Vidyalankar Institute of Technology Accredited A+ by NAAC	**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 (x_i, y_i).

2. Compute the gradient:

- gradient_w = - (y_i - (w * x_i + b)) * x_i

- gradient_b = - (y_i - (w * x_i + 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):

   - $m_t = \beta 1 * m_{(t-1)} + (1 - \beta 1) * gradient\_w$

2. Compute second-moment estimate:

   - $v_t = \beta 2 * v_{(t-1)} + (1 - \beta 2) * gradient\_w^2$

3. Bias correction:

   - $\hat{m}_t = m_t / (1 - \beta 1^t)$

   - $\hat{v}_t = v_t / (1 - \beta 2^t)$

4. Update parameters:

   - $w = w - (\eta / (sqrt(\hat{v}_t) + \varepsilon)) * \hat{m}_t$

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.

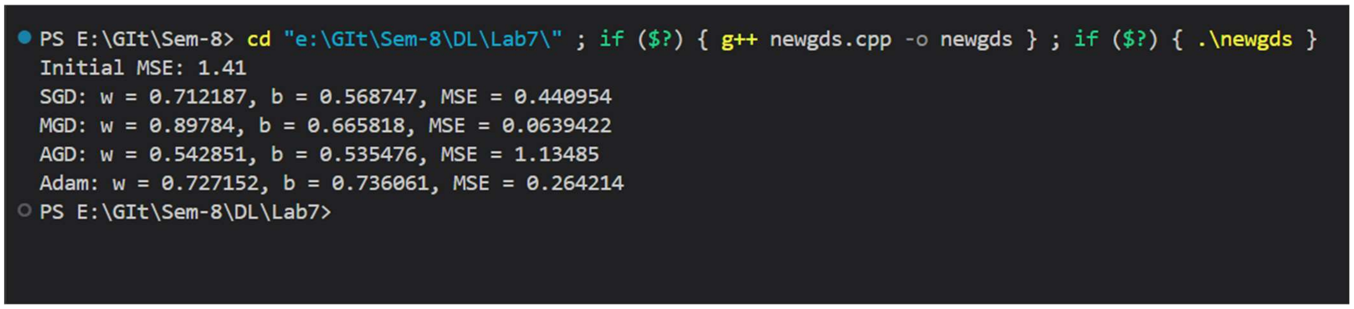
- Adam GD is widely used due to its efficiency and stability.

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 = 1e-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 = 1e-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:



```
● PS E:\GIt\Sem-8> cd "e:\GIt\Sem-8\DL\Lab7\" ; if ($?) { g++ newgds.cpp -o newgds } ; if ($?) { .\newgds }
  Initial MSE: 1.41
  SGD: w = 0.712187, b = 0.568747, MSE = 0.440954
  MGD: w = 0.89784, b = 0.665818, MSE = 0.0639422
  AGD: w = 0.542851, b = 0.535476, MSE = 1.13485
  Adam: w = 0.727152, b = 0.736061, MSE = 0.264214
○ PS E:\GIt\Sem-8\DL\Lab7>
```