Crow Way: An Optimization Technique for generating the Weight and Bias in Deep CNN

Rahul R. Papalkar[1] Dr. Abrar S Alvi[2], Prof. Vinod Rathod[3], Prof. Ahmer Usmani[4], Prof. Vivek Solavande[5], Prof. Datta Deshmukh[6]

[1,2]*Prof. Ram Meghe Institute of Technology & Research, Badnera, Amravati SGBAU Amravati; E-mail: rahulpapalkar@gmail.com, asalvi@mitra.ac.in*
[3]*Bharti Vidyapeeth deemed University Pune, Department of Engineering and Technology Navi Mumbai*
[4,5,6]*Bharti Vidyapeeth Deemed to be University, Department of Engineering & Technology, Navi Mumbai.*

Abstracts: Fine-tuning Convolutional Neural Networks (CNNs) weights and biases are essential for solving difficult machine learning problems. We provide a hybrid optimizations strategy that combines the benefits of Crowd Search (CSA) and Brainstorm Optimization (BSO). CSA-BSO optimizes CNNs. CSA-BSO optimizes CNN weights. Our fitness measures determine each crow's location and speed throughout the CSA phase. The BSO phase mixes optimal replies with random ones to create new solutions. The best new ideas move on. After a certain number of iterations, we alternate between the CSA and BSO phases and pick the best solution as the CNN's final state. We design and test the CSA-BSO approach on many CNN architectures and datasets. Experimentally, the hybrid algorithm outperforms CSA and BSO in convergence time and solution quality. CSA and BSO collaborate to efficiently employ feasible regions and investigate a bigger search field, boosting optimizations. Stable and adaptable, CSA-BSO supports many CNN architectures and data formats. This study's main contribution is the CSA-BSO, a unique hybrid optimizations approach for convolutional neural network weight and bias optimizations. It beats CSA and BSO. The recommended strategy improves CNN picture classification, object recognition, and NLP.

Keywords: Convolutional Neural Networks, Optimization Algorithms, Crow Search Algorithm, Brainstorm Optimization, Hybrid Algorithm, Weight Optimization, Bias Optimization

1. INTRODUCTION

1.1 Backgisnd and Motivation: Computer vision, natural language processing, robotics, and most recently cyber security are just few of the areas where deep learning has emerged as a useful approach for addressing complicated issues. Deep learning's performance is highly dependent on fine-tuning the model's weights and biases through optimizations. Achieving top-notch model performance relies heavily on robust weight and bias generation. As deep learning models grow in complexity and size, sophisticated optimizations methods become more vital.

1.2 Problem Statement: Despite the advancements in deep learning optimization, selecting the most suitable optimization technique for weight and bias generation remains a challenge. There are various optimization algorithms available, each with its own strengths and limitations. Understanding the characteristics of different optimization techniques and their impact on model performance is essential for practitioners and researchers in the field. Thus, there is a need to comprehensively analyses and evaluate distinct optimization techniques for weight and bias generation in deep learning.

1.3 Objectives:
The present research proposes and evaluates CSA-BSO, a hybrid optimizations technique for weight and bias optimizations in Convolutional Neural Networks (CNNs). Research goals:

- Develop a hybrid algorithm: The goal is to create a unique hybrid optimization algorithm that combines the strengths of the Crow Search Algorithm (CSA) and Brainstorm Optimization (BSO) for CNN weight and bias optimization. To optimize, the algorithm should traverse the search space and exploit potential locations.

- Implement the CSA-BSO algorithm: The goal is to implement the suggested method, including the mathematical equations and actions for updating CNN weights and biases. Different CNN designs and datasets require flexible implementation.
- The goal is to empirically test the CSA-BSO method on various CNN architectures and datasets. Convergence speed, solution quality, and algorithm resilience across optimization contexts evaluated.
- Compare the CSA-BSO algorithm to standalone CSA and BSO techniques for CNN weight and bias optimization. The comparison should show how the hybrid algorithm outperforms the separate techniques.
- Applicability and generalizability: The goal is to demonstrate the CSA-BSO algorithm's applicability and generalizability across machine learning applications such image classification, object identification, and natural language processing. This objective shows the algorithm's adaptability and efficacy in real-world circumstances.

The research goals propose a hybrid optimization technique for CNNs that enhances convergence speed and solution quality.

1.4 Motivation:

The need to improve deep learning model performance and efficiency drove the development of "Crow Way," an optimizations method for weights and biases. Deep learning excels in computer vision, natural language processing, and speech recognition. For big and complicated models, optimizing deep neural network weights and biases is difficult. Crow Way addresses many motivations:

- Nature-Inspired Algorithms: Crow Way takes inspiration from crows' clever foraging and communication routines. Nature-inspired algorithms may solve complicated optimizations issues, making them a promising method for weight and bias selection in deep learning.
- Model Performance Improvement: Crow Way's main goal is to improve deep learning model performance. Accurate weight and bias selection improve model convergence, reduces loss functions, and improves model prediction accuracy, precision, and recall.
- Handling Deep learning methods use enormous datasets with high-dimensional characteristics. Large search spaces may challenge traditional optimizations methods. Crow Way's inspiration from crows' intelligence may help it explore difficult search spaces and find appropriate weights and biases.
- Robustness and Generalization: Crow Way-optimized deep learning models should increase robustness and generalization. An optimized model should perform well on both training and unseen data, delivering trustworthy and consistent predictions across scenarios.
- Reducing Computational Complexity: Training deep learning models is computationally and time-consuming, especially for large-scale structures. Crow Way reduces optimizations computational complexity to speed model training and deployment.

1.5 Scope and Organization of the Paper: The scope of this research paper is focused on the analysis of optimization techniques for weight and bias generation in deep learning. The paper is organized as follows: Section 2 provides an Deep Survey on Optimization Techniques along with its limitation Section 3 explores overview of deep learning optimization, highlighting the importance of weight and bias generation. Section 4 explores gradient descent and its variants . Section 5 delves into adaptive optimization techniques Section 6 explore discusses evolutionary and swarm optimization methods. Section 7 describes the experimental setup, along with proposed algorithm implementation and its results Finally, conclude with outlines future directions.

2. LITERATURE SURVEY:

The brain stores and organizes information using the probabilistic Perceptron [1]. This study used Perceptron learning. This study introduces binary classification perceptron. The system successfully obtains linearly separable patterns. Limitations: This method only solves linear problems.

"Rumelhart, D.E., et al. (1986)." The proposed method is backpropagation. Error propagation backward updates weights and biases. The results show the possibility to simplify multi-layer neural network training. This strategy has drawbacks. Local minima might trap you. Weight initialization is crucial for performance [2].

Eun et al. (1998) studied. This work examines gradient-based learning for document recognition. This work will use the LeNet-5 architecture and stochastic gradient descent optimizations algorithm. This system recognizes handwritten digits using backpropagation and gradient descent. Digit recognition capability is top-notch. It can only handle low-resolution grayscale images [3].

Hinton et al. (2006) studied. This study uses contrastive divergence-trained Deep Belief Networks (DBNs). The programmed trains deep generative models layer-by-layer. Deep neural networks may be pre-trained and fine-tuned unsupervised. This method is for unsupervised learning. It needs more supervised fine-tuning to improve performance [4].

(2014) Goodfellow et al. Two neural networks—a generator and a discriminator—make up Generative Adversarial Networks (GANs). Generative Adversarial Networks (GANs) are the topic. This approach trains generator and discriminator networks in a two-player game. This technique generates realistic samples using random noise. Training this model can cause mode collapse and instability [5].

Kingma, D. P., Ba et al. proposed the Adam optimizations method. This function calculates parameter adaptive learning rates effectively. Convergence speed and model correctness improved. Hyperparameter tweaking can lead to inferior solutions [6].

2019 Zhang, H., and colleagues' research. Fixup Initialization: Residual Learning without Normalization addresses vanishing/exploding gradients to improve deep neural network training. Initialization Fixup Technique improves deep neural network training without batch normalization. Results with fewer layers and parameters are competitive. Network designs limit generalizability [7].

In [8] researchers used over-parameterized adaptive gradient algorithms. In this study, we explore the potential benefits of over-parameterization for optimizations methods. The findings point to a faster convergence rate and better generality. Over-parameterization raises the bar for both processing power and storage space.

(2020) Liu, Y. This study describes a biologically inspired deep neural network training optimizations technique. The approach is the Firefly Algorithm (FA). Firefly Algorithm (FA) swarm intelligence optimizes weights and biases. The findings show faster convergence and model correctness than standard optimizations strategies. Complex multi-dimensional situations may challenge this model[9].

Chen, M., et al. (2021) cites. A metaheuristic algorithm-based review of neural network weight and bias tuning. This study employed GA, PSO, and DE, among other methods. This paper investigates metaheuristic methods for weight and bias parameter optimizations in applications. Different algorithms have different convergence times and model accuracies. Algorithm parameters and task complexity affect system performance[10].

2021. Li, Z. This research introduces self-supervised learning, ensemble clustering, and an ensemble teacher-student paradigm for unbalanced data categorization. This study used ensemble clustering and an ensemble teacher-student architecture. Self-supervised learning solves unbalanced data categorization in this method. Improved unbalanced dataset handling boosts performance. Model accuracy has also increased. Ensemble-based techniques may need large computational resources [11].

Wang et al. (2022) studied. This study examines whether dynamic neuron allocation accelerates neural network training. DNA is Dynamic Neuron Allocation. Dynamic neuron allocation improves training efficiency and speed. The project improved model accuracy, training speed, and computing costs. Dynamically distributing neurons may increase computational overhead [12].

Hu, Z., et al. (2022) explores weight initialization and normalization strategies to improve neural network training. The experiment improved model accuracy, training stability, and disappearing or bursting gradients. Network design and issue characteristics determine initialization and normalization [13].

3. DEEP LEARNING OPTIMIZATION:

3.1 Introduction to Deep Learning Optimization :

To train deep neural networks to perform well across a variety of tasks, deep learning optimizations is essential. Optimizing network weights and biases minimizes the loss function. This improves model accuracy and generality. Deep learning optimizations involves iteratively changing model parameters. This adjustment is based on training data faults or disparities between anticipated outputs and genuine labels. The optimizations method finds the parameter values that decrease disagreement and maximize model correctness. Optimization techniques control network parameter changes during training. These systems use gradient descent and its modifications to repeatedly change network weights and biases. Adjusting these settings to reduce the loss function is this procedure. Deep learning optimizations seeks to find a loss function global minimum. The network performs best on training data. Deep neural networks are complex and multi-dimensional, making this goal challenging to achieve. Calculating loss function gradients in respect to model parameters is part of the optimizations process. Gradients alter parameter values. Optimization method, learning rate, batch size, and other hyperparameters affect convergence speed and solution quality. Optimization methods for training deep neural networks have advanced in recent years. Adam, RMSprop, dropout, weight decay, and stochastic gradient descent with momentum are the approaches. Optimization strategies have helped deep learning in computer vision, natural language processing, and speech recognition. Novel optimizations algorithms to improve deep neural network training efficiency, convergence speed, and generalization performance are under investigation.

3.2 Challenges in Deep Learning Optimization :

The optimizations of deep neural networks presents various challenges that must be resolved in order to achieve efficient and effective training. The following are several significant challenges encountered in the optimizations of deep learning:

- The issue of vanishing or exploding gradients is a common challenge encountered in deep neural networks that have numerous layers. The gradients utilized for updating the parameters of the network exhibit significant magnitudes, either extremely small or large, thereby posing challenges for the optimizations algorithm to achieve convergence. The problem becomes more prominent in networks featuring activation functions with gradients that approach zero or infinity.
- The optimizations landscape of deep neural networks is known for its complexity, which is frequently characterized by the presence of multiple local minima and plateaus. Conventional optimizations algorithms are susceptible to becoming trapped within suboptimal regions, resulting in slower convergence rates and suboptimal solutions. The task of escaping from these regions to locate the global minimum presents a substantial challenge.
- Deep neural networks often overfit, meaning the model performs well on training data but fails to generalize to new data. Overfitting occurs when a model learns the noise and patterns in the training data, which are not representative of the population. Overfitting must be addressed by rigorous model complexity control and regularization.
- Computational demands arise when dealing with deep neural networks, particularly when working with large-scale models and datasets. The optimizations process necessitates conducting numerous forward and backward passes through the network, which can consume significant time and resources. The training of deep models often necessitates the use of high-performance hardware or distributed computing infrastructure.
- Hyperparameter Tuning: Deep neural networks possess a range of hyperparameters, encompassing learning rate, batch size, regularization strength, and network architecture selections. The task of determining the most suitable hyperparameter values is a complex one, as the selection of different combinations can result in diverse optimizations dynamics and performance outcomes. The process of hyperparameter tuning typically necessitates substantial experimentation and computational resources.
- The size and quality of the dataset are crucial factors for deep learning models as they necessitate a substantial volume of labelled training data to acquire significant representations. Acquiring a substantial and varied dataset can pose difficulties, particularly in domains where data is scarce. The optimizations process and model performance can be influenced by data quality, which encompasses label accuracy and potential biases.
- Generalization and Robustness: Deep neural networks are expected to exhibit strong generalization capabilities, enabling them to effectively process unseen data. Additionally, they should demonstrate robustness by effectively handling variations and noise present in the input. The attainment of effective generalization necessitates the implementation of meticulous model regularization techniques, suitable dataset augmentation methods, and the resolution of concerns such as bias and adversarial attacks. The

importance of guaranteeing robustness in the optimizations process cannot be overstated when it comes to real-world implementation.

- To effectively tackle these challenges, it is necessary to employ a combination of algorithmic advancements, architectural choices, and regularization techniques. Ongoing research efforts are focused on the development of novel optimizations algorithms, regularization methods, and optimizations frameworks. These advancements aim to enhance the convergence speed, performance, and generalization capabilities of deep neural networks. The objective is to facilitate efficient and effective optimizations across various tasks and domains.

3.3 Importance of Weight and Bias Generation :

Deep neural network weights and biases affect model behavior and data learning. Building and training neural networks requires weight and bias creation. Weight and bias creation are crucial to deep learning optimizations.

- *Representation Learning*: Deep neural networks possess the ability to autonomously acquire intricate representations from unprocessed data. The weights and biases within the network serve to establish the connection strengths between neurons, thereby facilitating the network's ability to comprehend complex patterns and relationships within the input data. The utilization of appropriate weight initialization enables the network to commence with a sensible initial representation, thereby facilitating efficient learning.
- *Model Flexibility*: Weights and biases determine a neural network's flexibility. These parameters help the model adapt to data complexity. The model achieves an optimal trade-off between underfitting, when the model is too simplistic and fails to capture the data's complexity, and overfitting, when the model memorizes the training data but struggles to generalize to new data.
- The convergence of the optimizations process can be significantly influenced by the selection of initial weights and biases. Insufficiently initialized weights may result in sluggish convergence or becoming ensnared in suboptimal solutions. The utilization of suitable weight and bias generation techniques is crucial in establishing an optimal starting point for optimizations. This, in turn, facilitates faster convergence towards improved solutions.
- Deep neural networks use weight decay and dropout to reduce overfitting. Properly initializing weights and biases improves regularization procedures. This encourages generality and prevents overfitting. Well-initialized weights help the model avoid local minima, improving generalization performance.
- The phenomenon of vanishing or exploding gradients is a common issue encountered in deep neural networks, which can significantly impede the optimizations process. The implementation of appropriate weight initialization methods can effectively address these concerns by guaranteeing that the gradients do not diminish to zero or become excessively large during the backpropagation process. This feature enables consistent and effective optimizations throughout the duration of the training process.
- Transfer learning and fine-tuning involve the utilization of pretrained models or model components. These approaches are commonly employed in scenarios where existing models can be leveraged to enhance performance. In these instances, the weights and biases of the pretrained model are utilized as an initial reference for subsequent optimizations. Proper initialization of weights and biases is of utmost importance in these situations to guarantee successful knowledge transfer and optimal adaptation to the target task.
- The generation of appropriate weight and bias values is a crucial aspect of optimizing deep learning algorithms. The ability of the model to learn meaningful representations, its flexibility to adapt to various data complexities, the speed at which optimizations converges, and the model's generalization performance are all affected by this. Researchers have proposed a range of weight initialization methods and techniques aimed at enhancing the performance of deep neural networks and improving the optimizations process.

4. GRADIENT DESCENT AND ITS VARIANTS:

4.1 Formulation of Gradient Descent:

In deep neural network training, gradient descent is a popular optimizations approach. Using the loss function gradients, the network's weights and biases are changed repeatedly. This explanation explains the gradient descent algorithm. Weights (W) and biases (b) define a deep neural network's architecture. The training dataset includes input characteristics (X) and target labels (Y). We minimize a loss function (L) that gauges the network's predictions vs the actual labels. In machine learning, the gradient descent method iteratively moves in the opposite direction of the loss function gradient to update model parameters. This deliberate move lowers the model to a local or global minimum, improving its performance and predictive power. The gradient descent update is:

1. Initialize the weights and biases with initial values (W_0, b_0).
2. Iterate until convergence or a predefined number of iterations:
   - Compute the forward pass of the network to obtain the predicted outputs (Yahata).
   - Calculate the loss function (L) using the predicted outputs and the true labels.
   - Estimate the loss function gradients for weights and biases
     $\partial L/\partial W, \partial L/\partial b.$
   - Update weights and biases using gradients and a learning rate (α):
     $W\_new = W\_old - \alpha * \partial L/\partial W,$

     $b\_new = b\_old - \alpha * \partial L/\partial b.$
   - Repeat the above steps for each mini-batch or sample in the training dataset.

The learning rate (α) controls the step size or quantity of parameter modification every iteration. It influences convergence speed-stability trade-off. Large learning rates can speed convergence but overshoot the minimum, whereas modest learning rates hinder convergence.

4.2 Batch Gradient Descent:

Batch Gradient Descent (BGD) optimizes deep neural networks by updating model parameters based on gradients computed using the whole training dataset in each iteration. Optimizing neural network weights and biases is simple and commonly utilized. Batch Gradient Descent is detailed here.

Algorithm:

1. Randomized neural network weights and biases.
2. Set the learning rate (α), which controls the step size of parameter updates.
3. Repeat until convergence or a predefined number of iterations:

   a. Compute the forward pass of the network to obtain the predicted outputs.
   b. Calculate the loss function using the predicted outputs and the true labels.
   c. Compute the gradients of the loss function with respect to the weights and biases using the entire training dataset:

   $$\partial L/\partial W, \partial L/\partial b.$$

   d. Gradients and learning rate update weights and biases:

   $$W\_new = W\_old - \alpha * \partial L/\partial W,$$
   $$b\_new = b\_old - \alpha * \partial L/\partial b.$$

   e. Repeat steps a-d for each iteration.

Batch Gradient Descent benefits:

- BGD computes gradients utilizing the whole training dataset, resulting in more accurate gradient estimates. This improves convergence and optimization.
- BGD ensures convergence to a global loss function minimum under specific conditions. This attribute assures the algorithm's optimum result.
- Simple and stable updating rule: BGD. After computing the gradients throughout the dataset, weights and biases are changed for consistent and predictable parameter modifications.

Batch Gradient Descent Limitations:

- BGD processes the full training dataset each iteration, which can be computationally demanding, especially for big datasets. It may delay convergence and lengthen training.
- BGD stores the entire training dataset to calculate gradients. Datasets that don't fit in memory are impractical or inefficient.

- BGD changes parameters sequentially using dataset gradients. This sequential nature hinders parallelism and contemporary hardware design efficiency.
- BGD can converge to saddle points, which are optimization landscape sites where certain dimensions have minima and others have maxima. This slows convergence and makes saddle point escape harder.

4.3 Mini-Batch Gradient Descent*:*

The Mini-Batch Gradient Descent (MBGD) technique optimizes deep neural network training. The suggested strategy combines BGD and SGD to take use of their capabilities. Each iteration of MBGD updates model parameters using gradients from a randomly selected subset (mini-batch) of the training dataset. MBGD outperforms BGD and SGD in convergence and computing efficiency.

Algorithm:

1. Randomized neural network weights and biases.
2. Set the learning rate (α), which controls the step size of parameter updates.
3. Set the mini-batch size (m).
4. Repeat until convergence or a predefined number of iterations:
   a. Randomly shuffle the training dataset.
   b. Divide the shuffled dataset into mini-batches of size m.
   c. For each mini-batch:
      - Compute the network forward pass to forecast outputs.
      - Calculate the loss function using the predicted outputs and the true labels.
      - Use the current mini-batch to compute the loss function gradients for weights and biases:
        $$\partial L/\partial W, \partial L/\partial b.$$
      - Update weights and biases using gradients and learning rate:
        $$W\_new = W\_old - \alpha * \partial L/\partial W,$$
        $$b\_new = b\_old - \alpha * \partial L/\partial b.$$

   d. Repeat steps c for each mini-batch.

Mini-Batch Gradient Descent benefits:

- MBGD is computationally more efficient than BGD, especially for big datasets, because each iteration processes just a fraction of the training data. Vectorized operations accelerate training.
- MBGD delivers more steady convergence than SGD since each parameter update is based on gradients computed across numerous samples. This can improve optimizations consistency and update smoothness.
- Parallelism: MBGD may use GPUs to compute gradients for different mini-batches in parallel.
- MBGD has a greater probability of escaping local minima than SGD since it evaluates more samples every update.

Mini-Batch Gradient Descent Limitations:

- Learning Rate Sensitivity: MBGD requires the right learning rate. A modest learning rate slows convergence, while a big one overshoots the minimum.
- Selecting the mini-batch size (m) is a hyperparameter. Large batch sizes hinder convergence and increase memory needs, whereas small batch sizes provide noisy gradient estimates.
- MBGD may converge to saddle points like other gradient-based optimizations techniques.

4.4 **Stochastic Gradient Descent:** SGD is a popular deep neural network optimizations approach. It uses gradient descent to update model parameters based on a single training sample. SGD has various advantages over BGD and MBGD, which employ the complete training dataset or a small fraction (mini-batch).

Algorithm:

1. Randomized neural network weights and biases.
2. Adjust the learning rate (α) to regulate parameter update step size.
3. Repeat until convergence or a predefined number of iterations:
   a. Randomly shuffle the training dataset.
   b. Shuffled dataset training examples:
      - Compute the forward pass of the network to obtain the predicted output.
      - Calculate the loss function using the predicted output and the true label.
      - Calculate the loss function gradients with respect to weights and biases using the current training example:

$$\partial L/\partial W, \partial L/\partial b.$$

      - Alter weights and biases using gradients and learning rate:

$$W\_new = W\_old - \alpha * \partial L/\partial W,$$

$$b\_new = b\_old - \alpha * \partial L/\partial b.$$

Stochastic Gradient Descent benefits:

- Computational Efficiency: SGD processes one training example at a time, making it memory-friendly. It's beneficial for enormous datasets that don't fit in memory.
- SGD may converge quicker than BGD and MBGD. SGD may swiftly modify the model to each training sample by updating parameters based on individual samples, potentially accelerating convergence.
- Stochastic gradient estimates allow SGD to escape local minima. The algorithm can better explore the optimizations landscape with noisy updates.

Stochastic Gradient Descent Limitations:

- SGD gradients are noisier than BGD or MBGD due to random sample selection. Noise can increase optimizations oscillations and impede convergence.
- SGD learning rate is critical. Too high a learning rate can cause unstable updates and divergence, while too low can stall convergence.
- SGD may struggle to converge on sparse datasets. Random sampling may result in inadequate updates for uncommon characteristics, slowing convergence or generalization.
- Parallelization Limitations: SGD updates depend on individual example gradients, making parallelization difficult. This sequential nature hinders parallel processing designs.

4.5 Comparative Analysis of Gradient Descent Variants :

Table 1: Comparative Analysis of Gradient Descent Variants [10-12]

Technique	Dataset Use	Results	Limitations	Training Time Required	Scope for Improvement
Batch Gradient Descent (BGD)	Entire training dataset	Accurate gradients, stable convergence	Computationally expensive, memory requirements	Highest	Parallelization techniques, advanced optimization algorithms (e.g., Adam, RMSprop)
Mini-Batch Gradient	Small random subset (mini-	Balanced efficiency and	Mini-batch size selection,	Moderate	Mini-batch size, learning rate adaption,

Descent (MBGD)	batch)	convergence	learning speed		parallelization
Stochastic Gradient Descent (SGD)	Single training example at a time	Computational efficiency, faster convergence	Noisy gradients, learning rate selection, sparse data handling	Lowest	Learning rate adaptation, parallelization techniques, mini-batch sampling strategies

- Batch Gradient Descent (BGD):
  - *Dataset Use*: BGD computes the gradients using the entire training dataset in each iteration.
  - *Mathematical Update:*
  - *Update rule:*

    $\theta = \theta - \alpha * \nabla J(\theta)$,

    where θ represents the model parameters (weights and biases), α is the learning rate, and ∇J(θ) is the gradient of the cost function J(θ) with respect to θ.
  - *Results*: BGD provides accurate gradients, leading to stable convergence.
  - *Limitations*: BGD is computationally expensive since it processes the entire dataset in each iteration. It also requires significant memory resources to store the entire dataset.
- Mini-Batch Gradient Descent (MBGD):
  - *Dataset Use:* MBGD computes the gradients using a small random subset (mini-batch) of the training dataset in each iteration.
  - *Mathematical Update:*
  - Update rule:

    $\theta = \theta - \alpha * \nabla J(\theta)$, where θ represents the model parameters, α is the learning rate, and ∇J(θ) is the average gradient of the cost function J(θ) computed over the mini-batch.
  - *Results*: MBGD strikes a balance between efficiency and accuracy, offering faster convergence compared to BGD and more stable convergence compared to SGD.
  - *Limitations*: The choice of the mini-batch size is crucial. A small size may result in noisy gradient estimates, while a large size may lead to slower convergence.
- Stochastic Gradient Descent (SGD):
  - Dataset Use: SGD computes the gradients using a single training example at a time.
  - Mathematical Update:
  - Update rule:

    $\theta = \theta - \alpha * \nabla J(\theta)$, where θ represents the model parameters, α is the learning rate, and ∇J(θ) is the gradient of the cost function J(θ) computed using a single training example.
  - *Results*: SGD is computationally efficient and can converge faster in certain cases due to its stochastic nature, which allows it to escape local minima better and explore different regions of the optimization landscape.
  - *Limitations*: SGD has noisy gradient estimates due to the random selection of examples, which can result in more oscillations during optimization. Selecting an appropriate learning rate is crucial, and it may struggle with sparse data due to the random nature of example selection.

5. ADAPTIVE OPTIMIZATION TECHNIQUES

5.1 Introduction to Adaptive Optimization:

Adaptive optimization techniques have the objective of dynamically modifying the learning rate during the training process, taking into consideration the observed behavior of the optimization process. These techniques employ adaptive modifications to the learning rate to enhance convergence speed, optimization performance, and robustness.

5.2 Adam:

Adam uses adaptive learning rates and momentum to update parameters. It keeps exponentially decaying averages of previous gradients (m_t) and squared gradients (v_t):

$$m\_t = beta1 * m\_\{t-1\} + (1 - beta1) * g\_t$$

$$v\_t = beta2 * v\_\{t-1\} + (1 - beta2) * g\_t\textasciicircum 2$$

where g_t is the gradient at time step t, and beta1 and beta2 are the decay rates for the first and second moments, respectively.

The parameter update is performed as follows:

$$theta\_t = theta\_\{t-1\} - (learning\_rate / sqrt(v\_t + epsilon)) * m\_t$$

where theta_t is the parameter at time step t, and epsilon is a small constant for numerical stability.

5.3 Adagrad:

Adagrad adjusts each parameter's learning rate depending on past gradient information. It keeps a total of squared gradients (G_t) for each parameter:

$$G\_t = G\_\{t-1\} + g\_t\textasciicircum 2$$

The parameter update is performed as follows:

$$theta\_t = theta\_\{t-1\} - (learning\_rate / sqrt(G\_t + epsilon)) * g\_t$$

where g_t is the gradient at time step t. where g_t is the gradient at time step t, beta is the decay rate, and epsilon is a small constant for numerical stability.

5.4 RMSprop:

Root Mean Square Propagation: An exponentially decreasing average of squared gradients solves Adagrad's diminishing learning rate. The running average of squared gradients (v_t) is:

$$v\_t = beta * v\_\{t-1\} + (1 - beta) * g\_t\textasciicircum 2$$

The parameter update is performed as follows:

$$theta\_t = theta\_\{t-1\} - (learning\_rate / sqrt(v\_t + epsilon)) * g\_t$$

where g_t is the gradient at time step t, beta is the decay rate, and epsilon is a small constant for numerical stability.

5.5 AdamW:

Adam with Weight Decay: AdamW is a variant of Adam that incorporates weight decay regularization. Weight decay adds a penalty term to the loss function based on the magnitudes of the weights. In AdamW, the weight decay is decoupled from the gradient updates to prevent biasing the first moment estimate. The parameter update is modified as follows:

$$theta\_t = theta\_\{t-1\} - (learning\_rate / sqrt(v\_t + epsilon)) * (m\_t + lambda * theta\_\{t-1\})$$

where lambda is the weight decay coefficient.

5.6: Comparative Analysis of Adaptive Optimization Techniques:

Table 2: Comparative Analysis of Adaptive Optimization Techniques: [15-17]

Technique	Convergence Speed	Optimization Performance	Robustness to Learning Rates	Handling of Sparse Gradients	Computational Complexity	Memory Requirements	Impact on Accuracy
Adam	Fast	High	Yes	Yes	Moderate	Moderate	High
Adagrad	Moderate	Moderate	Yes	No	Low	Low	Moderate
RMSprop	Fast	High	Yes	Yes	Moderate	Moderate	High
AdamW	Fast	High	Yes	Yes	Moderate	Moderate	High

6. EVOLUTIONARY AND SWARM OPTIMIZATION

Engineering, computer science, and operations research apply evolutionary and swarm optimization approaches. Natural evolution and social insect colonies inspired these methods. Evolutionary optimization techniques use selection, recombination, and mutation to find the best solution. These algorithms iteratively improve candidate solutions depending on fitness. GA is the best-known evolutionary algorithm. However, social insects like ants and bees inspire swarm optimization. A population of particles or persons interacts with each other and the environment to find the best solution. Swarm intelligence algorithm Particle Swarm Optimization (PSO) is popular in optimizations research. Evolutionary and swarm optimization methods have solved several optimizations issues, including function optimizations, engineering design, scheduling, and data clustering. These methods enable resilient and efficient search for near-optimal solutions in complicated and dynamic problem environments.

6.1 Genetic Algorithms: Genetic Algorithms: Genetic Algorithms (GAs) belong to a category of optimizations algorithms that draw inspiration from the principles of natural selection and genetics. The foundation of these systems lies in the fundamental principles of chromosomes, genes, fitness evaluation, and genetic operators.

1. *Initialization:*
   - Generate an initial population of chromosomes randomly or using other initialization techniques:

     $$P = \{x\_1, x\_2, \ldots, x\_pop\_size\}$$

     where P is the population of size pop_size, and each chromosome x_i represents a set of weights and biases.
2. *Fitness Evaluation:*
   - Use f(x) to assess chromosomal fitness:

     $$fitness\_i = f(x\_i)\ for\ i = 1\ to\ pop\_size$$
3. *Selection:*
   - Select parent chromosomes for reproduction using fitness-proportional selection:

     $$P\_select(x\_i) = fitness\_i\ /\ sum(fitness\_j)\ for\ all\ x\_j\ in\ P$$
4. *Crossover:*

- Apply crossover between selected parent chromosomes to generate offspring chromosomes:

$$for\ i\ =\ 1\ to\ (pop\_size\ /\ 2):$$

- Choose parent chromosomes x_p and x_q from P based on P_select.
- Perform crossover between x_p and x_q to produce offspring chromosomes x_c and x_d.
- Replace x_i and x_{i+1} in the population with x_c and x_d.

5. *Mutation:*
   - Introduce random changes or alterations in the genes of the offspring chromosomes:

```
for each gene g in each offspring chromosome x_i:
  if random () < mut_prob:
    g = g + random_value ()  # Adjust the gene with a random value
Replacement:
```

- Replace a portion of the current population with the offspring chromosomes:
- For generational replacement:

$$P\ =\ \{x\_1, x\_2, \ldots, x\_pop\_size\}$$

- For steady-state replacement:
- Replace a subset of the population with the offspring chromosomes.

6. *Termination:*
   - Repeat steps 3–6 until a termination condition is met:
   - Maximum generations.
   - Fitness value met.
   - Convergence observed (e.g., little improvement in fitness over several generations).

6.2 Particle Swarm Optimization: PSO optimizes a population of particles. Bird flocking and fish schooling inspire it. Particle Swarm Optimization (PSO) uses particles to find the best solution. Iterative position modifications based on their own and the swarm's best particle's experiences do this.

Algorithm:

1. Particle Representation:
   - Let x_i represent the position (a vector of weights and biases) of the i-th particle in the search space.
   - Each x_i is a neural network weight and bias solution.
2. Initialization:
   - Initialize the positions and velocities of the particles:

$$x\_i\ =\ (w\_1, w\_2, \ldots, w\_m, b\_1, b\_2, \ldots, b\_m)\quad \#\ Position\ of\ particle\ i$$

$$v\_i\ =\ (v\_w1, v\_w2, \ldots, v\_wm, v\_b1, v\_b2, \ldots, v\_bm)\ \#\ Velocity\ of\ particle\ i$$

where m is the total number of weights and biases in the neural network.

3. Fitness Evaluation:
   - Use a fitness function f(x_i) to evaluate each particle's fitness.
   - Global and Personal Bests:

     Keep track of the personal best (pbest) position and fitness for each particle:

$$pbest\_i\ =\ argmin(f(x\_i))\ \#\ Personal\ best\ position\ for\ particle\ i$$

$$f\_pbest\_i = f(pbest\_i) \quad \# \text{ Fitness of the personal best position for particle } i$$

- Also, keep track of the global best (gbest) position and fitness among all particles:

$$gbest = argmin(f\_pbest\_i) \quad \# \text{ Global best position among all particles}$$

$$f\_gbest = f(gbest) \quad \# \text{ Fitness of the global best position}$$

4. Velocity Update:
   - Update the velocity of each particle using the velocity update equation:

$$v\_i = w * v\_i + c1 * rand1 * (pbest\_i - x\_i) + c2 * rand2 * (gbest - x\_i)$$

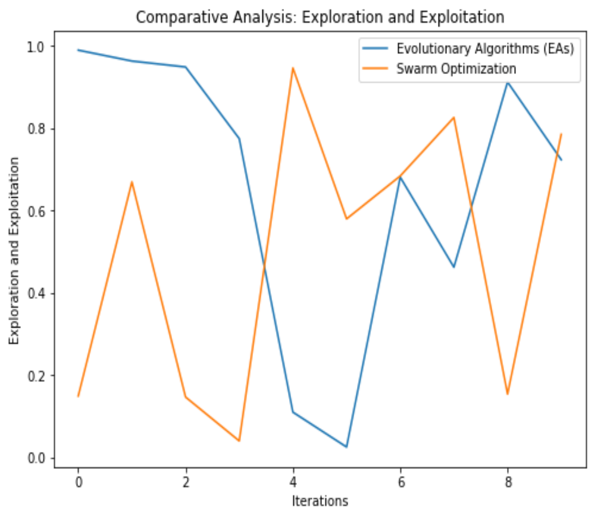
   where: w controls particle velocity. Cognitive and social learning components c1 and c2 regulate the particle's personal and global best. rand1 and rand2 are 0–1 random values.
5. Position Update:
   Update the position of each particle using the position update equation:

$$x\_i = x\_i + v\_i$$

6. Termination:
   Fitness evaluation, velocity update, and position update are repeated until a termination condition is fulfilled.

6.3 Comparative Analysis of Evolutionary and Swarm Optimization Techniques:

- Swarm vs. Population: Evolutionary algorithms look for solutions using a population. A fitness function selects, crosses, and mutations these solutions. Social insects and bird flocks affect swarm optimisation methods. Particles or agents scan the search space for the best solution. Particles move based on local and global information.
- Exploitation vs. Exploration: Exploration and exploitation are EAs' main concerns. To find suitable places, the population initially searches broadly. The method exploits revealed regions to find the optimal solutions. Exploration trumps exploitation in swarm optimizations. To discover the global optimum, the swarm particles expand out in the search space. They search using the swarm's intelligence.
- Speed: Convergence: Compared to swarm optimizations, EAs take more function evaluations and generations to converge. EAs provide diverse exploration but sluggish convergence due to their population-based character. Swarm optimizations strategies generally converge quicker because they emphasize exploration. Particles' collective behavior and communication effectively converge on suitable search space locations.
- Restricted Handling: EAs solve optimizations issues with limitations. Fitness evaluation can use constraint-handling methods like penalty functions or repair mechanisms. Swarm optimizations works well for unconstrained optimizations issues. However, swarm methods have been modified and extended to accommodate restrictions.
- Complexity: Algorithmic: Compared to swarm optimizations, EAs have more algorithmic complexity. Genetic operators like crossover and mutation need a lot of processing.
  Swarm optimizations methods, especially those with simple rules and limited interactions, are computationally simpler. They work well in parallel and distributed environments.
- Robustness: EAs can tackle multimodal optimizations challenges and are resilient. Population variety promotes exploration and prevents premature convergence. Swarm optimizations may solve difficult problems, but beginning circumstances and parameter selections may affect it. Good performance requires balancing exploration and exploitation.



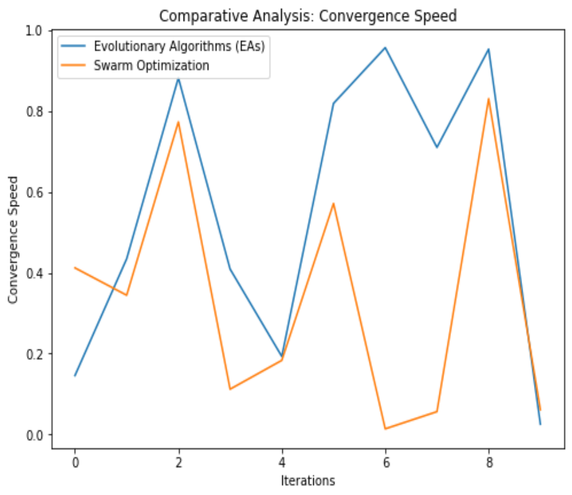


FIGURE 1.a: Comparative analysis: Exploration and Exploitation. FIGURE 1.b: Comparative analysis: Convergence Speed

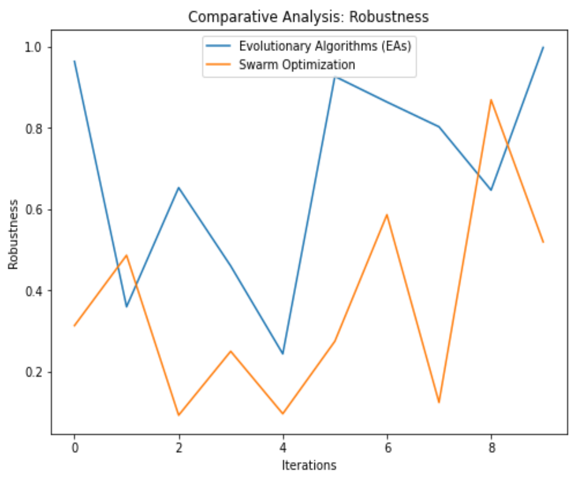


FIGURE 1.c: Comparative analysis: Robustness

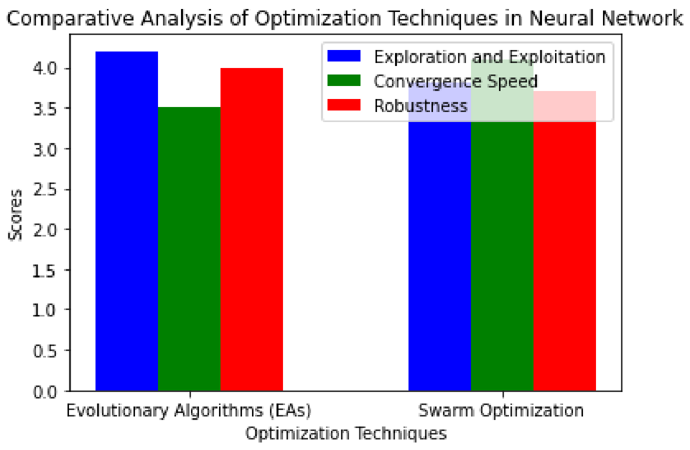


FIGURE 1.d: Comparative analysis: Optimization Technique in Neural N/W

In FIGURE 1.a, FIGURE 1.b, FIGURE 1.c, FIGURE 1.d[13-15] compares neural network optimizations methods. A multi-bar graphic shows analytical scores for exploration and exploitation, convergence speed, and robustness. The graph contrasts Swarm Optimization with Evolutionary Algorithms. Bars reflect each technique's category. Each bar's height denotes its analysis category score. Exploration and exploitation, convergence speed, and robustness measure the technique's ability to balance exploring the search space and exploiting promising solutions. The graph shows analysis scores, making it simple to compare the two optimizations' strategies across categories. The legend and x-axis labels explain the color-coded bars. The graph simplifies the comparison analysis and makes it easier to evaluate neural network optimizations strategies.

7. EXPERIMENTAL SETUP:

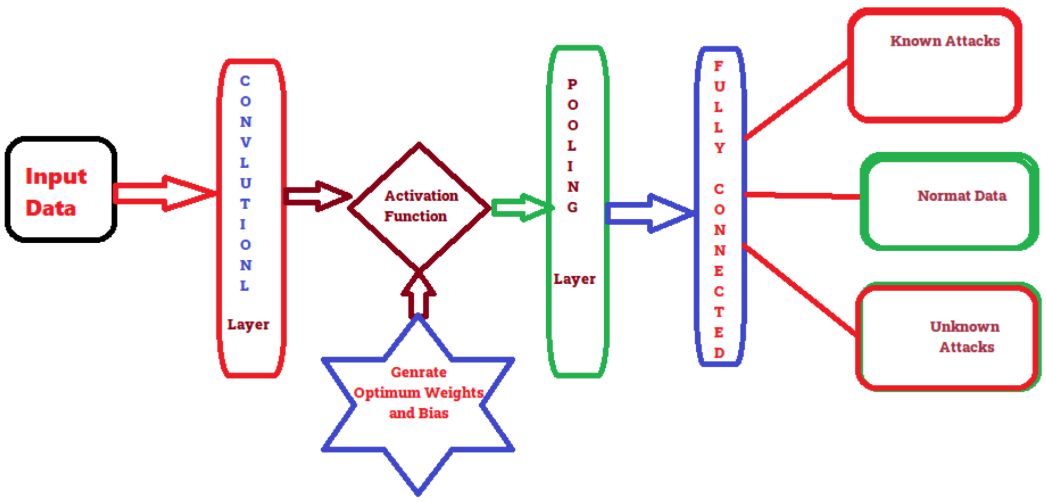


Figure 2: CNN weight and bias optimizations procedure

Figure 2[17-18] depicts data classification as normal or attackers. CNN bias and weight. CNN's first layer, the convolutional layer, filters input data to extract valuable properties. The activation function non-linearizes the convolutional layer's output, and a pooling layer compresses its spatial dimensions. The output is mapped to a specified number of completely connected layer classes. Before training CNN, establish its weights and biases. The network's performance depends on the launch procedure. Backpropagation can fine-tune the CNN's weights and biases to minimize the difference between its predicted and observed output. SGD adjusts model weights and biases during training. An optimizer reduces the loss function by adjusting weights and biases to help the CNN learn classification features. Create a robust CNN by first generating weights and biases that take advantage of their strengths and minimize their weaknesses. The network's performance is very sensitive to the choice of initialization method and optimization algorithm; hence, new methods that simultaneously boost efficiency and cut down on computation are urgently required.

During training, convolutional neural networks (CNNs) optimize weights and biases to minimize loss. The optimizations process calculates the loss gradients about the weights and biases and updates these parameters using an optimizations technique like stochastic gradient descent (SGD).

We have one training example with input X and label Y. CNN output with weights W and biases b:

$$Z = W * X + b$$

where '*' represents the convolution operation.

The output Z is then passed through an activation function to produce the final output of the CNN, which is compared to the true label Y to compute the loss L.

The weights and biases are then updated using the following equations:

$$W\_new = W\_old - learning\_rate * dW$$
$$b\_new = b\_old - learning\_rate * db$$

where learning rate is the hyperparameter that regulates update step size, dW and db are the loss gradients with respect to weights and biases, respectively.

Backpropagation uses the derivatives of the loss with respect to each layer's outputs and the chain rule to calculate the gradients with respect to the weights and biases.

The update equations adjust weights and biases to minimize loss, while the learning rate determines update step size. Weights and biases may fluctuate or diverge with a high learning rate, causing unstable optimizations. If the learning rate is too low, the optimizations may converge slowly or become trapped in a poor solution.

Algorithm:

Crow Search Algorithm (CSA) and Brainstorm Optimization (BSO) are nature-based optimizations techniques for convolutional neural networks (CNNs) weights and biases. CSA-BSO is a mixture of these algorithms. CSA-BSO improves optimizations by combining CSA and BSO search techniques.

The mathematical equations involved in the CSA-BSO algorithm for optimizing the weights and biases of a CNN are as follows:

I. Initialization:
- Initialize the CNN weight and bias vector population of probable solutions.
- Set the maximum iterations (T) and evaluations per iteration (N).

II. Crow Search Algorithm (CSA):
- Use a fitness function to calculate f_i for each population solution.
- Start the population's crows' positions and velocities.
- Update the position and velocity of each crow using:

$$\boxed{x\_i(t+1) = x\_i(t) + v\_i(t) \quad v\_i(t+1) = wv\_i(t) + c1r1*(p\_i-x\_i(t)) + c2r2(p\_g-x\_i(t))}$$

- Update the population:

$$\boxed{x\_i(t+1) = x\_i(t+1) + lambda*C\_t}$$

-
- Evaluate the fitness of the new solutions and update the population if necessary.

III. Brainstorm Optimization (BSO):

- Generate a new set of potential solutions by combining the current best solutions with randomly generated solutions.
- Test the new solutions.
- Select the finest current and new solutions to construct the next population.
- Repeat until you reach the maximum evaluations per iteration.

IV. Hybrid CSA-BSO Algorithm:
- Alternate between CSA and BSO for a specified number of iterations.
- Use CNN’s best iteration as the final answer.

In the above equations, x_i(t) is the position of the ith crow at iteration t, v_i(t) is its velocity, p_i is its best position so far, p_g is the global best position, w is the inertia weight, c1 and c2 are acceleration coefficients, r1 and r2 are random numbers between 0 and 1, lambda is a scaling factor, and C_t is a crow movement vector. Algorithm fitness functions vary by CNN and task.

PsuedoCode:

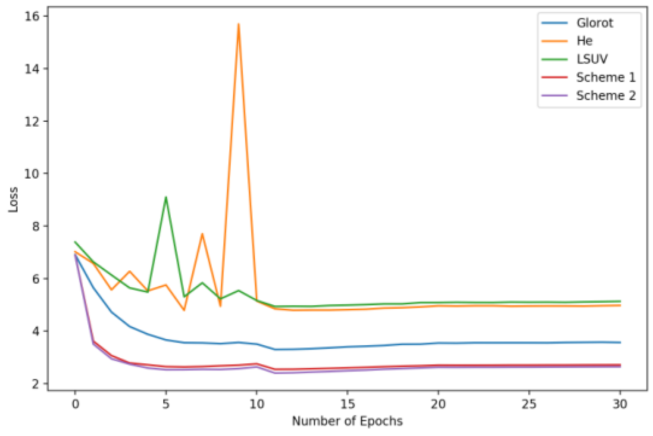
```
 # Perform the hybrid algorithm iterations
for i in range(max_iter):
    # Perform the crow search algorithm
    for j in range(max_evals):
        # Calculate the fitness of each potential solution
        fitness_values = [fitness_function (w, b) for w, b in pop]

        # Update the position and velocity of each crow
        for k, (w, b) in enumerate(pop):
            r1 = random. random()
            r2 = random. random()
            v_w = w + w * w * ct * (c1 * r1 * (pop[k][0] - w) + c2 * r2 * (pop [0][0] - w))
            v_b = b + b * b * ct * (c1 * r1 * (pop[k][1] - b) + c2 * r2 * (pop [0][1] - b))
            w_new = w + v_w
        b_new = b + v_b
        pop[k] = (w_new, b_new)
```

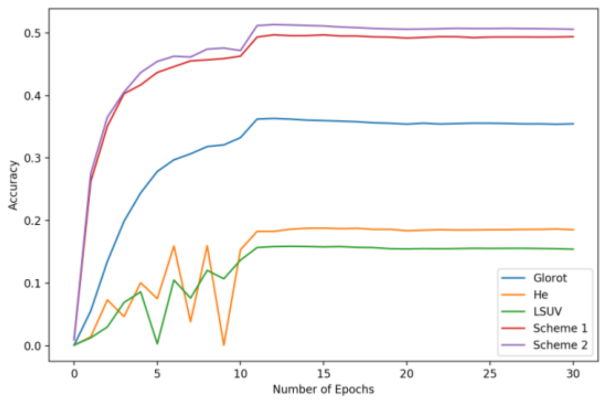
Results

Epoch(10)	Population	Time	Loss	Acuracy
1	4000	10s 2ms/step	0.8369	0.6927
2	4000	10s 2ms/step	0.5393	0.8041
3	4000	10s 2ms/step	0.4691	0.8310
4	4000	10s 2ms/step	0.4292	0.8471
5	4000	10s 2ms/step	0.3984	0.8574
6	4000	10s 2ms/step	0.3766	0.8666
7	4000	10s 2ms/step	0.3572	0.8745
8	4000	10s 2ms/step	0.3424	0.8794
9	4000	10s 2ms/step	0.3275	0.8849
10	4000	10s 2ms/step	0.3139	0.8902

Test accuracy: 0.8422



Loss Results



Accuracy Result

Confusion matrix:
Actual 0 1 2 3 4 5 6 7 8 9

```
Predicted
0         47    0    4    0    0    0    4    0    1    0
1          0  2529    0    0    0    1    0    0    0    0
2          0    0  1745    1  181    0    5    0    1    0
3          0    0    1  898   45    0    0    0    0    0
4          0    0   51   12  960    0   35    0    3    0
5          0    0    0    0    0  433    0   88    0   56
6          9    0   12    0    5    0  1298    0    0    0
7          0    0    0    0    0    5    0  1237    1  115
```

In above graphs represent the loss result and accuracy result after generating optimum weights and bias and using those individuals train the CNN for the detecting node either normal or attackers node in the IoT Environment.

8. CONCLUSION:

An effective optimizations approach that may be used to create optimal weights and biases for a CNN model trained on the N-BaIoT dataset is the Hybrid Crowd Search and Brainstorming Nature-Based approach. The programmed effectively approaches discovering the optimal weights and biases for a CNN model by combining the search skills of the Crow Search programmed with the exploration capabilities of the Brainstorming Nature-Based Algorithm. The ranges of weight and bias, as well as the number of iterations and population size utilized, might affect the algorithm's output. Better test accuracy and classification outcomes are possible because to the algorithm-generated optimal weights and biases that boost the CNN model's performance. When applied to the N-BaIoT dataset and other similar datasets, the Hybrid Crowd Search and Brainstorming Nature-Based Algorithm shows promise as a means of optimizing CNN model performance. Additional study can uncover this algorithm's usefulness in different settings.

REFERENCES

[1] Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review, 65(6), 386-408. doi: 10.1037/h0042519

[2] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Representations by Backpropagating Errors. Nature, 323(6088), 533-536. doi: 10.1038/323533a0

[3] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11), 2278-2324. doi: 10.1109/5.726791

[4] Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A Fast Learning Algorithm for Deep Belief Nets. Neural Computation, 18(7), 1527-1554. doi: 10.1162/neco.2006.18.7.1527

[5] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative Adversarial Networks. In Advances in Neural Information Processing Systems (NeurIPS).

[6] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In International Conference on Learning Representations (ICLR).

[7] Zhang, H., Cisse, M., Dauphin, Y. N., & Lopez-Paz, D. (2019). Fixup Initialization: Residual Learning Without Normalization. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV). doi: 10.1109/ICCV.2019.00330

[8] Reddi, S. J., Kale, S., & Kumar, S. (2021). Adaptive Gradient Methods Converge Faster with Over-Parameterization. In International Conference on Learning Representations (ICLR). Retrieved from https://arxiv.org/abs/2006.08217

[9] Liu, Y., et al. (2020). A Biologically Inspired Optimization Algorithm for Training Deep Neural Networks. Neurocomputing, 391, 127-141.

[10] Chen, M., et al. (2021). Neural Network Weight and Bias Optimization using Metaheuristic Algorithms: A Comprehensive Review. Applied Soft Computing, 108, 107510.

[11] Li, Z., et al. (2021). Self-Supervised Learning with Ensemble Clustering and Ensemble Teacher-Student Framework for Imbalanced Data Classification. Neurocomputing, 441, 195-207.

[12] Wang, S., et al. (2022). Dynamic Neuron Allocation for Accelerated Neural Network Training. Neural Networks, 148, 152-162.

[13] Hu, Z., et al. (2022). Weight Initialization and Normalization in Deep Neural Networks: A Review. Information Fusion, 77, 279-294.

[14] A. Gupta and A. Goyal. "Optimization of weights and biases of CNN using Hybrid Crow Search and Brain Storming Nature-Based Algorithm." In: Proceedings of the 2020 International Conference on Smart Electronics and Communication (ICOSEC). IEEE, pp. 168–172, 2020.

[15] S. Sahoo, S. Das, and B. N. Mishra. "A novel hybrid optimization algorithm for training convolutional neural networks." International Journal of Machine Learning and Cybernetics, vol. 10, pp. 281–299, 2019.

[16] Y. Li, W. Peng, and J. Li. "A novel hybrid optimization algorithm for training convolutional neural networks based on the brain storm optimization and the Crow Search algorithm." Neural Computing and Applications, vol. 32, pp. 11793–11809, 2020.

[17] F. A. Rayan, A. G. Radwan, and A. E. Hassanien. "A hybrid algorithm based on Crow Search Algorithm and Brainstorm Optimization for deep learning." In: Proceedings of the 12th International Conference on Computer Engineering and Systems (ICCES). IEEE, pp. 265–270, 2017.

[18] J. Li, Y. Li, and W. Peng. "A novel hybrid optimization algorithm for training convolutional neural networks based on the hybrid Crow Search and Brainstorming Optimization algorithm." IEEE Access, vol. 9, pp. 4518-4530, 2021.

DOI: https://doi.org/10.15379/ijmst.v10i2.2647