Optimization Techniques in Machine Learning and Deep Learning

Ashutosh.V.Patil [1], Gayatri.Y.Bhangle [2]

[1]Student, SCTR's Pune Institute of Computer Technology, Information Technology, Pune, Maharashtra, India
ashutoshpatil359@gmail.com
[2]Student, SCTR's Pune Institute of Computer Technology, Information Technology, Pune, Maharashtra, India
gayatribhangle18@gmail.com

Abstract

Optimization is an important part of machine learning and has attracted many research interests in this-field. As the amount of data continues to increase and the complexity of the model increases, optimization technology will face more challenges. Therefore, to solve these problems, this paper provides a reasonable analysis of different optimization methods in machine learning and deep learning. Here, we examine some methods such as gradient descent and their variants such as stochastic gradient, batch gradient, minibatch gradient, as well as advanced gradient methods such as power, RMSprop, Adam.
addition, we investigate evolutionary algorithms and Bayesian optimization as well as special techniques for training deep neural networks. By experimenting with and evaluating the effectiveness of various models, we demonstrate the advantages and limitations of each method and provide insight into the practical use of developing a good model.

1. Introduction

Machine learning[1] has fleetly advanced, getting a crucial research area and impacting colourful fields like translation, speech recognition[5], and image recommendation. At its core, machine learning relies heavily on optimization to minimize cost functions and tune model parameters from data. Maximization or Minimization is one of the fundamental pillars of machine learning [3].
The essence of most machine learning algorithms the most fundamental of machine learning is mathematical optimization, which deals with calculations of the parameters of a system designed to make decisions based on the information that is not available. This is now known to be the best choice for a given learning problem [6]. So, such success of optimization in ML has motivated extra-ordinary number of analysts and information researchers in various ranges to handle more profound challenges in machine learning issues and to plan the new algorithm or strategies which are broadly pertinent in these ranges.
As importance in optimization in machine learning and deep learning can be understood through three key aspects: (i) part in model performance (ii) effectiveness in training (iii) impact on modern operations. Similar in case it has different parameters like loss function, assigned weights, hyper parameters and other. So sometimes it can be also more challenging to the function which is defined for giving the optimized path or result for a corresponding function.[2]

Literature Survey

In [1], a study was conducted on various optimization methods used in training ML models. The study emphasizes the crucial role of optimization in fine-tuning hyperparameters to make less the error attribute, therefore enhancing the effectiveness and generalization. This paper discusses several enhancement strategies, including GD variants, η, DL problems and second order. Here principles, applications & advantages are outlined, with a focus on this applicability in overcoming challenges.
In [2], The paper titled "Optimization Algorithms for Machine Learning Models" explores various enhancement techniques used to enhance the efficiency of ML algorithms, specifically in predicting employee attrition. The study

covers several key optimization methods, including DL algorithms, clustering algorithm, normalizations and other GD's. Each algorithm's advantages, disadvantages, and implementation are discussed, with a attention on efficiency of correctness, speed and stability in the machine learning algorithms.

In [3], The paper titled "Optimization and plan of machine learning computational procedure for expectation of physical partition handle" presents the development with optimization of machine learning (ML) methods to predict and understand physical separation processes, specifically using adsorption for impurity removal from water. It achives high accuracy ($R^2 > 0.9$). The paper demonstrates that GB and ET models outperform RF, providing a robust approach for modelling adsorption with limited data.

In [4], The paper, "The Impact of Interdisciplinary Approaches on the Development of New Materials and Technologies", explores how interdisciplinary approaches, particularly the integration of materials science with other fields such as biology and engineering, have led to significant advancements in the development of new materials and technologies. The authors highlight the importance of collaborative efforts and the use of advanced computational techniques to design and optimize materials with novel properties.

In [5], "Overview of optimization calculations in cutting edge Neural systems", paper provides a comprehensive overview various algorithms used in NN focusing on 1st order, 2nd order and various IG methods. The authors classify and evaluate various optimization techniques, including Stochastic Gradient Descent (SGD), Adam, Quasi-Newton methods and more. The paper emphasizes on significance of these algorithms in improving the accuracy and efficiency of neural network training, particularly in challenging applications such as quantum computations and spiking neural networks.

2. Fundamentals of Optimization in machine Learning

Almost all machine learning algorithms can be designed as an optimization problem[1] for calculating minimum function value. Creating a model and creating a suitable role is the first step in the machine learning. After the objective function is created ,appropriate numbers or optimization tests are usually used to solve the problems to be solved. ML algorithms can be classified into supervised learning and unsupervised learning. So, below is the equation for the two types.

2.1. Supervised Learning Enhancement

In the directed approach , the objective was to discover the ideal mapping work f(x) to play down the error value of the preparing tests which are given to be as follows:

$$min_\theta \ \frac{1}{R} \ \sum_{i=0}^{r} G(y^i, f(x^i, \theta)) \qquad (1)$$

where R is the number of preparing tests work, x^i specifies point vector of the i_th tests, y^i is the comparing marker, and G down of error value[5].

2.2. Optimization of Unsupervised Learning

Clustering algorithms divides tests into multiple clusters pointing to play down contrasts between the tests within the same are one of kind from each other.[1][5] The k-means clustering calculate points to optimize the following low error function:

$$min_S + \sum_{d=1}^{d} \ \sum_{p \in S_d} \ ||x\text{-}\mu_d|| \qquad (2)$$

where:

d is the number of clusters, p is the include vector of tests, μd is the middle of cluster d, and Sd is the test set of cluster d

For a function $f(x)$ the gradient is:

$$\nabla f(x) = (\frac{\partial f}{\partial x1}, \frac{\partial f}{\partial x2}, \dots, \frac{\partial f}{\partial xn}) \qquad (2)$$

2.3. Formal Optimization Problem Statement: In machine learning, optimization problems emerge from the formulation of prediction and loss functions, which are encapsulated within the expected risk that the model seeks to minimize.[6]

As some of the algorithms in machine learning fails to train the deep neural networks so there the advance techniques originate. Again, let's look over the fundamental role of optimization in deep learning in short.

2.4. Training Deep Neural Networks

- **Objective Function:** In DL, optimization focuses on minimizing a low function that quantifies the mistake between anticipated and genuine values over potentially millions of parameters.

- **Complexity:** Deep neural networks have many layers and parameters, making the optimization problem highly complex. Effective optimization is crucial to ensure convergence and avoid issues like vanishing or exploding gradients. [6][2][5]

2.5. Gradient-Based Optimization

- **Algorithm Choices:** Advanced gradient-based optimization algorithms like Adam, RMSprop, and AdaGrad are essential for training deep learning models efficiently[6]. These methods adapt the learning rate and use past gradients to improve convergence.

- **Momentum and Adaptive Learning Rates:** Techniques like momentum and adaptive learning rates offer assistance speeding up training and improving the joining rate by adjusting the update rules based on past gradients[4].

2.6. Regularization and Generalization

- **Regularization:** Optimization also includes techniques[2] for regularization to prevent overfitting and enhance the model's capability to generalize to concealed information.

- **Hyper parameter Tuning:** Finding the best hyper parameters (e.g. learning rate, bunch size) through optimization procedures like search in grid or Bayesian optimization is crucial for enhancing the model's performance

2.7. System Overview:

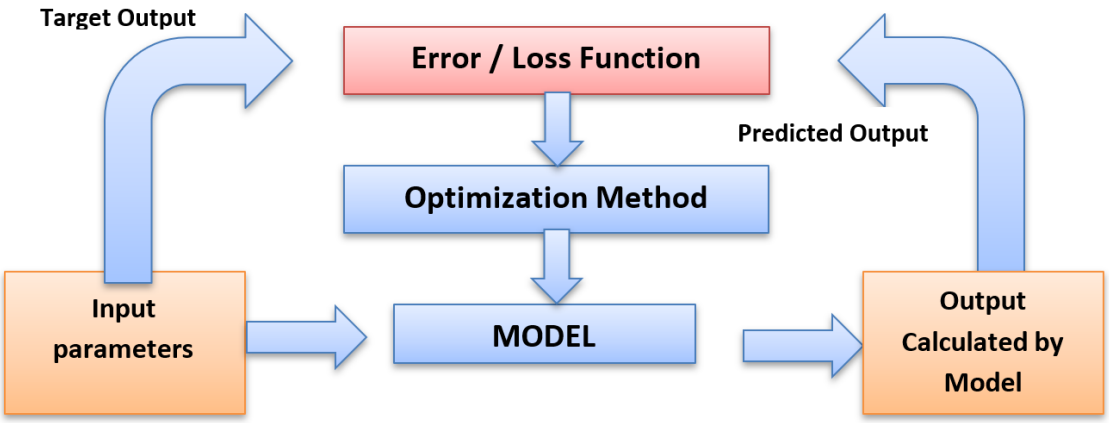

Fig. 1. Overall flow of the Optimization

3. Gradient optimization Techniques

3.1. Gradient Descent:

To model a problem efficiently, we aim to minimize cost, reduce computational power, and enhance performance. GD may be a crucial optimization algorithm utilized to discover the least of a function $f(x)$, significant in machine learning and Deep learning.[6][7][8] It works by iteratively adjusting parameters w to play down the less error function $J(w)$ which measures how well a model's predictions coordinate the real information. The objective is to play down this large cost function to progress model accuracy. Consider , a case of fitting a line to a set of data points using a linear equation: $\boldsymbol{y = mx + c}$ **where:**

- m is the incline of the line and c is the y-intercept

3.2. Challenges in line fitting

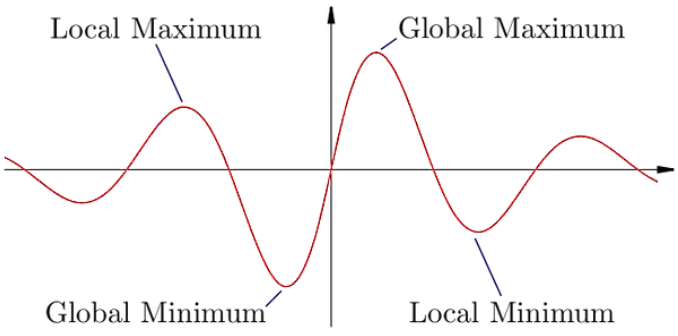

Fig. 2 Representation of Minima and Maxima

- **Global Minimum:**

The minima of global means where the cost function is at its absolute lowest across the entire domain. To find it, gradient descent iteratively adjusts parameters to minimize the cost function, though achieving this can be challenging with complex losses.[8]

Mathematical expression $for\ x = x_0$ to be at a global minimum could be:

$$f(x_0) \leq f(x)\ \forall\ x\ in\ entire\ domain$$

- **Local Minima:**

A local minimum is where the cost function is lower than at nearby points but may not be the lowest value across the entire domain. [8] This GD can meet to a nearby minima of local rather than the minima of global, particularly with complex functions.

Mathematical expression $for\ x = x_0$ to be at a global minimum could be:

$$f(x_0) \leq f(x)\ \forall\ x\ neighbourhood\ around\ x_0$$

- **Global Maxima:**

A global maximum is the highest point in a function's domain, with the function's value being the highest compared to all other points [6]. There can be one or multiple global maxima if the highest value is achieved at several points.

Mathematical expression $for\ x = x_0$ to be at a global minimum could be:

$$f(x_0) \geq f(x)\ \forall\ x\ in\ entire\ domain$$

- **Local Maxima:**

A local maximum is a point where the function's value is higher than at nearby points but not necessarily the highest value overall.

Mathematical expression $for\ x = x_0$ to be at a global minimum could be:

$$f(x_0) \geq f(x)\ \forall\ x\ neighbourhood\ around\ x_0$$

3.3. Derivation and formulation

We've covered the concepts of maxima and minima and their conditions. Now, let's derive the equations for $m\ and\ c$ the Gradient Descent Algorithm.[6][8] The general update for $y = ax + b$ formula in Gradient Descent is:

$$\theta = \theta - LR * \frac{\partial J(\theta)}{\partial \theta} \qquad (3)$$

For a parameter θ (which could be $a\ or\ b$) which defines the slope or intercept of a best fit line.

where:

- θ is the parameter being updated
- LR is the learning rate
- $\frac{\partial J(\theta)}{\partial \theta}$ is the angle gradient of the tests function J with regard to θ

- **Specific updates for m and c:**

$$a = a - LR * pd(a) \quad (4)$$

where:

- $pd(a)$ represents the fractional derivation of the tests function with regard to a, denoted as $\frac{\partial J}{\partial a}$

$$b = b - LR * pd(b) \quad (5)$$

where:

- $pd(b)$ represents the derivation of the tests function with regard to c, denoted as $\frac{\partial J}{\partial b}$

3.3.1. Interpretation

- **Gradient Calculation**: $pd(a)\ and\ pd(b)$ are the gradients showing the direction and rate of change for each parameter.
- **Learning Rate:** LR controls the step size in the direction opposite the gradient. A small LR means slow convergence, while a large LR might overshoot.

3.3.2. After this the mean squared error is given as

$$MSE = \frac{1}{n}\sum_{n=1}^{n} ((r_i - (ax_i + b)))^2 \quad (6)$$

So, by comparing the equations 1,2,3 we calculate the partial derivative of $\frac{\partial J}{\partial a}\ and\ \frac{\partial J}{\partial b}$. After formulating the equations,

$$pd(a) = \frac{-2}{q}\sum_{q=1}^{q} xii\ (r_i - (ax_i + g)) \quad (7)$$

$$pd(b) = \frac{-2}{p}\sum_{p=1}^{p} (r_i - (ax_i + g)) - 1 \quad (8)$$

Then you should update the parameters using chain rule :

$$w_1 = w_1 - LR * \frac{\partial}{\partial w_1} \quad (9)$$

$$w_2 = w_2 - LR * \frac{\partial}{\partial w_2} \quad (10)$$

Again for further calculation the steps could be:

- Define the values of $x\ and\ y$ ii) Assume values of $m\ and\ c$ iii) Calculate partial derivative of $J(m, c)$ iv) Update the parameters as

$$c_{new} = old - LR * \frac{\partial J}{\partial c}$$ and similarly for m (11)

In Machine Learning, Gradient Descent has limitations, leading to the development of its variants. In the graph below, the arrow indicates where the function [8]starts decreasing in cost. By calculating the derivative, we observed that :

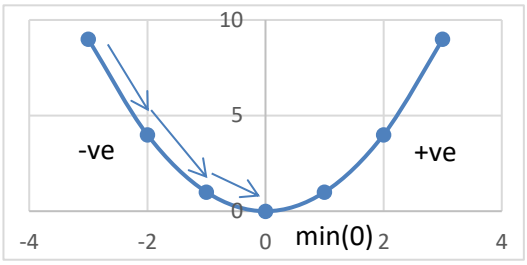

Fig. 3 Minimize the Loss function (x increases and f(x)decreases)

3.4. Limitations of Gradient Descent

- **High Computational Load:** Expensive and slow for large or high-dimensional datasets.
- **Memory Usage**: Requires a lot of memory to store and process large datasets.
- **Learning Rate Sensitivity**: Performance depends on the learning rate, which can affect convergence.

3.5. Types of Gradient Descent

3.5.1. Batch Gradient Descent

BGD calculates the steepest descent using the entire dataset, making it suitable for smaller datasets. It updates parameters based on this complete gradient computation. Batch Gradient Descent (BGD) [6][8]is a variant of the Gradient Descent optimization calculation utilized to discover the least of a error function. It's commonly employed in machine learning and DNN's to optimize the hyperparameters of a model. Similarly, name indicates the batch gradient it takes over a batch of samples to calculate the each of the training samples.

Given:

θ represents the parameters of the model, $J(\theta)$ is the cost function, α(alpha) is the learning rate

$$\theta = \theta - \alpha * \frac{1}{m}\sum_{n=1}^{n} \nabla_{\theta} J(\theta) \qquad (12)$$

here:

- m represents quantity of training examples
- $\nabla_{\theta} J(\theta)$ is the steepest of the cost function
- $J(\theta)$ with regard to the parameters θ

Limitations of Batch Gradient

- **Slow Convergence:** While working on large dataset, it can lead to slow processing.
- **Fixed Learning Rate & Not Suitable for Large Datasets:** A fixed learning rate may not suit all training stages, affecting convergence. High computational and memory demands.

It offers accurate gradient estimates and stable convergence, it can be computationally expensive [3]and memory-intensive, particularly for huge datasets. Variations like Mini-Batch Steepest Descent and progressed optimization calculations have been developed for addressing these limitations.

Algorithm 1. Batch Gradient Descent: Dataset features X, true labels y_{true}, number of epochs, learning rate η, Trained weights w, bias b

- Initialize w ← 1, b ← 0, n ← number of samples in X
- d ← number of features in X
    - Initialize empty lists for cost list and epoch list each epoch i from 1 to epochs

$$y_{predicted} = X * w + b \quad (13)$$

    - compute the gradient for weights (w_grad and b_grad) using:

$$w_{grad} = -\frac{2}{total_{sum}} * X^T * (y_{true} - y_{predicted} \quad (14)$$

$$b_{grad} = -\frac{2}{total\_sum} * X^T * (y_{true} - y_{predicted}) \quad (15)$$

- Update the weights

$$w = w - learning_{rate} * w_{grad}$$

$$b = b - learning_{rate} * b_{grad}$$

- i mod 20 = 0 Append cost to cost_list and i to epoch_list, w, b, cost,cost_list, epoch_list
    - Calculate the cost function :

$$\text{cost} = \frac{1}{total\_sum} * \sum (y_{true} - y_{predicted})^2 \quad (16)$$

3.5.2. Stochastic Steepest Descent (GD)

Stochastic steepest model progresses a parameters more frequently by utilizing the single data point (or small subset) at a time, making it suitable for large datasets and online learning. It offers faster convergence and lower memory utilization compared to clumped slope descent(BGD).
Gradient Descent move towards the[9] minimum slowly.
.
Given:
To update Stochastic Gradient Descent use:

$$\theta = \theta - \alpha * \nabla_\theta J(\theta, x_i, y_i) \quad (17)$$

where: $\nabla_\theta J(\theta, x_i, y_i)$ is the derivative of the function error $J(\theta)$ with regard to the parameters θ computed using single tests example[8] (x_i, y_i)

3.5.3. Limitations of Stochastic Gradient

- **Noisy Convergence:** Updates can be inconsistent, causing oscillations.
- **Hyperparameter Sensitivity:** Requires careful tuning of the learning curve.
- **Many Iterations:** May need many iterations to converge, leading to long training times.

SGD offers faster updates and lower memory usage but can suffer from noisy convergence and sensitivity to hyperparameters. [10] Mini-Batch SGD combines benefits of both SGD and Batch Gradient Descent to improve efficiency and stability.

Algorithm 2. Stochastic Gradient Descent: Dataset features X, true labels y_{true}, number of epochs, learning rate η, Trained weights w, bias b

- Initialize w ← 1, b ← 0, n ← number of samples in X
- Initialize empty lists for cost list and epoch list
- each epoch i from 1 to epochs
    - Randomly pick a sample o f any_index from 0 to *overall samples*
    - *sample x* ← *X*[*any_index*] *sample y* ← y_{true}[*any_index*]

      $y_{predicted} = Sample\_x * w + b$
    - compute the gradient for weights (w_grad and b_grad) using:

      $w_{grad} = -2 * sample\_x(y_{true} - y_{predicted})$

      $b_{grad} = -2 * (y_{true} - y_{predicted})$
- Update the weights

  $w = w - learning_{rate} * w_{grad}$

  $b = b - learning_{rate} * b_{grad}$
- i mod 100 = 0 Append cost to cost_list and i to epoch_list, w, b, cost,cost_list, epoch_list
- calculate the cost function

  cost = $square * (y_{true} - y_{predicted})$

3.5.4. Mini-Batch Optimization

Mini-Batch slope plunge combines various aspects of BGD and SGD by utilizing little, arbitrarily chosen subsets of the preparing information (mini-batches) to upgrade model parameters[10].

Given:

- θ represents the hyperparameters of the model
- J(θ) is the cost function, α is a learning rate
- $\{(x_{i1}, y_{i1}) \ldots . (x_{ib}, y_{ib})\}$ is a mini-batch of training examples with size b

Updating Mini-Batch Gradient Descent can be done using following rule:

$$\theta = \theta - \alpha * \frac{1}{b}\sum_{j=1}^{b} \nabla_{\theta} J\left(\theta, x_{ij}, y_{ij}\right) \qquad (18)$$

where:

$\frac{1}{b}\sum_{j=1}^{b} \nabla_{\theta} J\left(\theta, x_{ij}, y_{ij}\right)$ is the gradient of the cost function computed over mini-batch.

3.5.5. Limitations of Mini-Batch Gradient

- **Hyperparameter Tuning:** The mini-batch size affects performance; too small or too large can impact results.
- **Local Minima:** Can still converge to local minima but is generally more stable than pure SGD..
- **Learning Rate Sensitivity:** Requires careful tuning of less η to fit it more in line.

Mini-Batch optimization[10] is a balanced approach that improves efficiency and stability compared to pure BGD or SGD, making suitable for huge data and deep learning models.

Algorithm 3. Mini-Batch Gradient Descent: Dataset features X, true labels y_{true}, number of epochs, learning rate η, Trained weights w, bias b

- Initialize w ← 1, b ← 0, n ← number of samples in X
- Initialize empty lists for cost list and epoch list
    - each epoch *epoch* from 1 to epochs
      Shuffle X and y_{true} each mini-batch starting at *start* and ending at *end*
      $X\ batch \leftarrow X[start : end]$
      $y_{predicted} = Sample\_x * w + b$
    - compute the gradient for weights (w_grad and b_grad) using:

$$w_{grad} = -\frac{2}{len(y_{batch})} * X_{batch}^{T}(y_{batch} - y_{predicted})$$

$$b_{grad} = -\frac{2}{len(y_{batch})} * X_{batch}^{T}(y_{batch} - y_{predicted})$$

- Update the weights
    - $w = w - learning_{rate} * w_{grad}$
    - $b = b - learning_{rate} * b_{grad}$
- i mod 10 = 0 Append cost to cost_list and i to epoch_list, w, b, cost,cost_list, epoch_list
- calculate the cost function

$$\text{cost} = \frac{1}{total\_sum} * \sum(y_{true} - y_{predicted})^2 \quad (19)$$

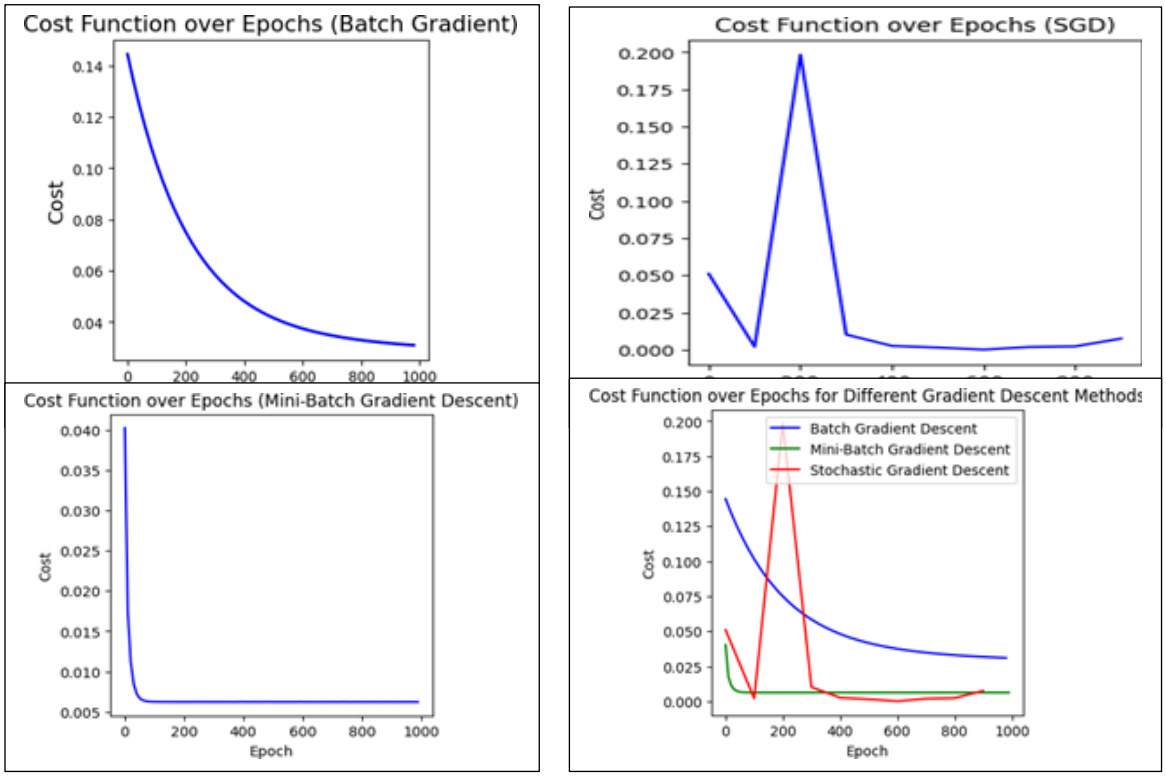

Fig. 4 Graphs Indicating their performance on the trained model Evaluation

4. Advance Gradient Algorithms

They address limitations such as noisy data and convergence issues by refining parameter updates, reducing the impact of outliers, and improving training stability.

4.1. Momentum:

Momentum accelerates gradient descent by combining the current gradient with a fraction of the previous update. This smooths out updates and helps navigate through areas with steep gradients, improving convergence[8][6].

Consider the following data points the time interval t and the weights b as follows [8]:

$t_1\, t_2\, \dots\dots\dots t_n\,,\, b_1\, b_2\, \dots\dots\dots b_n$

If the exponential moving average is applied to the time interval t_1 with the weight of b_1

$$\therefore V_1 = b_1 \qquad (20)$$

$$\therefore V_2 = \gamma * v_1 + b_2 \text{ (value of } \gamma\ 0 \leq \gamma \leq 1) \quad (21)$$

Substituting the value of V_1in equation (2) we get:

$$V_2 = \gamma * b_1 + b_2 \qquad (22)$$

Here the most priority is given to the $b_2\, followed\ by\ b_1$.,

Again for finding V_3 :

$$\therefore V_3 = \gamma * v_2 + b_3 \qquad (23)$$

Substituting the value of V_2in equation (ii) we get:

$$\therefore V_3 = \gamma\,(\gamma * v_1 + b_2) + b_3$$

$$\therefore V_3 = \gamma^2 * b_1 + \gamma * b_2 + b_3$$

So now the final equation looks like:

$$\therefore V_3 = \gamma^2 * b_1 + \gamma * b_2 + b_3 \qquad (24)$$

The momentum can be applied with SGD to remove the remaining noise of the graph.

$$w_{new} = w_{old} - \eta * \frac{\partial L}{\partial w_{old}} \quad (25)$$

Applying the exponential weighted average the equation we get is:

$$w_{new} = w_{old}\left[\gamma * V_{t-1} + \eta * \frac{\partial L}{\partial w_{old}}\right] \qquad (26)$$

where V_{t-1} :

$$V_{t-1} = 1 * \left[\frac{\partial L}{\partial w_{old}}\right] t + \gamma * \left[\frac{\partial L}{\partial w_{old}}\right] t_{-1} + \gamma^2 * \left[\frac{\partial L}{\partial w_{old}}\right] t_{-3} \dots\dots n \qquad (27)$$

Algorithm 1. Momentum: Dataset features X, true labels y_{true}, number of epochs, learning rate η, Trained weights w, bias b, momentum μ

- Initialize w ← 1, b ← 0, $v_W \leftarrow 0$ (velocity for weights), $v_b \leftarrow 0$ (velocity for bias)
  $n \leftarrow$ number of samples in X
- Initialize empty lists for cost list and epoch list
  - each epoch from 1 to epochs
    $$y_{predicted} = X * w + b$$
  - Compute the gradient for weights (w_grad,b_grad) using:
    $$b_{grad} = -\frac{2}{total_{samples}} * \sum(y_{true} - y_{predicted}) \quad (28)$$
    $$w_{grad} = -\frac{2}{total\_samples} * X_{batch}^{T} * (y_{batch} - y_{predicted})$$
- Update the velocity for weights and bias using:

$$V_w = momentum * V_w + learning_{rate} * w_{grad} \quad (29)$$
$$V_b = momentum * V_b + learning_{rate} * b_{grad} \quad (30)$$
$$w = w - V_w \qquad b = b - V_b$$

- i mod 10 = 0 Append cost to cost_list and i to epoch_list, w, b, cost,cost_list, epoch_list
- calculate the cost function as above

4.2. AdaGrad (Adaptive Gradient Algorithm)

Adaptive Steepest Descent [1][2] adjusts the η for each parameter independently based on past angles . Parameters with reliably huge angles of steepest get a less η , where those less steepest get bigger η [2]. The weight update formula is:

AdaGrad is:

$$w_{new} = w_{old} - \eta' * \frac{\partial L}{\partial w_{old}} \quad (31)$$

Now the values are clear so η' with respect to time t will be:

$$w_t = w_{t-1} - \eta'_t * \frac{\partial L}{\partial w_t} \quad (32)$$

where η'_t is given by,

$$\therefore \eta'_t = \frac{\eta}{\sqrt{\alpha_t} + \epsilon}$$

Here the value of the η is initial as 0.01 so as the learning rates will be get changed and the value of ϵ is any +ve number added to α_t to avoid a big error[4]. Now the α_t is given by:

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial L}{\partial w_i}\right)^2 \quad (33)$$

Similarly, as neural networks goes deeper so the $\alpha_t \uparrow$ increases and the learning rate $\eta \downarrow$ decreases.

AdaGrad's limitation is that it can grow very large over time, reducing the learning rate too much. To address this, variants like **AdaDelta** and **RMSprop** use a weighted average instead:

$$\therefore \eta'_t = \frac{\eta}{\sqrt{W_{avg_t}+\epsilon}} \quad (34)$$

Again the ϵ is the +ve number and the all-naming convention remains same as per the above AdaGrad algorithm. Where as the weighted average W_{avg_t} is given as:

$$W_{avg_t} = \gamma * W_{avg_{t-1}} + (1-\gamma) * \left(\frac{\partial L}{\partial w_t}\right)^2 \quad (35)$$

This approach prevents the learning rate from decreasing too quickly. Whereas in the case of W_{avg_t} we are restricting it for $\sum_{i=1}^{t}$ and the value of γ should be 0.95 in approximate cases. So, the value of W_{avg_t} will increase but not will get very-very high as α_t .

4.3. Adam Optimizer

Adam Optimizer combines the benefits of AdaGrad and RMSprop[8] by maintaining moving averages of both gradients and squared gradients.

4.3.1. Algorithm

i) Initially $V_{dw} = 0 \; and \; V_{db} = 0$ and $S_{dw} = 0 \; and \; S_{db} = 0$

$$w_t = w_{t-1} - \eta' * V_{dw} \quad , \quad b_t = w_{db\,t-1} - \eta' * V_{db}$$

$$\eta' = \frac{\eta}{\sqrt{S_{dw}} + \epsilon}$$

ii) On iteration on t inside epoch iii) Compute $\frac{\partial d}{dw}$ on current MBD

where $V_{dw} \; and \; V_{db}$ is given as:

$$V_{dw} = \beta V_{dw\,t-1} + (1-\beta) d_w * \frac{\partial L}{\partial w_{t-1}} \quad (36)$$

$$V_{db} = \beta V_{db\,t-1} + (1-\beta) d_w * \frac{\partial L}{\partial b_{t-1}} \quad (37)$$

So, now the weighted average parameters looks like as:

$$w = w - \eta * V_{dw}$$

$$b = b - \eta * V_{db}$$

So, this was the case with context of hierarchical learning to train DNN's.

Algorithm 2. Adam optimizer: Dataset features X, true labels y_{true}, number of epochs, learning rate η, β_1, β_2, ϵ, Trained weights w, bias b

- Initialize w ← 1, b ← 0, m_w ← 0 (biased first moment estimate for weights), m_b ← 0 (first moment for b), v_w ← 0 (raw 2nd moment for weights) v_b ← **0** (biased second raw moment estimate for bias), t ← **0** (time step)
- Initialize empty lists for cost list and epoch list
  - each epoch epoch from 1 to epochs t ← t+1

$$y_{predicted} = X * w + b$$

  - compute the gradient for weights (w_grad, b_grad) using:

$$w_{grad} = -\frac{2}{X.shape[0]} * X_{batch}^T * (y_{batch} - y_{predicted})$$

$$b_{grad} = -\frac{2}{X.shape[0]} * X_{batch}^T * (y_{batch} - y_{predicted})$$

  - Update the first moment estimate for weights (m_w, m_b) using:

$$m_w = \beta_1 * m_w + (1 - \beta_1) * w_{grad}$$
$$m_b = \beta_1 * m_b + (1 - \beta_1) * b_{grad}$$

  - Update Biased Second Raw Moment Estimate:

$$V_w = \beta_2 * V_w + (1 - \beta_2) * w_{grad}^2$$
$$V_b = \beta_2 * V_b + (1 - \beta_2) * b_{grad}^2$$

  - Compute Bias-Corrected Estimates: Compute the bias-corrected first moment estimate for weights (m_w_hat, m_b_hat) using:

$$m_b^{hat} = \frac{m_b}{1-\beta_1^t} \quad (38)$$

$$V_w^{hat} = \frac{V_w}{1-\beta_1^t} \text{ and } V_b^{hat} = \frac{V_b}{1-\beta_2^t} \quad (39)$$

- Update Weights and Bias:

$$w = w - \frac{\eta * m_w^{hat}}{\sqrt{V_w^{hat}} + \epsilon} \quad (40)$$

$$b = b - \frac{\eta * m_b^{hat}}{\sqrt{V_b^{hat}} + \epsilon} \quad (41)$$

- Compute Cost:

$$\text{cost} = -\frac{1}{X.shape[0]} * \sum (y_{true} - y_{predicted})^2$$

These were the Algorithms that were implemented on the Banglore_Housing_Dataset and the after the plotting of graphs we noticed the below results as:

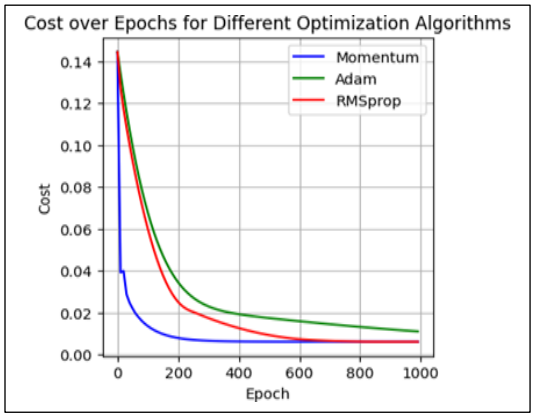

Fig. 5 Comparison of momentum vs Adam vs RMSprop

5. Second-Order Optimization

Second-Order Optimization methods improve upon first-order methods by incorporating insights of curvature regard of less error function, leading potentially faster convergence and better parameter updates. But they are computationally intensive. Approximation techniques help make these methods more practical for complex models[7][8].

5.1. Newton's Method

NM updates parameters using both the gradient and second order derivative matrix with loss equation, allowing the adaptive step sizes based on curvature[8][7].

Newton's Update Rule

Given a loss function L(θ) where θ are the parameters, NM updates parameters according below:

Compute Gradient:

$$g = \nabla L(\theta)$$

where g is the slope vector belonging to loss equation.

Compute Hessian Matrix:

$$H = \nabla^2 L(\theta)$$

where H is the second order derivative matrix and update parameters:

$$\theta_{t+1} = \theta_t - H^{-1}g$$

Where: H is inverse of the second-order Jacobin matrix, and g is the slope vector.

5.2. Broyden-Fletcher-Goldfarb Shanno and Limited Memory bfgs

BFGS and L-BFGS are QN methods that approximate the second-order Jacobin matrix to optimize parameter updates, especially in high-dimensional problems[7].

5.2.1. BFGS (Broyden-Fletcher-Goldfarb-Shanno) algorithm

$$\theta_{k+1} = \theta_k - \alpha_k H^{-1} g_k \quad (42)$$

Where α_k is the step size.

Update Hessian Approximation: Update the inverse Hessian approximation H_{k+1}^{-1} using:

$$H_{k+1}^{-1} = H_k^{-1} + \frac{y_k y_k^T}{y_k^T s_k} - \frac{H_k^{-1} s_k s_k^T H_k^{-1}}{s_k^T H_k^{-1} s_k} \quad (43)$$

where $s_k = \theta_{k+1} - \theta_k \; and \; y_k = g_{k+1} - g_k$

5.2.2. L-BFGS (Limited-memory BFGS) algorithm

$$\theta_{k+1} = \theta_k - \alpha_k H^{-1} g_k \quad (44)$$

Update Approximation: Use the limited-memory approximation to update the inverse Hessian matrix. These were the Algorithms that were implemented on the Banglore_Housing_Dataset and the after the plotting of graphs we noticed the below results as:

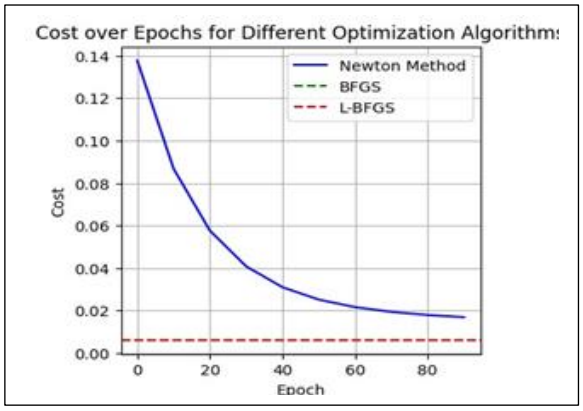

Fig. 6 Comparison of second order optimization

6 Evolutionary Algorithms

6.1 Evolutionary Algorithms Implementations

This technique leverages natural evolution principles to solve complex optimization problems. Evolutionary algorithms (EAs)[11] are probabilistic heuristics inspired by natural selection, where only the best-suited individuals survive and reproduce, iteratively guiding solutions toward an optimal or near-optimal outcome. Similar to Darwinian selection, inferior solutions are discarded, while superior ones evolve to form a new generation with the potential for improved performance..

6.2. Genetic Algorithm (GA):

The adaptive algorithms in DNN are progressive search methods inspired by natural selection. They use random searches and historical data to focus on promising solution areas[11][2]. GAs simulates "survival of the fittest" by evaluating and evolving a population of solutions each generation, with solution.

Algorithm 1. Genetic Algorithm: Optimization variables: Population, generations, mutation rates Hyperparameters: popoulas_size, formation in number, mutation frequency[8]

- Dataset with coordinates, population size, number of formations, mutation frequency
- Initialize *ga best values* as an empty list
  - Initialize *ga best values* as an empty list
  - Set population measures ⟵20
  - Set generations ⟵100
  - Set mutation freqn ⟵0.1
- Initialize population with *gf.sample*(*n = population size*).*to dict*(′*records*′)
  - each generation from 1 to generations
    - Initialize new population as an empty list and each of populace
    - Select guardian 1 utilizing selection(population, fitness)
    - Select guardian 2 utilizing selection(population, fitness)
- Make child utilizing crossover(both guardians)
  - Apply mutate(child, change rate)
  - Add child to unused population
  - individual⟵ min(population, key = wellness) populace ⟵ sort new populous (measure-1)
- Append perfect value with its above equation
- Append fitness(perfect value) to *ga* perfect *values* best value

6.3. Particle swarm optimization (PSO)

Optimization seeks the best parameter values to meet design goals at minimal cost, used in many scientific fields[2]. Traditional algorithms often struggle with issues like local optima and unknown search spaces. To overcome these, metaheuristics like (PSO), (GWO), (ACO), (GA), and cuckoo search were developed. This article delves into PSO, building on the fundamentals of stochastic optimization[9][11]

6.3.1. Key Concepts:

- Particles: Individuals in the population, representing potential solutions.
- Position: A vector representing a particle's current state in the search space.
- Velocity: The rate of change of the loss position, guiding the movement.
- Best Solution(P_i): Better position of a particle has achieved largely.
- Global good solution (G_i): Good global positions achieved by some particle in PSO.

6.3.2. Overview of mathematics

- Every fragment in PSO has a related position, speed, wellness value.
- Each molecule keeps track of the bestFit, bestFitPosition.
- A record of global_best, Fit_position and global_best, Fit_value is kept up.

6.3.3. Mathematical Formulation

i. Velocity Update

To change the speed of each fragment, the equation formula is:

$$z_i\ (e+1) = \omega v_i(e) + c_i r_i(u_i - v_i) + \ c_2 r2(h - v_i)\ (45)$$

Components:

- $z_i(e)$: Current speed of element i at times t.
- ω: Inertia weight, balancing exploration and exploitation.
- c_1: Cognitive coefficient, representing the influence of the personal best position.
- r_1: Any number between 0 and 1, introducing randomness as the cognitive component.
- u_i: Owns best position of fragment *i*.
- v_i: Current position of particle iii.
- c_2: Social coefficient, representing the influence of the global best position.
- r_2: Random number between 0 and 1, introducing randomness in the social component.
- h: Denotes global best position found by the swarm.

ii. Position update equation

The position of each molecule is balanced utilizing:

$$v_i\ (e+1) = v_i\ (e) + z_i(e+1) \qquad (46)$$

This equation moves the particle to a new place belonging to its ongoing position and the updated velocity.

Algorithm 2: Particle Swarm Optimization: Optimization variables: Particle positions, velocities, best positions, global best position, Parameters: weight of inertia *w*, coefficient *c*1, sociol coefficient *c*2[8]

- Initialize the dataset with coordinates, number of particles, number of loops, inertia weight *w*, C.coefficient *c*1, S.coefficient *c*2 Global best position and value.
- Initialize pso *best values* as an empty list Class Particle :
  - Initialize ***position*** ← ***np.array*([*x, y*])**
- Initialize ***speed*** ← ***array*[*any uniform*(−1, 1), *any uniform*(−1, 1)]**
- Initialize *best* ***position*** ← ***position*** Initialize *best* ***value*** ← ***float*(′*inf*′)** Method
- update velocity (global best position):
  - Generate unique num r1, r2 ***cognitive*** ← ***c*1 · *r*1** · (*best position*)
  - ***socio*** ← ***c*2 · *r*2** · (*global best site*)
  - *speed* ← *w* · *speed* + *cognitive* + *social* Method update position():
    position ← ***position*** + ***speed***
- Load data from "tsp.csv" and initialize particles with coordinates
  - Initialize *global best* ***value*** ← ***float*(′*inf*′)**
  - Initialize *global best* ***position*** ← ***np.array*([0, 0]) each** iteration from 1 to 100 each particle
  - Compute ***value*** ← ***objective*** *f* (*particle.position*) *value* < *particle.best value*
    Store the best value in best position with best global value with perfect global location
- Add g's best number to *pso better values global best place* , g's *value*

6.4. Differential Evolution (DE)

Mechanism: DE could be a population-based calculation algorithm. It optimizes a issue by repeatedly attempting to progress candidate arrangements based on a indicated degree of quality[4][2][8]. The key idea behind DE is the utilize of vector contrasts for perturbation of the population members, which helps to explore the search space efficiently.

Mechanism in Detail:

- Initialization: Each individual is represented as a vector x_i .
- Mutation:
  For each target vector x_i, a donor vector v_i is produced by including the weighted difference between two arbitrarily chosen populace vectors x_r2 and x_r3 by another randomly selected vector x_r1 [5][6]:

$$v_i = x_{r1} + F.(x_r2 - x_r3) \quad (47)$$

- Crossover:
  The crossover operation mixes the target vector x_i and the giver vector v_i to shape a trial vector u_i. The components in trial vecs are determined as follows:

$$u_i[j] = \begin{cases} v_i[j] & if\ randj \leq CR\ or\ j = j_{rand} \\ x_i[j] & otherwise \end{cases} \quad (48)$$

  CR is the crossover probability, $rand_j$ is a uniformly distributed random number, and j_{rand} are arbitrary chosen num's that ensures only one element by donor vecs [2]

- Iteration:
  Steps 2 to 4 are repeated for a predetermined number of generations or until a satisfactory solution is found.

Algorithm 3: Differential Evolution: Optimization variables: Population, mutation factor, crossover rate, generations, Parameters: populace size, mutation element, crossover rating, number of components

- Dataset with coordinates, populace size, mutation element, crossover rates, number of components and the best solution
- Initialize de_best_values as an empty list
  - Set populace size ← 10
  - Set mutation element ← 0.8
  - Set crossover rating ← 0.7
  - Set components ← 100
  - Initialize population with *np.array*([*x*,*y*]) for coordinates from *df*
- Randomly sample population to size population size each generation from 1 to generations
- Initialize new population as an empty list each individual *i* in range(population size)
  Generate unique indices *a idx*, *b idx*, *c idx* from population
  - *a*, *b*, *c* ← *population*[*a idx*], *population*[*b idx*], *population*[*c idx*]
- Mutation:
  - Compute *mutant vector* ← *a* + *mutation factor* · (*b* − *c*)
- Crossover:

- Compute crossover vector by mixing mutant vector and population[i] crossover vector[j] ← mutant vector[j]
  if *random.random*() < *crossover rate* else *population*[*i*][*j*]
- Selection:
  - If *objective function*(*crossover vector*) < *objective function*(*population*[*i*])

- **THEN**
- Append crossover vector to new population , append population[i] for new poeple
- Update populace to new populace_int
  - Compute excellent solution ← *min*(*populace*, *key* = *objective function*)
  - Append objective function(best solution) to *de best values* best solution

---

7. Bayesian Optimization

Bayesian Optimization is a probabilistic method for optimizing complex, unknown functions with expensive evaluations, like hyper parameter tuning. It uses a surrogate show, regularly a Gaussian Process, to surmised the function and efficiently [8] guide the search toward optimal solutions with minimal evaluations.

7.1. Working

- Initial Sampling: Begin with a small set of objective function samples to train the surrogate model.
- Prediction: The surrogate model, using a Gaussian Process, predicts the mean (expected output) and variance (uncertainty) of the objective function at new input points.
- Acquisition equation: This work equalizations investigation for and exploitation (sampling high-performance areas)[8][8] to suggest the next evaluation point.
- Evaluation and Update: Evaluate error function with suggested calculation, update surrogate structure result with refine predictions.
- Iteration: Repeat the process, using the surrogate model to progressively zero in on the global optimum.

Acquisition Function: Balancing Exploration and Exploitation[8][11]

In Bayesian Optimization, the acquisition function guides the selection of the next evaluation point by balancing:

i. **Exploration**: Investigating areas of the search space with high model uncertainty to discover potential new solutions.
ii. **Exploitation**: Focusing on regions in surrogate framework calculates the large efficiency to refine best-known solution.[4]

Using predictions from the surrogate model, typically a gaussian-methods, the this indicates the forward point to make efficient, balancing these two objectives.

i. Expected Improvement

- **Concept:** EI predicts in gain over of current best solution by considering both the likelihood and magnitude of improvement.
- **Mathematical Explanation:** EI calculates the expected gain over the best-observed value using the surrogate model's mean and variance.
- **Balance:** EI balances exploration (uncertainty) and exploitation (potential improvement).

Formula:

$$EI(x) = E[maz(0, f(x] - f(x^*))] \qquad (49)$$

Where f(x) is predicted value at point x, and *f(x*)*[4] is good value.

ii. Probability of Improvement

PI aims in maximizing of probability for improving over the best-observed value.

Mathematical Explanation: PI calculates the probability that the objective function at a new point will outperform the best-known value, based on the surrogate model's prediction.
Balance: PI tends to favor exploitation more than exploration, especially if the threshold for improvement is set close to the best-observed value.
Formula:

$$PI(x) = \emptyset\left(\frac{\mu(x) - f(x*) - £}{\sigma(x)}\right) \quad (50)$$

where: $\mu(x)$ & $\sigma(x)$ are (predicted mean and S.D) at point x, *f(x*)* is the best-observed value, and $\emptyset$ is the total dissemination work of the standard typical conveyance. £ is a little parameter utilized to control the exploration-exploitation trade-off.

iii. Upper Confidence Bound (UCB)

Concept: UCB (Upper Confidence Bound) encourages exploration by focusing on areas with high uncertainty.

Mathematical Explanation: It selects points based on the predicted mean plus a multiple of the standard deviation, balancing exploration and exploitation.

Balance: The weight of the uncertainty term can be adjusted to favor exploration or exploitation. Formula:

$$UCB(x) = \mu(x) + k.\sigma(x) \quad (51)$$

Where $\mu(x)$ and $\sigma(x)$ are the predicted mean and standard deviation at point x, and k is a parameter that controls the exploration-exploitation trade-off.

7.2. Optimization Process

Bayesian Optimization is an iterative process that refines a surrogate model and selects new evaluation points based on an acquisition function:

- Initialization: Evaluate the objective function at initial points, often chosen randomly or via space-filling designs like Latin Hypercube Sampling[2][4], and fit an initial surrogate model (e.g., Gaussian Process).
- Surrogate Model Update: Update the surrogate model with new objective function evaluations to better estimate the function and its uncertainty.
- Acquisition Function Calculation: Compute the acquisition function using the updated model to balance exploration and exploitation.
- Termination: Stop when the criterion is reached and output the best solution found.

Algorithm 4: Bayesian Optimization : Number of restarts for acquisition function, number of iterations for optimization Bounds: Latitude and longitude ranges

---

- Dataset X_{sample}, Y_{sample} , Gaussian Process Regressor gpr, bounds, number of n iterations
- Initialize *bo_best_values* as an empty list
    - each iteration i from 1 to *n iteration*

- Fit gpr on X_{sample} and Y_{sample}

- Propose the next location to evaluate using $propose\ location(X_{sample}, Y_{sample}, gpr, bounds)$
  - $next\ sample \leftarrow propose\ location\ \ next\ sample \leftarrow next\ sample.reshape(1, -1)$

- Obtain the objective function value for $next\ sample$
  - $next\ value \leftarrow objective\ function(next\ sample[0])$

- Update X_{sample} and Y_{sample} with $next\ sample$ and $next\ value$

- Track the best value found so far
  - $best\ index \leftarrow np.argmin(Y_{sample})$ Append $Y_{sample}[best\ index]$ to bo_best_values

- Compute $best\ solution \leftarrow X_{sample}[best\ index]$
- Compute $best\ value \leftarrow Y_{sample}[best\ index]$
- Print $best\ solution, best\ value$

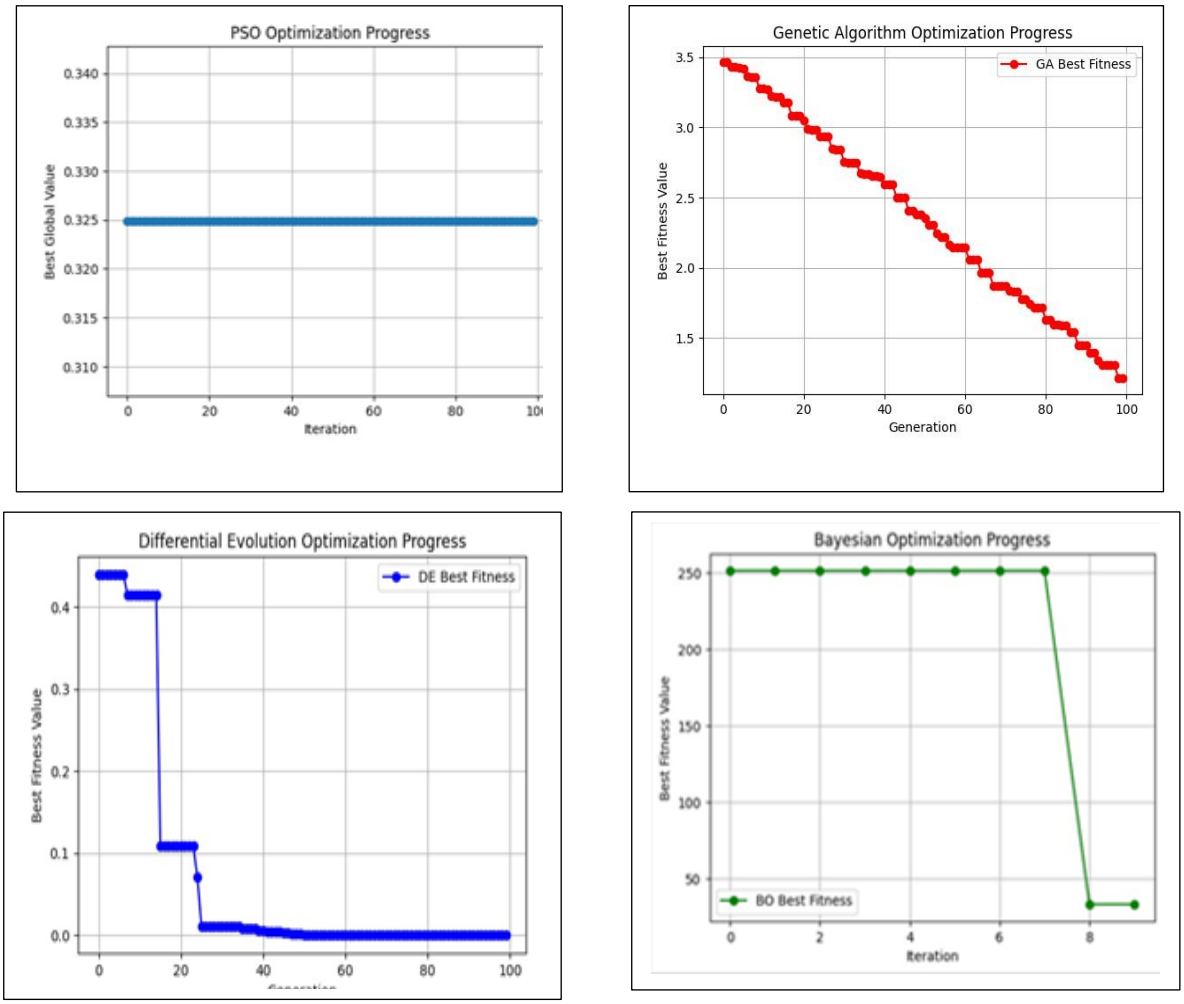

Fig. 7 Graphs Indicating their performance on the trained model Evaluation

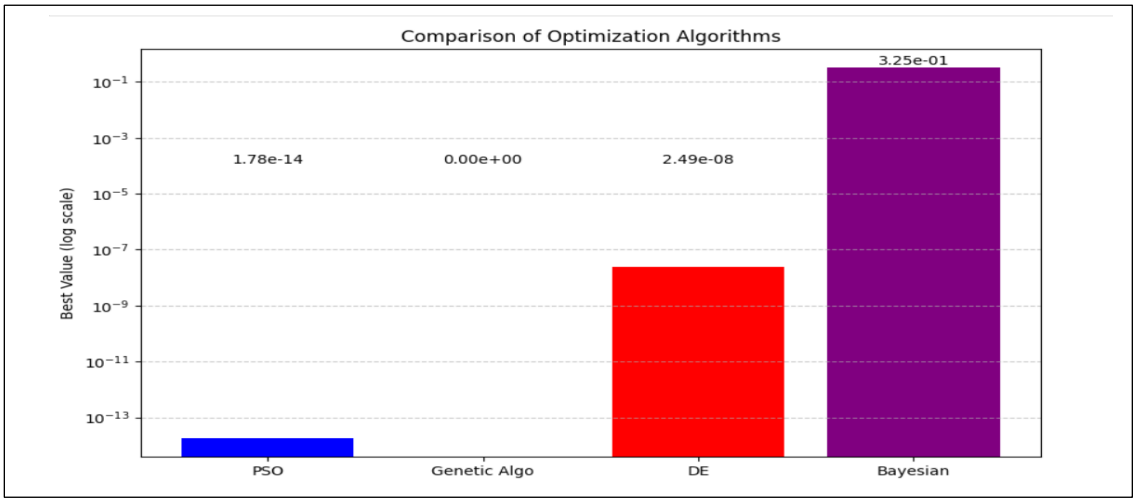

Fig. 8 Comparison of PSO , GA , DE , Bayesian optimizer

8. Challenges in Deep Learning Optimization

- **Non-Convexity:**
  Loss functions in deep learning are typically non-convex, with multiple local minima and saddle points. Unlike convex functions, where any local minimum is global, non-convex functions can have many zero-gradient points[11] that aren't optimal, making it challenging to find the global minimum and leading to potential local minima traps, especially in complex neural networks.
- **Vanishing/Exploding Gradients:**
  In deep neural networks, gradients can either vanish (slow learning with small updates) or explode (cause instability with large updates). This hinders training and may require careful initialization, normalization, or alternative architectures like ResNets.
- **High-Dimensional Spaces:**

  Deep learning models face slow and unpredictable convergence due to complex loss landscapes with many valleys and plateaus. The curse of dimensionality worsens this by exponentially increasing space volume and making gradients sparse and hard to interpret.

8.1. Regularization Techniques

Regularization[8] prevents overfitting by appending penalty to the error function, discouraging framework from exact fit too closely to training data. Known types are L1 and L2 regularization.

8.1.1. L1 Regularization (Lasso)

Definition: L1 regularization[6][8], too known as Lasso, include a large loss to the misfortune work that's relatively to the whole of the outright values of the framework parameters (weights).
Mathematical Formulation:
The model with L1 regularization can be stated as

$$Loss = Original\ Loss + \lambda \sum |w| \qquad (52)$$

where:

- w represents the framework parameters (weights),
- λ is the penalty term,
- $\sum |w|$ is whole of appropriate values of the weights.

Impact:
Encourages Sparsity: L1 regularization pushes some weights to zero, creating a sparse model.
Feature Selection: It helps identify important features by driving irrelevant weights to zero.

8.1.2. L2 Regularization (Ridge)

Definition: This regularization[6][8], too called as Ridge Regression, adds a large loss that is similar to whole of 2 of the frameowrks parameters (weights).
Mathematical Formulation:
Incase loss function, incorporating this, can be expressed as follows:

$$Loss = Original\ Loss + \lambda \sum |w^2| \qquad (53)$$

where:

w represents the hyperparameters (weights),
λ penalty term,
$\sum |w^2|$ is the sum of sqaures of the weights.

Impact:
Prevents Large Weights: Penalizes large weights, reducing model variance and improving generalization.
Smooth Solutions: Distributes weights more evenly across features, avoiding sparsity and including all features.

Note: For implementation codes please refer the below github links

- https://github.com/Gayuu03/Research
- https://github.com/bhak90-tesh/Optimization

The code has been implemented with the references to the above algorithms and [8][6][5] citations.

9. Results and Discussions

9.1. Dataset Description:

The dataset used in this analysis comprises information on residential properties in Bangalore, with a focus on three key attributes: area, number of bedrooms, and price. Area: This column represents the size of the property, measured in square feet It indicates a diverse range of property sizes from smaller units to larger homes. Bedrooms: This attribute indicates the number of bedrooms in the property. The dataset features properties with a varying number of bedrooms, from 1 to 6, Price: The price column denotes the cost of the property, expressed in lakhs of Indian Rupees (INR).

Link for dataset: https://www.kaggle.com/datasets/amitabhajoy/bengaluru-house-price-data

1) Gradient Descent Algorithms:

Table 1 Comparing MSE, MAE, R^2 of implemented algorithms

Optimization Algorithm	MSE	MAE	R^2
Batch Gradient	0.030	0.117	-1.085
Stochastic Gradient	0.014	0.078	0.004
Mini-Batch Gradient	0.006	0.044	0.578

- **MBD** is the best-performing algorithm all over the three, with the cheapest metrics and huge of third parameter from the table.

2) Advance Gradient Descent Algorithms:

Table 2 Comparing MSE, MAE, R^2 of implemented algorithms

Optimization Algorithm	MSE	MAE	R^2
Momentum Optimizer	0.006	0.043	0.578
Adam Optimizer	0.011	0.068	0.253
RMSprop Optimizer	0.006	0.043	0.579

- **RMSprop** is the best-performing algorithm among the three, with the lowest MSE and the highest R^2, although Momentum is very close in performance.

3) **Second Order Optimization Algorithms:**

Table 3 Comparing MSE, MAE, R^2 of implemented algorithms

Optimization Algorithm	MSE	MAE	R^2
Newton's Optimization	0.016	0.084	-0.095
BFGS Optimization	0.006	0.043	0.579
L-BFGS Optimization	0.006	0.043	0.579

- **BFGS and L-BFGS** is the best-performing algorithm among the three, with the lowest MSE and the highest R^2

So, here we can conclude that Mini-Batch GD, RMSprop and BFGS and L-BFGS are the best algorithms in their respective groups.

9.2. Dataset Description: The dataset used in this research is sourced from a file named tsp.csv. and contains geographical information, including latitude, longitude, and city names. The dataset contains 551 entries and 4 columns, with a mix of numeric and object (string) data types. Below is a summary of each column: Column 1: Contains numeric data (latitude coordinates), stored as float64. Column 2: Contains numeric data (longitude coordinates), stored as float64. Column 3 (City names): Contains string data representing city or location names. Column 4 (Unnamed: 3): Mostly contains null values, with only one non-null entry.

The Dataset was created by taking the latitude and longitude points from the google map with the help of Google_API.

1) Evolutionary Algorithms:

Table 4. Comparing the Best Value of implemented algorithms

Optimization Algorithm	Best Value	Mean Iterations to Best	Mean Best Value	Standard Dev. of Best Value
Particle Swarm Optimization	948.7	0.0	948.7	0.0
Genetic Algorithm	4.77	0.0	32.9	20.37
Differential Evolution	2.9	45.4	53.8	253.67
Bayesian Optimization	32.76	10	564.42	439.62

DE seems to be the best based on the best value and the closeness to the optimal solution, despite its higher variability and longer average iterations.

Overall, **Differential Evolution (DE)** appears to be the best choice given its extremely close best value to optimal. However, if consistency is critical, **PSO** might be preferred.

10 Conclusion:

This research provides a complete exploration of optimization strategies in ML, with a focus on their quantitative impact on model performance. Experiments were conducted across several optimization algorithms, evaluating their performance using various different key metrics. Among the gradient-based methods, Mini-Batch Gradient Descent demonstrated superior performance with an MSE of 0.006, an MAE of 0.044, and an R^2 of 0.578, outperforming both Batch and Stochastic Gradient Descent.

In advanced gradient descent algorithms, RMSprop stood out with an MSE of 0.006, an MAE of 0.043, and an R^2 of 0.579, closely followed by the Momentum Optimizer, which achieved nearly identical results.. When evaluating evolutionary algorithms, Differential Evolution (DE) emerged as the most effective, achieving a best value of 2.9, despite requiring an average of 45.4 iterations to reach this value. While Particle Swarm Optimization (PSO) provided a consistent best value of 948.7 with no variability, DE's proximity to the optimal solution makes it a preferable choice, especially when optimization accuracy is prioritized.

In summary, Mini-Batch Gradient Descent, RMSprop, and BFGS/L-BFGS were identified as the top-performing algorithms in their respective categories, demonstrating significant improvements in model accuracy and convergence speed. The study's results, supported by numerical performance evaluations, provide valuable insights for researchers and practitioners, contributing to the ongoing advancements over the fields of ML and DL.

References

[1] Training K. Karthick Department of Electrical and Electronics Engineering, GMR Institute of Technology, Rajam - 532127, Andhra Pradesh, India, "Comprehensive Overview of Optimization Techniques in Machine Learning" Control Systems and Optimization Letters, Vol. 2, No 1, 2024

[2] Networks Ruslan Abdulkadirov, Pavel Lyakhov and Nikolay Nagornov, "Survey of Optimization Algorithms in Modern Neural" Mathematics 2023, 11, 2466. https://doi.org/10.3390/math11112466

[3] Haiqing Li, Chairun Nasirin c , Azher M. Abed d , Dmitry Olegovich Bokov e,f , Lakshmi Thangavelug , Haydar Abdulameer Marhoon, Md Lutfor Rahman, " Optimization and design of machine learning computational technique for prediction of physical separation process", 1878-5352 2022 Published by Elsevier B.V. on behalf of King Saud University

[4] Raniah Zaheer Lecturer Department of CS, Najran University, Humera Shaziya Assistant Professor Informatics, Nizam College Osmania University, "A Study of the Optimization Algorithms in Deep Learning", International Conference on Inventive Systems and Control (ICISC-2022)

[5] L´eon Bottou, Frank E. Curtis‡ Jorge Nocedals, Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao "Optimization Methods for Large-Scale Machine Learning" Journal of Machine Learning Research, vol. 18, 2021.

[6] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao "A Survey of Optimization Methods from a Machine Learning Perspective", arXiv :1906.06821

[7] Xiaolong Li;Feng Feng;Shuxia Yan;Wei Zhang;Qi-Jun Zhang, "Simulation-Inserted Optimization of Four-Order Waveguide Filter Using Combined Quasi-Newton Method With Lagrangian Method" 2023 IEEE MTT-S International Conference

[8] Optimization for Machine Learning by Suvrit Sra, Sebastian Nowozin, and Stephen J. Wright & Optimization for Machine Learning and Machine Learning for Optimization by Rachid Chelouah, Patrick Siarry

[9] Carmina Fjellström,Kaj Nyström, "Deep learning, stochastic gradient descent and diffusion maps"

[10] Haobo Qi,Feifei Wang &Hansheng Wang, "Statistical Analysis of Fixed Mini-Batch Gradient Descent Estimator"

[11] David Shulman, Department of Chemical Engineering, Ariel University, Ariel, Israel 407000 vol. 18, 2023.