Application of Summation and Recurrence Relations in Loss Function Computation for Gradient Descent

Steven Owen Liauw - 13523103
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
owenliauw05@gmail.com, 13523103@std.stei.itb.ac.id

***Abstract*—With the vast development of artificial intelligence, machine learning is becoming more widely used. One of the loss function optimization algorithms in machine learning is gradient descent. The paper examines the role of summation and recurrence relations in determining loss functions for gradient descent, emphasizing their theoretical and computational significance. In order to provide insights into the convergence behavior and computational efficiency of gradient descent, summation and recurrence relations techniques are used to describe the cumulative influence of errors over training data. This paper highlight how crucial these mathematical concepts are for improving optimization strategies and offer a strong basis for future developments in machine learning.**

***Keywords*—Summation, Recurrence Relations, Gradient Descent.**

I. Introduction

In machine learning, gradient descent is a basic optimization technique that is widely used to iteratively minimize loss functions, enabling models to efficiently learn from data. Precise calculations of the loss function and its gradient, which directly influence parameter updates, are essential to the process's effectiveness. Mathematical methods like summation and recurrence relations, which are the foundation of many machine learning systems, are essential to these calculations.

To compute mistakes across the training data and provide a comprehensive assessment of the model's performance, summation is essential. It computes individual errors from every data point into one complete metric, enabling the model to evaluate its overall performance in relation to the whole dataset. This collection establishes the basis for determining essential metrics, like mean squared error or cross-entropy loss, which are important for directing the learning process. By accounting for the total impact of errors, summation guarantees that the gradient descent algorithm is guided by the overall patterns in the data, instead of individual cases of false prediction.

Recurrence relations streamline the sequential updates needed for gradient descent, allowing for a clearer depiction of iterative calculations. These relationships offer a structured method to connect every iteration based on the previous one, which simplifies computation and enhances clarity in the optimization procedure. By reflecting the iterative aspect of parameter changes, recurrence relations also help in analyzing convergence characteristics, assisting in assessing if the algorithm is moving toward an optimal solution. Collectively, these mathematical frameworks provide improved insight into the structure of loss functions, the interplay among model parameters and data, and the fundamental dynamics of convergence behavior.

This paper is intended to examine the application of summation and recurrence relations within gradient descent, emphasizing their theoretical significance and computational effects. This study offers a better insight into the role of these mathematical techniques in loss function computation by analyzing how they support the iterative learning process.

II. Theoretical Framework

A. Summation

Summation, represented by sigma notation, is a mathematical process used to find the total of a series of numbers. It offers a brief method to depict the sum of a sequence of terms.

$$\sum_{i=m}^{n} a_i$$

The summation notation instructs us to substitute each value of i from m (the lower bound) to n (the upper bound) into expression a_i, compute the terms, and then add them together. The result is the total sum of the sequence.

$$\sum_{i=m_1}^{m_2} \sum_{j=n_1}^{n_2} x_{ij}$$

Nested summation builds upon this idea by performing sums across several dimensions, with one summation taking place inside another. This method is frequently used to illustrate more intricate associations, like summing values throughout rows and columns of a dataset or merging various hierarchy levels.

B. Recurrence Relations

A recurrence relation is a mathematical expression that specifies each term in a sequence by referring to one or more of its earlier terms. It offers a structured method to create sequences or series that adhere to particular rules or patterns. For example, if a_n denotes the n_{th} term of a sequence, a recurrence relation might define a_n based on the previous terms in the sequence. This function embodies the conversion or connection between successive terms.

For instance, in the scenario of a straightforward recurrence relation, $a_n = f(a_{n-1})$, the function f illustrates how every term is obtained from the previous one. A typical example is the Fibonacci sequence, where the starting values are set as $a_0 = 1$ and $a_1 = 1$, with later terms calculated using the formula $a_n = a_{n-1} + a_{n-2}$. This relation produces the sequence: 1, 1, 2, 3, 5, 8, 13, ..., in which every term is the total of the two previous terms.

Recurrence relations are deemed linear homogeneous when they consist of a linear combination of previous terms without any extra external terms. This kind of relationship is expressed as:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$$

where $c_1, c_2, \ldots, c_k$ are constants, with c_k not equal to zero. The answer to a linear homogeneous recurrence relation is frequently obtained by proposing a general solution in the format $a_n = r^n$, with r representing a constant. Inserting this into the equation yields the characteristic equation:

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \cdots - c_k = 0.$$

The equation above is the characteristic equation of the recurrence relation, and its characteristic roots represent the solutions to the equation.

C. *Loss Function*

A loss function mathematically expresses the difference between a model's predicted output and the actual target values. It measures the difference, offering a metric that directs the optimization procedure in machine learning and statistical models. The main aim of the loss function is to assess the model's performance and act as the basis for modifying parameters to enhance accuracy.

A loss function, L, takes as input the predicted value, y_{pred}, and the actual target value, y_{true}, and outputs a scalar value that reflects the degree of error in the prediction. Formally, a loss function can be expressed as:

$$L(y_{\text{true}}, y_{\text{pred}})$$

For models with multiple predictions, the overall loss is computed by aggregating individual losses across all data points. This is commonly represented as:

$$L_{\text{total}} = \frac{1}{n}\sum_{i=1}^{n} L\left(y_{\text{true}}^{(i)}, y_{\text{pred}}^{(i)}\right)$$

Some loss functions in regression models, such as Mean Squared Error (MSE) and Mean Absolute Error (MAE), can be expressed as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(y_{\text{pred}}^{(i)} - y_{\text{true}}^{(i)}\right)^2$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|y_{\text{pred}}^{(i)} - y_{\text{true}}^{(i)}\right|$$

While other loss functions in classification models, such as Cross-Entropy Loss and Hinge Loss, can be expressed as:

$$\text{Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_{\text{true}}^{(i)}\log\left(y_{\text{pred}}^{(i)}\right) + \left(1 - y_{\text{true}}^{(i)}\right)\log\left(1 - y_{\text{pred}}^{(i)}\right)\right]$$

$$\text{Hinge Loss} = \frac{1}{n}\sum_{i=1}^{n} max\left(0, 1 - y_{\text{true}}^{(i)} \cdot y_{\text{pred}}^{(i)}\right)$$

The loss function acts as the objective to minimized throughout the training process. Optimization techniques, like gradient descent, adjust model parameters in iterations to minimize the loss. The gradient of the loss function directs the modifications, taking the model towards an ideal solution.

D. *Gradient Descent*

Gradient descent is a key optimization technique commonly used in machine learning, statistics, and computational mathematics to reduce a specific objective function. The algorithm works in iterations, modifying model parameters to minimize the function's value, eventually leading to an optimal solution. The iterative characteristic of gradient descent makes it especially useful for high-dimensional issues, where finding analytical solutions is frequently unfeasible.

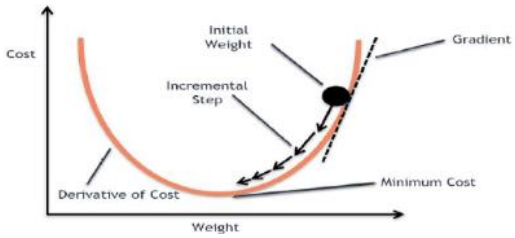


Figure 1. Gradient Descent Algorithm
Source: [2]

At its essence, gradient descent aims to find the minimum of a loss function, $L(\theta)$, where θ represents the parameters of a model. The loss function measures the difference between the predicted and actual values, and reducing this error guarantees that the model's predictions improve in accuracy. The gradient of the loss function, $\nabla L(\theta)$, provides the direction of steepest ascent, pointing toward the values of θ where the function grows most rapidly. By moving in the reverse direction of the gradient, the algorithm guarantees a reduction in the loss function value with every iteration. The general update rule for gradient descent is given by:

$$\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t)$$

Where α is the learning rate, a hyperparameter that determines the step size of each update. The choice of α is important: a

learning rate that is too small leads to slow convergence, while one that is too large risks overshooting the minimum or causing divergence. This balance is essential to ensure both efficiency and stability in the optimization process.

Gradient descent can be categorized into three primary types depending on the method used for calculating the gradient:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m} \nabla L\left(y^{(i)}, \widehat{y^{(i)}}\right)$$

1. Batch Gradient Descent: In this method, the gradient is calculated by using the complete dataset. Although it guarantees a steady decline towards the minimum, batch gradient descent can be costly in terms of computation for extensive datasets, since it necessitates handling all the data during each iteration.

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla L\left(y^{(i)}, \widehat{y^{(i)}}\right)$$

2. Stochastic Gradient Descent (SGD): Rather than using the full dataset, SGD determines the gradient using just one data point at each iteration. Although this approach improves computational efficiency, the updates may cause noise, resulting in oscillations near the minimum instead of a smooth convergence.

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{b}\sum_{i \in B} \nabla L\left(y^{(i)}, \widehat{y^{(i)}}\right)$$

3. Mini-Batch Gradient Descent: This version finds a middle ground between batch and stochastic methods by calculating the gradient over small groups (mini-batches) of the data. Mini-batch gradient descent is commonly used in practice because of its computational effectiveness and consistent updates.

The effectiveness of gradient descent depends on the characteristics of the loss function. For convex functions, since every local minimum is a global minimum, gradient descent is assured to reach the optimal solution, as long as the learning rate is selected correctly. Nonetheless, for non-convex functions, like those frequently found in neural networks, gradient descent might settle at a local minimum or saddle point instead of reaching the global minimum. Nonetheless, non-convex optimization frequently produces solutions that are adequately effective for real-world applications.

E. Linear Regression

Linear regression is a basic statistical and machine learning method used to model and examine the connection between one or more independent variables (features) and a dependent variable (target). The main objective of linear regression is to identify the optimal line or hyperplane that reduces the difference between the predicted and actual values of the dependent variable.

The linear regression model assumes that the relationship between the input features and the target variable is linear. For a single feature, the model is represented as:

$$f_{w,b}\left(x^{(i)}\right) = wx^{(i)} + b$$

Where:

- $f_{w,b}\left(x^{(i)}\right)$ is the predicted value for the i_{th} data point
- $x^{(i)}$ is the input feature value.
- w is the weight (slope of the line) that defines the impact of the feature.
- b is the bias (intercept), which shifts the line vertically.

For multiple features, the model generalizes to:

$$f_{w,b}\left(x^{(i)}\right) = w^{\top} x^{(i)} + b$$

Where:

- $x^{(i)}$ is the vector of feature values for the i_{th} data point.
- w is the vector of weights corresponding to the features.
- $w^{\top} x^{(i)}$ is the dot product of the weights and features.

To find the optimal values of w and b, gradient descent is often used. Gradient descent iteratively adjusts w and b in the direction that reduces the loss function. The update rules are given by:

$$w \leftarrow w - \alpha\frac{\partial L}{\partial w}, \quad b \leftarrow b - \alpha\frac{\partial L}{\partial b}$$

Where:

- α is the learning rate, controlling the step size.
- $\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$ are the gradients of the loss function with respect to w and b

For multiple features, the gradient with respect to the weights becomes:

$$\frac{\partial L}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right)x^{(i)}$$

III. IMPLEMENTATION

This implementation uses Python to perform simple linear regression through gradient descent, constructed completely from scratch. The aim is to highlight the importance of summation and recurrence relations in the computational steps.

A. Tools

In this implementation, we will make use of:

- Numpy
- Matplotlib
- Math

```python
import math, copy
import numpy as np
import matplotlib.pyplot as plt
```

Figure 2. Importing python libraries
Source: writer's archive

B. Load Dataset

```python
# Load artificial set
x_train = np.array([1.0, 2.0, 3.0, 4.0, 5.0])   #features
y_train = np.array([300.0, 500.0, 650.0, 450.0, 720.0])   #target value
```

Figure 3. Load artificial dataset
Source: writer's archive

C. Compute Loss Function

```python
def compute_loss(x, y, w, b):

    m = x.shape[0]
    cost = 0

    for i in range(m):
        f_wb = w * x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost
```

Figure 4. compute_loss function
Source: writer's archive

The compute_loss function determines the Mean Squared Error (MSE) cost associated with linear regression, It takes the input data x (attributes), y (outcomes), and the model parameters w (weights) and b (intercept). The function iterates through every data point, determines the predicted value using the linear equation, and calculates the square of the error between the predicted value and the real target value.

D. Equations

```python
def compute_gradient(x, y, w, b):
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0

    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_dw, dj_db
```

Figure 5. compute_gradient function
Source: writer's archive

The compute_gradient function determines the gradients of the loss function concerning the model parameters w (weight) and b (bias) in linear regression. The inputs consist of the feature values x, target values y, along with the current parameters w and b. For each data point, it calculates the error between the predicted value $f_{wb} = w \cdot x[i] + b$ and the actual target value $y[i]$. The gradient with respect to w (dj_dw) is computed by multiplying this error by the feature value $x[i]$ and the gradient with respect to b (dj_db) is the error itself. These individual gradients are totaled across all data points and subsequently averaged by dividing by the total number of data points. The function returns the averaged gradients (dj_dw) and (dj_db).

E. Gradient Descent

```python
def gradient_descent(x, y, w_in, b_in, alpha, num_iters, compute_loss, gradient_function):
    w = copy.deepcopy(w_in)
    J_history = []
    p_history = []
    b = b_in
    w = w_in

    for i in range(num_iters):
        dj_dw, dj_db = gradient_function(x, y, w, b)
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        if i < 100000:
            J_history.append(compute_loss(x, y, w, b))
            p_history.append([w, b])

        if i % math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4}: Loss {J_history[-1]:0.2e} ",
                  f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
                  f"w: {w: 0.3e}, b:{b: 0.5e}")

    return w, b, J_history, p_history
```

Figure 6. gradient_descent function
Source: writer's archive

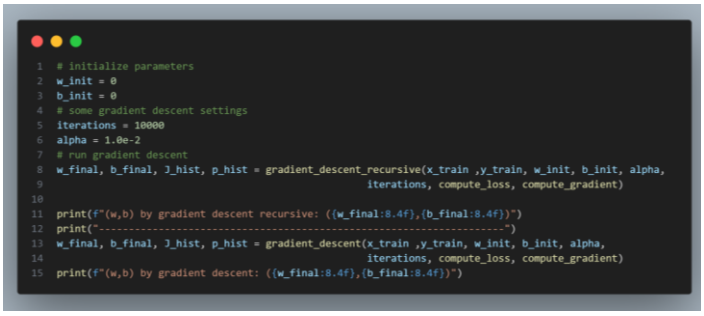
The gradient_descent function repeatedly modifies the model parameters w (weight) and b (bias) using the gradients derived from the gradient_function. Beginning with starting values of w and b , the function modifies them towards reducing the loss function. The learning rate (α) regulates the magnitude of the steps taken for these updates. The historical data of the loss values (J_history) and parameter values $(p\_history)$ is recorded for future visualization. After the specified number of iterations, the updated w,b and the history of loss and parameters are returned.

```python
def gradient_descent_recursive(x, y, w, b, alpha, num_iters,
                               compute_loss, gradient_function,
                               J_history=None, p_history=None, i=0):
    if J_history is None:
        J_history = []
    if p_history is None:
        p_history = []

    if i >= num_iters:
        return w, b, J_history, p_history

    dj_dw, dj_db = gradient_function(x, y, w, b)
    b = b - alpha * dj_db
    w = w - alpha * dj_dw

    if i < 100000:
        J_history.append(compute_loss(x, y, w, b))
        p_history.append([w, b])

    if i % max(1, math.ceil(num_iters / 10)) == 0:
        print(f"Iteration {i:4}: Loss {J_history[-1]:0.2e} ",
              f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
              f"w: {w: 0.3e}, b:{b: 0.5e}")

    return gradient_descent_recursive(x, y, w, b, alpha, num_iters,
                                      compute_loss, gradient_function,
                                      J_history, p_history, i + 1)
```

Figure 7. gradient_descent_recursive function
Source: writer's archive

Alternatively, we can also make gradient descent with a recursive approach, where each iteration is handled by a recursive call instead of using a loop. The base case checks if the current iteration index (i) exceeds the specified number of iterations (num_iters), in which case it stops the recursion and returns the final results. At each recursive step, the function calculates the gradients (dj_dw) and (dj_db) using the provided gradient_function. These gradients are used to modify the parameters w and b using the gradient descent formula. The revised values are subsequently saved in the history lists together with the calculated loss via the compute_loss function.

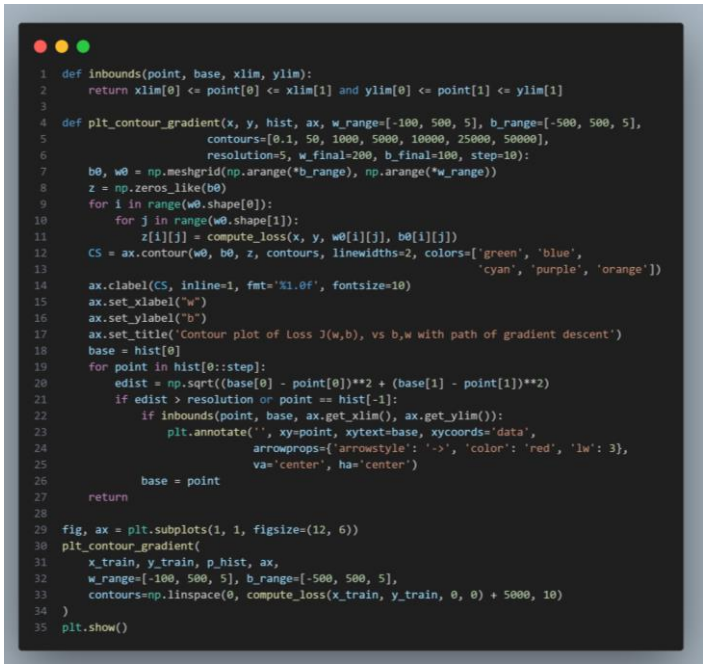
F. *Gradient Descent Visualization*



```python
# initialize parameters
w_init = 0
b_init = 0
# some gradient descent settings
iterations = 10000
alpha = 1.0e-2
# run gradient descent
w_final, b_final, J_hist, p_hist = gradient_descent_recursive(x_train ,y_train, w_init, b_init, alpha,
                                                              iterations, compute_loss, compute_gradient)

print(f"(w,b) by gradient descent recursive: ({w_final:8.4f},{b_final:8.4f})")
print("-----------------------------------------------------------------------")
w_final, b_final, J_hist, p_hist = gradient_descent(x_train ,y_train, w_init, b_init, alpha,
                                                    iterations, compute_loss, compute_gradient)
print(f"(w,b) by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
```

Figure 8. final w and b
Source: writer's archive

The code runs gradient descent twice, once using a recursive approach and once using an iterative approach to find the optimal values of w (weight) and b (bias) that minimize the lost function. It initializes the parameters and settings, then prints the final values of w and b for both methods, comparing their results.



```python
def inbounds(point, base, xlim, ylim):
    return xlim[0] <= point[0] <= xlim[1] and ylim[0] <= point[1] <= ylim[1]

def plt_contour_gradient(x, y, hist, ax, w_range=[-100, 500, 5], b_range=[-500, 500, 5],
                         contours=[0.1, 50, 1000, 5000, 10000, 25000, 50000],
                         resolution=5, w_final=200, b_final=100, step=10):
    b0, w0 = np.meshgrid(np.arange(*b_range), np.arange(*w_range))
    z = np.zeros_like(b0)
    for i in range(w0.shape[0]):
        for j in range(w0.shape[1]):
            z[i][j] = compute_loss(x, y, w0[i][j], b0[i][j])
    CS = ax.contour(w0, b0, z, contours, linewidths=2, colors=['green', 'blue',
                                                               'cyan', 'purple', 'orange'])
    ax.clabel(CS, inline=1, fmt='%1.0f', fontsize=10)
    ax.set_xlabel("w")
    ax.set_ylabel("b")
    ax.set_title('Contour plot of Loss J(w,b), vs b,w with path of gradient descent')
    base = hist[0]
    for point in hist[0::step]:
        edist = np.sqrt((base[0] - point[0])**2 + (base[1] - point[1])**2)
        if edist > resolution or point == hist[-1]:
            if inbounds(point, base, ax.get_xlim(), ax.get_ylim()):
                plt.annotate('', xy=point, xytext=base, xycoords='data',
                             arrowprops={'arrowstyle': '->', 'color': 'red', 'lw': 3},
                             va='center', ha='center')
            base = point
    return

fig, ax = plt.subplots(1, 1, figsize=(12, 6))
plt_contour_gradient(
    x_train, y_train, p_hist, ax,
    w_range=[-100, 500, 5], b_range=[-500, 500, 5],
    contours=np.linspace(0, compute_loss(x_train, y_train, 0, 0) + 5000, 10)
)
plt.show()
```

Fig 9. plot_contour_gradient function
Source: writer's archive

The code runs a function that plots the contour of a loss function $J(w, b)$ for linear regression and overlays the gradient descent path. A helper function, inbounds, also ensures that arrows are only plotted if they fall within the visible plot range.

```python
def plot_loss_vs_iteration(J_hist):
    fig,axes = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
    ax1, ax2 = axes
    ax1.plot(J_hist[:100])
    ax2.plot(1000 + np.arange(len(J_hist[1000:])), J_hist[1000:])
    ax1.set_title("Loss vs. iteration (start)")
    ax2.set_title("Loss vs. iteration (end)")
    ax1.set_ylabel('Loss')
    ax2.set_ylabel('Loss')
    ax1.set_xlabel('Iteration step')
    ax2.set_xlabel('Iteration step')
    plt.show()

plot_loss_vs_iteration(J_hist)
```

Figure 10. plot_loss_vs_iteration function
Source: writer's archive

The code runs a function, plot_loss_vs_iteration, that displays how the loss function behaves across iterations in the process of gradient descent. It creates two subplots, left plot displays the loss function values over the initial 100 iterations to observe the early stages rapid changes, while right plot displays the loss function values starting from the 1000th iteration onward to observe the convergence at later stages.

IV. RESULTS

A. *Final w and b*

```
Iteration    0: Loss 1.18e+05  dj_dw: -1.730e+03, dj_db: -5.240e+02   w:  1.730e+01, b: 5.24000e+00
Iteration 1000: Loss 5.00e+03  dj_dw:  2.123e+00, dj_db: -7.666e+00   w:  9.154e+01, b: 2.41730e+02
Iteration 2000: Loss 4.82e+03  dj_dw:  3.910e-01, dj_db: -1.412e+00   w:  8.131e+01, b: 2.78663e+02
Iteration 3000: Loss 4.81e+03  dj_dw:  7.202e-02, dj_db: -2.600e-01   w:  7.943e+01, b: 2.85465e+02
Iteration 4000: Loss 4.81e+03  dj_dw:  1.326e-02, dj_db: -4.788e-02   w:  7.908e+01, b: 2.86717e+02
Iteration 5000: Loss 4.81e+03  dj_dw:  2.443e-03, dj_db: -8.819e-03   w:  7.901e+01, b: 2.86948e+02
Iteration 6000: Loss 4.81e+03  dj_dw:  4.499e-04, dj_db: -1.624e-03   w:  7.900e+01, b: 2.86990e+02
Iteration 7000: Loss 4.81e+03  dj_dw:  8.285e-05, dj_db: -2.991e-04   w:  7.900e+01, b: 2.86998e+02
Iteration 8000: Loss 4.81e+03  dj_dw:  1.526e-05, dj_db: -5.509e-05   w:  7.900e+01, b: 2.87000e+02
Iteration 9000: Loss 4.81e+03  dj_dw:  2.810e-06, dj_db: -1.015e-05   w:  7.900e+01, b: 2.87000e+02
(w,b) by gradient descent recursive: ( 79.0000,287.0000)
---------------------------------------------------------------------
Iteration    0: Loss 1.18e+05  dj_dw: -1.730e+03, dj_db: -5.240e+02   w:  1.730e+01, b: 5.24000e+00
Iteration 1000: Loss 5.00e+03  dj_dw:  2.123e+00, dj_db: -7.666e+00   w:  9.154e+01, b: 2.41730e+02
Iteration 2000: Loss 4.82e+03  dj_dw:  3.910e-01, dj_db: -1.412e+00   w:  8.131e+01, b: 2.78663e+02
Iteration 3000: Loss 4.81e+03  dj_dw:  7.202e-02, dj_db: -2.600e-01   w:  7.943e+01, b: 2.85465e+02
Iteration 4000: Loss 4.81e+03  dj_dw:  1.326e-02, dj_db: -4.788e-02   w:  7.908e+01, b: 2.86717e+02
Iteration 5000: Loss 4.81e+03  dj_dw:  2.443e-03, dj_db: -8.819e-03   w:  7.901e+01, b: 2.86948e+02
Iteration 6000: Loss 4.81e+03  dj_dw:  4.499e-04, dj_db: -1.624e-03   w:  7.900e+01, b: 2.86990e+02
Iteration 7000: Loss 4.81e+03  dj_dw:  8.285e-05, dj_db: -2.991e-04   w:  7.900e+01, b: 2.86998e+02
Iteration 8000: Loss 4.81e+03  dj_dw:  1.526e-05, dj_db: -5.509e-05   w:  7.900e+01, b: 2.87000e+02
Iteration 9000: Loss 4.81e+03  dj_dw:  2.810e-06, dj_db: -1.015e-05   w:  7.900e+01, b: 2.87000e+02
(w,b) by gradient descent: ( 79.0000,287.0000)
```

Figure 11. Gradient decent *w* and *b*
Source: writer's archive

This result shows that the recursive or iterative approach, does not influence the convergence of the gradient descent algorithm, provided that the mathematical calculations and hyperparameters (learning rate and iteration count) stay the same. Both methods successfully reduce the loss function, resulting in identical optimal parameters.

B. *Gradient descent contour map*

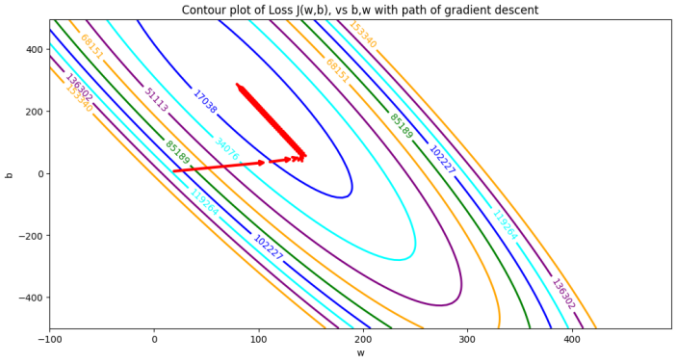


Figure 12. Gradient descent contour map
Source: writer's archive

The gradient descent contour map illustrates that during each iteration, the algorithm gradually shifts toward areas of reduced loss, ultimately stopping at the minimum point where the loss function reaches its lowest value. This is depicted on the contour map as a collection of red dots and arrows that outline the path of the parameters over the loss landscape. As the gradient descent algorithm nears the optimal values, the updates to the parameters shrink, ensuring smooth convergence and preventing overshooting. This visual representation validates that the optimization procedure is operating properly and effectively minimizes the error at every stage.

C. *Plot loss vs iteration*

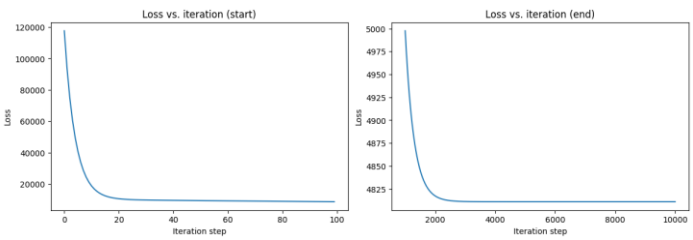


Figure 13. Loss vs iteration
Source: writer's archive

The graph shows loss in relation to iterations indicates that initially in gradient descent, the loss declines rapidly as the algorithm makes larger strides driven by steep gradients. Gradually, the expenses drop at a slower rate, and the curve levels off as the algorithm nears the minimum. Ultimately, the loss stabilize, indicating that the algorithm has reached or is nearing the optimal parameters.

V. Conclusion

This research explored the importance of summation and recurrence relations in the gradient descent optimization method. Summation provides a clear way to combine the total error from all training samples, forming the basis for loss function calculations. This ensures that the optimization process focuses on the model's overall performance rather than individual cases.

Recurrence relations allow for a systematic and repetitive method for adjusting parameters. It is possible to standardize and simplify the calculations needed for each step by describing the gradient descent updates as recurrence formulae. This guarantees steady progress in lowering the loss function and offers a precise mathematical foundation for understanding the algorithm's convergence behavior.

Summation and recurrence relations, when combined, form the core of gradient descent, underpinning both its theoretical frameworks and real-world applications. These mathematical concepts are crucial for enabling optimization in machine learning models and ensure that gradient descent works efficiently whether used recursively or iteratively.

VI. Appendix

Link Video:
https://www.canva.com/design/DAGbCzc9EMM/PtwpeA7qbBHVizJL0-mldQ/view?utm_content=DAGbCzc9EMM&utm_campaign=designshare&utm_medium=link&utm_source=recording_view

VII. Acknowledgment

The completion of this paper would not have been possible without the support and encouragement of many people. First, I thank God for providing the strength, wisdom, and determination to finish this work.

I sincerely thank Dr. Ir. Rinaldi, M.T, the lecturer for Discrete Mathematics (K-01), whose insightful lectures and helpful feedback were crucial in developing the concepts examined in this paper.

I am also deeply grateful to my parents for their constant support, love, and encouragement throughout my studies. Finally, I extend my thanks to my friends, whose guidance and companionship made this work more enjoyable and meaningful.

References

[1] Ganie, A.G., Dadvandipour, S. From big data to smart data: a sample gradient descent approach for machine learning. J

Big Data 10, 162 (2023). https://doi.org/10.1186/s40537-023-00839-9.
[2] Hikmat, Saad & Abdulazeez, Adnan & Saad, Hikmat & Haji, Adnan & Mohsin, Abdulazeez. (2021). COMPARISON OF OPTIMIZATION TECHNIQUES BASED ON GRADIENT DESCENT ALGORITHM: A REVIEW PJAEE, 18 (4) (2021) COMPARISON OF OPTIMIZATION TECHNIQUES BASED ON GRADIENT DESCENT ALGORITHM: A REVIEW Comparison Of Optimization Techniques Based On Gradient Descent Algorithm: A Review--Palarch's Journal Of Archaeology Of Egypt/Egyptology 18(4).
[3] "Homepage Rinaldi Munir." https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/10-Rekursi-dan-relasi-rekurens-(Bagian2)-2023.pdf (accessed Dec. 28, 2024).
[4] Jupudi, Lakshmi & Scholar, Research. (2016). Stochastic Gradient Descent using Linear Regression with Python. IJAERA. 2. 519 -525.
[5] K. H. Rosen, "Discrete Mathematics and Its Applications," 8th ed. New York, NY: McGraw Hill, 2019.
[6] Manorathna, Rukshan. (2020). Linear Regression with Gradient Descent.
[7] Rani, Parvathy & Devi, Rani. (2014). Gradient descent based linear regression approach for modeling PID parameters. 2014 International Conference on Power Signals Control and Computations, EPSCICON 2014. 1-4. 10.1109/EPSCICON.2014.6887482.
[8] Sarmento, Rui & Costa, Vera. (2017). Introduction to Linear Regression. 10.4018/978-1-68318-016-6.ch006.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 31 Desember 2024

Steven Owen - 13523103