COSC 6364: Advance Numerical Analysis

Assignment – 2

By Dinesh Narlakanti (2083649)

Gradient descent is an optimization algorithm that we commonly use in machine learning and AI to minimize the cost or loss function of a model. Basically, the idea is to update the model parameters iteratively by moving in the direction of steepest descent of the gradient. The gradient is the vector of partial derivatives of the cost function with respect to each parameter, and the lambda determines how much we update the parameters at each iteration. By repeating this process until we converge to a minimum of the cost function, we can optimize the model to perform better on our task. Of course, the convergence point depends on the initial starting point and the lambda we choose.

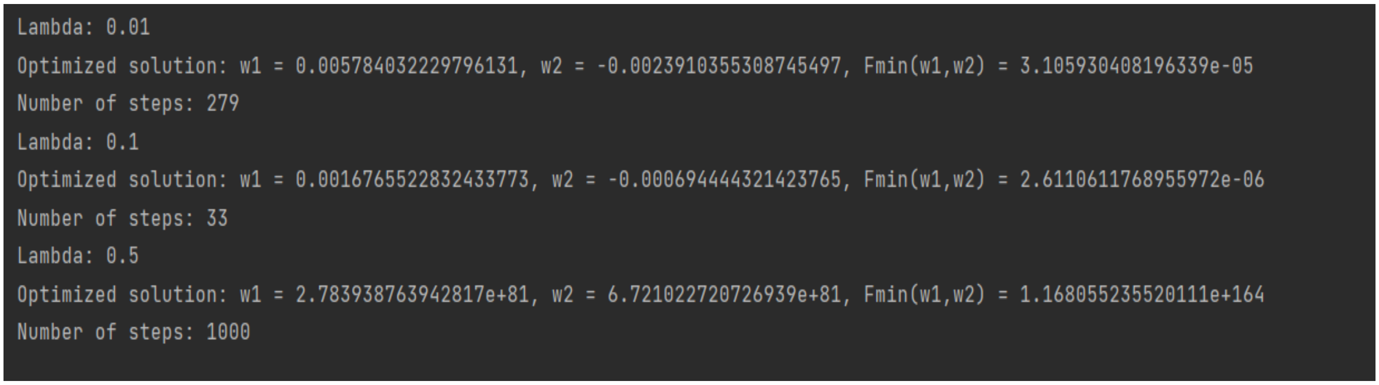
Code Explanation:

The first part of the code defines two functions, f(Wstart) and df(Wstart), which represent the cost function and its derivative, respectively. In this example, the cost function is defined as f(Wstart) = Wstart[0] ** 2 + Wstart[0] * Wstart[1] + 2 * Wstart[1] ** 2, where Wstart is an array of two parameters. The derivative of this cost function with respect to each parameter is computed in df(Wstart), which returns an array of two values.

The code then sets some hyperparameters, including the lambdas, which are the step sizes used to update the parameters at each iteration of the algorithm. Three lambdas are defined as 0.01, 0.1, and 0.5. The maximum number of iterations Nmax is set to 1000 and the tolerance toler is set to 1e-6, which determines when the algorithm has converged.

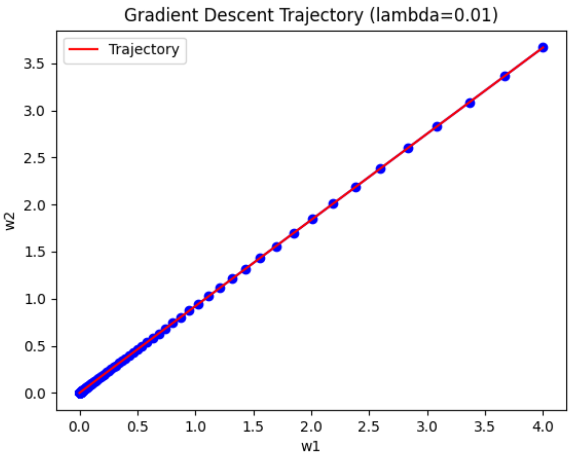
The code then enters a loop that iterates over the lambdas. For each lambda, the initial point Wstart is set to [1.0, 1.0], and the algorithm proceeds to update the parameters of the cost function using gradient descent. The algorithm iteratively computes the gradient of the cost function and updates the parameters in the direction of steepest descent of the gradient. The iteration continues until the change in the function value is smaller than the tolerance value or the maximum number of iterations is reached.

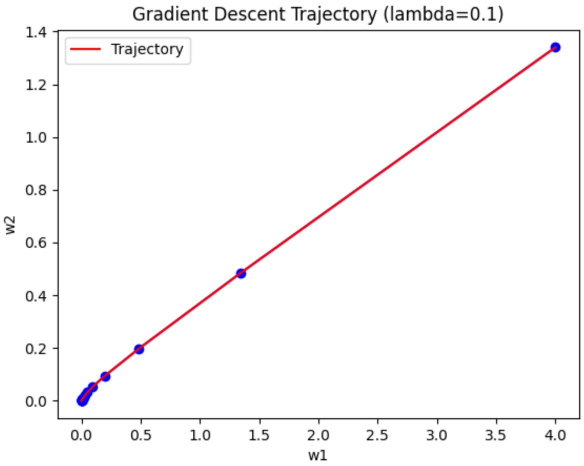
Outputs:

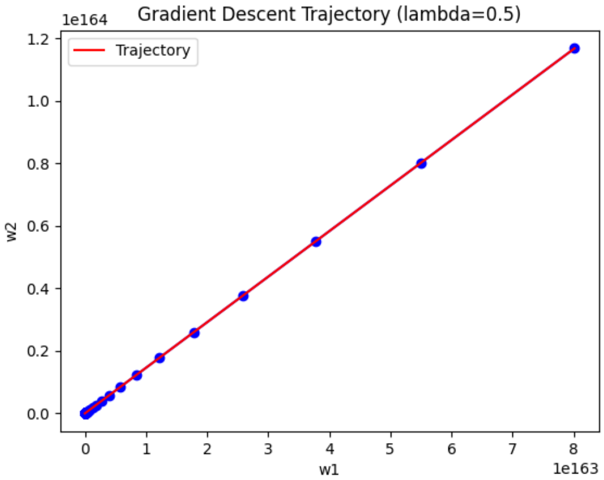


```
Lambda: 0.01
Optimized solution: w1 = 0.005784032229796131, w2 = -0.0023910355308745497, Fmin(w1,w2) = 3.105930408196339e-05
Number of steps: 279
Lambda: 0.1
Optimized solution: w1 = 0.0016765522832433773, w2 = -0.000694444321423765, Fmin(w1,w2) = 2.6110611768955972e-06
Number of steps: 33
Lambda: 0.5
Optimized solution: w1 = 2.783938763942817e+81, w2 = 6.721022720726939e+81, Fmin(w1,w2) = 1.168055235520111e+164
Number of steps: 1000
```

Trajectory at different lamda values:







Analysis of the Output:

The output shows the results of running gradient descent with three different lambdas, λ = 0.01, 0.1, and 0.5.

For λ = 0.01, the algorithm converges after 279 steps and finds an optimized solution with w1 = 0.005784032229796131 and w2 = -0.0023910355308745497, achieving the minimum value of the cost function Fmin(w1,w2) = 3.105930408196339e-05

For λ = 0.1, the algorithm converges after only 33 steps and finds an optimized solution with w1 = 0.0016765522832433773 and w2 = -0.000694444321423765, achieving the minimum value of the cost function Fmin(w1,w2) = 2.6110611768955972e-06.

However, for λ = 0.5, the algorithm fails to converge within the maximum number of steps (1000) and produces very large values for w1 and w2, which is a sign that the algorithm is overshooting the minimum. The minimum value of the cost function Fmin(w1,w2) = 1.168055235520111e+164 is not a valid solution, and the algorithm should be re-tuned with a smaller lambda or different hyperparameters

In general, the results show that the choice of lambda has a significant impact on the performance of the gradient descent algorithm. A smaller lambda leads to slower convergence but better accuracy, while a larger lambda may converge faster but may overshoot the minimum or even diverge. It is important to choose an appropriate lambda that balances convergence speed and accuracy.

Analysis of the graphs:

The trajectory plots show the sequence of (w1, w2) pairs visited by the algorithm during the optimization process, where each point corresponds to the parameter values at a specific iteration. The blue dots indicate the visited points, while the red line represents the trajectory connecting the visited points.

For λ = 0.01 and λ = 0.1, the trajectory plots show a smooth, descending curve that approaches the minimum point in a relatively small number of steps, which is consistent with the convergence behavior observed in the output. The trajectory for λ = 0.1 is more steep than that of λ = 0.01, which is indicative of the faster convergence rate for the larger lambda.

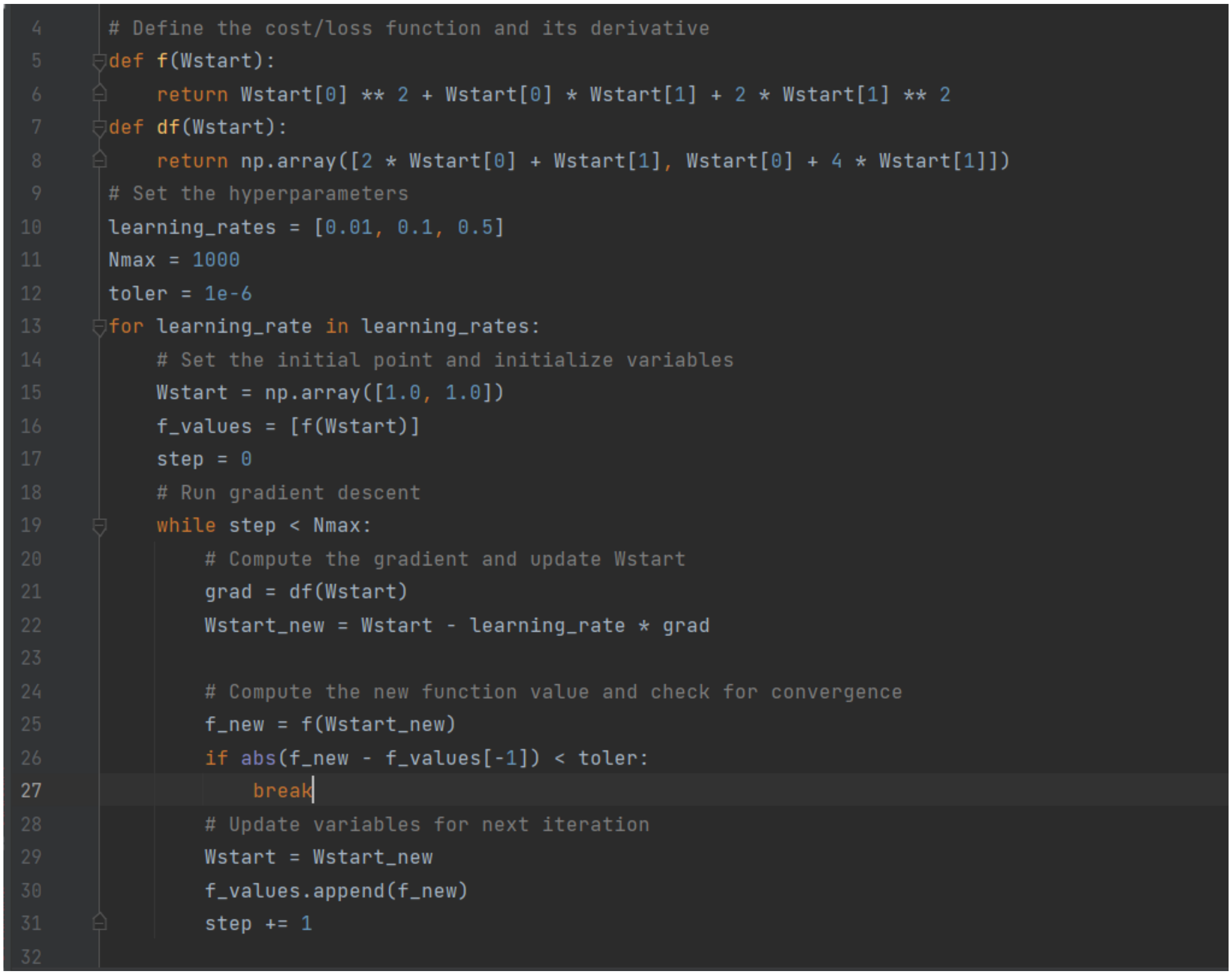
However, for λ = 0.5, the trajectory plot is erratic and does not show a clear descending path towards the minimum. Instead, the algorithm oscillates back and forth between different regions of the parameter space, indicating that the lambda is too large, and the algorithm is overshooting the minimum.

Overall, the trajectory plots provide a useful visualization of the optimization process and can help in understanding the behavior of the algorithm for different lambdas. They also highlight the importance of tuning the lambda to achieve the optimal trade-off between convergence speed and stability.

Failure of gradient descent:

The gradient descent fails for a lambda of 0.5, as it results in divergent behavior and the algorithm is unable to converge to a minimum. A lambda that is too high can cause the algorithm to overshoot the minimum, leading to divergent behavior, while a lambda that is too low can result in very slow convergence. The choice of the lambda often requires trial and error experimentation to find an appropriate value that allows the algorithm to converge efficiently. There are also advanced techniques like adaptive lambda methods that can automatically adjust the lambda during the optimization process to improve convergence.

Code:



```
# Define the cost/loss function and its derivative
def f(Wstart):
    return Wstart[0] ** 2 + Wstart[0] * Wstart[1] + 2 * Wstart[1] ** 2
def df(Wstart):
    return np.array([2 * Wstart[0] + Wstart[1], Wstart[0] + 4 * Wstart[1]])
# Set the hyperparameters
learning_rates = [0.01, 0.1, 0.5]
Nmax = 1000
toler = 1e-6
for learning_rate in learning_rates:
    # Set the initial point and initialize variables
    Wstart = np.array([1.0, 1.0])
    f_values = [f(Wstart)]
    step = 0
    # Run gradient descent
    while step < Nmax:
        # Compute the gradient and update Wstart
        grad = df(Wstart)
        Wstart_new = Wstart - learning_rate * grad

        # Compute the new function value and check for convergence
        f_new = f(Wstart_new)
        if abs(f_new - f_values[-1]) < toler:
            break
        # Update variables for next iteration
        Wstart = Wstart_new
        f_values.append(f_new)
        step += 1

```