|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

Experiment No. 07

|  |  |
| --- | --- |
| Semester | B.E. Semester VIII – Computer Engineering |
| Subject | Deep Learning Lab |
| Subject Professor In-charge | Prof. Kavita Shirsat |
| Academic Year | 2024-25 |

|  |  |
| --- | --- |
| Student Name | Deep Salunkhe |
| Roll Number | 21102A0014 |

**Title:** Implement MGD ,(Adagrad GD) and Adam optimization algorithm

Optimization algorithms are essential in machine learning for minimizing error functions like Mean Squared Error (MSE). This report compares Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent (MGD), Adaptive Gradient Descent (Adagrad), and Adam Gradient Descent (Adam GD) in terms of accuracy, loss, and convergence speed.

**1. Stochastic Gradient Descent (SGD)**

SGD updates model parameters using a single training example at a time. This makes it faster per iteration but introduces high variance in updates.

Steps:

1. Select a random training sample (xᵢ, yᵢ).

2. Compute the gradient:

- gradient\_w = - (yᵢ - (w \* xᵢ + b)) \* xᵢ

- gradient\_b = - (yᵢ - (w \* xᵢ + b))

3. Update parameters:

- w = w - η \* gradient\_w

- b = b - η \* gradient\_b

Advantages:

- Efficient for large datasets.

- Can escape local minima due to randomness.

Disadvantages:

- High variance in updates.

- Can be unstable and slow to converge.

**2. Mini-Batch Gradient Descent (MGD)**

MGD processes small batches of data rather than a single sample or the entire dataset. This balances stability and efficiency.

Steps:

1. Divide dataset into small batches.

2. Compute gradients for each batch.

3. Apply updates with momentum:

- v\_w = β \* v\_w + η \* gradient\_w

- v\_b = β \* v\_b + η \* gradient\_b

- w = w - v\_w

- b = b - v\_b

Advantages:

- More stable than SGD.

- Faster than full-batch gradient descent.

Disadvantages:

- Batch size must be carefully chosen.

- May still struggle with complex optimization landscapes.

**3. Adaptive Gradient Descent (Adagrad)**

Adagrad adjusts the learning rate for each parameter based on past gradients, making it effective for sparse data.

Steps:

1. Accumulate squared gradients:

- G\_w = G\_w + gradient\_w^2

- G\_b = G\_b + gradient\_b^2

2. Update parameters:

- w = w - (η / sqrt(G\_w + ε)) \* gradient\_w

- b = b - (η / sqrt(G\_b + ε)) \* gradient\_b

Advantages:

- Adjusts learning rate dynamically.

- Works well for sparse datasets.

Disadvantages:

- Learning rate decreases over time.

- May stop learning too early.

**4. Adam Gradient Descent (Adam GD)**

Adam combines momentum from MGD with an adaptive learning rate like Adagrad, making it widely used.

Steps:

1. Compute first-moment estimate (momentum):

- m\_t = β1 \* m\_(t-1) + (1 - β1) \* gradient\_w

2. Compute second-moment estimate:

- v\_t = β2 \* v\_(t-1) + (1 - β2) \* gradient\_w^2

3. Bias correction:

- m̂\_t = m\_t / (1 - β1^t)

- v̂\_t = v\_t / (1 - β2^t)

4. Update parameters:

- w = w - (η / (sqrt(v̂\_t) + ε)) \* m̂\_t

Advantages:

- Fast convergence.

- Works well for deep learning and noisy gradients.

Disadvantages:

- Requires careful tuning.

- May not generalize to all problems.

**5. Comparative Analysis**

SGD is suitable for large datasets but has unstable updates due to high variance. MGD improves stability while maintaining efficiency but needs careful batch size selection. Adagrad adapts learning rates dynamically, making it effective for sparse data, but its learning rate decays too much over time. Adam GD provides the fastest and most stable convergence, combining the benefits of MGD and Adagrad.

**6. Conclusion**

- SGD is efficient but unstable.

- MGD balances speed and stability.

- Adagrad adapts learning rates but slows down over time.

- Adam GD is widely used due to its efficiency and stability.

Among these, Adam Gradient Descent is the most effective for most applications due to its adaptability and fast convergence.

**Implementation:**

**#include** <iostream>

**#include** <vector>

**#include** <cmath>

**#include** <fstream>

**using** **namespace** std;

// Mean Squared Error Function

**float** mse(**float** w, **float** b, vector<**float**> x\_values, vector<**float**> y\_values, **int** n) {

**float** sum **=** 0;

**for** (**int** i **=** 0; i **<** n; i**++**) {

**float** y\_pred **=** x\_values**[**i**]** **\*** w **+** b;

        sum **+=** pow((y\_values**[**i**]** **-** y\_pred), 2);

    }

**return** sum **/** (2 **\*** n);

}

// Stochastic Gradient Descent

**void** sgd(**float** w, **float** b, **float** neta, vector<**float**> x\_values, vector<**float**> y\_values, **int** n) {

**for** (**int** i **=** 0; i **<** n; i**++**) {

**float** y\_pred **=** x\_values**[**i**]** **\*** w **+** b;

**float** dw **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred) **\*** x\_values**[**i**]**;

**float** db **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred);

        w **-=** neta **\*** dw;

        b **-=** neta **\*** db;

    }

    cout **<<** "SGD: w = " **<<** w **<<** ", b = " **<<** b **<<** ", MSE = " **<<** mse(w, b, x\_values, y\_values, n) **<<** endl;

}

// Mini Batch Gradient Descent with Momentum

**void** mgd(**float** w, **float** b, **float** neta, **float** beta, vector<**float**> x\_values, vector<**float**> y\_values, **int** n) {

**float** v\_w **=** 0, v\_b **=** 0;

**for** (**int** i **=** 0; i **<** n; i**++**) {

**float** y\_pred **=** x\_values**[**i**]** **\*** w **+** b;

**float** dw **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred) **\*** x\_values**[**i**]**;

**float** db **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred);

        v\_w **=** beta **\*** v\_w **+** neta **\*** dw;

        v\_b **=** beta **\*** v\_b **+** neta **\*** db;

        w **-=** v\_w;

        b **-=** v\_b;

    }

    cout **<<** "MGD: w = " **<<** w **<<** ", b = " **<<** b **<<** ", MSE = " **<<** mse(w, b, x\_values, y\_values, n) **<<** endl;

}

// Adaptive Gradient Descent

**void** agd(**float** w, **float** b, **float** neta, vector<**float**> x\_values, vector<**float**> y\_values, **int** n) {

**float** alpha\_w **=** 0, alpha\_b **=** 0, epsilon **=** 1**e-**8;

**for** (**int** i **=** 0; i **<** n; i**++**) {

**float** y\_pred **=** x\_values**[**i**]** **\*** w **+** b;

**float** dw **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred) **\*** x\_values**[**i**]**;

**float** db **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred);

        alpha\_w **+=** pow(dw, 2);

        alpha\_b **+=** pow(db, 2);

        w **-=** (neta **/** sqrt(alpha\_w **+** epsilon)) **\*** dw;

        b **-=** (neta **/** sqrt(alpha\_b **+** epsilon)) **\*** db;

    }

    cout **<<** "AGD: w = " **<<** w **<<** ", b = " **<<** b **<<** ", MSE = " **<<** mse(w, b, x\_values, y\_values, n) **<<** endl;

}

// Adam Gradient Descent

**void** adam(**float** w, **float** b, **float** neta, **float** beta1, **float** beta2, vector<**float**> x\_values, vector<**float**> y\_values, **int** n) {

**float** m\_w **=** 0, m\_b **=** 0, v\_w **=** 0, v\_b **=** 0, epsilon **=** 1**e-**8;

**for** (**int** i **=** 0; i **<** n; i**++**) {

**float** y\_pred **=** x\_values**[**i**]** **\*** w **+** b;

**float** dw **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred) **\*** x\_values**[**i**]**;

**float** db **=** **-**1 **\*** (y\_values**[**i**]** **-** y\_pred);

        m\_w **=** beta1 **\*** m\_w **+** (1 **-** beta1) **\*** dw;

        m\_b **=** beta1 **\*** m\_b **+** (1 **-** beta1) **\*** db;

        v\_w **=** beta2 **\*** v\_w **+** (1 **-** beta2) **\*** pow(dw, 2);

        v\_b **=** beta2 **\*** v\_b **+** (1 **-** beta2) **\*** pow(db, 2);

        w **-=** (neta **\*** m\_w) **/** (sqrt(v\_w) **+** epsilon);

        b **-=** (neta **\*** m\_b) **/** (sqrt(v\_b) **+** epsilon);

    }

    cout **<<** "Adam: w = " **<<** w **<<** ", b = " **<<** b **<<** ", MSE = " **<<** mse(w, b, x\_values, y\_values, n) **<<** endl;

}

**int** main() {

**float** w **=** 0.5, b **=** 0.5, neta **=** 0.01, beta1 **=** 0.9, beta2 **=** 0.999, beta **=** 0.9;

    vector**<float>** x\_values **=** {1, 2, 3, 4, 5};

    vector**<float>** y\_values **=** {2.2, 2.8, 3.6, 4.5, 5.1};

**int** n **=** x\_values.size();

    cout **<<** "Initial MSE: " **<<** mse(w, b, x\_values, y\_values, n) **<<** endl;

    sgd(w, b, neta, x\_values, y\_values, n);

    mgd(w, b, neta, beta, x\_values, y\_values, n);

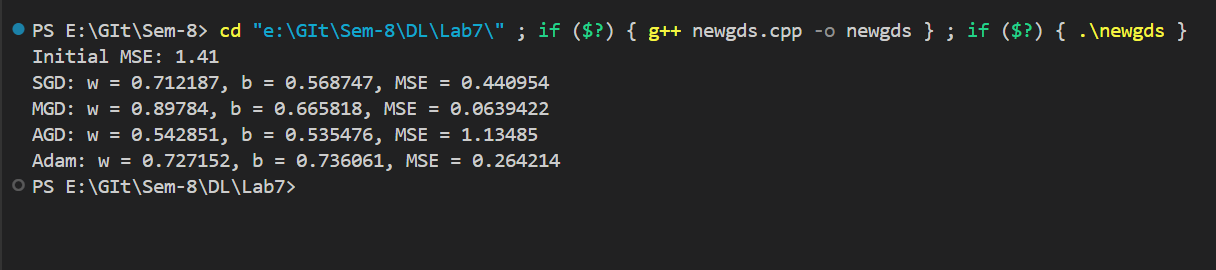
    agd(w, b, neta, x\_values, y\_values, n);

    adam(w, b, neta, beta1, beta2, x\_values, y\_values, n);

**return** 0;

}

**Output:**

****