Project Based Learning Report

on

Implementation of Gradient Decent Algorithm”

Submitted in the partial fulfilment of the requirements
For the Project based learning in (Fuzzy logic, Neural Networks & Genetic Algorithms) in
Electronics & Communication Engineering

By

2014111767	SHIVAM SHASHWAT
2014111111	PRIYANKA SINGH
2014111112	ANKUSH SINGH

Under the guidance of Course In-charge

Prof V.P Kaduskar

Department of Electronics & Communication Engineering

Bharati Vidyapeeth
(Deemed to be University)
College of Engineering,
Pune – 411043

Academic Year: 2022-23

Bharati Vidyapeeth
(Deemed to be University)
College of Engineering,
Pune – 411043

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

CERTIFICATE

Certified that the Project Based Learning report entitled, "Implementation of Gradient Decent Algorithm" is work done by

2014111767	SHIVAM SHASHWAT
2014111111	PRIYANKA SINGH
2014111112	ANKUSH SINGH

in partial fulfilment of the requirements for the award of credits for Project Based Learning (PBL) in Fuzzy logic, Neural Networks &Genetic Algorithms of Bachelor of Technology Semester V, in Electronics and Communication.

Date:

Prof. V.P kaduskar   Dr. Arundhati A.Shinde

Course In-charge   Professor & Head

Index:

Problem Statement:

Let’s take a simple quadratic function defined as:

$$f(x) = x^2 - 4x + 1$$

Because it is an univariate function a gradient function is:

$$\frac{df(x)}{dx} = 2x - 4$$

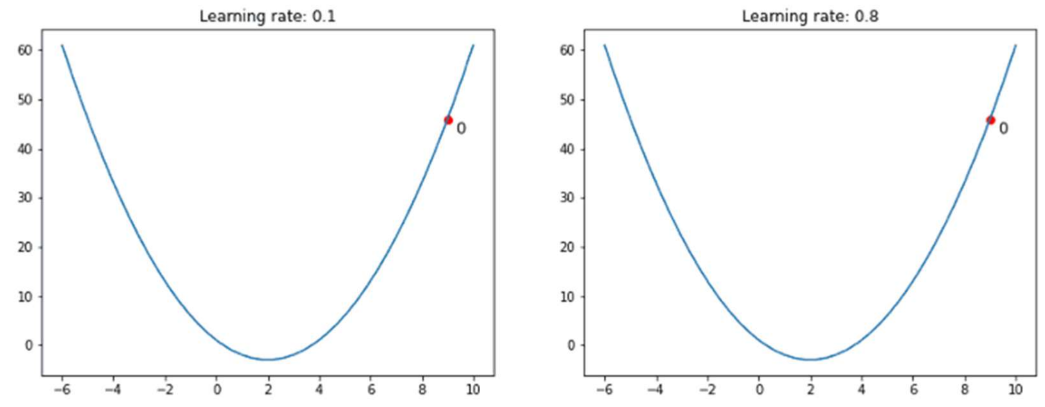
Let’s write these functions in Python:

For this function, by taking a learning rate of 0.1 and starting point at x=9 we can easily calculate each step by hand. Let’s do it for the first 3 steps:

$$x_1 = 9 - 0.1 \cdot (2 \cdot 9 - 4) = 7.6$$
$$x_2 = 7.6 - 0.1 \cdot (2 \cdot 7.6 - 4) = 6.48$$
$$x_3 = 6.48 - 0.1 \cdot (2 \cdot 6.48 - 4) = 5.584$$

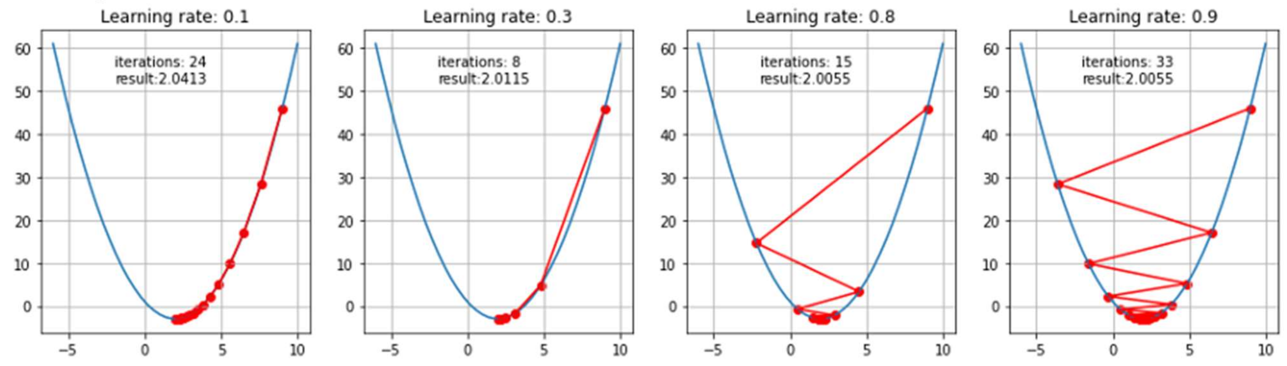
The python function is called by:

The animation below shows steps taken by the GD algorithm for learning rates of 0.1 and 0.8. As you see, for the smaller learning rate, as the algorithm approaches the minimum the steps are getting gradually smaller. For a bigger learning rate, it is jumping from one side to another before converging.



First 10 steps taken by GD for small and big learning rate; Image by author

Trajectories, number of iterations and the final converged result (within tolerance) for various learning rates are shown below:



THEORY:

Gradient descent (GD) is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. This method is commonly used in *machine learning* (ML) and *deep learning*(DL) to minimise a cost/loss function (e.g. in a linear regression). Due to its importance and ease of implementation, this algorithm is usually taught at the beginning of almost all machine learning courses.

However, its use is not limited to ML/DL only, it's being widely used also in areas like:

- control engineering (robotics, chemical, etc.)
- computer games
- mechanical engineering

That's why today we will get a deep dive into the math, implementation and behaviour of first-order gradient descent algorithm. We will navigate the custom (cost) function directly to find its minimum, so there will be no underlying data like in typical ML tutorials — we will be more flexible in terms of a function's shape.

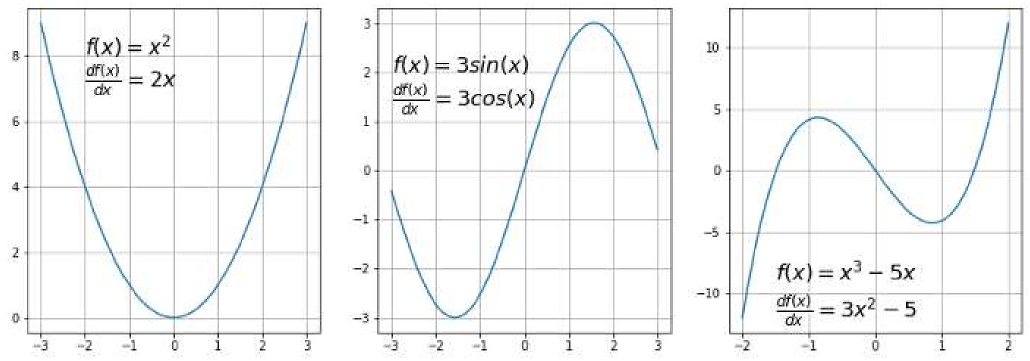
This method was proposed before the era of modern computers and there was an intensive development meantime which led to numerous improved versions of it but in this article, we're going to use a basic/vanilla gradient descent implemented in Python.

2. Function requirements

Gradient descent algorithm does not work for all functions. There are two specific requirements. A function has to be:

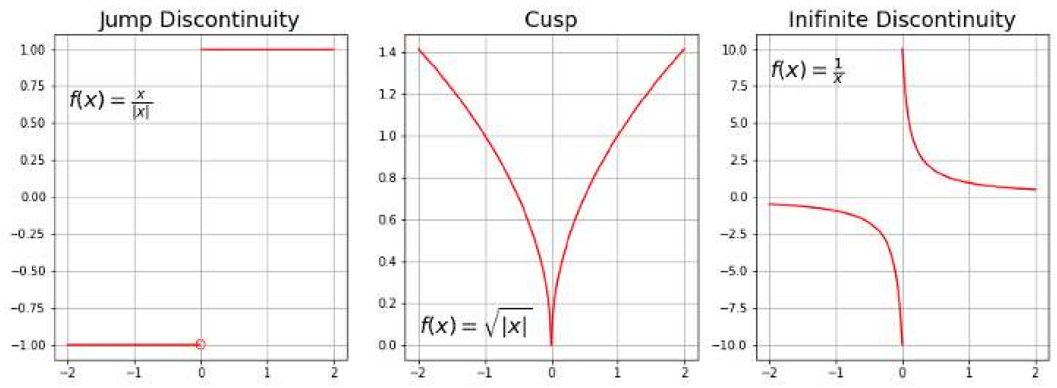
- **differentiable**
- **convex**

First, what does it mean it has to be **differentiable**? If a function is differentiable, it has a derivative for each point in its domain — not all functions meet these criteria. First, let’s see some examples of functions meeting this criterion:



Examples of differentiable functions; Image by author

Typical non-differentiable functions have a step a cusp or a discontinuity:



Examples of non-differentiable functions; Image by author

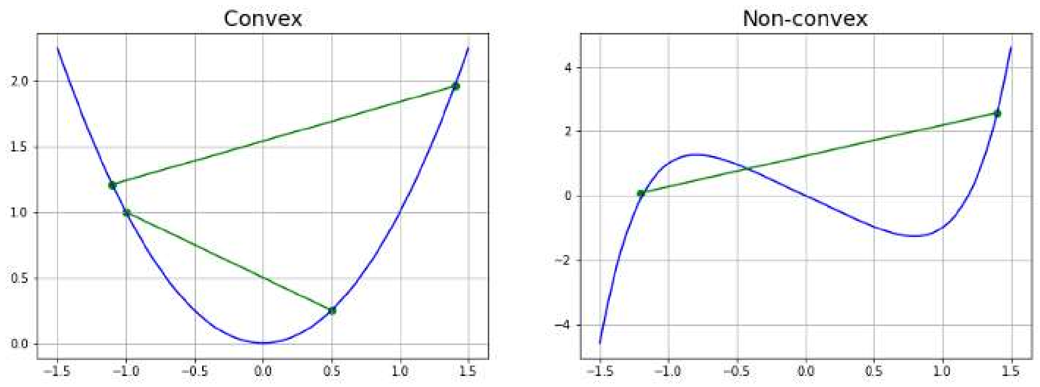
Next requirement — **function has to be convex**. For a univariate function, this means that the line segment connecting two function's points lays on or above its curve (it does not cross it). If it does it means that it has a local minimum which is not a global one.

Mathematically, for two points x_1, x_2 laying on the function's curve this condition is expressed as:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda)f(x_2)$$

where λ denotes a point's location on a section line and its value has to be between 0 (left point) and 1 (right point), e.g. λ=0.5 means a location in the middle.

Below there are two functions with exemplary section lines.



Exemplary convex and non-convex functions; Image by author

Another way to check mathematically if a univariate function is convex is to calculate the second derivative and check if its value is always bigger than 0.

$$\frac{d^2 f(x)}{dx^2} > 0$$

Let’s do a simple example (*warning: calculus ahead!*).
GIF via giphy

Let’s investigate a simple quadratic function given by:

$$f(x) = x^2 - x + 3$$

Its first and second derivative are:

$$\frac{df(x)}{dx} = 2x - 1, \quad \frac{d^2 f(x)}{dx^2} = 2$$

Because the second derivative is always bigger than 0, our function is strictly convex.

It is also possible to use **quasi-convex functions** with a gradient descent algorithm. However, often they have so-called **saddle points** (called also minimax points) where the algorithm can get stuck (we will demonstrate it later in the article). An example of a quasi-convex function is:

$$f(x) = x^4 - 2x^3 + 2$$

$$\frac{df(x)}{dx} = 4x^3 - 6x^2 = x^2(4x - 6)$$

Let’s stop here for a moment. We see that the first derivative equal zero at x=0 and x=1.5. These places are candidates for function’s extrema (minimum or maximum)— the slope is zero there. But first we have to check the second derivative first.

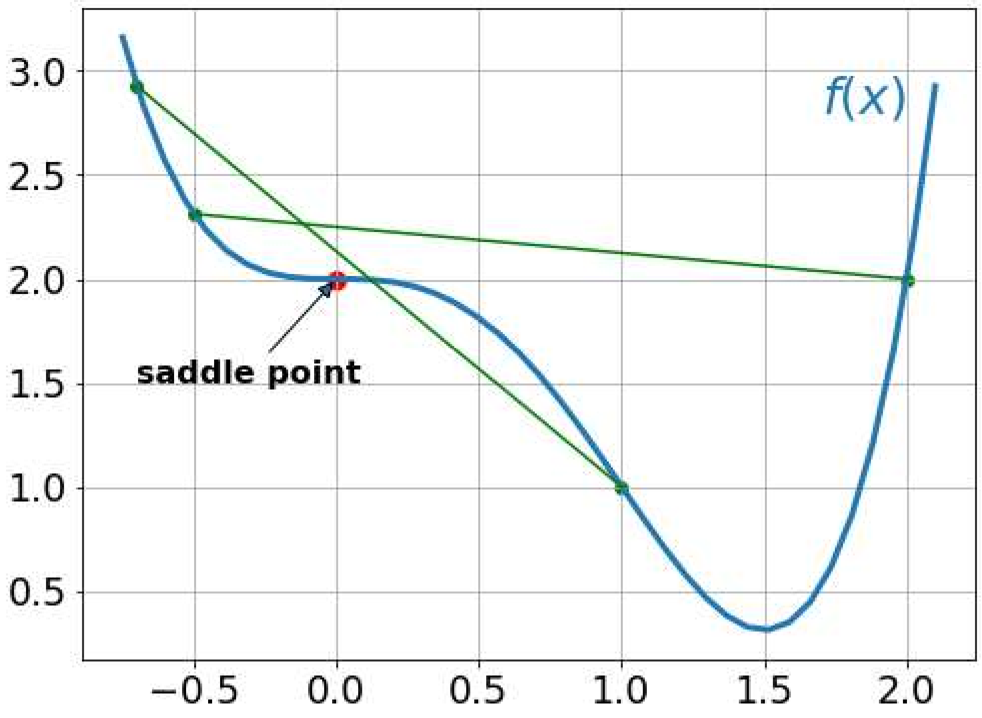
$$\frac{d^2 f(x)}{dx^2} = 12x^2 + 12x = 12x(x - 1)$$

The value of this expression is zero for *x=0* and *x=1*. These locations are called an inflexion point — a place where the curvature changes sign — meaning it changes from convex to concave or vice-versa. By analysing this equation, we conclude that:

- for x<0: function is convex
- for 0<x<1: function is concave (the 2nd derivative < 0)
- for x>1: function is convex again

Now we see that point *x=0* has both first and second derivative equal to zero meaning this is a saddle point and point x=1.5 is a global minimum.

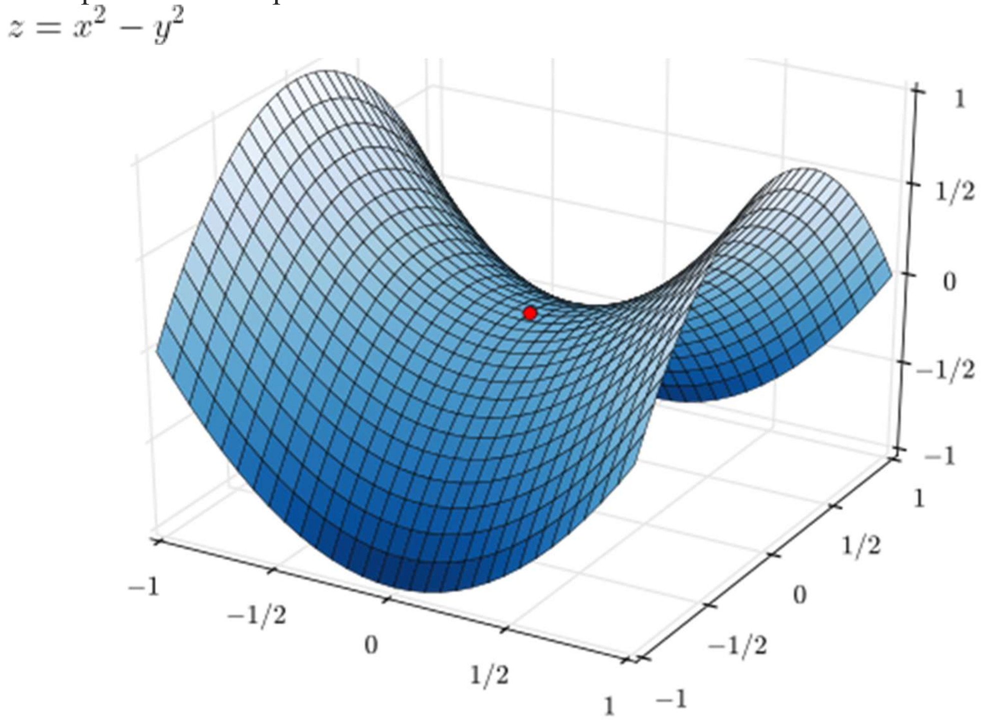
Let's look at the graph of this function. As calculated before a saddle point is at *x=0* and minimum at *x=1.5*.



Semi-convex function with a saddle point; Image by author

For multivariate functions the most appropriate check if a point is a saddle point is to calculate a Hessian matrix which involves a bit more complex calculations and is beyond the scope of this article.

Example of a saddle point in a bivariate function is show below.



Nicoguaro, CC BY 3.0, via Wikimedia Commons

3. Gradient

Before jumping into code one more thing has to be explained — what is a gradient. Intuitively it is a slope of a curve at a given point in a specified direction.

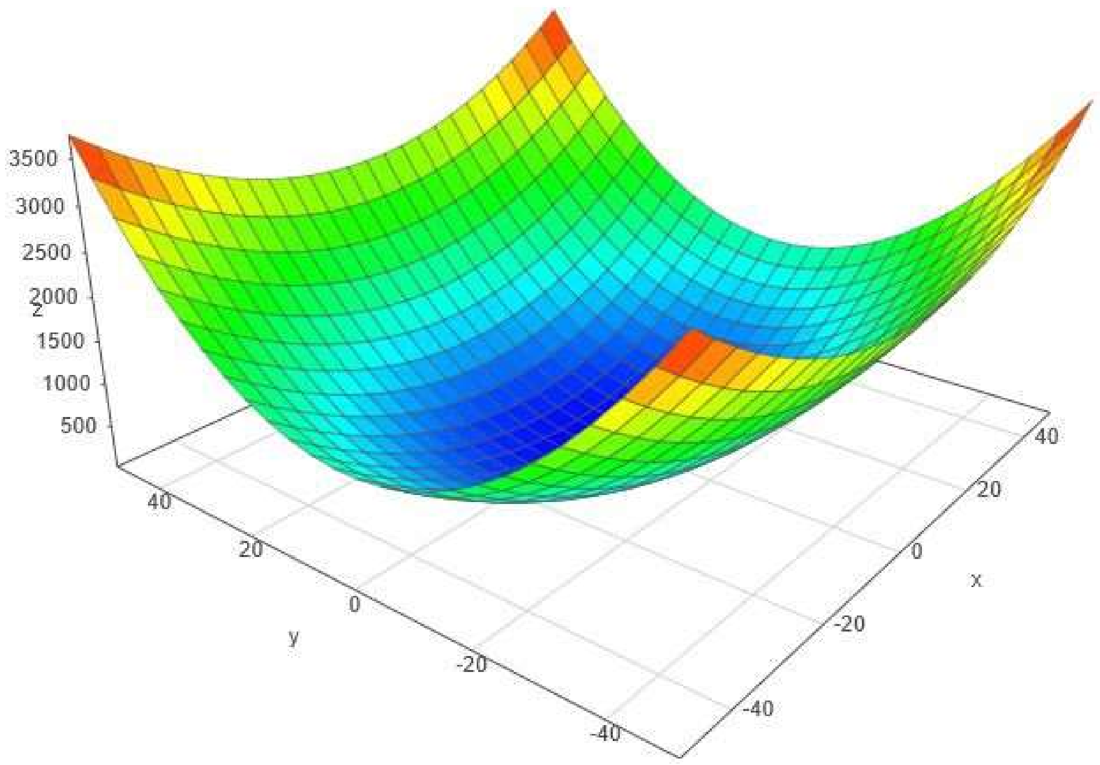
In the case of **a univariate function**, it is simply the **first derivative at a selected point**. In the case of **a multivariate function**, it is a **vector of derivatives** in each main direction (along variable axes). Because we are interested only in a slope along one axis and we don't care about others these derivatives are called **partial derivatives**.

A gradient for an n-dimensional function f(x) at a given point p is defined as follows:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

The upside-down triangle is a so-called *nabla* symbol and you read it "del". To better understand how to calculate it let's do a hand calculation for an exemplary 2-dimensional function below.

$$f(x) = 0.5x^2 + y^2$$



3D plot; Image by author

Let's assume we are interested in a gradient at point p(10,10):

$$\frac{\partial f(x,y)}{\partial x} = x, \quad \frac{\partial f(x,y)}{\partial y} = 2y$$

|

so consequently:

$$\nabla f(x, y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$$

$$\nabla f(10, 10) = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

By looking at these values we conclude that the slope is twice steeper along the y axis.

4. Gradient Descent Algorithm

Gradient Descent Algorithm iteratively calculates the next point using gradient at the current position, scales it (by a learning rate) and subtracts obtained value from the current position (makes a step). It subtracts the value because we want to minimise the function (to maximise it would be adding). This process can be written as:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

There's an important parameter **η** which scales the gradient and thus controls the step size. In machine learning, it is called **learning rate** and have a strong influence on performance.

- The smaller learning rate the longer GD converges, or may reach maximum iteration before reaching the optimum point
- If learning rate is too big the algorithm may not converge to the optimal point (jump around) or even to diverge completely.

In summary, Gradient Descent method's steps are:

1. choose a starting point (initialisation)

2. calculate gradient at this point
3. make a scaled step in the opposite direction to the gradient (objective: minimise)
4. repeat points 2 and 3 until one of the criteria is met:

- maximum number of iterations reached
- step size is smaller than the tolerance (due to scaling or a small gradient).

Below, there's an exemplary implementation of the Gradient Descent algorithm (with steps tracking):

This function takes 5 parameters:

1. **starting poin**t - in our case, we define it manually but in practice, it is often a random initialisation

2. **gradient functio**n - has to be specified before-hand

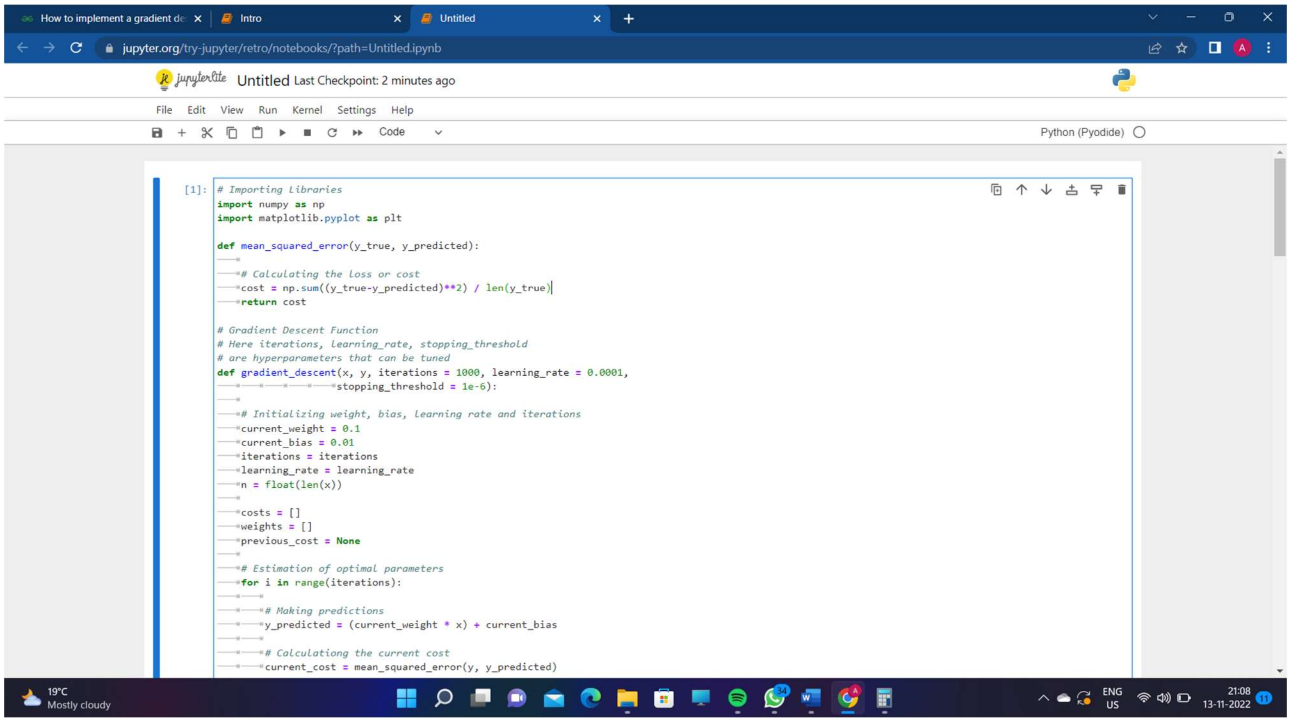
3. **learning rate** - scaling factor for step sizes

4. maximum number of iterations

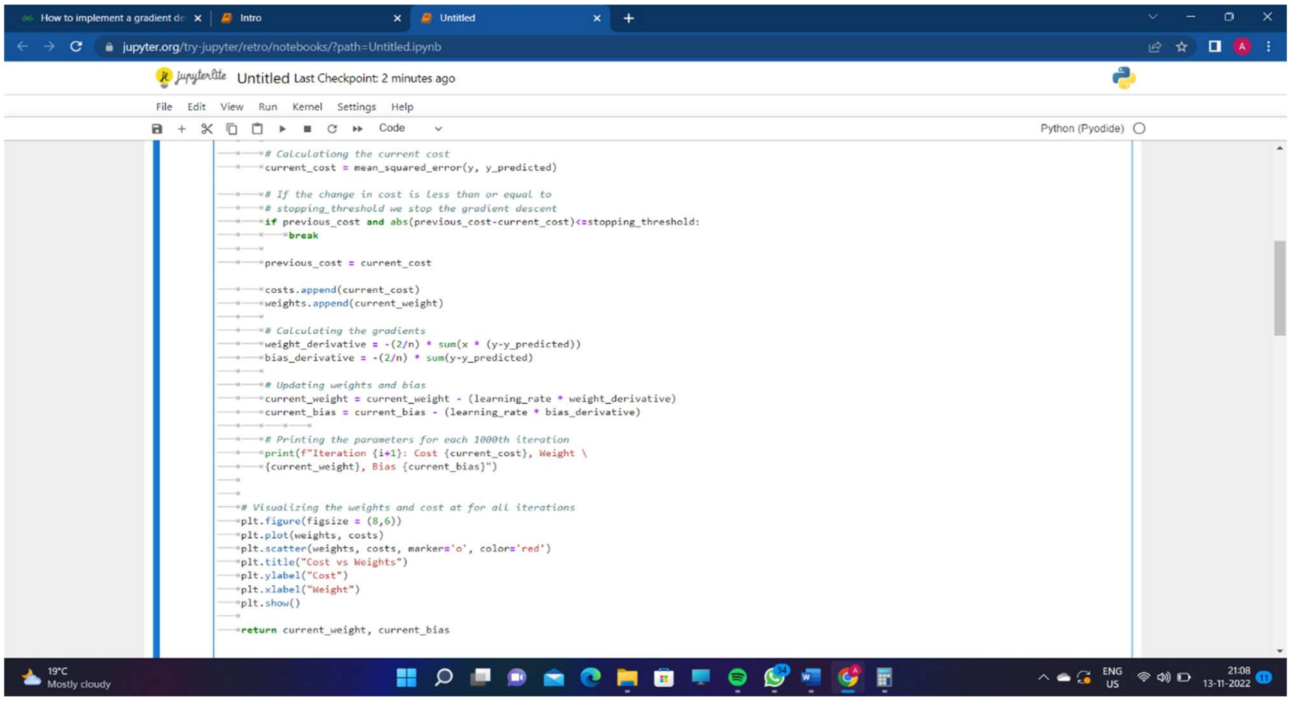
5. tolerance to conditionally stop the algorithm (in this case a default value is 0.01)

Python code

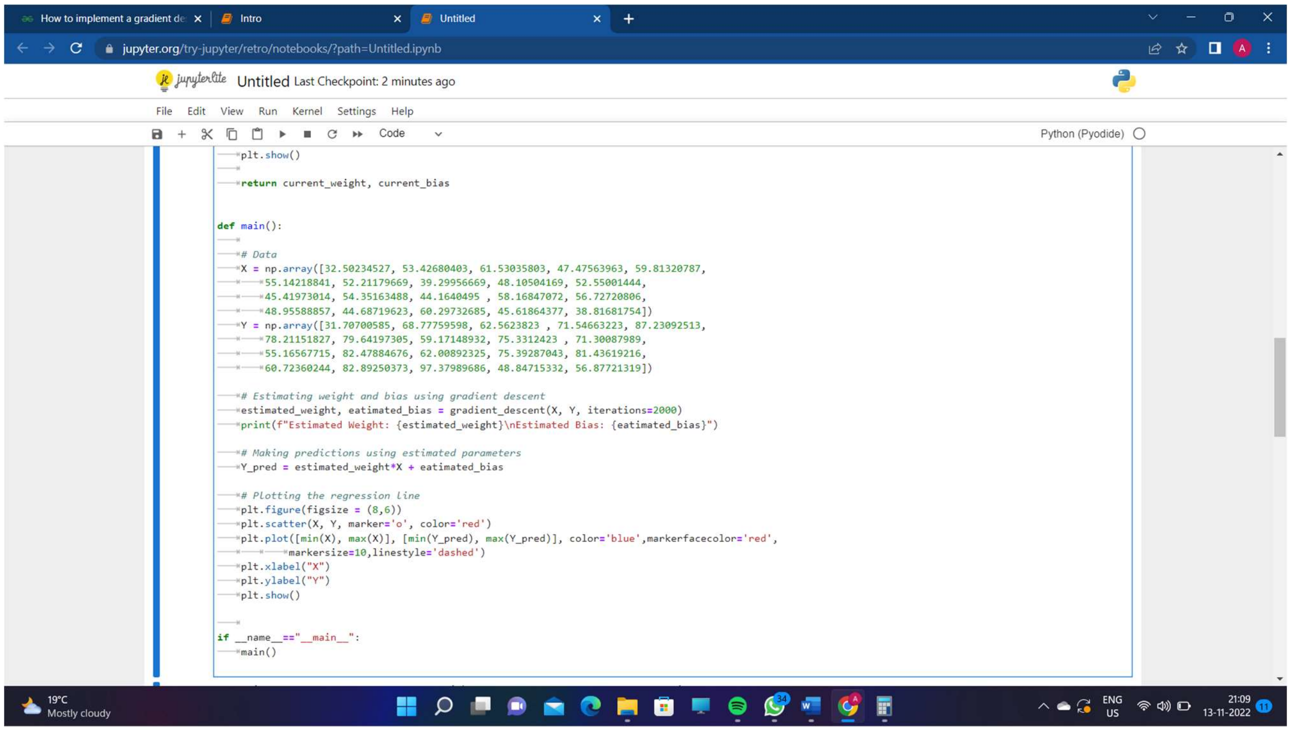
Input code


How to implement a gradient de
Intro
Untitled
jupyter.org/try-jupyter/retro/notebooks/?path=Untitled.ipynb
jupyterlite Untitled Last Checkpoint: 2 minutes ago
File Edit View Run Kernel Settings Help
Code
Python (Pyodide)
19°C
Mostly cloudy
ENG
US
21:08
13-11-2022

```
[1]: # Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

def mean_squared_error(y_true, y_predicted):
    
    # Calculating the loss or cost
    cost = np.sum((y_true-y_predicted)**2) / len(y_true)
    return cost

# Gradient Descent Function
# Here iterations, learning_rate, stopping_threshold
# are hyperparameters that can be tuned
def gradient_descent(x, y, iterations = 1000, learning_rate = 0.0001,
                     stopping_threshold = 1e-6):
    
    # Initializing weight, bias, learning rate and iterations
    current_weight = 0.1
    current_bias = 0.01
    iterations = iterations
    learning_rate = learning_rate
    n = float(len(x))
    
    costs = []
    weights = []
    previous_cost = None
    
    # Estimation of optimal parameters
    for i in range(iterations):
        
        # Making predictions
        y_predicted = (current_weight * x) + current_bias
        
        # Calculationg the current cost
        current_cost = mean_squared_error(y, y_predicted)
```


How to implement a gradient de
Intro
Untitled
jupyter.org/try-jupyter/retro/notebooks/?path=Untitled.ipynb
jupyterlite Untitled Last Checkpoint: 2 minutes ago
File Edit View Run Kernel Settings Help
Code
Python (Pyodide)
19°C
Mostly cloudy
ENG
US
21:08
13-11-2022

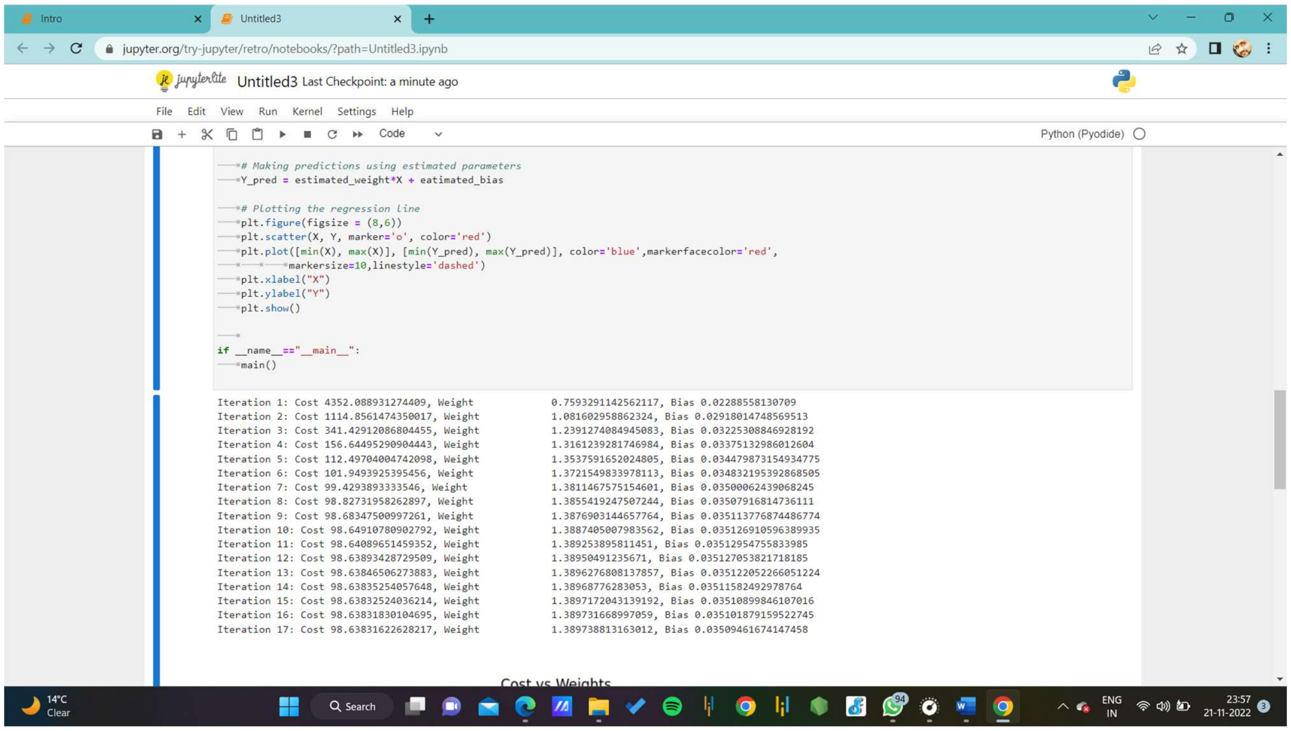
```python
        # Calculationg the current cost
        current_cost = mean_squared_error(y, y_predicted)

        # If the change in cost is less than or equal to
        # stopping_threshold we stop the gradient descent
        if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)

        # Calculating the gradients
        weight_derivative = -(2/n) * sum(x * (y-y_predicted))
        bias_derivative = -(2/n) * sum(y-y_predicted)

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)

        # Printing the parameters for each 1000th iteration
        print(f"Iteration {i+1}: Cost {current_cost}, Weight \
        {current_weight}, Bias {current_bias}")


    # Visualizing the weights and cost at for all iterations
    plt.figure(figsize = (8,6))
    plt.plot(weights, costs)
    plt.scatter(weights, costs, marker='o', color='red')
    plt.title("Cost vs Weights")
    plt.ylabel("Cost")
    plt.xlabel("Weight")
    plt.show()

    return current_weight, current_bias
```


How to implement a gradient de
Intro
Untitled
jupyter.org/try-jupyter/retro/notebooks/?path=Untitled.ipynb
jupyterlite Untitled Last Checkpoint: 2 minutes ago
File Edit View Run Kernel Settings Help
Code
Python (Pyodide)
19°C
Mostly cloudy
ENG
US
21:09
13-11-2022

```python
    plt.show()

    return current_weight, current_bias


def main():

    # Data
    X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,
        55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,
        45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,
        48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])
    Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,
        78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,
        55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,
        60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent
    estimated_weight, eatimated_bias = gradient_descent(X, Y, iterations=2000)
    print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {eatimated_bias}")

    # Making predictions using estimated parameters
    Y_pred = estimated_weight*X + eatimated_bias

    # Plotting the regression line
    plt.figure(figsize = (8,6))
    plt.scatter(X, Y, marker='o', color='red')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfacecolor='red',
            markersize=10,linestyle='dashed')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()


if __name__=="__main__":
    main()
```

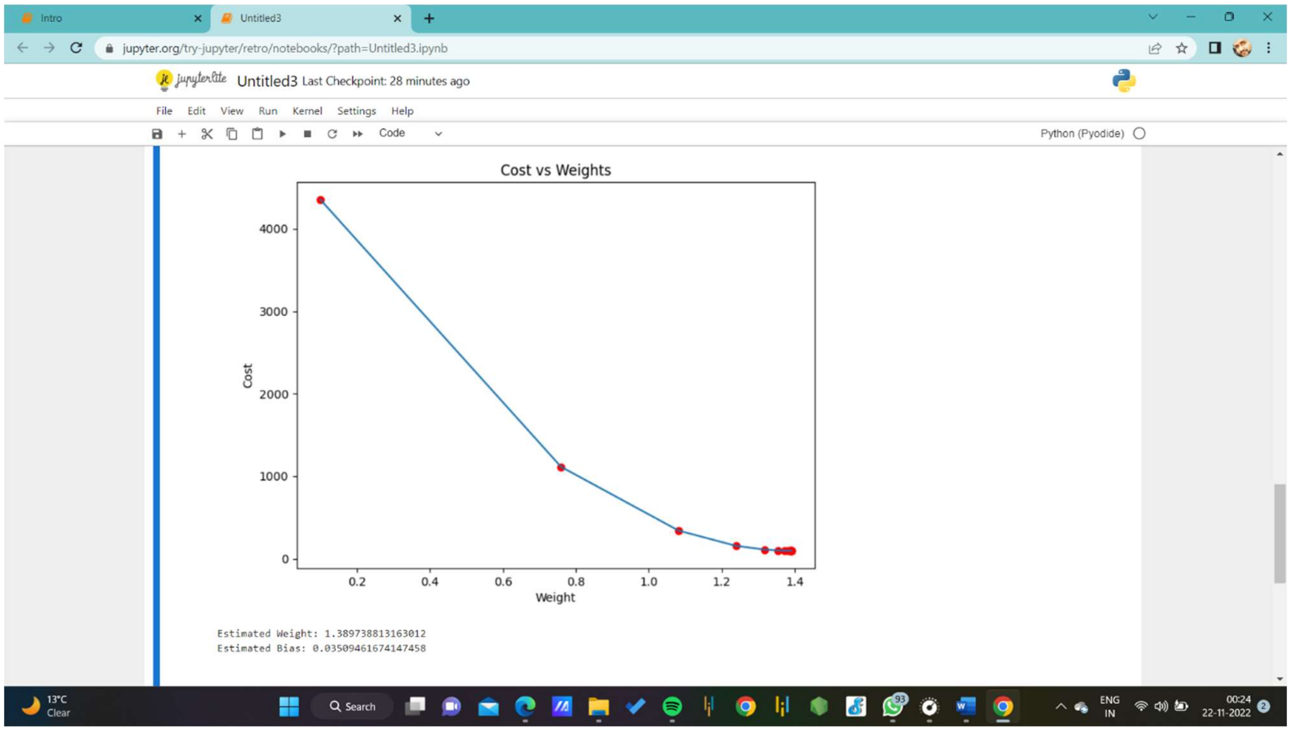
Result & Analysis:

<u>OUTPUT CODE</u>

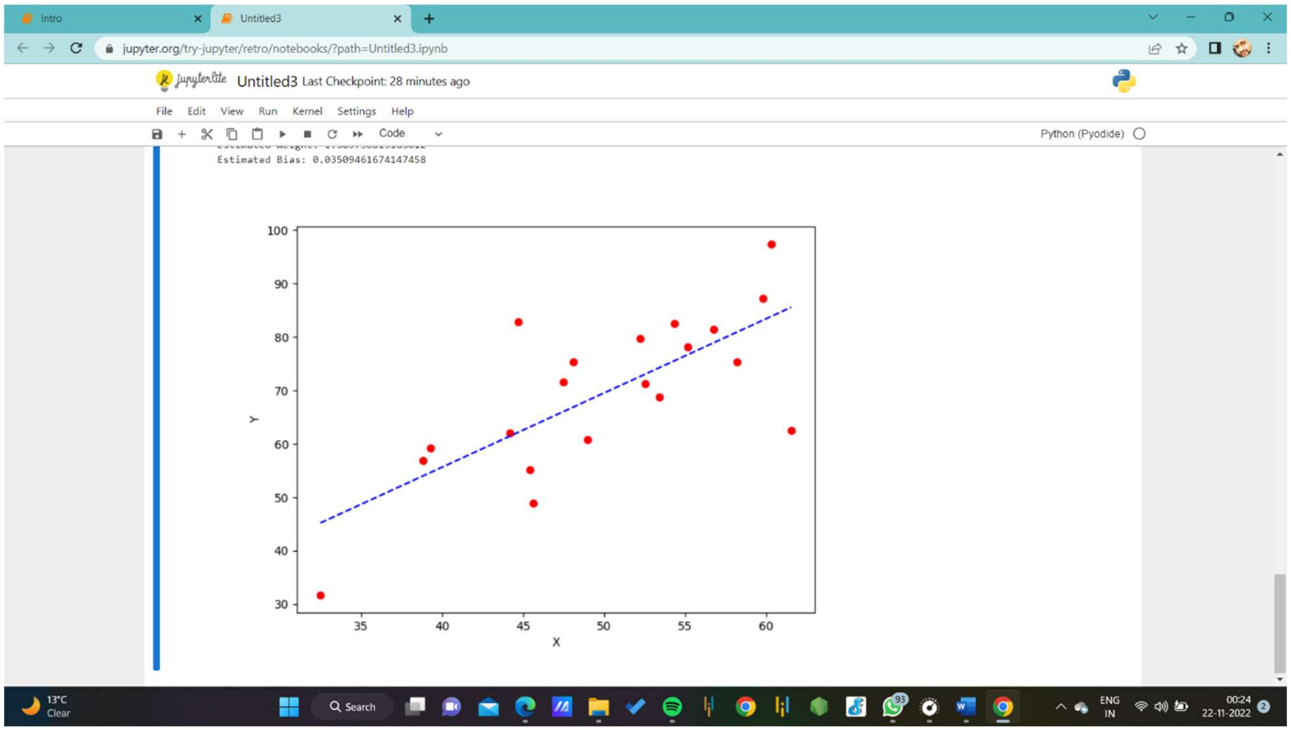

Intro
Untitled3
jupyter.org/try-jupyter/retro/notebooks/?path=Untitled3.ipynb
jupyterlite
Untitled3 Last Checkpoint: a minute ago
File Edit View Run Kernel Settings Help
Code
Python (Pyodide)
14°C Clear
Search
ENG IN
23:57 21-11-2022

```
    # Making predictions using estimated parameters
    Y_pred = estimated_weight*X + eatimated_bias

    # Plotting the regression line
    plt.figure(figsize = (8,6))
    plt.scatter(X, Y, marker='o', color='red')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfacecolor='red',
                    markersize=10,linestyle='dashed')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()


if __name__=="__main__":
    main()
```

```
Iteration 1: Cost 4352.088931274409, Weight           0.7593291142562117, Bias 0.02288558130709
Iteration 2: Cost 1114.8561474350017, Weight          1.081602958862324, Bias 0.02918014748569513
Iteration 3: Cost 341.42912086804455, Weight          1.2391274084945083, Bias 0.03225308846928192
Iteration 4: Cost 156.64495290904443, Weight          1.3161239281746984, Bias 0.03375132986012604
Iteration 5: Cost 112.49704004742098, Weight          1.3537591652024805, Bias 0.034479873154934775
Iteration 6: Cost 101.9493925395456, Weight           1.3721549833978113, Bias 0.034832195392868505
Iteration 7: Cost 99.4293893333546, Weight            1.3811467575154601, Bias 0.03500062439068245
Iteration 8: Cost 98.82731958262897, Weight           1.3855419247507244, Bias 0.03507916814736111
Iteration 9: Cost 98.68347500997261, Weight           1.3876903144657764, Bias 0.035113776874486774
Iteration 10: Cost 98.64910780902792, Weight          1.3887405007983562, Bias 0.035126910596389935
Iteration 11: Cost 98.64089651459352, Weight          1.389253895811451, Bias 0.03512954755833985
Iteration 12: Cost 98.63893428729509, Weight          1.38950491235671, Bias 0.035127053821718185
Iteration 13: Cost 98.63846506273883, Weight          1.3896276808137857, Bias 0.035122052266051224
Iteration 14: Cost 98.63835254057648, Weight          1.38968776283053, Bias 0.03511582492978764
Iteration 15: Cost 98.63832524036214, Weight          1.3897172043139192, Bias 0.03510899846107016
Iteration 16: Cost 98.63831830104695, Weight          1.389731668997059, Bias 0.035101879159522745
Iteration 17: Cost 98.63831622628217, Weight          1.389738813163012, Bias 0.03509461674147458
```



```
Estimated Weight: 1.389738813163012
Estimated Bias: 0.03509461674147458
```



```
Estimated Bias: 0.03509461674147458
```

Project outcome:

CO3: To create awareness of the application areas of neural network techniques.

CO4: To provide alternative solutions to the conventional problem-solving techniques in signal processing, pattern recognition, and classification, control system

Project Conclusion:

we checked how a Gradient Decent algorithm works, when can it be used and what are some common challenges when using it. I hope this will be a good starting point for you to explore more advanced gradient-based optimisation methods like Momentum or Nesterov (Accelerated) Gradient Descent, RMSprop, ADAM or second-order ones like the Newton-Ralphson algorithm.

Appendix

CODE: -

```
# Importing Libraries

import numpy as np

import matplotlib.pyplot as plt

def mean_squared_error(y_true, y_predicted):

# Calculating the loss or cost

cost = np.sum((y_true-y_predicted)**2) / len(y_true)

return cost

# Gradient Descent Function

# Here iterations, learning_rate, stopping_threshold

# are hyperparameters that can be tuned

def gradient_descent(x, y, iterations = 1000, learning_rate = 0.0001,

stopping_threshold = 1e-6):

# Initializing weight, bias, learning rate and iterations

current_weight = 0.1

current_bias = 0.01
```

```
iterations = iterations

learning_rate = learning_rate

            n = float(len(x))

              costs = []

            weights = []

            previous_cost = None

            # Estimation of optimal parameters

            for i in range(iterations):

            # Making predictions

            y_predicted = (current_weight * x) + current_bias

            # Calculationg the current cost

            current_cost = mean_squared_error(y, y_predicted)

             # If the change in cost is less than or equal to

            # stopping_threshold we stop the gradient descent

            if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:

                     break

                      previous_cost = current_cost
```

```
          costs.append(current_cost)

        weights.append(current_weight)

          # Calculating the gradients

        weight_derivative = -(2/n) * sum(x * (y-y_predicted))

        bias_derivative = -(2/n) * sum(y-y_predicted)

        # Updating weights and bias

        current_weight = current_weight - (learning_rate * weight_derivative)

        current_bias = current_bias - (learning_rate * bias_derivative)

        # Printing the parameters for each 1000th iteration

        print(f"Iteration {i+1}: Cost {current_cost}, Weight \

        {current_weight}, Bias {current_bias}")

# Visualizing the weights and cost at for all iterations

  plt.figure(figsize = (8,6))

  plt.plot(weights, costs)

  plt.scatter(weights, costs, marker='o', color='red')

  plt.title("Cost vs Weights")

  plt.ylabel("Cost")
```

```
            plt.xlabel("Weight")

            plt.show()

    return current_weight, current_bias

    def main():

            # Data

            X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,

                    55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

                    45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

                    48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])

            Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,

                    78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,

                    55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,

                    60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent

            estimated_weight, eatimated_bias = gradient_descent(X, Y, iterations=2000)

            print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {eatimated_bias}")

              # Making predictions using estimated parameters
```

```
        Y_pred = estimated_weight*X + eatimated_bias

        # Plotting the regression line

        plt.figure(figsize = (8,6))

        plt.scatter(X, Y, marker='o', color='red')

        plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfacecolor='red',

                        markersize=10, linestyle='dashed')

        plt.xlabel("X")

        plt.ylabel("Y")

        plt.show()

if __name__=="__main__":

        main()
```

GitHub Link: - https://github.com/shivamshashwat26/New-folder