Computational Complexity of Gradient Descent Algorithm

Mr. Nishchal J
Mr Neel Bhandari
CSE, R.V. College of Engineering
Bengaluru, India

Abstract: Information is mounting exponentially, and the world is moving to hunt knowledge with the help of Big Data. The labelled data is used for automated learning and data analysis which is termed as Machine Learning. Linear Regression is a statistical method for predictive analysis. Gradient Descent is the process which uses cost function on gradients for minimizing the complexity in computing mean square error. This work presents an insight into the different types of Gradient descent algorithms namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent, which are implemented on a Linear regression dataset, and hence determine the computational complexity and other factors like learning rate, batch size and number of iterations which affect the efficiency of the algorithm.

Keywords: *Linear Regression, Gradient Descent, Machine Learning, Artificial Neural Networks, Learning rate, Data Analytics, Time Complexity, epochs.*

I. INTRODUCTION

Machine Learning is a learning technique consisting of certain algorithms that identify patterns to expect the potential data, or to execute crucial decision making under uncertainty situations. Gaining Knowledge, understanding the patterns, image detection and face recognition are a few phrases which machine learning defines precisely. This refers to transformation for completing a job related to artificial intelligence (AI). A model

designed should preserve the features of the algorithm adopting to the modifications in its environment and figuring necessary actions by possibly predicting the effects.

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. Gradient descent was invented by French mathematician Louis Augustin Cauchy in 1847. It is based on the idea that instead of immediately guessing the best solution for the problem, gradient descent iteratively takes steps in the direction in which the solution becomes better and better. The process of changing the parameter values involves differential Calculus, specifically calculating the “derivative” of the cost function. The derivative gives the slope of the function at a specific point. In other words, it specifies how to scale a small change in the input to obtain the corresponding change in the output. The derivative is therefore useful for minimizing the cost function because it tells us how to change the parameters in order to make a small improvement in finding the function’s minimum.

Gradient Descent Algorithm can be implemented in three ways namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent. This paper aims to deduce and compare the computational efficiencies of the three stated algorithms by implementing them on a Linear Regression Dataset and simultaneously vary some of the parameters like batch size, number of iterations

learning rate, etc. and see its effect on the efficiency of the algorithm.

Gradient descent is widely used in its raw form in Linear and Logistic regression analysis of data. Mini-batch is the most widely used approach to solve regression or classification problems. However, a few optimizations on this algorithm gives birth to the foundational algorithms used in Artificial Neural Networks, like ADAM, RMS PROP, ADAGRAD etc. which work on the same principle as that of gradient descent but is more optimized in terms of time complexity.

II. LINEAR REGRESSION

In statistics, linear regression is a linear approach to modelling the relationship between a scalar response or dependent variable and one or more explanatory variables or independent variables. The case with one independent variable is termed as simple linear regression. For more than one independent variable, the process is termed as multiple linear regression. This term is distinct from multivariate linear regression, where multiple dependent variables are predicted, rather than a single scalar variable. A general equation of a simple linear regression would look like this:

$$Y = A + B * X \quad (\text{eq.1})$$

Where Y represents a single dimensional output vector, X represents a single dimensional input vector, A is the explanatory or independent variable, whereas B is the scalar response or the dependent variable. In regression problems like these, the main aim is to determine the set of optimum values of (A,B) which exactly makes the input and the output vector fit into the equation 1.

Fig. 1: Regression Analysis

Fig. 1 shows the regression analysis, which basically involves scatter plotting the data points with X on x-axis and Y on y-axis, and then analysing the type of regression it belongs to, in this case it appears to be linear.

The values of A and B is determined by adopting various algorithms out of which the most used ones are Newton's Normal equation method and Gradient Descent algorithm.

III. GRADIENT DESCENT

Before analysing the Gradient descent algorithm, it is important to know some equations and terms.

$$H(\theta) = X * \theta \quad (\text{eq.2})$$

The (eq.1) can be re-written as (eq.2) is called the Prediction function where θ represents an $n*1$ matrix representing n number of features or variables for each data point (A and B in eqn.1). The matrix X is an $m*n$ vector with m number of data points and n features. The aim of linear regression is to keep approximating the values of θ matrix (A and B in eqn.1) iteratively to reach an optimal value which satisfies fits the given data in eqn.2.

$$J(\theta) = \frac{1}{2*m} \sum_i^m (H(x^i) - y^i)^2 \quad (\text{eq.3})$$

The eq.3 is called the Cost function, In ML, cost functions are used to estimate how badly models are performing.

Simply put, a cost function is a measure of how wrong the model is, in terms of its ability to estimate the relationship between X and Y. This is typically expressed as a square of difference or distance between the predicted value and the actual value. Here the summation of such differences over m samples gives us the net cost for the parameters θ which is approximated.

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla J(\theta)$ with respect to the parameters (eq.4). The learning rate α determines the size of the steps we take to reach a (local) minimum.

In other words, the direction of the slope of the surface created by the objective function downhill is followed until a valley is reached. At this point the model has optimized the weights such that they minimize the cost function. This process is integral to the ML process, because it greatly expedites the learning process. Gradient descent, therefore, enables the learning process to make corrective updates to the learned estimates that move the model towards an optimal combination of parameters.

$$\theta = \theta - \alpha \cdot \nabla J(\theta) \quad (\text{eq.4})$$

The learning rate α determines how fast or slow should the update of the parameters be while performing gradient descent. If the learning rate is too high, the local minima seen in fig.2 may be skipped completely due to a very big update, else if the learning rate is too small, then it would take a large number of updates or iterations to reach the minima point, which can cost a lot of time. Hence, selection of the optimum learning rate is important while performing the gradient descent algorithm on a data.

Fig. 2: Gradient Descent Approach

Repeat Until Convergence

$$\left\{ \begin{array}{l} \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \end{array} \right\} \quad (\text{eq.5})$$

On finding the first order partial derivative of the cost function $J(\theta)$ and substituting back to the equation, we get the finalized Gradient descent equation shown in eq.5. A general Gradient descent equation for multivariate problems is the same as eq.5 with more than two parameters to be updated in θ matrix. The above step is to be repeated until the cost function attains a minimum value. The cost function is dependent on the θ directly, hence after every update of the parameters, cost value must be calculated to check if it has reached a minimal point.

Fig. 3: Contour Plot: After every Iteration

The number of iterations it takes to converge to the minima is of great interest as it determines the efficiency of the algorithm. The number of iterations depend on the type of data fed into the model, however, for a given dataset, parameters like Batch Size and learning rate influence the convergence of the cost to its minimum value.

IV. TYPES

Various variants of gradient descent are defined based on how we use the data to calculate derivative of cost function in gradient descent. Depending upon the quantity of data used, the time complexity and accuracy of the algorithms differs from one another.

A. Batch Gradient Descent

It is the primary type of gradient descent in which the complete dataset will be used to compute the gradient of cost function. As there is a need to calculate the gradient on the entire dataset just to perform one update, batch gradient descent can be very slow and is unmalleable for datasets that don't fit in memory. After initializing the parameter with arbitrary values, we calculate gradient of cost function using the eq.5 given above iteratively where 'm' is the number of training examples. The update of the variables and the decrease in the cost function is gradual and smooth as shown in fig 3, because all the values in training set are used at a time to update the parameters in one single iteration.

B. Stochastic Gradient Descent

Batch Gradient Descent turns out to be a slower algorithm as all the training examples are taken for each update. So, for faster computation, it is preferred to use stochastic gradient descent. As a first step of the algorithm, the whole training set is

to be randomized. Then, for updation of every parameter only one training example is used in every iteration to compute the gradient of cost function. As it uses one training example in every iteration this algorithm is faster for larger data set.

The reason for shuffling the dataset is to primarily induce randomness in the dataset. In SGD, as updating the parameter values takes place for each training sample individually, there is a lot of noise created in the cost function updates as seen in Fig.4, hence, to decrease this, shuffling of dataset in mandatory.

Fig. 4 Contour Plot: SGD Cost Function

In SGD, one might not achieve accuracy, but the computation of results is faster. After initializing the parameter with arbitrary values the gradient of cost function is calculated using eq.6 as follows:

$$\begin{aligned}
 &\text{Repeat } \{ \\
 &\quad \text{for } i := 1, \dots, m \{ \\
 &\quad \quad \theta_j := \theta_j - \alpha(h_{\theta}(x^{(i)}) - y^{(i)})x_j^{(i)} \\
 &\quad \quad \quad (\text{for every } j = 0, \dots, n) \\
 &\quad \} \\
 &\}
 \end{aligned}
 \tag{eq.6}$$

where, 'm' is the number of training examples. SGD never actually converges like batch gradient descent does but ends up wandering around some region close to the global minimum as seen in Fig. 4.

SGD algorithm unlike Batched Gradient Descent, doesn't depend on the total number of training examples, however learning rate influences its working tremendously. The noise in the update can also be regularised by selecting the appropriate α . However, it is observed that SGD wanders around the global minima in lesser number of iterations due to multiple updates of parameters in a single iteration.

C. Mini-Batch Gradient Descent

Mini batch algorithm is the most favourable and widely used algorithm that makes precise and faster results using a batch of 'k' training examples. Hence the whole training set is shuffled and divided into batches of 'k' examples, which is then used for updating the gradient of the cost. Common mini-batch sizes range between 16 and 256 but can vary for different applications.

After initializing the parameter with discretional values, gradient of cost function is calculated using the following:

$$\begin{aligned} &\text{Repeat } \{ \\ &\quad \text{for } i = 1, 11, 21, 31, \dots, 991 \{ \\ &\quad \quad \theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_{\theta}(x^{(k)}) - y^{(k)}) x_j^{(k)} \\ &\quad \quad \text{(for every } j = 0, \dots, n) \} \} \quad (\text{eq. 7}) \end{aligned}$$

where 'b' is number of batches taken and 'm' is the total number training examples.

From eq.7 it is inferred that the update of parameters happens in multiple batches of fixed training examples, hence due to presence of multiple batches, noise in the cost function gradient is expected. However, when compared with the Stochastic Gradient approach, the reduction in noise is significant as seen in fig. 5 due to the batched training approach.

Mini-batch gradient descent seeks to strike a balance between the robustness of

stochastic gradient descent and the efficiency of Batch Gradient Descent. It is the foremost common implementation of gradient descent utilized in the field of deep learning. It reduces the variance of the parameter updates, which can lead to more stable convergence and it makes use of highly optimized matrix, that makes computing of gradient very efficient.

Fig 5. Contour Plot: MBGD Cost Function

V. EXPERIMENTAL ANALYSIS OF GRADIENT DESCENT

A. Aim

To perform the three types of gradient descent algorithms namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent on a simple Regression dataset and hence deduce and compare the computational complexity of all the three algorithms.

B. Technological Stack Required

The experiment was conducted on Google Colab Platform. Colaboratory is a free Jupyter notebook environment provided by Google where free GPUs and TPUs can be used to train ML models.

The dataset contained 700 training examples of a simple linear regression data which was manually made to conduct this experiment; hence no pre-processing of data was required, it was mostly noise free.

The algorithm was written and trained in Python language. Additional modules such as NumPy, Random, Sklearn.metrics and Matplotlib for vector multiplication, RMS error finding and graph plotting, were imported.

C. Data Visualisation and Analysis

The chosen dataset contained 700 datapoints with only one input parameter X and one output parameter Y. A sample of the dataset is shown here:

X	Y
24	21.54945
50	47.46446
15	17.218
38	36.586
87	87.28898
36	32.46387
12	10.780
81	80.7634
25	24.61215
5	6.963319
16	11.23757
24	24.60324

Table 1. Sample Regression Dataset

On plotting these values on scatter plot graph using the Matplotlib library, the data could be visualised as follows:

Fig. 6: Scatterplot: Regression Dataset

From fig. 6, it was clearly deducible that the data was in accordance with linear regression. Small deviations are usually common in large datasets, however it the above graph provided clear evidence to use it for training a Linear regression model.

D. Control Parameters

Parameters kept for observation were, the number of training samples taken 'm', learning rate ' α ', batch size 'k', cost estimated 'J' and finally the number of basic steps required to complete training 't'.

These parameters were varied across all the three algorithms to realise their effect on performance and efficiency.

E. Model Training, Evaluation and Observation

The dataset was trained on all the three models namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent. Test Parameters were varied, and the observations were recorded for further analysis.

The python function used to compute cost (J) is as follows:

```
def cost(a,b,m,x1,y1):
    h1=a+b*x1
    return(np.sum(np.square(h1-y1))/(2*m))
```

Fig. 7: Cost Function

This function is called after every iteration to check if the cost has reached the minimum or has saturated to a value when the parameters are updated.

a. Batch Gradient Descent

The python program for implementing Batch Gradient Descent is as follows:

```
x_train=np.array(x[:128])
y_train=np.array(y[:128])
m=128
a=0.1
b=1
h=a+b*x_train
n=1
r=0.00001
while(n):
    n+=1
    if(cost(a,b,m,x_train,y_train)<=0):
        print(acc(a,b,x_train,y_train),n-1,a,b)
        break
    else:
        a=a- r*(((a+b*x_train)-y_train).sum())/m
        b=b- r*(((a+b*x_train)-y_train)*x_train).sum())/m
        print(cost(a,b,m,x_train,y_train))
```

Fig. 8: Batch Gradient Descent

X_train represents the training set and Y_train represents the actual output of the line. Here, the basic operation is the update of 'a' and 'b' which are the parameters of the regression line. Learning rate α is set to 0.00001 which determines the leap gradient takes every iteration.

$Y = a + b \cdot X$ is the prediction equation taken in which a and b are the variables which are in need to be found out through BGD

The following data was tabulated after running the batch gradient descent on varying number of 'm' and learning rate α :

M	Iterations	Learning rate
16	16971	0.001
128	13746	0.0007
256	18083	0.0006
512	17226	0.0006
700	16478	0.0006

Table 2: Test parameters vs Iterations taken

Table 2 shows the variation of number of iterations or epochs taken to reach the minimum cost for the corresponding learning rates and training samples taken.

a	b	cost
-1.1078	1.0006	2.5526
0.3157	0.9923	3.9486
-0.2547	1.001	3.8982
-0.197	1.005	3.9096
-0.0893	1.0003	3.9282

Table 3: The dependent variables and Cost

The second variable 'b' converges eventually to a constant value of 1.00x over large number of data points taken, whereas variable 'a' converges when 'm' reaches 700. Looking at the iterations in Table 2., the number of iterations taken to reach the minimal cost is very high even for a small training set of 16 samples.

Loss	No. basic steps performed
2.236	33942
2.8545	27492
2.7929	36166
2.7999	34452
2.8259	32956

Table 4: The RMS loss and Number of Basic Steps Performed
(the above table is incomplete, please refer Section F for further information)

The loss of the model is evaluated by calculating the root mean square loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters 'a' and 'b'. This step is performed twice every iteration (for 'a' and 'b'). Therefore, the total number of basic steps performed, or the complexity is:

$$O(2*I)$$

Where I is the number of iterations ran to reach the minimal cost.

b. Stochastic Gradient Descent

The python program for implementing Stochastic Gradient Descent is as follows:

```
x_train=x[:700]
y_train=y[:700]
m=700
a=0.1
b=10
h=a+b*x_train
n=1
r=0.001
while(n):
    n+=1
    if(cost(a,b,m,x_train,y_train)<=4568.281576429381):
        print(acc(a,b,x_train,y_train),n-1,a,b)
        if((preva-a) < 1 and (prevb-b) < 1):
            print(preva,a)
            break
    else:
        preva=a
        prevb=b
        for s in range(m):
            a=a- (r*((a+b*x_train[s])-y_train[s]))/1
            b=b- ((r*((a+b*x_train[s])-y_train[s])*x_train[s]))/1
        print(cost(a,b,m,x_train,y_train),a,b)
```

Fig. 9: Stochastic Gradient Descent

X_train represents the training set and Y_train represents the actual output of the line. Here, the basic operation is the update of 'a' and 'b' which are the parameters of the regression line. Learning rate α is set to 0.001 which determines the leap, gradient takes every iteration.

$Y = a + b*X$ is the prediction equation taken in which a and b are the variables which are in need to be found out through SGD

The following data was tabulated after running the Stochastic gradient descent on varying number of 'm' and learning rate α :

M	Iterations	Learning rate
16	10511	0.00005
128	12956	0.00001
256	5202	0.00001
512	4501	0.00001
700	4701	0.000003

Table 5: Test parameters vs Iterations taken

Table 5 shows the variation of number of iterations or epochs taken to reach the minimum cost for the corresponding learning rates and training samples taken.

a	b	cost
-1.1014	1.0028	2.5576
0.3252	0.9912	3.9496
-0.3196	1.0071	3.9385
-0.2508	1.0059	3.9448
-0.1041	1.0007	3.9283

Table 6: The dependent variables and Cost

The second variable 'b' converges eventually to a constant value of 1.00x over large number of data points taken, whereas variable 'a' converges when 'm' reaches 700. Looking at the iterations in Table 5., the number of iterations taken to reach the minimal cost is high for smaller datasets however becomes very low for larger training datasets taken.

Loss	No. basic steps performed
2.25	336352
2.8641	3316736
2.8222	2663424
2.8197	4609024
2.8259	6581400

Table 7: The RMS loss and Number of Basic Steps Performed

The loss of the model is evaluated by calculating the root mean square loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters 'a' and 'b'. This step is performed for '2*m' number of times every iteration (for 'a' and 'b'). Therefore, the total number of basic steps performed, or the complexity is given by:

$$O(2*m*I)$$

Where I is the number of iterations ran to reach the minimal cost. The number of steps taken is tabulated in Table 7.

c. Mini-Batch Gradient Descent

The python program for implementing Mini-Batch Gradient Descent is as follows:

```
x_train=x[:700]
y_train=y[:700]
a=1
b=1
m=700
k=32
h=a+b*x_train
n=1
r=0.0001
while(n):
    n+=1
    if(cost(a,b,m,x_train,y_train)<=3.9283):
        print(acc(a,b,x_train,y_train),n-1,a,b)
        if((preva-a) < 1 and (prevb-b) < 1):
            print(preva,a)
            break
    else:
        preva=a
        prevb=b
        for s in range(int(m/k)):
            a=a- (r*((a+b*x_train[s*k:s*k+k])-y_train[s*k:s*k+k]).sum())/k
            b=b- ((r*((a+b*x_train[s*k:s*k+k])-y_train[s*k:s*k+k])*x_train[s*k:s*k+k]).sum())
        print(cost(a,b,m,x_train,y_train))
```

Fig. 10: Mini-Batch Gradient Descent

X_train represents the training set and Y_train represents the actual output of the line. Here, the basic operation is the update of 'a' and 'b' which are the parameters of the regression line. Learning rate α is set to 0.0001 which determines the leap, gradient takes every iteration.

$Y = a + b*X$ is the prediction equation taken in which a and b are the variables which are in need to be found out through MBGD

The following data was tabulated after running the Mini-Batch gradient descent on varying number of 'm' and learning rate α and batch size 'k':

M	Iterations	Learning rate	Batch Size
16	19871	0.001	4
128	9557	0.0001	8
256	7941	0.0001	8
512	5988	0.0001	16
700	6641	0.0001	32

Table 8: Test parameters vs Iterations taken

Table 8 shows the variation of number of iterations or epochs taken to reach the minimum cost for the corresponding learning rates and training samples taken.

a	b	cost
-1.5487	0.9902	2.9169
0.3514	0.9884	3.9623
-0.306	1.0078	3.9549
-0.2159	1.0046	3.9338
-0.0894	1.0003	3.9282

Table 9: The dependent variables and Cost (the above table is incomplete, please refer Section F for further information)

The second variable 'b' converges eventually to a constant value of 1.00x over large number of data points taken, whereas variable 'a' converges when 'm' reaches 700. The values of 'a', 'b' and cost are very similar to those which were obtained in mini-batch Gradient Descent, however the iterations (in Table 8.) taken to reach the minimal cost is high for smaller datasets however gradually lowers for larger training datasets taken.

Loss	No. basic steps performed
2.3318	158968
2.8159	305824
2.7929	508224
2.819	383232
2.8259	278922

Table 10: The RMS loss and Number of Basic Steps Performed

The loss of the model is evaluated by calculating the root mean square loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters 'a' and 'b'. This step is performed for '2*m/k' number of times every iteration (for 'a' and 'b').

Here k represents the batch size, and (m/k) represents the number of batches present in the dataset.

As update happens for each batch in one iteration therefore, the total number of basic steps performed, or the complexity is given:

$$O(2*(m/k)*I)$$

Where 'I' is the number of iterations ran to reach the minimal cost. The number of steps taken is tabulated in Table 10.

F. Graphical Analysis of the Model

The linear regression model trained on three different algorithms, gave a similar output for various set of inputs given.

Graphs were plotted using modules like Matplotlib which superimposed the scatter plot representing the actual or the original output values, whereas the red line on the graph represents the predicted output line from the linear regression model.

The values of 'a' and 'b' and the predicted line for different sizes of inputs and their graphical representation is as follows:

$$a = -1.1014, b = 1.0006,$$

$$Y = -1.1014 + 1.0006*X$$

Fig. 11: Predicted Vs Actual for m=16,

$$a = 0.3252, b = 0.9912,$$

$$Y = 0.3252 + 0.9912*X$$

Fig. 12: Predicted Vs Actual for m=128

$$a = -0.306, b = 1.0007,$$

$$Y = -0.306 + 1.0007*X$$

Fig. 13: Predicted Vs Actual for m=512

$$a = -0.0894, b = 1.0003,$$

$$Y = -0.0894 + 1.0003 * X$$

Fig. 14: Predicted Vs Actual for m=700

G. Analysis and Inference

Let's compare the observations of all the three algorithms made in the above experimental study.

From what is outwardly seen, taking test parameters into consideration, the number of iterations taken for SGD to reach the minimal value is the least when compared to the other two algorithms. Thus, SGD has an advantage of lesser iterations over the other two algorithms.

$$SGD < MBGD < BGD$$

Next is the comparison of the RMS loss computed for all the algorithms, it is observed that, the loss is least for Batched Gradient descent whereas it is high in SGD. Thus, BGD has an advantage of lower loss in the computation of the model

$$BGD < MBGD < SGD$$

The last and the most important comparison of the algorithms is with respect number of basic steps performed. Outwardly it is seen that BGD takes very less steps to complete computation whereas SGD takes the most as seen in observation tables. But those tables are incomplete. Though, theoretically the above said is true, in practical implementations, it seen that MBGD and SGD takes lesser computation

time to train the model whereas BGD takes the most.

This is because, in Batch gradient Descent, though in each iteration, update of 'a' and 'b' happened only once, the time consumption for this update was a lot, i.e. each update required a traversal through all the datapoints in training sample and then a final summation. Considering the traversal and summation also to be significant we formulate a new time complexity for BGD:

$$O(2*I)*m$$

This changes the Table 4 to the following:

No. basic steps performed
543072
3518976
9258496
17639424
23069200

Table 11: Corrected BGD Basic Number of Steps.

Moving further, the same applies to Mini-Batch Gradient Descent, where in, if we consider the time complexity of traversing the batch elements of size 'k' and the summation, then, the new time complexity of MBGD:

$$O(2*(m/k)*I)*k$$

Table 10 is modified to the following:

No. basic steps performed
635872
2446592
4065792
6131712
8925504

Table 12: Corrected MBGD Basic Number of Steps.

However, in the case of SGD, updating of the parameters takes place by individual datapoints, hence no extra summation or traversal is required. Hence the table in Table 7 correctly identifies the basic number of steps as well as the complexity of the algorithm.

It is to be noted that, there is a tough competition between MBGD and SGD with regards to the time complexity. Though theoretically, SGD should have the lowest time complexity, in practical situations, on many datasets, MBGD outperforms SGD offering lower time complexity.

From the above analysis, we can conclude that Stochastic Gradient descent and Mini-Batch Gradient descent take lesser time to train the model and reach the minima when compared to Batch gradient descent which clearly takes a lot more than other two from the fig. 18, fig. 19 and fig. 14.

$$O(2*I)*m > O(2*(m/k)*I)*k \approx O(2*m*I)$$

$$BGD < MBGD \approx SGD$$

VI. EXPERIMENTAL ANALYSIS OF GRADIENT DESCENT USING LOGISTIC REGRESSION

A. Aim:

To perform the three types of gradient descent algorithms namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent on a simple Regression dataset and hence deduce and compare the computational complexity of all the three algorithms.

B. Technological Stack Requirement

The experiment was conducted on Google Colab Platform. Colaboratory is a free Jupyter notebook environment provided by Google where free GPUs and TPUs can be used to train ML models.

The dataset contained 700 training examples of a simple linear regression data which was manually made to conduct this experiment; hence no pre-processing of data was required, it was mostly noise free.

The algorithm was written and trained in Python language. Additional modules such as NumPy, Random, Sklearn.metrics and Matplotlib for vector multiplication, RMS error finding and graph plotting, were imported.

C. Data Visualisation:

The data being used in this part of our analysis for Logistic regression is one where 2 sets of features are tallied and a column of targets consisting of 1 or 0 is present, which provided the tally of whether the parameter has passed a certain threshold. The following is a table that presents a certain sample of the dataset.

Feature 1	Feature 2	Targets
9.43624594E-01	6.79782992E-01	1
-8.68018214E-01	-2.53921797E-01	0
-1.76747111E+00	9.9992369E-01	0
5.72275908E-01	3.94578577E+00	1
-1.43602637E+00	-1.30764497E+00	0
-1.06475862E+00	1.11871846E+00	0
9.72359074E-01	-1.52979682E-01	1
-1.51469537E+00	-2.73244894E+00	0
2.13551898E+00	1.31799260E+00	1
-2.38046447E-01	7.76711032E-01	0
-6.20566756E-01	-1.49496599E+00	1
-3.35735665E+00	2.02011582E+00	0

Table 13: Logistic Regression Sample Dataset

On plotting these points on a scatter plot, we receive the following plot.

The red points represent those who have 1 in the corresponding target section, while the blue points represent those who have 0 in the target section.

Fig 15: Scatter plot of Data Sample

D. Control Parameters

Parameters kept for observation were, the number of training samples taken ‘m’, learning rate, batch size ‘bs’, cost estimated ‘loss’ and finally the number of basic steps required to complete training ‘count’.

These parameters were varied across all the three algorithms to realize their effect on performance and efficiency.

E. Model Training, Evaluation and Observation

The dataset was trained on all the three models namely, Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent across 3 notebooks, with subtle variations to suite shapes of data frames present in the algorithms. Test Parameters were varied, and the observations were recorded for further analysis.

The python function used to compute gradient descent(J) is as follows:

```
def gradient_descent(X, h, y):
    return np.dot(X.T, (h - y))
def update_weight_loss(weight, learning_rate, gradient):
    return weight - learning_rate * gradient
```

Fig. 16: Gradient Descent and Update Functions

This function is called after every iteration to check if the cost has reached the minimum or has saturated to a value when the parameters are updated.

The following functions provide the mathematical models used to calculate the parameters we have considered i.e gradient and loss.

```
def sigmoid(x):
    sig = np.longdouble(1.0)/(np.longdouble(1.0) + np.longdouble(np.exp(-x)))
    return sig
def prediction(x,weights):
    pred= sigmoid(np.dot(x,weights))
    return pred
def loss(X,weights,y):
    ls=-y*np.log(prediction(X,weights))+ (1-y)*np.log(1-prediction(X,weights))
    return ls
```

Fig 17: Gradient Descent Functions

Here loss function essentially represents the cost that the model undergoes over the course of the entire training. This is the factor we are trying to minimise. Gradient is then used to calculate the direction and to what extent the function has to travel in order to minis the cost.

a. Batch Gradient Descent

The python program for implementing Batch Gradient Descent is as follows:

```
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,train_size=0.8)
X=(x_train-(x_train).mean())/10
Y=y_train
def train(features,targets,epochs,learning_rate):
    global weights;
    weights=np.zeros((X.shape[1], 1))
    print(features.shape)
    print(targets.shape)
    loss_sum=0;
    grad=0;
    for e in range(epochs):
        loss_sum=loss(features,weights,targets)/features.shape[0]
        grad=gradient_descent(features,prediction(features,weights),targets)
        weights=update_weight_loss(weights,learning_rate,grad)
```

Fig 18: Training Function

‘features’ represents the training set that we want to implement and targets represents the actual output of the line. Here, the basic operation is the update of ‘loss’ and ‘grad’ which provide the cost of the model and the gradient output respectively.

‘Learning rate’ is set variably which determines the leap gradient takes every iteration.

$Y = (X * \text{Weights})$ is the prediction equation taken in which weights represents the list of variables we need to determine through BGD

The following data was tabulated after running the batch gradient descent on varying number of ‘epochs’ and learning rate 0.0001:

Epochs	Iterations
16	6400
32	12800
64	25600
128	51200
256	102400
512	204800
1024	409600

Table. 14: Test parameter vs Iterations

The reason we have taken a constant learning rate is that the corresponding cost that comes with these iterations can provide an accurate understanding of how many iterations are required for minimal loss.

Weight 1	Weight 2	Cost
0.5471519	-0.00494	0.46137
0.90968	-0.007	0.35585
1.3834	-0.0074	0.26532
1.9319	-0.0009690	0.20435
2.5078	0.0177	0.17048
3.05653	0.051	0.15488
3.50774	0.09202	0.14944

Table. 15: The dependent variables and Cost

We can see that after a certain number of iterations, the cost has become minimised for this set. We see that weights start to converge to certain values that are

incredibly close to each other and produce the same cost.

Loss	Basic Steps Performed
0.2885	40000
0.2527	56300
0.2255	69600
0.20792	84800
0.2007	92800

Table. 16: The Log loss and Number of Basic Steps Performed

The loss of the model is evaluated by calculating the log loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters weights. This step is performed twice every iteration (for both weights). Therefore, the total number of basic steps performed, or the complexity is:

$$O(2*I)$$

Where I is the number of iterations ran to reach the minimal cost.

b. Stochastic Gradient Descent:

The python program for implementing Stochastic Gradient Descent is as follows:

```
def sigmoid(x):
    sig = np.longdouble(1.0)/(np.longdouble(1.0) + np.longdouble(np.exp(-x)))
    return sig
def prediction(x,weights):
    pred= sigmoid(np.dot(x,weights))
    return pred
def loss(X,weights,y):
    ls=-y*np.log(prediction(X,weights))+ (1-y)*np.log(1-prediction(X,weights))
    return ls
def gradient_descent(X, b, y):
    global count
    count+=1
    X=X.reshape(2,1)
    grad=np.dot(X,(b - y))
    return grad
def update_weight_loss(weight, learning_rate, gradient):
    gradient=gradient.reshape(2,1)
    return weight - learning_rate * gradient

def train(features,targets,epochs,learning_rate):
    global count
    count=0
    n_records, n_features = features.shape
    global weights
    weights = np.random.normal(scale=1 / n_features**.5, size=n_features)
    print(features.shape)
    print(targets.shape)
    print(weights.shape)
    weights=weights.reshape(2,1)
    loss_sum=0
    grad=0
    for e in range(epochs):
        for x,y in zip(features,targets):
            loss_sum=loss(x,weights,y)
            grad=gradient_descent(x,prediction(x,weights),y)
            weights=update_weight_loss(weights,learning_rate,grad)
        loss_sum=np.mean(loss(features,weights,targets))
        print(loss_t)
    print(weights)
    print(count)
```

Fig 19: Training Function

‘features’ represents the training set that we want to implement and targets represents the actual output of the line. Here, the basic operation is the update of ‘loss’ and ‘grad’ which provide the cost of the model and the gradient output respectively. ‘learning_rate’ is set variably which determines the leap gradient takes every iteration.

$Y = (X * \text{Weights})$ is the prediction equation taken in which weights represents the list of variables we need to determine through BGD

The following data was tabulated after running the stochastic gradient descent on varying number of ‘epochs’ and learning rate 0.0001:

Epochs	Iterations
16	6400
32	12800
64	25600
128	51200
256	102400
512	204800
1024	409600

Table 17: Test parameters vs Iterations taken

Table 17 shows the variation of number of iterations or epochs taken to reach the minimum cost for the corresponding learning rates and training samples taken.

As can be noted, the number of iterations remain the same as BGD. However, as we will move on to see, the number of epochs

required are lesser to reach the minimum cost

We can see that after a certain number of iterations, the cost has become minimised for this set. We see that weights start to converge to certain values that are

incredibly close to each other and produce the same cost.

Moreover, to understand a comparison with the BGD algorithm, notice how fast the jump of both the weights as well as the cost of the SGD is. It is in line with the ability to have faster analysis for the same epochs as BGD.

The difference in cost between BGD and SGD is attributed to the shuffling of data required in each epoch of SGD in order to access random features in every iteration.

Loss	Basic Steps Performed
-0.088	53600
-0.033	69600
-0.0266	84800
-0.0145	102400
-0.003	204800

Table 18: The RMS loss and Number of Basic Steps Performed

The loss of the model is evaluated by calculating the log loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters weights. This step is performed twice every iteration (for both weights). Therefore, the total number of basic steps performed, or the complexity is:

$$O(2 * m * I)$$

Where I is the number of iterations ran to reach the minimal cost. The number of steps taken is tabulated in Table 18.

c. Mini-Batch Gradient Descent

The python program for implementing Mini-Batch Gradient Descent is as follows:

```
def train2(features,targets,epochs,learning_rate,bs):
    global weights;
    n_records, n_features = features.shape
    weights = np.random.normal(scale=1 / n_features**.5, size=n_features)
    weights=weights.reshape(2,1)
    targets=targets.reshape(400,1)
    loss_sum=0;
    grad=0;
    i=0
    global count
    for e in range(epochs):
        print("xxxxx")
        while (i+bs)<=398 :
            row=features[i:(i+bs)]
            column=targets[i:(i+bs)]
            loss_sum=loss(row,weights,column)/row.shape[0]
            grad=gradient_descent(row,prediction(row,weights),column,bs)
            weights=update_weight_loss(weights,learning_rate,grad)
            count+=4
            print(count)
            i=i+bs;
        i=0
    print(weights)
    print(loss_sum)
    print(count)
```

Fig. 20: Mini-Batch Gradient Descent

‘features’ represents the training set that we want to implement and targets represents the actual output of the line. Here, the basic operation is the update of ‘loss’ and ‘grad’ which provide the cost of the model and the gradient output respectively. ‘learning_rate’ is set variably which determines the leap gradient takes every iteration.

$Y = (X * \text{Weights})$ is the prediction equation taken in which weights represents the list of variables we need to determine through BGD

The following data was tabulated after running the mini batch gradient descent on varying number of ‘epochs’ and learning rates that are variable.

Loss	Basic Steps Performed
0.0245	33660
0.0167	53460
0.0120	73260
0.0105	89100
0.0086	112860

Table 19: Test parameters vs Iterations

Table 19 shows the variation of number of iterations or epochs taken to reach the minimum cost for the corresponding learning rates and training samples taken.

Weight 1	Weight 2	Cost
0.7874	-0.0613	0.4203
2.1553	-1.3981	0.33763
1.4960	-0.3805	0.23136
1.8184	-0.0062	0.18760
2.5635	0.1251	0.1992
3.0358	0.0494	0.1946
3.6032	0.04746	0.1832

Table 20: The dependent variables and Cost (the above table is incomplete, please refer Section F for further information)

The second variable ‘b’ converges eventually to a constant value of 1.00x over large number of data points taken, whereas variable ‘a’ converges when ‘m’ reaches 700. The values of ‘a’, ‘b’ and cost are very similar to those which were obtained in mini-batch Gradient Descent, however the iterations taken to reach the minimal cost is high for smaller datasets however gradually lowers for larger training datasets taken.

The loss of the model is evaluated by calculating the root mean square loss. The main aim of this paper is to determine the complexity of computation of this algorithm. The basic step is the update of the parameters ‘a’ and ‘b’. This step is performed for ‘ $2 * m/k$ ’ number of times every iteration (for ‘a’ and ‘b’).

Here k represents the batch size, and (m/k) represents the number of batches present in the dataset.

As update happens for each batch in one iteration therefore, the total number of basic steps performed, or the complexity is given by:

$$O(2*(m/k)*I)$$

Where I is the number of iterations ran to reach the minimal cost.

F. Graphical Analysis of the Model

The linear regression model trained on three different algorithms, gave a similar output for various set of inputs given.

Graphs were plotted using modules like Matplotlib which superimposed the scatter plot representing the actual or the original output features, whereas the red line on the graph represents the predicted output line from the logistic regression model.

The values of weights and the predicted line for different sizes of inputs and their graphical representation is as follows:

$$\text{weights} = [-0.0086, 3.6875]$$

$$Y = \text{weights} * X$$

Fig. 21: Predicted Vs Actual for Batch Gradient Descent

$$\text{Weights} = [-0.0072, 3.6879]$$

$$Y = \text{weights} * X$$

Fig 22: Predicted Vs Actual for SGD

$$\text{weights} = [-0.022, 3.7811]$$

$$Y = \text{weights} * X$$

Fig. 23: Predicted Vs Actual for Mini Batch Gradient Descent

G. Analysis and Inference

Let's compare the observations of all the three algorithms made in the above experimental study.

From what is outwardly seen, taking test parameters into consideration, the number of iterations taken for SGD to reach the minimal value is the least when compared to the other two algorithms. Notice the cost reduction in SGD compared to the other algorithms, and what you derive is that the

least cost for equal number of epochs comes from SGD.

Thus, SGD has an advantage of lesser iterations over the other two algorithms.

$$\text{SGD} < \text{MBGD} < \text{BGD}$$

Next is the comparison of the log loss computed for all the algorithms, it is observed that, the loss is least for Batched Gradient descent whereas it is high in SGD. Thus, BGD has an advantage of lower loss in the computation of the model

$$\text{BGD} < \text{MBGD} < \text{SGD}$$

The last and the most important comparison of the algorithms is with respect number of basic steps performed. Outwardly it is seen that BGD takes very less steps to complete computation whereas SGD takes the most as seen in observation tables. But those tables are incomplete. Though, theoretically the above said is true, in practical implementations, it seen that MBGD and SGD takes lesser computation time to train the model whereas BGD takes the most.

This is because, in Batch gradient Descent, though in each iteration, update of 'a' and 'b' happened only once, the time consumption for this update was a lot, i.e. each update required a traversal through all the datapoints in training sample and then a final summation. Considering the traversal and summation also to be significant we formulate a new time complexity for BGD:

$$O(2*I)*m$$

However, in the case of SGD, updating of the parameters takes place by individual datapoints, hence no extra summation or traversal is required. Hence the table in fig.14 correctly identifies the basic number of steps as well as the complexity of the algorithm.

It is to be noted that, there is a tough competition between MBGD and SGD with regards to the time complexity. Though theoretically, SGD should have the lowest time complexity, in practical situations, on many datasets, MBGD outperforms SGD offering lower time complexity.

From the above analysis, we can conclude that Stochastic Gradient descent and Mini-Batch Gradient descent take lesser time to train the model and reach the minima when compared to Batch gradient descent which clearly takes a lot more than other two from the fig. 18, fig. 19 and fig. 14.

$$O(2*I)*m > O(2*(m/k)*I)*k \approx O(2*m*I)$$

VII. CONCLUSION

Batched Gradient Descent algorithm performs well on small sized datasets with minimal loss in the model computation, however, in practical situations, due to robustness of datasets, BGD fails to perform well due its high time complexity.

In practical applications, Stochastic Gradient descent is used only for datasets having about 1000-2000 datapoints, but it is not suitable for higher number of data points due to the noise it generates during parameter updation and a high loss is encountered in the computational results.

Hence, Mini-Batch Gradient descent is the most widely used algorithm to perform any kind of Regression analysis, be it Linear or Logistic. Mini-batch Gradient Descent forms the basis for the construction of Artificial Neural Networks. Further optimizations are done which completely helps in removing the noise present in MBGD, and further improves its computational complexity. Famous optimizers like RMS Prop, ADAM, ADAGRAD etc. are all based on Mini-Batch Gradient Descent algorithm.

References:

1. Stochastic Gradient Descent using Linear Regression with PythonJ V N Lakshmi Research Scholar Department of Computer Science and Application SCSVMV University, Kanchipuram, India (PDF) Stochastic Gradient Descent using Linear Regression with Python.
2. CSC 576: Stochastic Gradient “Descent” Algorithm Ji Liu Department of Computer Sciences, University of Rochester
3. An overview of gradient descent optimization algorithms* Sebastian Ruder Insight Centre for Data Analytics, NUI Galway Aylien Ltd., Dublin
ruder.sebastian@gmail.com
4. 3 Types of Gradient Descent Algorithms for Small & Large Data Sets, Author: Rashmi Jain
5. A Study of Gradient-Based Algorithms Rasmus Hallén 02 - 03 - 2017
6. The history of Gradient Descent by mi-academy Oct 4, 2018 Machine Learning
7. Understanding the 3 Primary Types of Gradient Descent ODSC - Open Data Science
8. Wikipedia and various other websites for raw data and definitions.