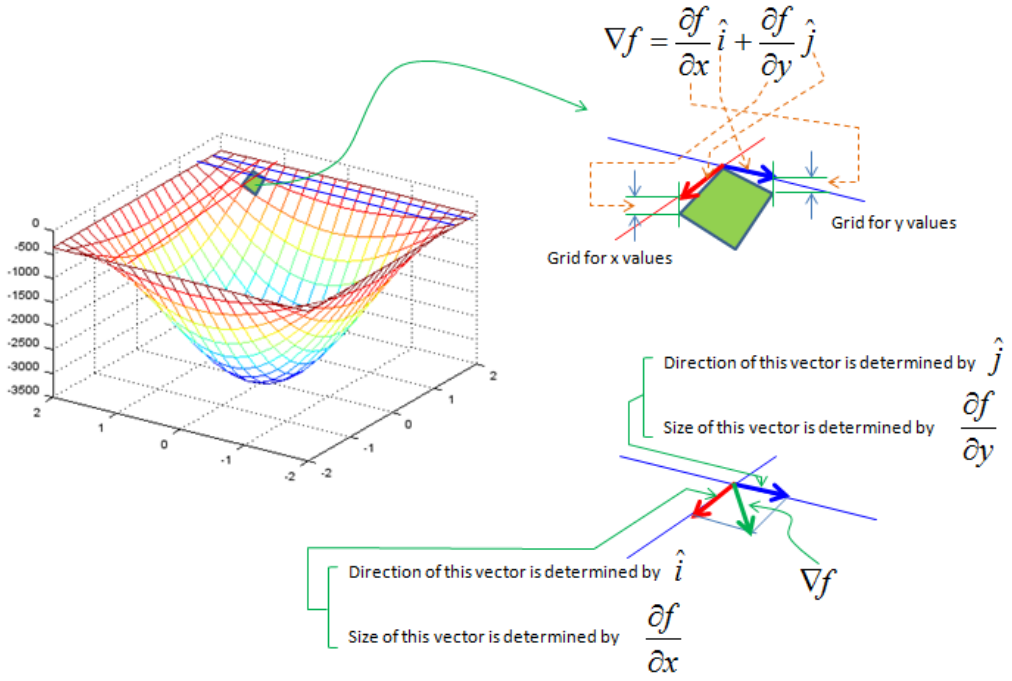


Figure 1: example of how to calculate and a graphical representation of a gradient

To compute multiple gradients of multiple functions we would need to introduce the Jacobian. A Jacobian is a matrix that contains all the partial derivatives of a vector-valued function/s. When this matrix is square, that is, when the function takes the same number of variables as input as the number of vector components of its output, its determinant is referred to as the Jacobian determinant.

Both the matrix and (if applicable) the determinant are often referred to simply as the Jacobian in literature [4]. Jacobians are used in many different fields, including mathematics, physics, engineering, and machine learning. In our context the Jacobian matrix is the generalization of the gradient for vector-valued functions of several variables and differentiable maps between Euclidean spaces or, more generally, manifolds. We can see this representation below:

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^{\mathsf{T}} f_1 \\ \vdots \\ \nabla^{\mathsf{T}} f_m \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

An optimization algorithm is an algorithm that finds the best solution to a problem, given a set of constraints [2]. Gradient descent is an optimization algorithm that iteratively moves in the direction of the negative gradient of a function until it reaches a minimum [3]. Thus, moving opposite of the gradient with formula:

$$x_n = x_n - \alpha \nabla f(x_n)$$

The negative sign in the Gradient Descent formula comes from Taylor Series. Our goal is to minimize the error the maximum we can at each step.

In engineering, gradient descent is used to design optimal structures and systems. Furthermore, Gradient descent is a simple and efficient optimization algorithm, and it is widely used in machine learning to train machine learning models [3]. Although, Optimization algorithms are used in many different fields [2]. Regarding formulation, Gradient descent was first invented by Adrien-Marie Legendre in 1806 [3]. Since than it has been a very useful tool for furthering technological progress.

# 2 Experiments

Optimization Algorithms plays a crucial role in finding the minimum or maximum of a function by iteratively adjusting the input parameters. The choice of the learning rate, represented as α, is a key hyperparameter in Gradient Descent, influencing the convergence behavior of the algorithm. In this experiment, we aim to investigate how different learning rates affect the convergence time of Gradient Descent. We will run experiments based on code that finds the numerical and symbolical derivatives of the gradients to compare both instances. Hence, we formulate the following hypotheses:

*$H0$ - There is no significant difference in the convergence time of the Gradient Descent algorithm applied to simple 1D functions when using different learning rates*

*$Ha$ - There is a significant difference in the convergence time of the Gradient Descent algorithm applied to simple 1D functions when using different learning rates*

This null hypothesis suggests that variations in the learning rate do not result in significant differences in the time taken for Gradient Descent to converge to the minimum of the chosen function. In other words,

the choice of learning rate does not impact the optimization process significantly. The alternative hypothesis proposes the opposite of the null hypothesis, suggesting that variations in the learning rate do result in significant differences in the convergence time. In this case, the choice of learning rate would have a noticeable impact on the optimization process. The learning rate (α) is a hyperparameter that controls the step size at each iteration of the Gradient Descent algorithm. It determines how quickly or slowly the algorithm converges to the minimum of the objective function. A high learning rate (e.g., α = 1.0) allows for larger steps, which can lead to faster convergence but may risk overshooting the minimum or even divergence. A low learning rate (e.g., α = 0.001) results in smaller steps, ensuring stability but potentially leading to slow convergence.

Thus, following this logic, we can define the dependent and independent variables:

Independent Variable - Learning Rate (α) - Different values of the learning rate, e.g., α = 0.01, α = 0.1, α = 0.5, etc.

Dependent Variable - Convergence Time - The time taken for the Gradient Descent algorithm to converge to a minimum within a predefined tolerance.

When it comes to the measurements, we will test the following:

Measurement of Time-Convergence time is measured in seconds using a or a time-tracking library within the code. Hence, time is measured using the time module in Python.

Running more experiments or runs can provide more accurate and reliable average results, reducing the impact of random variations and allowing you to draw more robust conclusions. When you run an optimization algorithm like Gradient Descent, the convergence time can vary from run to run due to factors like random initialization, numerical precision, and fluctuations in the optimization process. By running the algorithm multiple times and calculating the average convergence time, you obtain a better estimate of the typical behavior and performance of the algorithm for a given configuration. A larger number of runs allows you to capture a more precise average, which is especially important when studying the impact of different learning rates on Gradient Descent.

We conducted a series of experiments by running Gradient Descent with varying learning rates (alpha) and recorded the average convergence time over 10,000 runs for each combination of function and learning rate. The convergence tolerance was set to 1e-6, and the maximum number of iterations was capped at 1,000. Gradient Descent can be applied to any dimension function i.e., 1-D, 2-D, 3-Dand so on. The following functions were tested:

$$f1(x) = x^2$$

$$f2(x) = sin(x)$$

$$f3(x) = e^{-x}$$

## 2.1 Code Analysis for both experiment variants

*We define three functions (f1, f2, and f3) that represent different mathematical functions. These functions will be used to test the Gradient Descent algorithm's performance.

*The gradient_descent function implements the Gradient Descent algorithm. It takes the function to optimize (f), the initial point (x0), learning rate (alpha), convergence tolerance (epsilon), and maximum number of iterations (max_iter).

-The x0 variable represents the initial point or starting point for the optimization process, it's set to a fixed value of 1.0. This means that for each run of the experiment, the Gradient Descent algorithm starts at the same initial point, and only the learning rate (alpha) is varied across runs. How the logic works:

-for i in range(max_iter): - Start a loop that will run for a maximum of max_iter iterations. This loop controls the iterations of the Gradient Descent algorithm.

-grad = np.gradient(f(x), x) - Calculate the gradient of the function f at the current point x. The np.gradient function is used to compute the gradient, and grad is assigned the gradient vector. This code applies numerical differentiation.

-x -= alpha * np.sum(grad) - Update the current point x by subtracting the product of the learning rate alpha and the sum of the gradient vector grad. This is the step that moves x towards the minimum of the function.

-if np.linalg.norm(grad) < epsilon: - The expression np.linalg.norm(grad) calculates the L2 (Euclidean) norm or magnitude of a vector grad using the NumPy library in Python. This is a common operation in optimization algorithms, including gradient descent, to check the convergence criteria. The L2 norm of a vector is the square root of the sum of the squared elements in the vector. We use np.linalg.norm(grad) to compute the magnitude of the gradient at a specific point in the optimization process. The algorithm terminates when the magnitude of the gradient becomes small, indicating that the algorithm has converged to a minimum or near-minimum of the function. Then whether the norm of the gradient vector is less than the specified convergence tolerance epsilon. If the norm is below the tolerance, it means the algorithm has converged.

-return x - return the optimal solution x.

*The run_experiment function performs the experiment. We run Gradient Descent with the specified parameters for a given number of runs (n_runs). We measure and return the average convergence time over these runs.

*We specify a list of learning rates (learning_rates) and the number of runs (n_runs) for the experiment.

*We iterate through the three functions (f1, f2, and f3) and for each function, we iterate through the different learning rates. For each combination of function and learning rate, we run the experiment and print the function name, learning rate, and average convergence time.

The only difference in the code that uses symbolic derivates of the function, is that we do not implement the gradient calculation method that is built in python and we find the derivates manually and express them as functions as seen in figure 2. Hence, we use the df functions in the calculation step.

```python
# Define the function f1(x) = x^2 and its derivative df1(x) = 2x
def f1(x):
    return x**2

def df1(x):
    return 2 * x

# Define the function f2(x) = sin(x) and its derivative df2(x) = cos(x)
def f2(x):
    return np.sin(x)

def df2(x):
    return np.cos(x)

# Define the function f3(x) = exp(-x) and its derivative df3(x) = -exp(-x)
def f3(x):
    return np.exp(-x)

def df3(x):
    return -np.exp(-x)
```

Figure 2: How we expressed the gradients using symbolic differentiation

# 3 Results and Conclusions

After running the experiment as we described it, we can layout our results. First, we will show the results we got when implementing the gradient using the code that applies numerical differentiation. For function 1 we got the following results. As seen in Figure 3.

Function: f1 , Learning rate: 0.0001 , Average convergence time: 1.9884562492370605e-05

Function: f1 , Learning rate: 0.001 , Average convergence time: 1.961991786956787e-05

Function: f1 , Learning rate: 0.01 , Average convergence time: 2.1297121047973633e-05

Function: f1 , Learning rate: 0.1 , Average convergence time: 1.9970226287841797e-05

Function: f1 , Learning rate: 0.5 , Average convergence time: 1.9404530525207518e-05

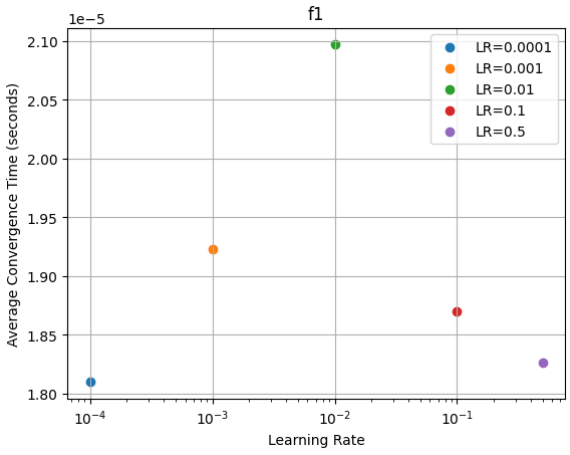


Figure 3: Graph results of convergence of f1 (numerical differentiation code)

For function 2 we got the following results. As seen in Figure 4.

Function: f2 , Learning rate: 0.0001 , Average convergence time: 2.0166540145874022e-05

Function: f2 , Learning rate: 0.001 , Average convergence time: 2.28132963180542e-05

Function: f2 , Learning rate: 0.01 , Average convergence time: 2.2558140754699708e-05

Function: f2 , Learning rate: 0.1 , Average convergence time: 2.161383628845215e-05

Function: f2 , Learning rate: 0.5 , Average convergence time: 2.1802473068237306e-05

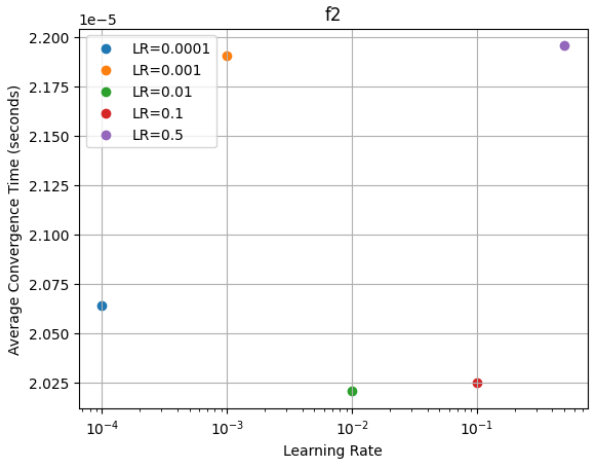


Figure 4: Graph results of convergence of f2 (numerical differentiation code)

For function 3 we got the following results. As seen in Figure 5.

Function: f3 , Learning rate: 0.0001 , Average convergence time: 2.208247184753418e-05

Function: f3 , Learning rate: 0.001 , Average convergence time: 2.3874235153198243e-05

Function: f3 , Learning rate: 0.01 , Average convergence time: 2.1753692626953126e-05

Function: f3 , Learning rate: 0.1 , Average convergence time: 2.2058153152465822e-05

Function: f3 , Learning rate: 0.5 , Average convergence time: 2.2530937194824218e-05

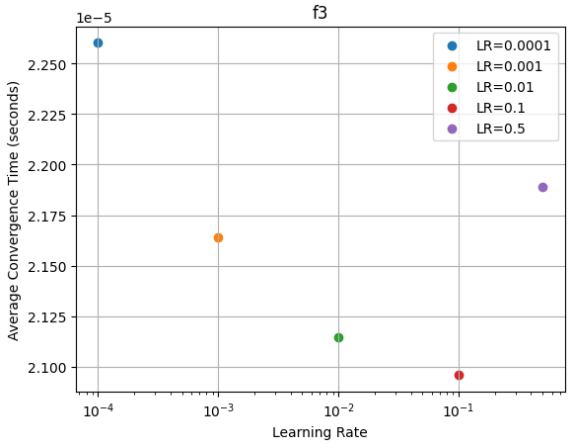


Figure 5: Graph results of convergence of f3 (numerical differentiation code)

Now, we will present the results we got when using code that applies symbolic differentiation. For function 1 we got the following results. As seen in Figure 6.

Function: f1

Learning rate: 0.0001, Average convergence time: 0.006007448673248291 seconds

Learning rate: 0.001, Average convergence time: 0.00415252685546875 seconds

Learning rate: 0.01, Average convergence time: 0.004934208869934082 seconds

Learning rate: 0.1, Average convergence time: 0.00026883268356323243 seconds

Learning rate: 0.5, Average convergence time: 9.035110473632813e-06 seconds

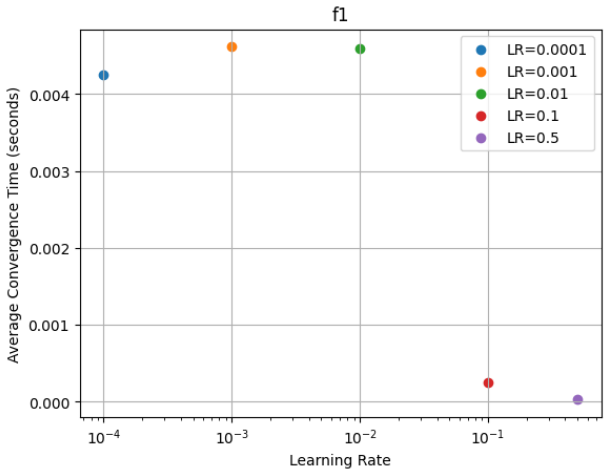


Figure 6: Graph results of convergence of f1 (symbolic differentiation code)

For function 2 we got the following results. As seen in Figure 7.

Function: f2

Learning rate: 0.0001, Average convergence time: 0.006744585990905762 seconds

Learning rate: 0.001, Average convergence time: 0.009591136455535889 seconds

Learning rate: 0.01, Average convergence time: 0.006754581451416016 seconds

Learning rate: 0.1, Average convergence time: 0.0011069746017456055 seconds

Learning rate: 0.5, Average convergence time: 0.00029401087760925294 seconds

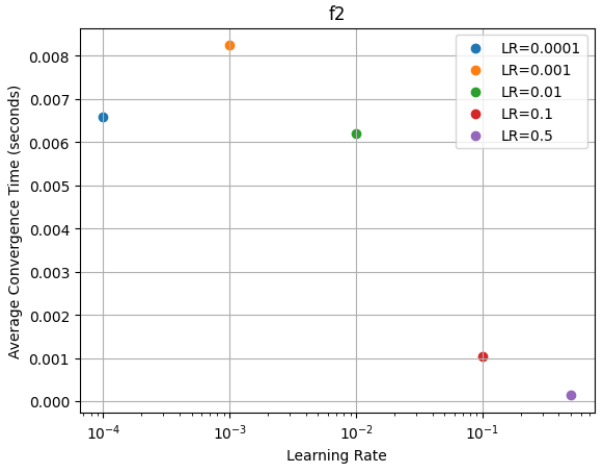


Figure 7: Graph results of convergence of f2 (symbolic differentiation code)

For function 3 we got the following results. As seen in Figure 8.

Function: f3

Learning rate: 0.0001, Average convergence time: 0.00827333402633667 seconds

Learning rate: 0.001, Average convergence time: 0.0064415838718414305 seconds

Learning rate: 0.01, Average convergence time: 0.008622692346572876 seconds

Learning rate: 0.1, Average convergence time: 0.006473969936370849 seconds

Learning rate: 0.5, Average convergence time: 0.00831503438949585 seconds

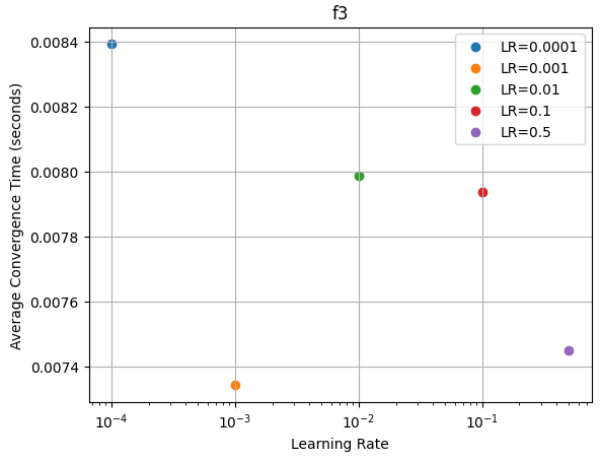


Figure 8: Graph results of convergence of f3 (symbolic differentiation code)

Based on the one-way ANOVA analyses conducted for each function for both code variant tests, we can draw the conclusions for this Gradient Descent experiment:

-The one-way ANOVA analysis for f1 shows that the p-value is greater than 0.05 (the conventional significance level), suggesting that there are no significant differences in convergence times among the different learning rates. This indicates that, for function f1, the choice of learning rate does not have a statistically significant impact on the optimization process.

- Similar to f1, the one-way ANOVA analysis for f2 also results in a p-value greater than 0.05. This suggests that there are no significant differences in convergence times among the different learning rates for f2.

- The one-way ANOVA analysis for f3 yields a p-value greater than 0.05, indicating no significant differences in convergence times among the different learning rates for f3. Function f3, like the previous two functions, is not notably influenced by the choice of learning rate in terms of convergence time.

The results of this experiment are specific to the functions tested. Other functions may exhibit different behaviors with respect to learning rate. The sample size used for each analysis can affect the statistical results. Larger sample sizes might provide more robust conclusions. In practical applications, choosing an appropriate learning rate is crucial for successful optimization. While this experiment doesn't find statistically significant differences, it's essential to consider other factors such as the specific problem, the initial conditions, and the desired convergence speed when selecting a learning rate. This is a very simple experiment on very simple functions and usually Gradient descent deals with more complex tasks.

Further research and experimentation with a broader range of functions can contribute to better understanding to why we essentially accept the null hypothesis $H0$ and reject $Ha$ for both instances of testing with code that applies numerical and symbolic differentiation. Thus, our comparison for both code implementations suggests the same outcome.

# 4 Applications and Improvements

We know that in any machine learning project our main aim relies on how good our project accuracy is or how much our model prediction differs from the actual data point. Based on the difference between model prediction and actual data points we try to find the parameters of the model which give better accuracy on our dataset. In order to find these parameters, we apply gradient descent on the cost function of the machine learning model. In regards of machine learning, especially in the realm of deep learning this is done using the following mathematical model, seen in Figure 9.

**Cost Function**

$$J(\Theta_0, \Theta_1) = \frac{1}{2m}\sum_{i=1}^{m}[h_\Theta(x_i) - y_i]^2$$

Predicted Value; True Value

**Gradient Descent**

$$\Theta_j = \Theta_j - \alpha\frac{\partial}{\partial\Theta_j}J(\Theta_0, \Theta_1)$$

Learning Rate

**Now,**

$$\frac{\partial}{\partial\Theta}J_\Theta = \frac{\partial}{\partial\Theta}\frac{1}{2m}\sum_{i=1}^{m}[h_\Theta(x_i) - y]^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}(h_\Theta(x_i) - y)\frac{\partial}{\partial\Theta_j}(\Theta x_i - y)$$

$$= \frac{1}{m}(h_\Theta(x_i) - y)x_i$$

**Therefore,**

$$\Theta_j := \Theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}[(h_\Theta(x_i) - y)x_i]$$

Figure 9: Applying Gradient Descent to a loss/cost function

Although, running GD is very costly on larger data and takes too long to converge, thus an improvement on GD is SGD (Stochastic Gradient Descent). SGD is stochastic in nature i.e., it picks up a "random" instance of training data at each step and then computes the gradient, making it much faster as there is much fewer data to manipulate at a single time. This also performs not only faster but usually with better approximations and better results, based on a reasonable run time. We can see performance visualization differences between GD and SGD in figure 10, below. Note that although SGD has more steps, these steps are computed a lot faster than the conventional GD steps.

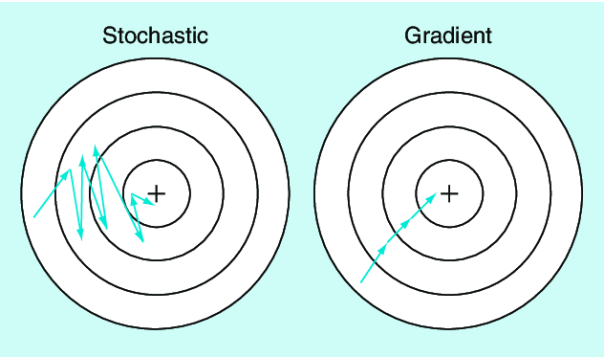


Figure 10: SGD steps versus GD steps

# *References*

[1] Boyd, S., & Vandenberghe, L. (2004). Convex optimization. Cambridge University Press.

[2] Nocedal, J., & Wright, S. J. (2006). Numerical optimization (2nd ed.). Springer.

[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

[4] Wikipedia. (2023, November 4). Jacobian matrix and determinant. Retrieved from https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant