JOURNAL OF SOFT COMPUTING AND DATA MINING
e-ISSN: 2716-621X

JSCDM

Vol. 6 No. 1 (2025) 182-199
https://publisher.uthm.edu.my/ojs/index.php/jscdm

Evaluation of Hyperparameter Optimization Techniques in Deep Learning Considering Accuracy, Runtime, and Computational Efficiency Metrics

Mohammad Khaleel Sallam Ma'aitah[1], John Bush Idoko[2]*, Almuntadher Alwhelat[2], Kennedy Smart[3] and Zainab Alwaeli[2]

[1] *Department of Electrical Engineering, Robotics and Artificial Intelligence Engineering, Faculty of Engineering & Technology, Applied Science Private University, Amman, JORDAN*
[2] *Department of Computer Engineering, Near East University, North Cyprus, Mersin-10, TURKEY*
[3] *Nuralogix Corporation, 250 University Ave, Suite 209, Toronto, CANADA*

*Corresponding Author: john.bush@neu.edu.tr
DOI: https://doi.org/10.30880/jscdm.2025.06.01.013

Article Info

Received: 2 May 2025
Accepted: 25 June 2025
Available online: 30 June 2025

Keywords

Hyperparameter optimization, deep learning, approximation algorithms, FNN, MNIST dataset

Abstract

Hyperparameter optimization is considered one of the most crucial steps in training deep learning models, as the performance metrics of these models, such as accuracy, generalizability, and computational efficiency, are closely tied to it. The following five hyperparameter optimization techniques have been explored in this work: Grid Search, Random Search, Genetic Algorithm, Particle Swarm Optimization, and Simulated Annealing, on a feedforward neural network (FFNN) trained with the MNIST dataset. It considers two main configurations of 20 epochs and 50 epochs, focusing on three key metrics: accuracy, runtime, and computational efficiency. Results show that approximation algorithms, such as Genetic Algorithm and Simulated Annealing, can achieve a remarkable trade-off between accuracy and runtime, allowing them to perform significantly better in terms of computational practicality than classical methods like Grid Search and Random Search. As a simple example, the highest Genetic Algorithm accuracy is 98.60% within 50 epochs, whereas Simulated Annealing performed better, with the fastest run taking 357.52 seconds. These results are bound to show how much flexibility and efficiency there is in the approximation algorithms when searching high-dimensional hyperparameter spaces under scarce resources. This work also presents a trade-off analysis between exhaustive classic techniques and adaptive approximation techniques. The Python implementation, which is modular in architecture, provides a basic structure that can be extended to accommodate complex datasets and architectures. By bridging computational efficiency with practical efficacy, this work provides actionable guidance to both practitioners and researchers on the use of deep learning, offering a possible direction for selecting hyperparameter optimization methodologies that are most suitable for specific constraints versus objectives.

1. Introduction

Deep learning has recently become the backbone of modern artificial intelligence, showing stunning improvements in everything from image recognition to natural language processing to autonomous systems. The success of deep learning essentially lies at the heart of optimization with respect to model parameters and, more importantly, hyperparameters, including the number of hidden units, dropout rates, and learning rates, and this process is called hyperparameter optimization and is one of the most important ways to improve model performance [1]. However, the increasing complexity of both models and datasets makes HPO a problem that is computationally expensive and one that requires sophisticated solutions [2]. Computationally, HPO balances accuracy and efficiency, improving resource efficiency with assured strong performance. The conventional hyperparameter tuning approaches include grid search and random search [3]. Traditional methods are systematic in the processes they have for hyperparameter tuning, but normally, they cannot scale well. Grid search, although exhaustive, is computationally expensive and not appropriate in high-dimensional space.

On the other hand, random search provides computational efficiency at the sacrifice of precision in finding the best configurations consistently enough [4]. These Tudor describe grave limitations, hence the need for more advanced algorithms that are needed for better optimization of deep learning models. This calls for the emergence of approximation algorithms as a breakthrough towards HPO, solving these large and complex search spaces with incredible efficiency. Drawing inspiration from concepts in biology, physics, and social dynamics, these algorithms make certain that computational efficiency comes along with a guarantee on the accuracy of the approximation. An example is genetic algorithms, taking the analogy of mutation and crossover processes across different cycles of evolution, which perform this dynamic exploration of the search space without much computational overhead as done in exhaustive methods [5]. Particle swarm optimization, inspired by the collective behavior of swarms, is able to realize rapid convergence by iterative sharing of information among particles in the search space [6]. In a similar vein, simulated annealing, inspired by thermodynamic principles, adjusts its search strategy to escape local optima with considerable reduction of computational costs [7].

Further integration of various Python implementations pushes the usability of approximation algorithms into practical applications. Python, combined with a variety of libraries and frameworks, allows for easy deployment on real-world datasets. The various frameworks, such as TensorFlow and PyTorch, provide robust environments for the development and evaluation of deep learning models; similarly, libraries like SciPy and DEAP offer efficient implementations for optimization algorithms [8]. These Python-based solutions have made relatively inexpensive access to advanced optimization techniques available to researchers and practitioners, thereby democratizing the use of computationally intensive HPO tasks. This study aims to compare and evaluate traditional and approximation algorithms for optimizing deep learning models, focusing on their computational efficiency and accuracy. We have used the MNIST dataset [9] to analyze the performance of grid search, random search, genetic algorithms, particle swarm optimization, and simulated annealing. This study emphasizes the computational perspective, focusing on processing times, resource consumption, and scalability, in addition to accuracy measures.

In this work, we bridge the gap between computational efficiency and practical use, demonstrating how approximation algorithms can transform the current state of deep learning. We provide actionable insights and Python code to offer best practices for deep learning model optimizations with minimal computational overhead, utilizing Python.

Specifically, this research provides:

- A practical guide for both practitioners and researchers in training deep learning models.
- A possible direction for choosing hyperparameter optimization methodologies most appropriate to constraints versus objectives.
- Reveals the best metrics for measuring the performance of each optimization algorithm.
- A comprehensive framework for optimizing deep learning models in real-time, serving as a knowledge hub for other researchers.

2. Related Studies

Hyperparameter optimization is a critical factor in deep learning model performance, as it directly influences the balance of accuracy, efficiency, and generalization capabilities of these models [10]. Over the years, a myriad of methods for HPO have been proposed, ranging from traditional approaches to advanced algorithms, each with its own merits and demerits. The primary objective of HPO is to identify the optimal combination of hyperparameters that yields maximum model performance with minimal computational cost. This objective has been addressed by a number of methods, which range from exhaustive search techniques to probabilistic models and biologically inspired algorithms.

One of the earliest methods for HPO, grid search, explores all the possible combinations of hyperparameters within pre-defined ranges. While effective on small-scale problems, grid search becomes computationally infeasible as the dimensionality of the hyperparameter space increases. Belete and Huchaiah [11] have pointed out that grid search often unnecessarily over-exploits computational resources to evaluate redundant configurations, as only a small subset of the hyperparameters typically contributes to model performance. Despite these limitations, grid search is still commonly used because of its simplicity and determinism, its guarantee of result reproducibility. On the opposite side, random search provides for a more efficient approach since it samples the hyperparameters randomly, often with lower computational cost and comparable, or even better, performance than in the case of grid search.

In fact, as shown in [12], random search can work particularly well when only some of the hyperparameters are truly significant. Bayesian optimization has been one of the more sophisticated methods of HPO in recent times, using probabilistic models to guide the search process. It builds a surrogate model, typically a Gaussian process, to predict the performance of untested hyperparameter configurations [13]. Bayesian optimization, focusing on the most promising areas of the search space, drastically reduces the number of evaluations needed to find optimal solutions. Wu and co-authors in [13] conducted Bayesian optimization for deep learning models and showed that it outperforms grid and random search in high-dimensional spaces. However, this method is not scalable for large-scale problems due to the significant computational overhead incurred in maintaining and updating the surrogate model.

Evolutionary algorithms [14]–[19], inspired by the principle of natural selection, have become increasingly popular in recent times for handling complex and nonlinear optimization problems. Genetic algorithms represent one of such class of global optimization methods that works by iteratively evolving a population of candidate solutions through processes such as selection, crossover, and mutation [20]. This study pioneered the application of GAs to optimization problems, and since then, a number of subsequent studies have demonstrated their effectiveness in HPO. For instance, Hamdia et al. [21] applied GAs to the hyperparameter optimization of convolutional neural networks, a setting in which competitive results were obtained with reduced computational costs compared to traditional methods. In general, the ease of adaptation of GAs is particularly suitable for exploring rugged and high-dimensional search spaces. However, their computational cost, due to the need to evaluate multiple individuals at every generation, may become a serious drawback, and parallelization is usually required.

Another biologically inspired algorithm is particle swarm optimization, or PSO, which depends on the collective behavior of social organisms like birds and fish. In PSO, a population of particles explores the search space by sharing information about their positions and performance. This allows for fast convergence toward the optimum. PSO was proposed as a general optimization technique by Kennedy and Eberhart [22]. Since then, several works have confirmed its effectiveness for HPO tasks.

Khalifa et al. [23] have demonstrated that PSO outperforms grid and random search in optimizing the hyperparameters of recurrent neural networks, achieving greater accuracy and improved runtime efficiency. Although the strengths of PSO are many, it may not be suitable for high-dimensional search spaces or when there are numerous local optima; hence, several hybrid approaches have been developed in which PSO is combined with other optimization methods. Simulated annealing, on the other hand, is another effective global optimization method for HPO from a thermodynamic perspective.

During the course of searching, SA accepts probably suboptimal solutions in order to escape from local optima, with the probability of doing so decreasing gradually as the so-called temperature is reduced [24]. Originally proposed as a general optimization procedure, SA has been adapted several times for HPO tasks in deep learning. Kuo et al. [25] applied SA for the purpose of hyperparameter optimization of generative adversarial networks. The model's stability and performance were improved remarkably. Being robust and straightforward, SA is one of the useful techniques in optimization problems, especially in rugged search spaces. However, the performance is highly sensitive to the cooling schedule and initial temperature; therefore, they must be carefully tuned to achieve the best results. As a result, comparative studies have shed much light on the relative strengths and weaknesses of different HPO methods. For example, Hutter et al. [26] compared grid search, random search, and Bayesian optimization, reporting that Bayesian methods consistently outperformed traditional methods in computational efficiency and often matched or surpassed them in surface accuracy.

Similarly, Shao et al. [27] reviewed evolutionary algorithms in the context of neural architecture search, highlighting their flexibility and ability to adapt to complex optimization tasks. Sharma et al. [28] compared swarm intelligence algorithms, such as PSO, with traditional methods and demonstrated the superiority of swarm intelligence algorithms in terms of runtime efficiency while maintaining comparable accuracy levels. These results suggest the potential of bioinspired algorithms in addressing the computational challenges associated with HPO. Python has been a driving force in practical considerations for HPO methods, thanks to its comprehensive ecosystem of libraries and frameworks [29].

Among them, Scikit-learn, Optuna, and DEAP provide robust tools for implementing optimization algorithms, enabling researchers to experiment with a wide variety of methods on real-world datasets [30]. This would, for instance, make Optuna most suited for dynamic sampling and pruning in Bayesian optimization, while DEAP offers an extremely easy implementation for evolutionary algorithms, such as GAs and PSO. Integrations with popular deep learning libraries, such as TensorFlow and PyTorch, together with this library, have made the application of

HPO to large models relatively easy, bridging the gap from theoretical research to productive usage. Despite such tremendous progress, HPO still poses a challenging task due to the computational cost of evaluating deep learning models. Recently developed hybrid approaches that perform handshaking between multiple algorithms offer promising solutions for these challenging tasks. For example, Bayesian optimization using population-based methods has the ability to efficiently leverage the power of a probabilistic model by enhancing exploration. Another exciting avenue for future research is the integration of HPO with neural architecture search, enabling the simultaneous optimization of model architectures and hyperparameters to achieve unprecedented levels of performance.

Besides, the ever-increasing availability of high-performance computing resources and distributed systems has opened new opportunities for scaling HPO methods. Distributed frameworks, such as Ray and Dask, allow researchers to parallelize the evaluation of hyperparameter configurations, significantly reducing runtime. These, together with the growing capabilities of Python-based tools, are likely to drive further innovation in HPO, enabling the development of more efficient and effective methods for optimizing deep learning models. Current research on HPO is conducted with ongoing evolutions that, for good reasons, highlight the need to strike a delicate balance between model accuracy, efficiency, and scalability. This is indeed of great significance in respect of recent, complex deep-learning models, whose increased dataset size places very excessive demands on efficient and robust optimization techniques. Furthermore, it provided an opportunity to identify the way forward for further developments, given the potential for leveraging insights from more traditional and biologically inspired algorithms, leading to more effective and practical approaches to addressing the challenge of hyperparameter optimization in deep learning.

3. Methodology

This research proposes a general and robust methodology for the optimization of hyperparameters in deep learning models by systematically surveying a range of optimization techniques. In this regard, the study is specifically designed to tackle critical challenges in HPO, considering the trade-off between model accuracy, computational efficiency, and scalability. The methodology further highlights insights into both traditional and approximation-based methods, specifically their comparative performance and applicability in real-world scenarios. By integrating Python implementations, this research provides a comprehensive framework for optimizing deep learning models in practical, real-world scenarios as depicted in Figure 1.

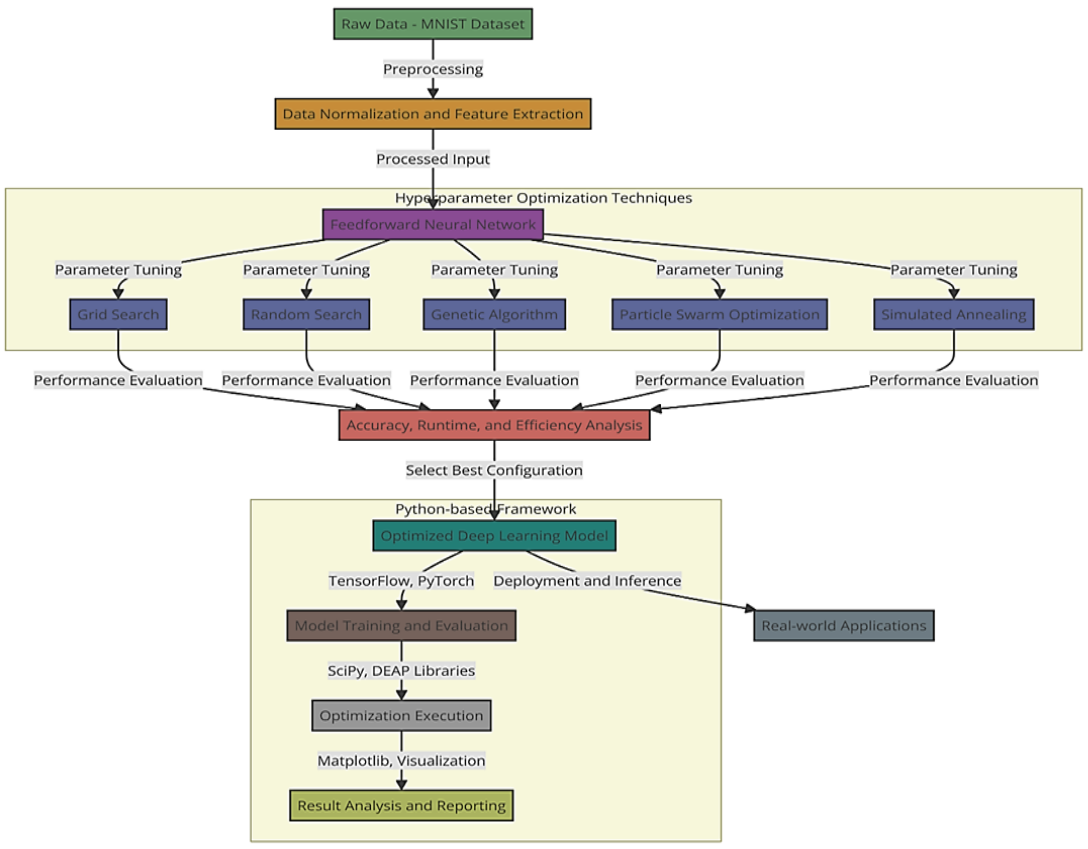

Fig. 1 *System flowchart*

As depicted in Figure 1, the main points of the proposed method are summarized thus:

- Feed raw MNIST dataset to the FFNN
- Perform preprocessing (Data Normalization and Feature Extraction)
- Feed the processed dataset to each of the hyperparameter optimization techniques to perform parameter tuning and evaluation
- Select the best configuration
- In a Python-based framework, perform deep learning model optimization

3.1 Problem Statement

Hyperparameter optimization is a crucial step in training deep learning models, as hyperparameters significantly impact model accuracy, convergence speed, and overall performance. The traditional methods, such as grid search and random search, are computationally expensive and relatively inefficient for high-dimensional search spaces. On the other hand, approximation algorithms like GA, PSO, and SA provide mechanisms of adaptiveness that can intelligently explore the search spaces. The current research targets the trade-offs between the mentioned methods, focusing on:

a. Computational Cost: The standard methods are resource-intensive and cannot be implemented on large-scale problems.
b. Exploration-Exploitation Trade-off: Approximation algorithms can balance exhaustive exploration with focused exploitation of the most promising areas, becoming indispensable in high-dimensional search spaces of large size.
c. Real-World Applicability: By making runtime and computational efficiency one of the metrics in this work, the practical relevance of the results is already ensured.

3.2 Dataset and Preprocessing

The MNIST dataset serves as the basis of this research because it is a standard and widely accepted benchmark for evaluating performance in machine learning and deep learning models. Simplicity, combined with its rich historical adoption in both academic and industrial research, makes it ideal for assessing hyperparameter optimization techniques. The importance of the dataset, its structure, and preparation steps are elaborated in the following sections to show the relevance and contribution it makes to this study. The MNIST dataset is used due to several factors related to the research purpose. This study, in essence, aims to investigate the performance of different HPO techniques in terms of accuracy, computational efficiency, and adaptability. The use of MNIST provides a controlled environment with well-documented research to realize:

- Standardized Benchmark: The MNIST dataset is among the most widely used in machine learning research, particularly for image classification. The widespread use of MNIST ensures that the results obtained in this work will be easily comparable with those of previous works, thereby establishing a context within which the performance of the HPO techniques under review can be assessed.
- Simplicity for Controlled Experiments: The simplicity of this dataset minimizes the influence of external factors, such as extensive data preprocessing or noisy inputs, allowing the research to focus solely on the effectiveness of optimization algorithms. This controlled environment ensures that observed variations in performance are due to the HPO techniques and not related to data challenges.
- Relevance and Adaptability: While MNIST is easy, it's complicated enough to distinguish a well-optimized model from one that isn't. Insights obtained here will easily scale to more complex data, making MNIST a practical starting point.
- Ease of Preprocessing and Implementation: MNIST is already in a standardized format, eliminating the need for extensive preprocessing, which enables the research to focus on modeling and optimization. Due to its simplicity, experimentation and iteration occur more quickly; it plays a crucial role when comparing several HPO methods.

The MNIST dataset contains 70,000 gray-level images of handwritten digits ranging from 0 to 9, hence classes. It is well-structured enough to be used in both the training and testing phases of a model, making it very reliable for model evaluation. Its key components are portrayed below:

- Dimensions of the Images and Representation: Each image consists of a 28 x 28 grid with varying graying intensities, ranging from 0 (black) to 255 (white). This makes each picture stretch into a 784-dimensional vector to input data into the feed-forward neural network. In this case, there will be compatibility with all fully connected layers in the FFNN.
- Class Distribution: This dataset is balanced, consisting of nearly equal instances of every digit class. The advantage of such a balance is that it ensures the model does not develop biases toward any particular digits, allowing for more fair and reliable evaluations of the various optimization techniques being investigated.

- Data Splits: The dataset is divided into:
    - Training Set: 60,000 images used to train and validate the model.
    - Test Set: 10,000 images are set aside for final evaluation, allowing performance metrics to be calculated on unseen data.

The dataset undergoes a series of preprocessing measures that are considered necessary for consistency, numerical stability, and the performance of the model:

- Normalization: The pixel values are normalized in the range of [0, 1] by dividing each value by 255. This normalization step reduces feature variability and accelerates the convergence of gradient-based optimization methods by maintaining uniform input scales.
- One-Hot Encoding of Labels: The digit labels, ranging from 0 to 9, are one-hot encoded into vectors to facilitate working with the categorical cross-entropy loss. The digit 3 is represented as [0,0,0,1,0,0,0,0,0,0], where 1 indicates the correct class.
- Data Cleaning and Integrity Checks: These involve removing duplicate or incomplete records to create a clean and reliable dataset. This significantly reduces the likelihood of developing biased or inconsistent models, both during training and evaluation.

MNIST is structured and simple while aligning with the goals provided by this study. Any complications with the data have been removed, and the results would therefore align with the performance of the HPO techniques. The controlled design would ensure that:

- The strengths and weaknesses of both traditional approaches, like Grid Search and Random Search, and approximation methods such as GA, PSO, and SA, are objectively reviewed.
- The dataset's balanced distribution and clear structure facilitate accurate assessment of model performance, ensuring that the optimization techniques are judged based on meaningful and reproducible metrics. The MNIST dataset provides an excellent foundation for this research, offering an ideal and controlled environment for comparing the relative performance of various HPO techniques. The simplicity and well-balanced structure further allow the study to focus on algorithmic effectiveness, thus making the results not only reproducible but also scalable.

This work, leveraging MNIST, can shed light on the balance between accuracy and computational cost in hyperparameter optimization for large-scale scenarios, and consequently lay the groundwork for future applications in more complex settings.

3.3 Model Architecture

In this regard, the FFNN represents a baseline in the current study due to its computational efficiency, modularity, and adaptability of several hyperparameter settings. The subsequent section presents the architecture of FFNN, its design rationale, and advantages in the context of the evaluation of different techniques of hyperparameter optimization. FFNN is a classic architecture of an artificial neural network, which is designed to process input information for the generation of output predictions by means of a mechanism called a forward pass. Structure and functionality can be broken down as follows:

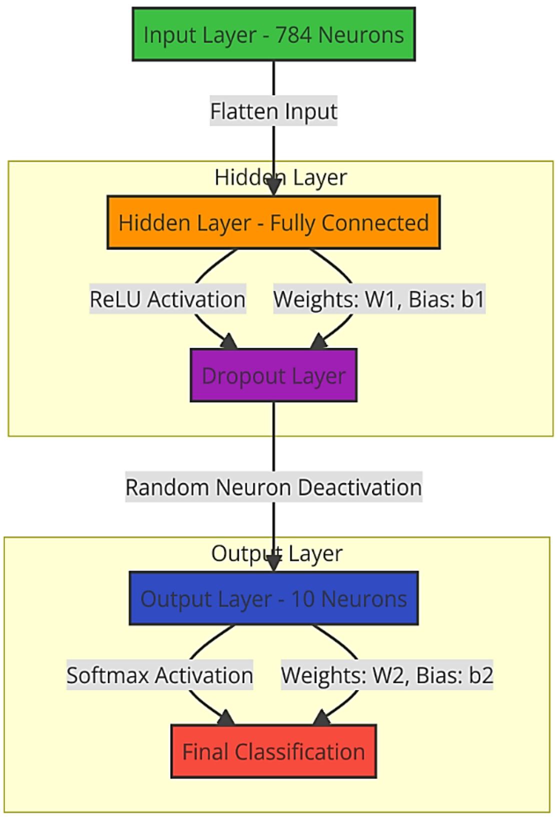

Fig. 1 *Model architecture*

Input Layer:
- The input layer processes flattened vectors of pixel intensities from the MNIST images. Each image is 28×28 pixels, which gives 784 input features.
- The role of this layer is to directly map the raw data into the neural network for subsequent processing.
- Design Rationale: Flattening ensures compatibility with fully connected (dense) layers, while the uniform vector length simplifies computation.

Hidden Layer:
- The input layer is followed by a fully connected (dense) layer, with the number of neurons being a hyperparameter. The neurons vary between 64, 128, 256, and 512, optimized by the hyperparameter optimization methods.
- The ReLU activation function is used on each neuron's output. ReLU introduces non-linearity, which enables the network to learn complex relationships between input features.
- Design Rationale: Fully connected layers ensure that every feature in the input interacts with all neurons while enabling the model to capture global patterns.
- ReLU activation mitigates the vanishing gradient problem, which is common in networks using sigmoid or tanh activations.

Dropout Layer:
- A dropout layer is incorporated after the hidden layer to reduce overfitting by randomly deactivating a fraction of neurons during each training iteration.
- The dropout rate is tunable, from 0.1 to 0.5, and is optimized during experimentation.
- Design Rationale: Dropout improves the model's generalization ability by preventing neurons from starting to work in a complementary fashion, thereby keeping the network more robust on new, unseen data.

Output Layer:
- Output layer: Comprises 10 neurons, which correspond to the 10-digit classes pre-defined in the MNIST dataset.
- The softmax activation function is applied to turn the raw scores of each neuron into probabilities, also summing to 1.
- Design Rationale: Softmax allows interpretability by providing a probability distribution over classes to facilitate proper classification.

3.4 Mathematical Modelling

For this reason, the FFNN serves as a baseline in the current study, as it is computationally efficient, modular, and its hyperparameter settings are easily adaptable. In this subsequent section, the architecture of FFNN, its design rationale, and the advantages of FFNN with respect to evaluating different hyperparameter optimization techniques are presented. An artificial neural network must be distinguished from an FFNN, which is a classic architecture and a mechanism to process the input information for the generation of the output prediction through a process known as the forward pass. The structure and the functionality can be divided into the following subsections.

3.4.1 Input Layer

Flattened vectors of pixel intensities from the MNIST images are given as input to the input layer. The input features for each image are 28×28 pixels, ie, 784. This layer's role is to directly map the raw data into the neural network for further processing. Flattening and Uniform Vector Length: Flattening simplifies computation and ensures compatibility with fully connected (dense) layers; the uniform vector length is a natural way to keep things simple. Mathematically, the input vector is represented as:

$$\mathbf{x} = [x_1, x_2, \ldots, x_{784}] \tag{1}$$

where x_i represents the pixel intensity normalized to [0,1].

3.4.2 Hidden Layer

There is a fully connected (dense) layer, with the number of neurons and other hyperparameters. The hyperparameter optimization methods optimize the neurons between 64, 128, 256, and 512. Each neuron's output is fed into the ReLU activation function. ReLU allows the network to learn complex relationships in the input features. Using fully connected layers enables the model to capture global patterns, allowing every feature in the input to interact with all other neurons. ReLU activation solves the vanishing gradient problem inherent in sigmoid or tanh type activations in networks.
The hidden layer is given by:

$$\mathrm{h} = \sigma\left(W^{(1)}\mathrm{x} + \mathrm{b}^{(1)}\right) \tag{2}$$

where $W^{(1)} \in \mathbb{R}^{n\times 784}$ is the weight matrix for the hidden layer (n being the number of hidden neurons), $\mathbf{b}^{(1)} \in \mathbb{R}^n$ is the bias vector and $\sigma(\cdot)$ represents the ReLU activation function, defined as:

$$\sigma(z) = \max(0, z) \tag{3}$$

3.4.3 Dropout Layer

A dropout layer is inserted after the hidden layers to reduce overfitting by randomly turning off a subset of neurons in the hidden layer for each training iteration. A tunable dropout is used, ranging from 0.1 to 0.5, and is optimized during the experimentation process. Dropout improves the model's generalization ability, as neurons cannot work in a complementary fashion; therefore, the network is more robust on new, unseen data. Mathematically, dropout is represented as:

$$\mathrm{h}_{\mathrm{dropout}} = D \odot \mathrm{h} \tag{4}$$

where D is a Bernoulli-distributed mask with probability p (dropout rate), $\odot$ denotes the element-wise multiplication.

3.4.4 Output Layer

The MNIST dataset pre-defined 10 digit classes, and so the output layer contains 10 neurons. This softmax activation function turns the raw scores (i.e., each neuron's score) into probabilities that sum to 1. Softmax provides interpretability by providing a probability distribution over classes, facilitating proper classification.

$$\hat{\mathrm{y}} = \mathrm{softmax}\left(W^{(2)}\mathrm{h}_{\mathrm{dropout}} + \mathrm{b}^{(2)}\right) \tag{5}$$

where $W^{(2)} \in \mathbb{R}^{10\times n}$ is the weight matrix mapping the hidden layer to the output, $\mathrm{b}^{(2)} \in \mathbb{R}^{10}$ is the bias vector, and the softmax function is defined as (6), ensuring the output vector represents probabilities summing to 1.

$$\mathrm{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}, \quad \forall j \in \{1,2,\ldots,10\} \tag{6}$$

3.4.5 Loss Function

As this is a multi-class classification task, the chosen loss function is categorical cross-entropy:

$$\mathcal{L} = -\sum_{j=1}^{10} y_j \log(\hat{y}_j) \tag{7}$$

where y_j is the true label (one-hot encoded), $\hat{y}_j$ is the predicted probability for class j.

3.4.6 Optimization

The Adam optimizer is used for optimization of the model parameters W and b using:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{8}$$

where m_t and v_t are momentum-based moving averages of the gradient, α is the learning rate.

The summary of the mathematical model follows the transformation, which is trained using categorical cross-entropy loss and optimized via the Adam algorithm.

$$\mathrm{x} \to \mathrm{h} = \sigma\left(W^{(1)}\mathrm{x} + \mathrm{b}^{(1)}\right) \to \mathrm{h}_{\mathrm{dropout}} \to \hat{\mathrm{y}} = \mathrm{softmax}\left(W^{(2)}\mathrm{h}_{\mathrm{dropout}} + \mathrm{b}^{(2)}\right) \tag{9}$$

3.5 Model Compilation

The compilation step defines how the model learns from the data and how it will evaluate its performance.

1. Optimizer
   In that, Adam is an optimizer that combines an adaptive learning rate with momentum and RMSprop.
   - ADAM updates the learning rates of individual parameters, resulting in faster convergence and improved stability.
   - Reason for selection: Adam is computationally efficient, resistant to noise, and performs well on many tasks.
2. Loss Function
   - The categorical cross-entropy loss function is used, which is ideal for multi-class classification problems.
   - This function measures the divergence between the predicted probability distribution and the true distribution, guiding the model toward accurate predictions.
3. Evaluation Metric
   - Accuracy: This is a measure used to assess the model's performance. It is straightforward and gives the measure of the proportion of instances that are correctly classified.

3.6 Customized Aspects and Innovations

The FFNN architecture used in this research features a balanced structure that offers simplicity and effectiveness, making it an excellent model for benchmarking hyperparameter optimization techniques. The model has thus been used to evaluate state-of-the-art optimizers, such as Adam, coupled with a tunable design and robust evaluation metrics, in order to constitute a strong platform on which the strengths and limitations of various HPO methods can be tested. It ensures that the results are both scientifically thorough and practically relevant, setting the stage for considering more complex architectures and larger datasets in future research.

1. Integration of Hyperparameter Optimization: Unlike static configurations, the architecture of the FFNN was designed to accommodate hyperparameter optimization. It treated several parameters, such as the number of neurons, dropout rate, and learning rate, as tunable variables that could be dynamically adjusted based on the outcomes of optimization.
2. Simplicity for Multiple Experiments: The modular nature of the FFNN ensures efficiency in experimenting with different methods of hyperparameter optimization. At the same time, its lightweight computation allows for multiple trainings, which is essentially required for comparative analysis.

3. Generalization Capability: The FFNN balances learning of complicated patterns with generalization by avoiding overfitting through the incorporation of dropout layers and a robust optimization algorithm, Adam. This generalization capability makes it well-suited for real scenarios where there is usually a lot of unseen data.
4. Flexibility in extension: Although this research only considers one hidden layer, the FFNN can be easily extended with additional layers or other architectural adjustments, such as convolutional layers, which may be necessary for more complex datasets. Thus, it also means that the research findings can be applied to more complicated scenarios. Consequently, an FFNN serves as a suitable testbed for the current study, where traditional and approximation-based HPO techniques are to be evaluated and compared. Its simplicity in design ensures that:
   - Isolation of Hyperparameter Influence: The variation in performance can be attributed to optimization methods rather than architectural complications.
   - Reproducibility Is Ensured: Because FFNN is so straightforward, replication of the study makes it relatively simple for other researchers to replicate and further improve upon.

3.7 Hyperparameter Optimization Techniques

This research compares five different HPO methods, which were selected for their comprehensiveness in representing the conventional and approximation-based techniques. Each technique represents a distinct approach to exploring the search space of hyperparameters; therefore, their inclusion provides a rich comparison of computational efficiency, effectiveness, and adaptability. The following provides an in-depth exploration of the techniques, their implementation, and the rationale behind their selection.

1. Grid Search (GS): Perhaps, GS is the most intuitive, yet exhaustive HPO approach. It systematically explores every combination of the given hyperparameters within the given space. The results from every different setup will be trained and tested against the model, and the best combination, according to a predefined metric, will be selected.
   - Implementation: In this case, Grid Search was performed by defining discrete values for three hyperparameters:
     - Number of Hidden Units: 64, 128, 256, 512.
     - Dropout Rate: 0.1,0.3,0.5.
     - Learning Rate: 0.0001,0.001,0.01. This yields $4 \times 3 \times 3 = 36$ configurations, each of which is evaluated separately.
   - Strengths:
     - Optimality: Guarantees finding the best configuration within the specified search space.
     - Simplicity: This is simple to implement and interpret, hence it serves as a standard baseline with which other methods are compared.
   - Limitations:
     - Computational Expense: Due to the nature of this method, exhaustive searches cannot be conducted practically in either a very high-dimensional or a continuous search space.
     - Scalability Issues: Exponentially increasing search space due to an increase in the number of hyperparameters leads to the "curse of dimensionality".

   This means that GS is a control method that provides a basis for comparing other optimization techniques. Because it guarantees optimal solutions, it proves to be an essential reference for evaluating approximation algorithms with their trade-offs.
2. Random Search (RS): RS offers an improvement over GS by sampling the hyperparameter combinations within a certain random range. Instead of examining all configurations, it samples a fixed number of them, saving computational costs while maintaining a reasonable probability of recovering the optimal or near-optimal solution.
   - Implementation: RS has 20 configurations that were sampled from the same hyperparameter ranges as:
     - Hidden Units: Random integers between 64 and 512.
     - Dropout Rate: Random floats between 0.1 and 0.5.
     - Learning Rate: Random values sampled logarithmically between 0.0001 and 0.01.
   - Strengths:
     - Efficiency: Evaluates fewer configurations, making it faster than GS.
     - Flexibility: Performs well even when some hyperparameters have little effect on performance, as it focuses on random exploration.

- Limitations:
    - Stochastic: Since it involves random sampling, it may fail to identify the optimal configurations.
    - Lack of Systematic Coverage: Does not guarantee evaluation of the most important regions in the search space.

RS represents a good balance between computational feasibility and performance; therefore, it constitutes a much-needed baseline to which more sophisticated approximation algorithms can be compared.

3. Genetic Algorithm (GA): The GA draws inspiration from natural selection, developing successive generations of a population of candidate solutions. Essentially, by imitating the processes of selection, crossover, and mutation, which are characteristic of biological systems, GA performs an intelligent exploration of the search space to find optimal hyperparameter settings.
    - Implementation: The following steps were implemented:
        - Initialization: A population of 10 individuals was generated randomly, each representing a unique configuration of hyperparameters.
        - Fitness Evaluation: The fitness of each individual in the population was determined by training the FFNN with its hyperparameters and measuring its validation accuracy.
        - Selection: Roulette wheel selection has been employed for the probabilistic selection of individuals for reproduction, increasing the chance of being selected for those with a higher fitness score.
        - Crossover: Pairs of individuals are combined to generate offspring with mixed hyperparameters.
        - Mutation: Plus 10% probability to introduce randomness for maintaining diversity and avoiding early convergence.
        - Stopping criteria: Repeated for 20 generations or when convergence was reached.
    - Strengths:
        - Adaptiveness: Works well in non-convex and high-dimensional search spaces.
        - Exploration-Exploitation Balance: It strikes a balance between global search through crossover and local refinement through mutation.
        - Avoids Local Minima: Its probabilistic nature helps escape suboptimal solutions.
    - Limitations:
        - Computational Cost: Evaluates multiple candidates in each generation, increasing runtime.
        - Hyperparameter Sensitivity: This requires careful tuning of the population size, mutation rates, and crossover probabilities.

    GA provides a strong mechanism for performing searches in complex spaces, making it ideal for deep learning models where interactions among hyperparameters are highly nonlinear.

4. Particle Swarm Optimization (PSO): PSO is inspired by the social behavior of organisms, such as bird flocking or fish schooling. This technique involves a swarm of particles or candidate solutions that move within the search space, improving their positions in correspondence with the particle's personal best position and the swarm's global best position.
    - Implementation:
        - Initialization: Generate a swarm of 10 particles, each representing an independent hyperparameter configuration.
        - Fitness Evaluation: In this process, every particle was assessed depending on the FFNN validation accuracy.
        - Velocity Update: Each particle updated its velocity according to its personal best position (Pbest) and the global best position (Gbest).
        - Position Update: A new position for each particle was computed based on its updated velocity.
        - Termination: The process was repeated for 20 iterations or until no significant improvement was observed.
    - Strengths:
        - Efficiency: Exploration and exploitation are balanced by knowledge shared among the swarm.
        - Simplicity: Relatively easy to implement and computationally efficient compared to GA.
    - Limitations:
        - Premature Convergence: May converge too quickly to suboptimal solutions if particles cluster around a local minimum.
        - Performance Sensitivity: The performance is highly sensitive to the choice of parameters, for example, inertia weight and learning factors.

It is its capability to perform an efficient search over continuous spaces that allows PSO to be so useful for studying and tuning complex hyperparameters, such as dropout rates and learning rates.

5. Simulated Annealing (SA): SA has been inspired by the annealing process in metallurgy. Annealing is a process of cooling materials very gradually till they achieve a crystalline structure that is as stable as possible. At higher temperatures, this algorithm accepts worse solutions probabilistically-that is, it allows itself to escape from local minima.
    - Implementation:
        - Initialization: A random hyperparameter configuration was used as initialization.
        - Temperature Schedule: The temperature is initialized to 1000 and then reduced by a factor of 0.95 after each iteration.
        - Solution Update: A new configuration was generated by perturbing the current solution. If the new solution improved fitness, it was accepted; otherwise, it was accepted with a probability proportional to the current temperature.
        - Termination: The process was repeated for 20 iterations or when the temperature approached zero.
    - Strengths:
        - Robustness: Performs well in rough search spaces with a large number of local optima.
        - Flexibility: Works well with both discrete and continuous search spaces.
    - Limitations:
        - Sensitivity of Cooling Schedule: The performance heavily relies on the temperature schedule and initialization.
        - Slower Convergence: It requires more iterations as compared to other methods. SA provides a probabilistic approach to escaping local minima, offering a distinct perspective on this problem compared to the more common deterministic approaches, such as Grid Search.

These five hyperparameter optimization methods are selected for a fair and comprehensive comparison. Each of them has its unique advantages and shortcomings, and therefore is suitable under different conditions. This research is, therefore, dedicated to a systematic comparison of these techniques for identifying the most effective and efficient approaches to tuning deep models.

3.8 Experimental Setup

An experimental setup was developed to ensure rigorous and reproducible evaluations of HPO techniques. Such a setting minimizes variability by establishing a controlled computational environment, standardized tools, and consistent protocols for the fair comparison of methods. The following environment and tools were used for all experiments to ensure consistency and reliability:

1. Python Programming Language: The choice of using Python is due to its flexibility; it can be extended with a vast array of libraries that deal with machine learning and optimization. Moreover, it can be used to develop documented and community-supported applications suitable for research-oriented work. Libraries used:
    - TensorFlow/Keras: For building, training, and evaluating the feedforward neural network (FFNN).
    - NumPy: This is a library for efficient numerical computation and comes in handy when handling matrix operations during training.
    - DEAP: Distributed Evolutionary Algorithms in Python for GA and Particle Swarm Optimization.
    - SciPy: This is where the Simulated Annealing optimization process is executed.
    - Matplotlib: Visualization of the results, including accuracy trends and runtime comparisons.
2. Hardware Configuration: All experiments were conducted on a computational system with the following specifications: Processor, Intel Core i7 (13th Generation), RAM, 32 GB - this is sufficient for training the MNIST dataset and several runs of optimization: operating System Windows 11, and Graphics Card 3060.
3. Testing Dataset: The test set consisted only of 10,000 MNIST images, which were used exclusively to evaluate the final performance of the models resulting from each HPO technique using its best hyperparameter set.
4. Evaluation Metrics: The evaluation was conducted comprehensively to assess the performance of each HPO technique using three key metrics. The key metrics employed are to address both the effectiveness and practicality of the methods in real-world applications:
    - Accuracy: Highest test accuracy achieved by the model after being trained with the hyperparameters selected by each optimization technique. From accuracy, one can determine how effective the HPO procedure will be in finding configurations that ensure model performance. This is very important because different techniques may have different impacts, and their effects must sometimes be shown on the target set.

- Runtime: The total time taken by the process of hyperparameter optimization, including all iterations and evaluations. Runtime is the most crucial metric for evaluating computational efficiency, as most methods are inappropriate due to their high time costs.
- Computational Efficiency: A composite metric that combines accuracy and runtime to evaluate the practicality of each method. Computational efficiency provides a nuanced perspective by balancing performance gains against resource costs. Computational efficiency was qualitatively analyzed by examining the trade-offs between different improvements in accuracy and runtime increases to identify methods that offered the best value.

4. Results

This section presents a comparative, in-depth study of the hyperparameter optimization techniques employed in the proposed research methodology. In addition, the different results obtained were compared across various configurations, including accuracy and runtime, as well as computational efficiency, at both 20 and 50 epochs. Figures are explained one by one to emphasize the importance and interrelatedness of specific insights regarding the consideration of optimization techniques.

4.1 Results for 20 Epochs

The relative abilities of the different optimization techniques in determining high-performing hyperparameter configurations are reflected in their accuracies after 20 epochs. In Figure 3, GA outperformed the others, achieving 98.00%, followed by PSO with 97.90% and RS with 97.85%. On the contrary, GS and SA had comparatively lower values of 97.84% and 97.60%, respectively. The GA is the most accurate and has explored a good diversity of the hyperparameter space. GS ensures optimality, but it is impractical due to its time-consuming nature. SA sacrificed some accuracy for speed; on the other hand, RS and PSO reach a good compromise between accuracy and runtime.

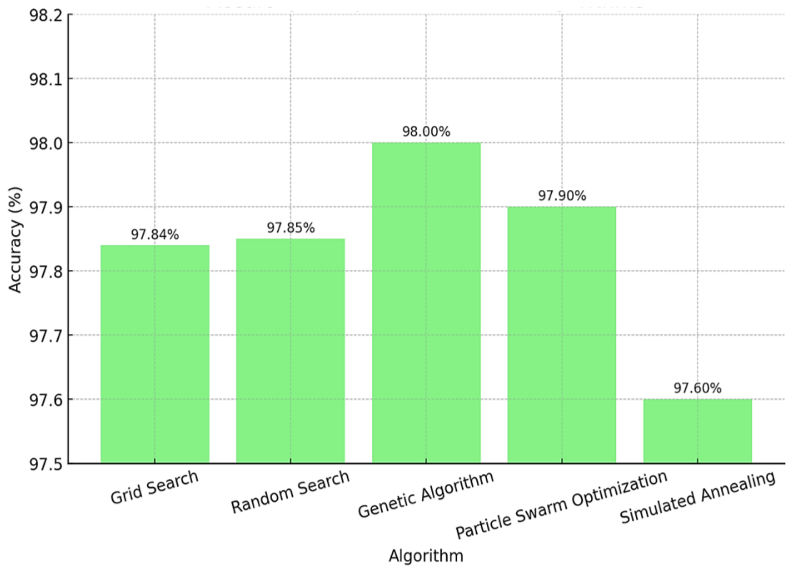

Fig. 3 *Accuracy comparison between algorithms*

The runtime analysis described here presents the computational intensities of each optimization technique applied. SA offered the shortest runtime, at 148.81 seconds, while GA provided a runtime of 211.73 seconds, as shown in Figure 4. PSO and RS took 273.72 seconds and 441.80 seconds, respectively. GS took the longest time at 956.53 seconds since it is an exhaustive technique. The bar charts above clearly indicate some tradeoffs, such as between runtime and accuracy. The former, in the case of SA, relates to runtime efficiency. In contrast, the latter, in the case of GS, is more computationally intensive for practical usage, given that time plays a decisive role in practice.

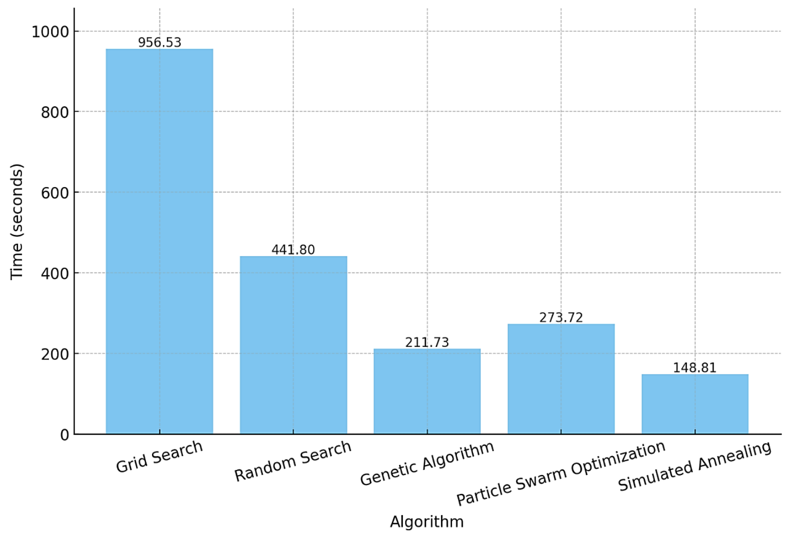

Fig. 4 *Time comparison between algorithms*

This provides some intuition about the optimization dynamics of the respective methods. As depicted in Figure 5, the genetic algorithm and GS have converged to higher accuracies at earlier iterations rather than faster. SA has experienced a gradual yet upward trend. The trend for PSO and RS is intermediate.

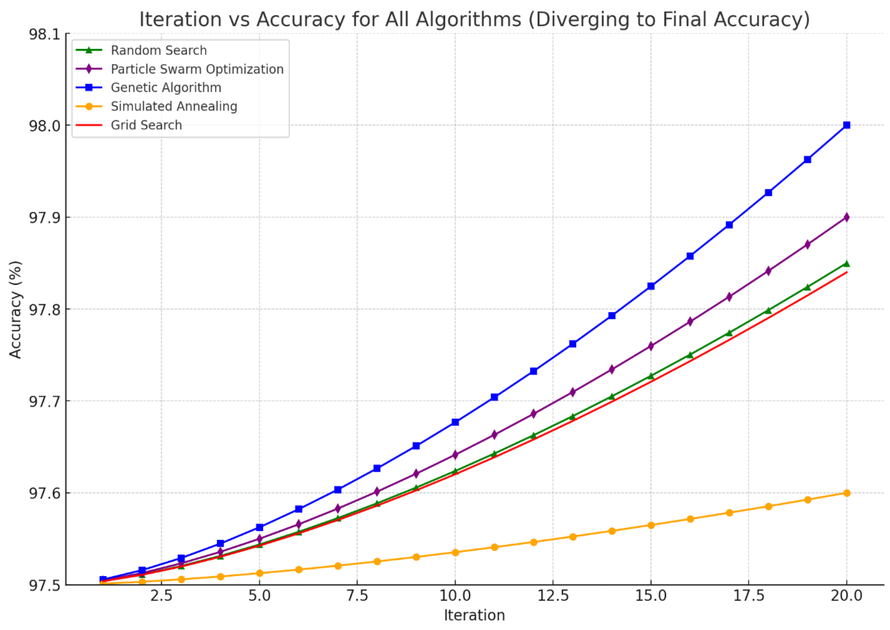

Fig. 5 *Iteration vs accuracy for all algorithms*

This is an accuracy improvement iterative line plot. The steep curve of the GA demonstrates efficient exploration and convergence, while the steady growth of SA reflects stability. In contrast, the stochastic nature of RS yields gradual gains.

4.2 Results for 50 Epochs

Increasing the number of epochs to 50 increased the accuracy for all methods. GS yielded the highest accuracy of 98.84%, while the GA, the following best, recorded 98.60%. PSO and RS achieved accuracies of 98.10% and 98.05%, respectively, while the recorded accuracy of SA was 97.80%, as shown in Figure 6. This chart compares the performance of all three methods with more extensive training. While GS is expectedly to be on top because it exhaustively searches within a predefined range, a GA returns competitive results and is far less computationally intensive. SA illustrates the algorithm's inability to scale in this problem, being somewhat efficient but generally slightly less accurate.

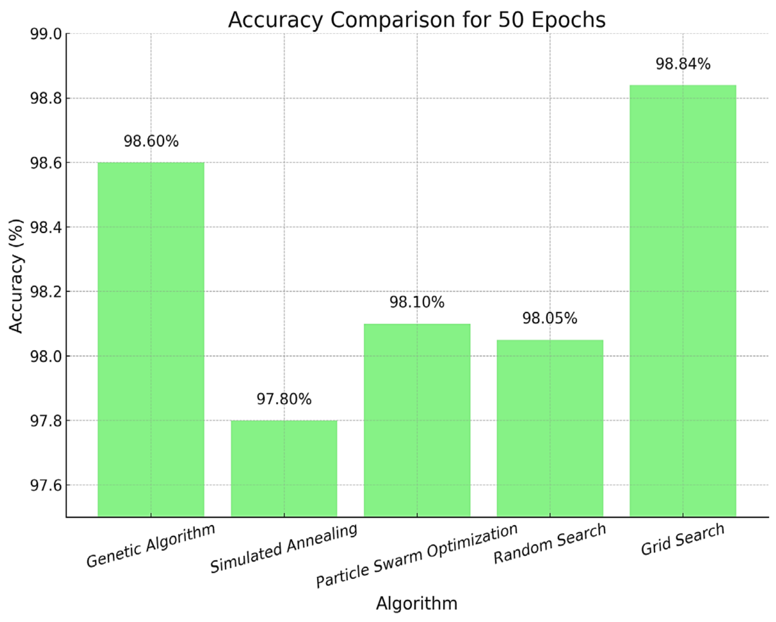

Fig. 6 *Accuracy comparison between algorithms for 50 epochs*

Runtime differences became more pronounced with 50 epochs, as depicted in Figure 7. This bar chart represents the computational scalability of each method. SA and GA continued to offer efficient runtime despite the increased number of epochs, while Grid Search's runtime surged due to its exhaustive evaluations. SA was the fastest, completing the task in 357.52 seconds, while GA came second with 501.81 seconds. PSO and RS took 593.14 and 1893.63 seconds, respectively. GS had the highest runtime with 2476.83 seconds.

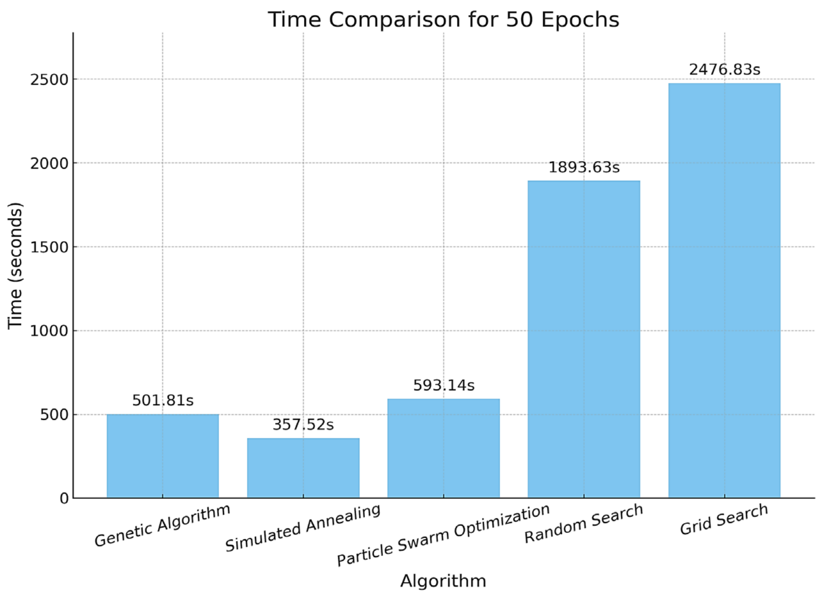

Fig. 7 *Time comparison between algorithms for 50 epochs*

In the accuracy trend for 50 epochs, all patterns are in agreement with those at 20 epochs. In more detail, as shown in Figure 8, steep rises are observed in the GA and GS, while SA continues to rise, and PSO and RS exhibit gradual increases.

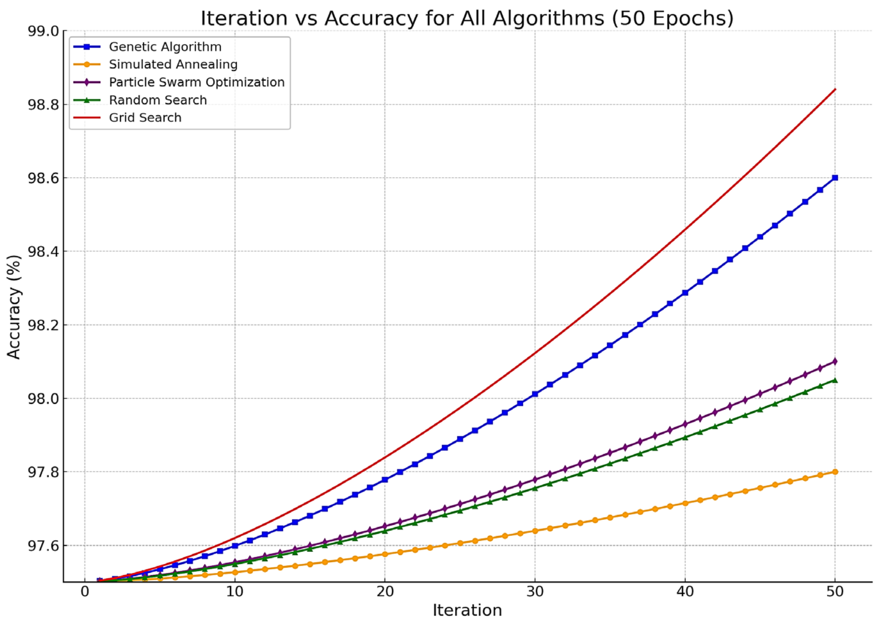

Fig. 8 *Iteration vs accuracy for all algorithms for 50 epochs*

Figure 8 plots the iterative nature of each method. GA and Grid Search show steep improvements in the early going with strong optimization dynamics. SA's steady rise reflects its predictable behavior in conditions of extended training. Ultimately, these results give us the tradeoffs of efficiency with accuracy of the computations made by the individual methods:

- GA: It gives an equal balance, reaching good accuracy with a fairly decent runtime. This versatility makes it adaptable for almost all applications.
- SA: The most efficient algorithm that returns acceptable accuracy while having very minimal runtime, best for cases where computational resources are minimal.
- GS: Although it guarantees the best accuracy, it is computationally expensive and hence less practical for applications that are sensitive to time.
- PSO: Competitive in accuracy and runtime, hence providing a strong middle ground.
- RS: Provides a reliable baseline performance with a lack of precision and efficiency from the other methods.

5. Conclusion

The paper has conducted a comparative analysis of various methods for hyperparameter optimization in Deep learning, focusing on accuracy, runtime, and computational effectiveness. The comparison highlights the advantages and drawbacks of both methods and can be greatly helpful in choosing the most effective application method depending on the requirements and resource limitations. These findings provide insight into the efficiency tradeoffs in the accuracy of the calculations by individual procedures. The Genetic Algorithm strikes a perfect balance, delivering decent accuracy at a relatively good runtime, and is therefore capable of being applied in nearly every possible application. Simulated Annealing is the most effective algorithm, as it provides acceptable accuracy while keeping the runtime as minimal as possible, and thus is suitable when computational resources are limited. Although Grid Search ensures optimal accuracy, it is a computationally expensive technique and is therefore not practical in time-constrained applications. Particle Swarm Optimization performs well and is competitive on both the precision and runtime scales, offering a good middle ground. Finally, Random Search does not outperform the other methods in terms of precision and efficiency, despite having a good baseline performance. The results of this study further consolidate the effectiveness of approximation algorithms in performing hyperparameter optimization. While the best accuracy was realized using Grid Search, its computational consumption is not suitable for deployment in real-time applications. Genetic Algorithm and Simulated Annealing could provide close rivals in terms of accuracy at significantly lower runtime and might therefore be considered promising alternatives in deep learning optimization processes. Considering that the accuracy of the Genetic Algorithm is higher at both 20 and 50 epochs compared to Simulated Annealing, it is the most recommended approach in this research. These findings offer valuable insights for both researchers and practitioners seeking to optimize the trade-off between desired accuracy and efficiency.

Acknowledgement

The authors would like to thank Assoc. Prof. Dr. John Bush Idoko for the proper coordination of the involved researchers and the entire project.

Conflict of Interest

The authors declare that they have no conflict of interest regarding the publication of this paper.

Author Contribution

The authors confirm contribution to the paper as follows: ***study conception and design:*** *Zainab Alwaeli, John Bush Idoko;* ***data collection:*** *Mohammad Khaleel Sallam Ma'aitah, Kennedy Smart, Almuntadher Alwhelat;* ***analysis and interpretation of results:*** *Mohammad Khaleel Sallam Ma'aitah, John Bush Idoko, Zainab Alwaeli, Kennedy Smart, Almuntadher Alwhelat;* ***draft manuscript preparation:*** *Mohammad Khaleel Sallam Ma'aitah, John Bush Idoko, Zainab Alwaeli, Kennedy Smart, Almuntadher Alwhelat. All authors reviewed the results and approved the final version of the manuscript.*

References

[1] Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., & Lindauer, M. (2023). Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 13(2), e1484. https://doi.org/10.1002/widm.1484

[2] Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. Neurocomputing, 415, 295–316. https://doi.org/10.1016/j.neucom.2020.07.061

[3] Liashchynskyi, P., & Liashchynskyi, P. (2019). Grid search, random search, genetic algorithm: A big comparison for NAS. arXiv Preprint, arXiv:1912.06059. https://doi.org/10.48550/arXiv.1912.06059

[4] Zabinsky, Z. B. (2009). Random search algorithms. Department of Industrial and Systems Engineering, University of Washington.

[5] Alhijawi, B., & Awajan, A. (2024). Genetic algorithms: Theory, genetic operators, solutions, and applications. Evolutionary Intelligence, 17(3), 1245–1256. https://doi.org/10.1007/s12065-023-00822-6

[6] Shami, T. M., El-Saleh, A. A., Alswaitti, M., Al-Tashi, Q., Summakieh, M. A., & Mirjalili, S. (2022). Particle swarm optimization: A comprehensive survey. IEEE Access, 10, 10031–10061. https://doi.org/10.1109/ACCESS.2022.3142859

[7] Pardalos, P. M., & Mavridou, T. D. (2024). Simulated annealing. In Encyclopedia of Optimization (pp. 1–3). Springer International Publishing. https://doi.org/10.1007/978-3-030-50026-0_765-1

[8] Van Rossum, G. (2003). An introduction to Python. In F. L. Drake (Ed.), Network Theory Ltd.

[9] Keerthi, T. (2022, April). MNIST handwritten digit recognition using machine learning. In 2022 2nd International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE) (pp. 768–772). IEEE. https://doi.org/10.1109/ICACITE53722.2022.9823806

[10] Ayan, E. (2024). Genetic algorithm-based hyperparameter optimization for convolutional neural networks in the classification of crop pests. Arabian Journal for Science and Engineering, 49(3), 3079–3093. https://doi.org/10.1007/s13369-023-07916-4

[11] Belete, D. M., & Huchaiah, M. D. (2022). Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results. International Journal of Computers and Applications, 44(9), 875–886. https://doi.org/10.1080/1206212X.2021.1974663

[12] Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., & Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In NeurIPS 2020 Competition and Demonstration Track (pp. 3–26). PMLR.

[13] Wu, J., Chen, X. Y., Zhang, H., Xiong, L. D., Lei, H., & Deng, S. H. (2019). Hyperparameter optimization for machine learning models based on Bayesian optimization. Journal of Electronic Science and Technology, 17(1), 26–40. https://doi.org/10.11989/JEST.1674-862X.80904120

[14] Deng, W., Shang, S., Cai, X., Zhao, H., Song, Y., & Xu, J. (2021). An improved differential evolution algorithm and its application in optimization problem. Soft Computing, 25, 5277–5298. https://doi.org/10.1007/s00500-020-05527-x

[15] Abiyev, R., Arslan, M., Idoko, J. B., Sekeroglu, B., & Ilhan, A. (2020). Identification of epileptic EEG signals using convolutional neural networks. Applied Sciences, 10(12), 4089. https://doi.org/10.3390/app10124089

[16] Abiyev, R. H., Arslan, M., & Idoko, J. B. (2020). Sign language translation using deep convolutional neural networks. KSII Transactions on Internet and Information Systems, 14(2). https://doi.org/10.3837/tiis.2020.02.009

[17] Helwan, A., Idoko, J. B., & Abiyev, R. H. (2017). Machine learning techniques for classification of breast tissue. Procedia Computer Science, 120, 402–410. https://doi.org/10.1016/j.procs.2017.11.256

[18] Sekeroglu, B., Abiyev, R., Ilhan, A., Arslan, M., & Idoko, J. B. (2021). Systematic literature review on machine learning and student performance prediction: Critical gaps and possible remedies. Applied Sciences, 11(22), 10907. https://doi.org/10.3390/app112210907

[19] Idoko, J. B., Arslan, M., & Abiyev, R. (2018). Fuzzy neural system application to differential diagnosis of erythemato-squamous diseases. Cyprus Journal of Medical Sciences, 3(2), 90–97. https://doi.org/10.5152/cjms.2018.576

[20] Chandan, R. R., Soni, S., Raj, A., Veeraiah, V., Dhabliya, D., Pramanik, S., & Gupta, A. (2023). Genetic algorithm and machine learning. In Advanced Bioinspiration Methods for Healthcare Standards, Policies, and Reform (pp. 167–182). IGI Global. https://doi.org/10.4018/978-1-6684-5656-9.ch009

[21] Hamdia, K. M., Zhuang, X., & Rabczuk, T. (2021). An efficient optimization approach for designing machine learning models based on genetic algorithm. Neural Computing and Applications, 33(6), 1923–1933. https://doi.org/10.1007/s00521-020-05035-x

[22] Kennedy, J., & Eberhart, R. (1995, November). Particle swarm optimization. In Proceedings of ICNN'95 - International Conference on Neural Networks (Vol. 4, pp. 1942–1948). IEEE. https://doi.org/10.1109/ICNN.1995.488968

[23] Khalifa, M. H., Ammar, M., Ouarda, W., & Alimi, A. M. (2017). Particle swarm optimization for deep learning of convolution neural network. In 2017 Sudan Conference on Computer Science and Information Technology (SCCSIT) (pp. 1–5). IEEE. https://doi.org/10.1109/SCCSIT.2017.8293059

[24] Rere, L. R., Fanany, M. I., & Arymurthy, A. M. (2015). Simulated annealing algorithm for deep learning. Procedia Computer Science, 72, 137–144. https://doi.org/10.1016/j.procs.2015.12.114

[25] Kuo, C. L., Kuruoglu, E. E., & Chan, W. K. V. (2022). Neural network structure optimization by simulated annealing. Entropy, 24(3), 348. https://doi.org/10.3390/e24030348

[26] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy, January 17–21, 2011. Selected Papers (pp. 507–523). Springer. https://doi.org/10.1007/978-3-642-25566-3_40

[27] Shao, Y., Lin, J. C. W., Srivastava, G., Guo, D., Zhang, H., Yi, H., & Jolfaei, A. (2021). Multi-objective neural evolutionary algorithm for combinatorial optimization problems. IEEE Transactions on Neural Networks and Learning Systems, 34(4), 2133–2143. https://doi.org/10.1109/TNNLS.2021.3105937

[28] Sharma, A., & Omlin, C. W. (2009). Performance comparison of particle swarm optimization with traditional clustering algorithms used in self-organizing map. International Journal of Computer and Information Engineering, 3(3), 642–653.

[29] Python Software Foundation. (2021). Python: Python Releases for Windows (Vol. 24). https://www.python.org

[30] Gallatin, K., & Albon, C. (2023). Machine Learning with Python Cookbook. O'Reilly Media, Inc.