MOHAMED KHIDER UNIVERSITY - BISKRA
FACULTY OF SCIENCE AND TECHNOLOGY
DEPT. ELECTRICAL ENGINEERING
DIVISION OF ELECTRONICS
Ref: .....

جامعة محمد خيضر - بسكرة
كلية العلوم والتكنولوجيا
قسم الهندسة الكهربائية
شعبة الإلكترونيك
المرجع: .....

Thesis title:

Metaheuristics for Deep learning architectures.
Application on computer vision problems.

By

DJENAIHI ELHANI

Thesis submitted to the Department of Electrical Engineering in candidacy for the Degree of **Doctorate (3rd Cycle)** in **Biometrics and Surveillance (Surveillance)**.

Publicly defended on **June 14, 2025**.

Members of the jury:

President:	Pr.Salim SBAA	Professor	University of Biskra
Supervisor:	Pr. Ahmed Chaouki MEGHERBI	Professor	University of Biskra
Examiner:	Pr. Athmane ZITOUNI	Professor	University of Biskra
Examiner:	Pr. Abdelkrim OUAFI	Professor	University of Biskra
Examiner:	Pr. Saber BENHARZALLAH	Professor	University of Batna2
Examiner:	Pr. Abdelmalik TALEB-AHMED	Professor	Univ. Polytechnique Hauts-de-France

2024/2025

DEDICATION

This work is dedicated
To my mother, whose boundless love, sacrifices, and unwavering support have propelled me to accomplish my goals.
To my family, whose encouragement and inspiration have become deeply ingrained in me, filling me with excitement.
To my dear brothers and sisters, I am grateful for your constant companionship and unwavering support.
To my teachers, I am deeply appreciative of the invaluable knowledge, passion, and wisdom you have shared with me, which have profoundly shaped my academic journey.
To my incredible friends, Your steadfast support and encouragement have been a cornerstone of my academic journey.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God, the Almighty, for granting me countless blessings, knowledge, and opportunities, enabling me to complete this thesis successfully.

I extend my sincere appreciation to Professor Ahmed Chaouki Megherbi, my supervisor, and Professor Fadi Dornaika and Professor Abdelmalik Taleb-Ahmed, my co-advisors, for their invaluable help, support, and companionship throughout the years of this project.

I sincerely thank the defense committee for their valuable time and effort in reading and evaluating this thesis. Their expertise and critical evaluation have significantly contributed to the quality and credibility of this research. Their insightful comments and suggestions have helped shape this thesis into a valuable contribution to the scientific community. I am truly grateful for their commitment and dedication in reviewing and assessing this work.

I would like to acknowledge the LESIA laboratory at the University of Biskra for their crucial support and resources throughout my research journey. The availability of necessary facilities, equipment, and funding provided by the laboratory has played a vital role in successfully completing this thesis. I appreciate their dedication to fostering research excellence and their role in furthering knowledge in the sector.

I thank my family, friends, and partner for their unwavering love, support, and encouragement during this academic journey. Your patience, understanding, and sacrifices have been a source of my strength and motivation.

Abstract

Deep Neural Networks (DNN) have proven helpful in computer vision tasks, but designing their architecture for optimal performance is challenging. This task demands considerable time and expertise due to the numerous parameter ranges involved. In response to this challenge, our thesis introduces three approaches to address this issue. First, we use a Particle Swarm Optimization (PSO) version known as particle swarm optimization without velocity equation (PSWV). The second method is a hybrid algorithm that combines PSO with Genetic Algorithm (GA), and the third method is MPSO (Mutation-enhanced PSO for CNN Architecture Optimization). These techniques minimize human intervention in CNN design, achieve quick convergence, and reduce the time needed to find the best CNN design. All three approaches employ a varying-length encoding strategy to represent particles within the population and integrate new particle updating methods. We compare the proposed algorithms with several contemporary methods in the literature, including 27 recent approaches, and extensively evaluate them on nine benchmark datasets used for classification tasks. Based on empirical findings, the pswvCNN, GAPSO, and MPSO approaches effectively identify CNN structures that perform similarly to standard designs.

Keywords: Metaheuristic, Optimization, PSO, GA, Hyperparameters, and CNN.

الملخص:

أثبتت الشبكات العصبية العميقة (DNNs) أنها مفيدة في مهام تصنيف الصور، لكن تصميم بنيتها للأداء الأمثل يمثل تحديًا. تتطلب هذه المهمة وقتًا وخبرة كبيرين بسبب نطاقات البارامترات العديدة المعنية. واستجابة لهذا التحدي، تقدم أطروحتنا ثلاثة نهج لمعالجة هذه المسألة. أولاً، نستخدم نسخة تحسين سرب الجسيمات (PSO) المعروفة باسم PSO بدون معادلة السرعة (PSWV). الطريقة الثانية هي خوارزمية هجينة تجمع بين PSO والخوارزميات الجينية (GA)، والطريقة الثالثة هي MPSO (PSO المعزز بالطفرة لتحسين بنية CNN). تقلل هذه التقنيات من التدخل البشري في تصميم CNN، وتحقق تقاربًا سريعًا، وتقلل من الوقت اللازم للعثور على أفضل تصميم CNN. تستخدم جميع الأساليب الثلاثة استراتيجية ترميز مختلفة الطول لتمثيل الجسيمات داخل المجموعة ودمج طرق جديدة لتحديث الجسيمات. نقارن الخوارزميات المقترحة بالعديد من الأساليب المعاصرة في الأدبيات، بما في ذلك 27 نهجًا حديثًا، ونقيمها على نطاق واسع على تسع مجموعات بيانات مرجعية تستخدم لمهام التصنيف. بناءً على النتائج التجريبية، تحدد مناهج pswvCNN و GAPSO و MPSO بشكل فعال هياكل CNN التي تعمل بشكل مشابه للتصميمات القياسية.

الكلمات المفتاحية: الخوارزميات الميتاهورية، التحسين، والمعلمات الفائقة، الخوارزمية الجينية (GA)، تحسين سرب الجسيمات (PSO)، والشبكة العصبية التلافيفية (CNN).

Publications in journals

- [1] **D. Elhani**, A.C. Megherbi, A. Zitouni, F. Dornaika, S. Sbaa, A. Taleb-Ahmed, "Optimizing Convolutional Neural Networks Architecture Using a Modified Particle Swarm Optimization for Image Classification", Expert Systems with Applications, vol.11, pp.395–407, May. 2023.

Publications in international conferences

- [2] **D. Elhani**, A.C. Megherbi. "MPSO: Mutation-Enhanced Particle Swarm Optimization for CNN Architecture Optimization." 2024 International Conference on Advances in Electrical and Communication Technologies (ICAECOT). IEEE, 2024.
- **D. Elhani**, A.C. Megherbi, A. Zitouni, S. Sbaa, A. Taleb-Ahmed, "Hybrid optimization method for deep neural network architectures," International Conference on Electrical Engineering (ISPA'2024), Biskra, April 2024.

TABLE OF CONTENTS

List of Tables

List of Figures

Figure **Page**

List of Acronyms

AI	Artificial Intelligence
ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
HPO	Automated hyperparameter optimization
AutoML	automated machine learning
BSO	Bat Swarm Optimization
CNN	Convolutional Neural Networks
DL	Deep Learning
DE	Differential Evolution
DNN	Deep Neural Networks
FA	Firefly Algorithm
GLS	Guided Local Search
GA	Genetic Algorithm
GP	Genetic Programming
GRASP	Greedy Randomized Adaptive Search Procedure
HC	Hill Clumbing
ILS	Iterated Local Search
LSTM	long short-term memory
MNIST	Mixed National Institute of Standards and Technology
ML	Machine Learning
MBO	Monarch Butterfly Optimization
CNTK	Microsoft Cognitive Toolkit

PSO	Particle Swarm Optimization
PSWV	particle swarm optimization without velocity equation
RNN	recurrent neural networks
SGD	Stochastic Gradient Descent
SOS	Symbiotic Organism Search Algorithm
SA	Simulated Annealing
TS	Tabu Search
VNS	Variable Neighborhood Search

GENERAL INTRODUCTION

Context

Deep Learning (DL) has become a potent method for addressing intricate issues in diverse domains, such as natural language processing, computer vision, and pattern identification. The Convolutional Neural Networks (CNN) is a widely used DL architecture renowned for its high performance in tasks linked to images. Nevertheless, creating the most efficient CNN structures is difficult because it requires careful consideration of numerous hyperparameters, including layer count, filter dimensions, learning rates, and activation functions.

Traditionally, optimizing CNN architectures has depended on manual methods such as random search, grid search, or Bayesian optimization. These approaches are time-consuming and labor-intensive and often produce suboptimal results. Consequently, researchers have begun exploring alternative methods to automate and enhance the optimization process.

In this context, the application of metaheuristic algorithms has attracted a lot of interest. These algorithms are optimization techniques inspired by natural or abstract phenomena, such as GA, PSO, Ant Colony Optimization (ACO), and Simulated Annealing (SA). These algorithms offer advantages in exploring complex search spaces, providing alternative search strategies, and finding better solutions than traditional methods.

This thesis explores and develops new metaheuristic algorithms for improving CNN setups. The main goal is to use these algorithms to navigate the complicated settings space and find the best ones for CNNs. By automating this process, these algorithms can help reduce the need for human involvement while making CNN models perform better. Additionally, this study aims to deal with the many challenges of optimizing CNN setups, like needing specialized knowledge, dealing with complex computations, and lacking good ways to search for the best settings. By introducing and testing these new metaheuristic algorithms, this thesis aims to simplify the process for researchers and practitioners to enhance CNN setups across various domains.

Problem statements

The CNN, or Convolutional Neural Network, is a remarkably efficient deep learning model that excels in a wide range of machine learning applications, particularly in the field of computer vision. These models are specifically built to handle organised grid-like data, such as photographs, by utilising convolutional layers to extract hierarchical characteristics from the input. Convolutional Neural Networks (CNNs) have revolutionized various domains within the field of computer vision, including image classification, object detection, and semantic segmentation, by learning relevant features directly from raw data. Their architecture is inspired by animal visual cortex organization [3]. CNNs have emerged as powerful tools for understanding image content, excelling in tasks such as image segmentation, classification, and detection [4]. What sets CNNs apart is their ability to process raw data directly, eliminating the need for explicit feature extraction, as their architecture inherently learns and extracts important image features.

The advancement of CNN models is largely attributed to the extensive expertise and research efforts of human experts, leading to significant achievements with models like VGG16 [5], ResNet [6], Inception [7], DenseNet [8], and SENet [9]. These models have been meticulously designed to cater to the unique attributes and requirements of their respective problem domains. Each solution provides a highly optimized approach that integrates particular architectural choices and approaches to efficiently address various challenges in computer vision.

VGG16, for example, has demonstrated the benefits of deeper networks with numerous layers and smaller convolutional filter sizes. Inception has highlighted the efficiency of deep networks built using the network-in-network concept. ResNets have introduced shortcut connections to facilitate training by creating direct links between a layer's inputs and outputs. DenseNets have enhanced connectivity by linking the outputs of each layer to all subsequent layers, resembling fully connected neural networks. These models, each with their unique approaches to image understanding, have significantly contributed to the progress in the field of computer vision.

The building blocks of a Convolutional Neural Network (CNN) architecture include convolutional layers, pooling layers, and fully connected layers.

Convolutional layers: These layers apply filters to input data, extracting features hierarchically. Filters slide across the input, detecting patterns like edges and textures.

Pooling layers: Pooling layers down-sample feature maps, reducing spatial dimensions while retaining essential information. Max pooling and average pooling are common operations.

Fully connected layers: These layers integrate features from convolutional and pooling

layers for decision-making. They connect every neuron in one layer to every neuron in the next, facilitating complex feature combinations.

These components work synergistically to enable CNNs to learn hierarchical representations of input data, making them powerful tools for various tasks, including image classification, object detection, and semantic segmentation.

In CNNs, two sets of parameters are crucial: trainable parameters, adjusted during training to optimize performance, and hyperparameters, predefined before training. Hyperparameters, like learning rate and network depth, control CNN behavior. Finding the optimal hyperparameter values is computationally intensive, requiring exploring various configurations to identify the best performance. This optimization process demands sophisticated algorithms to efficiently search the extensive configuration space and achieve an optimal solution.

Hyperparameter optimization encompasses various strategies, including grid search [10], random search [11], and Bayesian optimization [12], and metaheuristic algorithms, which play prominent roles in the field. Grid search is the standard approach for thoroughly exploring the hyperparameter configuration space. It systematically evaluates all possible combinations within the search space to identify the optimal configuration. On the other hand, random search defines a bounded domain of hyperparameter values as the search space and randomly samples points within this domain.

Another effective approach is Bayesian optimization, which integrates prior knowledge and observed outcomes to iteratively refine the search for optimal hyperparameters. It utilizes probabilistic models to represent the objective function and selects the next hyperparameter configuration based on an acquisition function. This methodology enables efficient exploration of the hyperparameter space.

Furthermore, metaheuristic algorithms are commonly utilized for hyperparameter optimization. These algorithms draw inspiration from natural phenomena or problem-solving heuristics to navigate the hyperparameter space effectively. Examples of such algorithms include GA, PSO, and SA. These methods offer flexibility and excel at handling complex and nonlinear optimization problems.

The objective of hyperparameter optimization in CNNs is to identify the most favorable values for the hyperparameters that maximize the model's performance and capacity to generalize. Hyperparameters are predetermined parameters that are established prior to the training process and are not acquired through the data. They have a substantial influence on the behavior of the CNN and can have a profound impact on its performance.

The main objectives of hyperparameter optimization in CNNs are:

1. **Improve Model Performance**: Optimize hyperparameter values to achieve the highest

accuracy or lowest error for tasks like image classification, object detection, or semantic segmentation. This optimization leads to superior performance on both training and validation datasets [13].

2. **Enhance Generalization**: Mitigate overfitting by determining favorable hyperparameter values that enable the model to effectively capture inherent patterns in the data, thereby improving its generalization capacity to unseen data [13].

3. **Speed up Training**: Accelerate the convergence of the training process by optimizing hyperparameters such as learning rate, batch size, momentum, and weight initialization. This efficiency is particularly crucial for handling large datasets or complex architectures, reducing computational costs and training time [14].

4. **Robustness and Stability**: Fine-tune hyperparameters to make the CNN model more robust and stable across different datasets or data distributions. This optimization ensures consistent performance across various scenarios and enhances the model's ability to generalize well, adapt to noise, and maintain stability during training [15].

5. **Minimize Manual Intervention**: Automate the hyperparameter optimization process to reduce the need for manual intervention and trial-and-error experimentation. Automated methods such as grid search, random search, and Bayesian optimization systematically explore the hyperparameter space to find optimal settings efficiently, enhancing productivity and enabling the development of sophisticated models in less time [16].

The architectural hyperparameters, including the layer count, filter quantity, layer types (convolutional or pooling), and fully connected layer count, significantly impact the optimization process with Stochastic Gradient Descent (SGD). Thoughtful selection and tuning of these hyperparameters, combined with appropriate regularization techniques and learning rate settings, can effectively address optimization challenges and enhance the neural network's performance. Selecting the correct hyperparameters to address issues associated with the ReLU activation function in CNNs is crucial. Achieving a balanced approach to hyperparameter tuning is essential for overcoming these challenges and ensuring optimal network performance.

Metaheuristic algorithms have garnered substantial attention for hyperparameter optimization due to their ability to explore complex search spaces and find near-optimal solutions efficiently. These algorithms offer flexible and robust optimization approaches that can overcome the limitations of traditional methods. These characteristics make metaheuristic algorithms powerful tools for hyperparameter optimization in deep learning architectures. Leveraging their strengths, researchers, and practitioners can effectively tune hyperparameters and discover

optimal or near-optimal configurations, leading to improved performance and efficiency in various computer vision tasks. certain researchers have made significant contributions to the automated evolution of CNN architectures, showcasing the potential of this approach to enhance CNN design by using: GA [17–21], Genetic Programming (GP) [22], PSO [1, 17, 23–30], Symbiotic Organism Search Algorithm (SOS) [31], Monarch Butterfly Optimization (MBO) [32], Firefly Algorithm (FA) [33] and hybrid algorithm [21, 34]

In this thesis, we present novel methods designed to optimize CNN architecture and improve the performance of CNN models across tasks like image classification. Through an exploration of various optimization techniques, we've developed approaches to select optimal hyperparameter configurations automatically. These methodologies reduce the need for manual tuning and decrease reliance on human expertise, thus streamlining the hyperparameter optimization process for CNNs.

Contributions

The thesis presents the following major contributions:

1. In the first approach (chapter 2):

   (1) This study introduces a fresh variant of the PSO algorithm, which operates without employing velocity equations. This novel approach optimizes convolutional neural network (CNN) architectures with variable particle lengths. By sidestepping the traditional velocity equations, our algorithm achieves a reduction in the requisite number of iterations to attain viable solutions. Consequently, this advancement enhances the overall efficiency of the optimization process.

   (2) Fresh Approach to Particle Position Update: Departing from the conventional velocity equation, each particle now adjusts its position by considering a blend of its individual and global best positions across the entire population. This novel strategy obviates the necessity for velocity calculations, streamlining the update process while upholding optimization efficacy.

   (3) Rapid Optimization algorithm for CNN architecture Design: This thesis introduces a swift optimization algorithm tailored to enhance the design of convolutional neural network architectures. This algorithm adeptly explores the architectural design space by harnessing the advantages of the particle swarm-without-velocity optimization approach, efficiently identifying optimal solutions.

These contributions collectively aim to enhance the optimization process for deep learning architectures by introducing a novel variant of the PSO algorithm, a new update strategy, and a fast optimization algorithm. The experimental evaluation of these contributions demonstrates their effectiveness and potential in achieving superior results in CNN architectural design optimization.

2. In the second proposed approach (chapter 3):

   (1) Incorporating GA Techniques for Particle Swarm Update: This approach substitutes traditional particle position and velocity update methods with strategies inspired by genetic algorithms. It entails integrating "selection", "crossover", and "mutation" to modify swarm particles. By adopting these genetic algorithm strategies, the algorithm can explore a wider solution space and potentially surpass local optima, thereby enhancing optimization results.

   (2) The method enhances architectural diversity and exploration by leveraging mutation operators, introducing randomness into the optimization process.

3. In the third proposed approach (chapter 4):

   (1) Introducing a groundbreaking approach, our method combines PSO with mutation operators to optimize Deep Neural Network (DNN) architectures.

   (2) The approach encourages the use of mutation operators to foster architectural variation and exploration, as these operators introduce a level of randomness.

Thesis Structure

The thesis delineates its structure, outlining how the chapters are organized and arranged. This overview serves as a guide for readers, assisting them in navigating the content and emphasizing the thesis's key components.

The initial section of the thesis comprises an introduction that fulfills various roles. It commences by furnishing essential background information, defining the research problem, and accentuating the thesis's contribution. Moreover, the introduction aids in contextualizing, signifying, and delineating the study's objectives. Additionally, it furnishes an outline of the thesis structure, offering readers a preview of what lies ahead in subsequent chapters.

Chapter 1 delves into a comprehensive review of the existing literature on deep learning and metaheuristic algorithms. By exhaustively exploring these areas, we aim to deepen comprehension and identify the research gap our thesis seeks to address.

Chapter 2 of the thesis focuses on elucidating and conducting experiments of the first contribution, namely, the optimization of CNN architecture using a modified PSO algorithm. This chapter's primary aim is to comprehensively analyze the algorithm's efficacy in identifying optimal architectures for convolutional neural networks.

Chapter 3 of the thesis elucidates and conducts experiments related to the first contribution, optimizing CNN architecture using a hybrid method. The principal objective of this chapter is to provide an in-depth analysis of the hybrid algorithm's effectiveness in identifying optimal architectures for convolutional neural networks.

Chapter 4 introduces a novel variant of PSO that integrates mutation mechanisms from genetic algorithms, providing detailed insights into this method.

The final segment serves as a comprehensive conclusion to our research endeavor, summarizing our findings and outlining future directions for the field.

CHAPTER 1

LITERATURE REVIEW

Contents

1.1 Introduction

This section offers a detailed overview of two essential domains: deep learning and metaheuristic algorithms. Deep learning, powered by artificial neural networks, has transformed problem-solving by allowing systems to learn from extensive datasets and make intelligent decisions. In contrast, metaheuristic algorithms, inspired by natural and social processes, provide powerful optimization techniques.

The chapter begins by explaining the foundational principles of deep learning, drawing parallels with the complex functioning of the human brain. It explores deep learning architectures like CNN and recurrent neural networks (RNN), highlighting their ability to extract intricate patterns from raw data. Additionally, it discusses the advantages and challenges of deep learning, setting the stage for integrating metaheuristic algorithms.

Subsequently, the chapter examines metaheuristic algorithms inspired by natural phenomena such as genetic evolution and swarm behavior. It outlines the mechanisms and strengths of algorithms like GA, PSO, and Differential Evolution (DE).

Furthermore, the chapter explores the fusion of deep learning and metaheuristic algorithms, which has shown promise in enhancing model performance. It covers various applications of metaheuristic algorithms in deep learning, including hyperparameter optimization and architecture search.

Additionally, the chapter discusses the benefits and challenges of integrating metaheuristic algorithms into DL frameworks. It highlights the potential for automating and optimizing deep learning models, reducing manual tuning efforts, and improving efficiency.

1.2 An overview of Deep Learning

1.2.1 Introduction

An overview of DL encompasses a wide range of essential concepts and techniques in the field. It begins with an exploration of machine learning, the broader discipline from which deep learning arises, covering key paradigms such as supervised, unsupervised, and reinforcement learning.

DL is a branch of machine learning that emphasizes the use of neural networks with numerous layers to enable the learning of hierarchical representations of features. This includes various types of deep learning networks like CNN and RNN, each designed for specific tasks and data modalities.

Hyperparameter optimization methods are also discussed, highlighting the importance of fine-tuning external configurations to improve model performance. Additionally, the overview includes an examination of deep learning software frameworks such as TensorFlow, PyTorch, and Keras, which provide the necessary tools and libraries for efficiently building, training, and deploying deep learning models.

1.2.2 Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) (see Figure 1.1), that enables computers to learn from data and improve their performance over time without being explicitly programmed. It involves a variety of algorithms and techniques that allow machines to recognize patterns, make predictions, and extract insights from complex datasets. The primary goal of machine learning is to develop systems that can automatically learn and adapt from experience, making it particularly valuable in scenarios where traditional rule-based programming is impractical due to the complexity or scale of the data involved.

1.2.3 Deep Learning

DL is a subfield of machine learning (see Figure 1.1) that focuses on algorithms inspired by the structure and function of the brain's neural networks. It aims to model high-level abstractions in data using multiple layers of processing units, also known as artificial neurons or nodes, in hierarchical architectures. These architectures, often called deep neural networks (DNNs), can learn data representations with multiple levels of abstraction, allowing them to perform tasks such as image and speech recognition, natural language processing, and reinforcement learning [35, 36].

In contrast to traditional ML techniques, DL may automate the process of learning feature sets for different tasks. This implies that DL can do feature discovery and adaptation approaches autonomously, reducing the requirement for explicit feature engineering and improving classification performance [35, 38]. Figure 1.2 illustrates the distinctions between traditional ML and DL.

1.2.4 Types of Deep Learning networks

This section offers a concise overview of several renowned categories of DL networks, including RNN and CNN. However, CNNs are discussed in more detail due to their significant importance and widespread use in various applications compared to other network types.

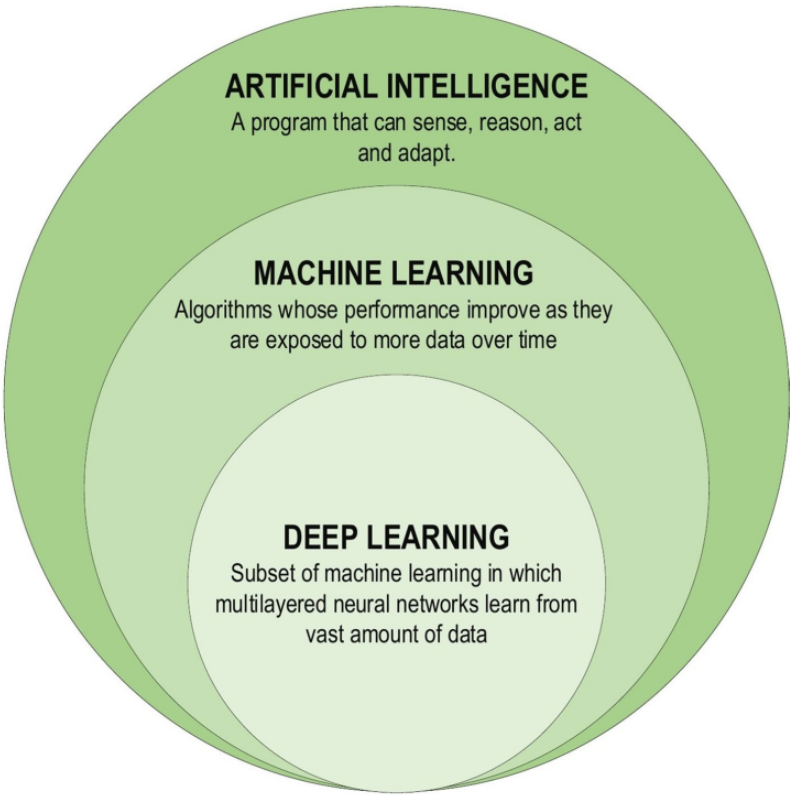

Figure 1.1: Deep learning family [37].

1.2.4.1 Reccurent Neural Networks (RNN)

RNNs are widely used algorithms in DL, commonly applied in tasks such as speech processing and Natural Language Processing (NLP). Unlike conventional neural networks, RNNs are tailored to handle sequential data, leveraging the inherent structure within sequences to extract crucial information for various applications. For example, understanding the contextual meaning of individual words within a sentence is essential. RNNs can be viewed as comprising units of short-term memory, with layers denoted as x (input), y (output), and s (hidden state). Figure 1.3 illustrates a typical unfolded RNN diagram for a given input sequence.

Pascanu et al. [39] proposed three techniques for deep RNNs: "Hidden-to-Hidden," "Hidden-to-Output," and "Input-to-Hidden." These methods aim to overcome learning challenges in deep networks and exploit the advantages of deeper RNN architectures.

However, one of the primary challenges with RNNs is their susceptibility to exploding and vanishing gradients during training. This occurs due to the repeated multiplication of large or small derivatives, causing gradients to either exponentially grow or decay, which degrades performance over time as the network processes new inputs. long short-term memory (LSTM) networks incorporate recurrent connections to memory blocks within the network,

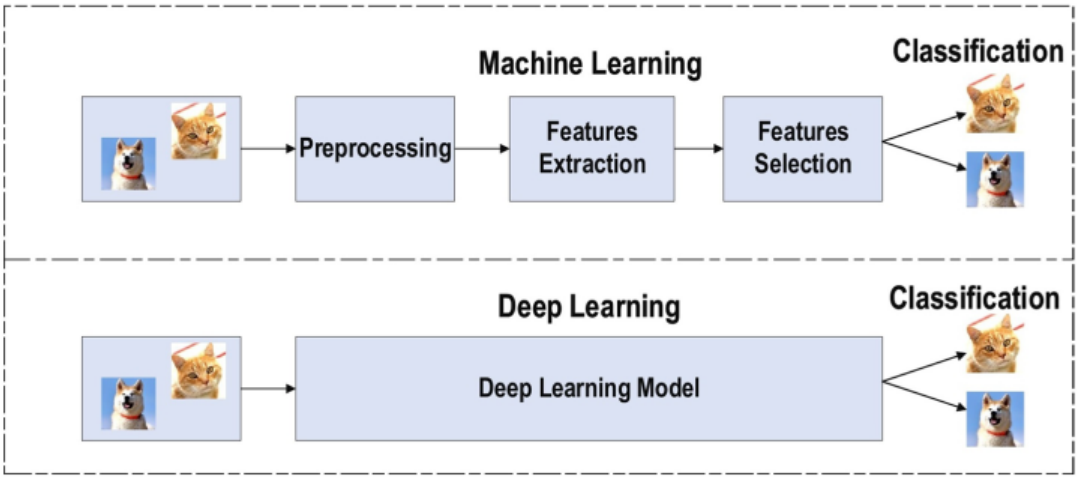

Figure 1.2: A comparison of traditional ML with DL [37].

each containing memory cells capable of retaining temporal states and gated units that regulate information flow. Additionally, residual connections are used in very deep networks to address the vanishing gradient problem.

While RNNs offer several advantages, it is worth noting that Convolutional Neural Networks (CNNs) are generally considered more powerful, with RNNs demonstrating lower feature compatibility compared to CNNs.

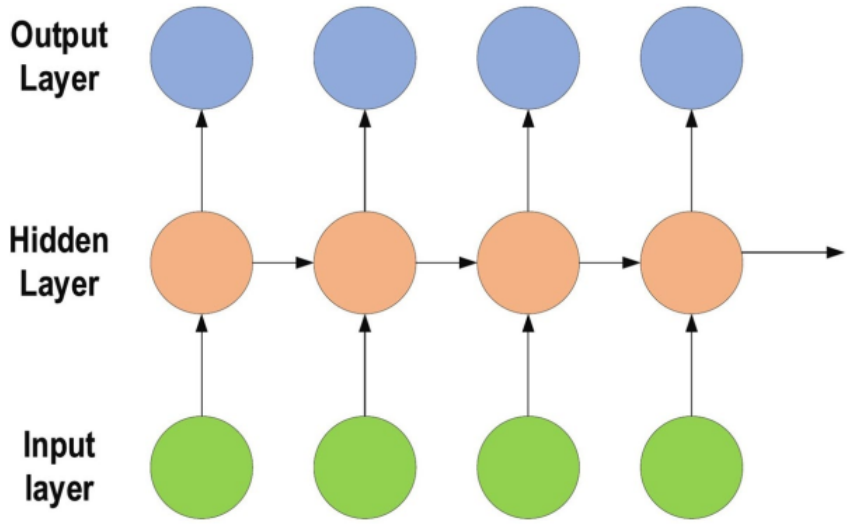

Figure 1.3: Typical unfolded RNN diagram [37].

1.2.4.2 Convolutional Neural Networks (CNN)

CNN represents a pivotal advancement in the field of DL, particularly suited for tasks involving visual imagery. Inspired by the biological visual cortex's organization [3], CNNs are structured to automatically learn hierarchical representations of data, specifically adept at processing grid-like data such as images. They have demonstrated exceptional performance in tasks like image classification, object detection, and image segmentation, setting new benchmarks in accuracy and efficiency compared to traditional ML techniques. Through layers that perform convolution operations and pooling to extract and enhance features, CNNs excel at capturing intricate patterns and spatial dependencies within images, making them indispensable in fields ranging from computer vision and medical imaging to autonomous driving and natural language processing [40–43].

A frequently employed convolutional neural network (CNN) design, resembling the structure of a multilayer perceptron (MLP), comprises of several convolutional layers, which are subsequently followed by subsampling (pooling) layers. The network concludes with fully connected (FC) layers. An illustration of a Convolutional Neural Network (CNN) structure for the purpose of image categorization is presented in Figure 1.4.

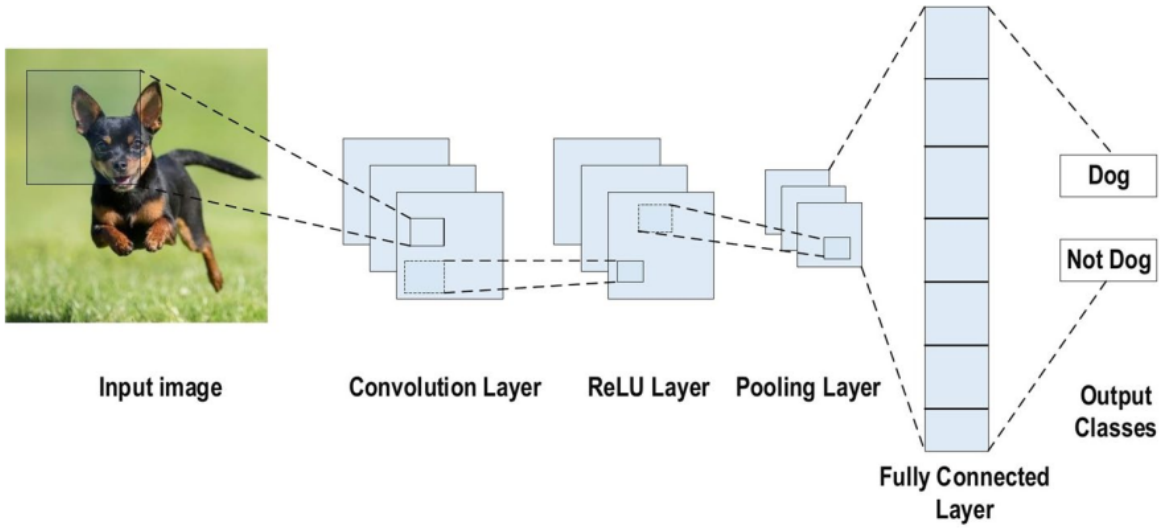

Figure 1.4: CNN Structure for Image Classification [37].

Layers of CNN

The key components of a CNN consist of convolutional layers, rectified linear unit (ReLU) layers, pooling layers, and fully connected layers.

1. **Convolutional Layer**: This is a crucial component of Convolutional Neural Networks (CNNs) architecture. These layers conduct convolution operations on input data, using small filters called kernels that slide across the input to extract features like edges, textures, and shapes. As multiple convolutional layers are stacked, the network learns more complex features.

   To illustrate the convolutional operation, consider a simple example: a (4 x 4) grayscale image and a (2 x 2) kernel. The kernel moves horizontally and vertically across the image, computing a dot product at each position between its values and the corresponding image region it covers. This dot product sums up the products of these values to produce a single scalar value, which becomes an entry in the output feature map.

   The kernel continues sliding over the image until it covers all possible positions, generating a collection of scalar values that form the output feature map. This map represents how the kernel interacts with different parts of the input image.

   Refer to Figure 1.5 for a visual depiction. Here, the light green area represents the (2 x 2) kernel, and the light blue area denotes the corresponding region of the input image. The multiplication of their values and the summation of these products (highlighted in light orange) determine the entries in the output feature map.

   Through this process, convolutional layers systematically analyze the input data, layer by layer, to build a detailed representation of its features. This capability enables CNNs to achieve high accuracy in image recognition and classification tasks.

   In the previous example, no padding is applied to the input image, and a stride of one (the step size for movement across vertical or horizontal locations) is used for the kernel. However, different stride values can be utilized. Increasing the stride value results in a lower-dimensional feature map. Padding is essential for preserving the border size information of the input image; it quickly recovers border features. Adding padding increases the input image size, which leads to a larger output feature map size. These convolutional layers provide several core benefits.

2. **Pooling Layer**: is a fundamental layer of convolutional neural networks (CNNs) used for feature extraction. Its primary purpose is to reduce the input feature maps' spatial dimensions, thereby decreasing the computational complexity of subsequent layers while preserving essential information. Pooling is typically applied independently to each feature map using a sliding window (pooling window) to traverse the input feature map. Common pooling operations include max pooling, average pooling, and min pooling, where the maximum, average, or minimum value within the pooling window is selected as

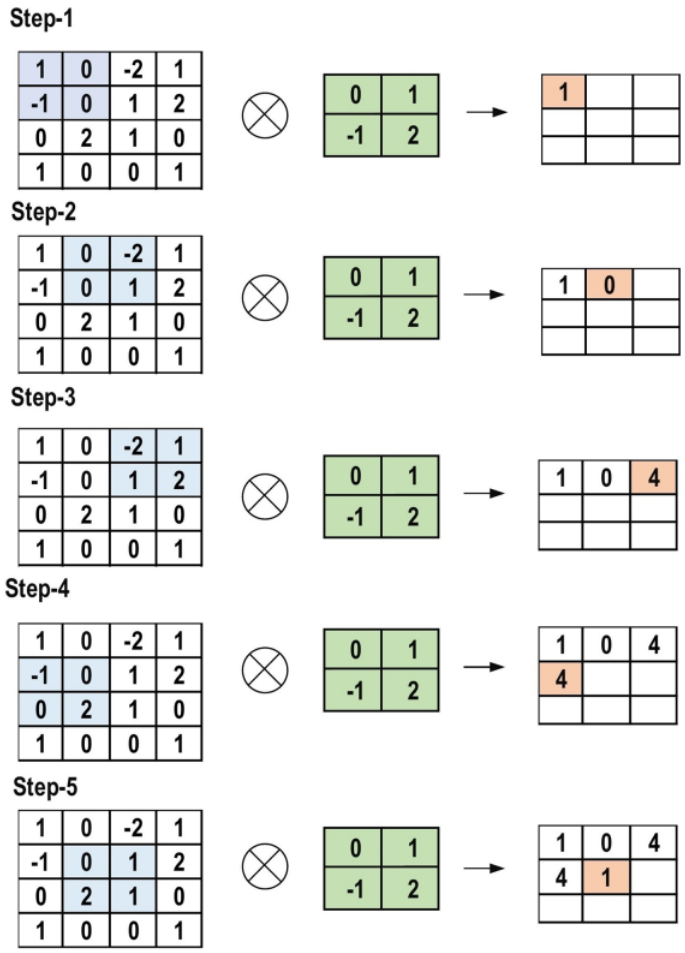

Figure 1.5: The main computations performed at each convolutional layer step [37].

the output value for that region. This downsampling operation helps achieve translation invariance and reduces overfitting in CNN models. Figure 1.6 illustrates these three pooling operations.

3. **Activation Function**, The activation function is essential for introducing non-linearity to the network's output. The presence of non-linearity is crucial for the model to acquire intricate patterns and generate precise predictions. [37].

   The predominant activation functions utilized in CNNs and other deep neural networks encompass [44]:

   * **Sigmoid**: This activation function accepts real numbers as input and constrains the output to fall within the range of zero to one. The sigmoid function is characterized by an S-shaped curve and can be expressed mathematically by: Equation 1.1 [37].

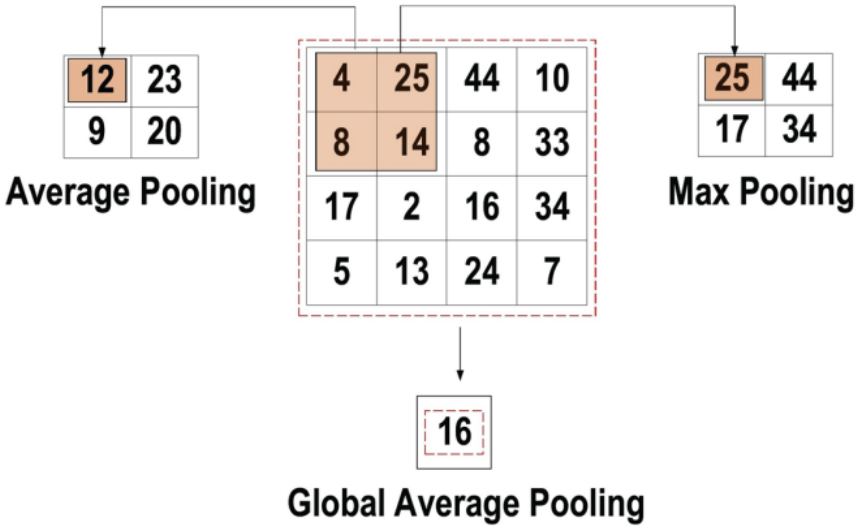

Figure 1.6: Three types of pooling operations [37].

$$f(x)_{sigm} = \frac{1}{1+e^{-x}} \tag{1.1}$$

* **Tanh**: Like the sigmoid function, tanh accepts real numbers as input, but its output is limited to -1 and 1. Its mathematical representation is shown in Equation 1.2 [37].

$$f(x)_{tanh} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{1.2}$$

* **ReLU (Rectified Linear Unit)**: which stands for Rectified Linear Unit, is an activation function commonly used in Convolutional Neural Networks (CNNs) and other deep learning models. It introduces non-linearity to the network by outputting the input directly if it is positive, and zero otherwise. Mathematically, ReLU is defined as Equation 1.3 [37].

$$f(x)_{ReLU} = max(0,x) \tag{1.3}$$

* **Leaky ReLU**: Unlike ReLU, which sets negative inputs to zero, Leaky ReLU addresses the "dying ReLU" problem where neurons can become inactive during training, halting their learning process. It introduces a small slope for negative inputs, allowing neurons

to still contribute to the gradient flow and learn from data. Mathematically, Leaky ReLU can be represented as shown in Equation 1.4 [37].

$$f(x)_{LeakyReLU} = \begin{cases} x, & x > 0 \\ mx, & x \leq 0 \end{cases} \tag{1.4}$$

4. **Fully Connected Layer**: A Fully Connected Layer, also called a dense layer, is where each neuron is connected to every neuron in the previous layer. This means that each neuron in a fully connected layer receives input from all neurons in the preceding layer. The output of each neuron in the fully connected layer is computed using a weighted sum of the inputs, followed by an activation function. This structure is illustrated in Figure 1.7.

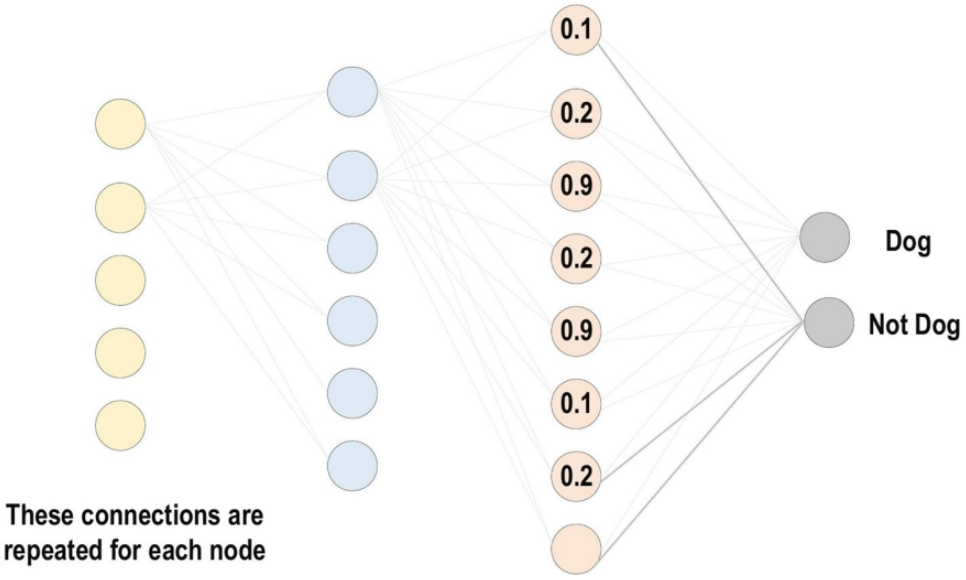

Figure 1.7: Fully connected layer [37].

5. **Loss Functions**: The preceding section discussed various layer types within a CNN architecture. The final classification result is obtained from the output layer, the last layer of the CNN. Specific loss functions are used in this layer to measure the predicted error across the training samples. This error represents the difference between the actual and predicted outputs, and it is minimized during the CNN learning process [37].

   To compute the error, the loss function requires two parameters: the CNN's predicted output (often called the prediction) and the actual output (referred to as the label).

Different loss functions are used for different types of problems. Below is a brief overview of some common types of loss functions [37].

(a) **Cross-Entropy or Softmax Loss Function**: The cross-entropy loss function, also known as the log loss function, is frequently used to evaluate the performance of CNN models. It outputs a probability within the range of [0, 1]. This function commonly replaces the squared error loss function in multi-class classification problems. In the output layer, softmax activations are typically employed to produce the output as a probability distribution. The mathematical representation of the output class probability is shown in Equation 1.5 [37].

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^{N} e_k^a} \tag{1.5}$$

Here, e^{a_i} represents the non-normalized output from the preceding layer, while N represents the number of neurons in the output layer. Finally, the mathematical representation of cross-entropy loss function is Equation 1.6 [37].

$$H(p,y) = -\sum_{i=1}^{N} y_i log(p_i) \tag{1.6}$$

(b) **Euclidean Loss Function**: The Euclidean Loss Function, also referred to as the mean square error, finds extensive application in regression tasks. It quantifies the disparity between predicted and actual values within a dataset. The mathematical representation of the estimated Euclidean loss is indicated by Equation 1.7 [37].

$$H(p,y) = \frac{1}{2N} \sum_{i=1}^{N} (p_i - y_i)^2 \tag{1.7}$$

(C) **Hinge Loss Function**: is frequently utilized in binary classification tasks, particularly in the context of maximum-margin-based classification. This function holds significance for Support Vector Machines (SVMs), which utilize the hinge loss function. In this framework, the optimizer maximizes the margin between dual objective classes. The mathematical expression of the Hinge Loss Function is denoted by Equation 1.8 [37].

$$H(p,y) = \frac{1}{2N}\sum_{i=1}^{N}(max(0,(m-2y_i-1)p_i) \tag{1.8}$$

The margin m is commonly set to 1. Additionally, the predicted output is p_i, while the desired output is y_i.

1.2.4.3 Hyperparameters CNN

Hyperparameters are essential configurations defined before the training process begins in the context of CNNs. They play a critical role in shaping the architecture and behavior of the CNN model.

These hyperparameters are distinct from the model's internal parameters, such as weights and biases, which are learned from the training data. Instead, they are external settings that guide the network's design and training process. Hyperparameters can generally be classified into two main types:

1- **Hyperparameters with Discrete Choices**: Some hyperparameters involve selecting from a discrete set of options. For example, choosing the type of activation function (e.g., ReLU, Sigmoid) or determining the number of layers in a neural network falls under this category. The machine learning algorithm explores various combinations of these discrete choices to identify the best-performing model.

2- **Hyperparameters with Continuous Values**: Other hyperparameters can take on continuous values within a specified range. Examples include the learning rate in gradient descent optimization. These hyperparameters require searching across a range of possible values to determine the optimal setting.

Hyperparameters can also be divided into two types on the other hand:

a) Hyperparameters that Determine the CNN Structure: These hyperparameters shape the architecture.

b) Hyperparameters that Determine the CNN Training: These hyperparameters control various aspects of the network's training process.

Regularization to CNN

In the context of CNN models, overfitting is a significant issue that affects the ability of the model to generalize well. Overfitting occurs when the model performs exceptionally well on the training data but fails to perform adequately on test data (unseen data). On the other hand, underfitting is the opposite scenario, where the model fails to learn sufficiently from the

Table 1.1: Hyperparameters that Determine the CNN Structure.

Hyperparameters	Description
Kernel(Filter) Size	The size of the kernel in a convolutional layer.
Number of feature map	The number of kernels in a convolutional layer.
Stride	number of pixels or units that a filter or kernel is moved across the input image or feature map.
Padding	refers to adding extra rows and columns of zeros to the input image or feature map.
Pooling type	How calculate the value for each patch on the feature map (average, Max).
Number of layers	The total number of layers that make up the network.
Number of neurons	The count of neurons in a fully-connected layer.

Table 1.2: Hyperparameters that Determine the CNN Training.

Hyperparameters	Description
Learning rate	Amount of change in weight that is updated during learning.
Batch Size	Group size to divide training data into several groups.
Weight initialization	Initialize the weights with small random numbers(Xavier initialization, He initialization).
Loss function	Function to calculate error (MSE, cross-entropy, etc.)
Optimiser	Is argument required for compiling the model (SGD, Adam, RMSprop, Adadelta, etc.)
Epoch	refers to a complete iteration of the entire training dataset during the training process.
Dropout rate	The method drops out units in neural network according to the desired probability.
Activation function	Neuron's activation function (ReLU, elu, sigmoid, etc.)

training data. When a model performs well on both the training and testing data, it is considered to be well-fitted or appropriately fitted. These three types of fitting are depicted in Figure 1.8.

To address overfitting, various intuitive techniques and concepts are employed to regularize the model. Regularization methods are used to prevent the model from fitting the training data too closely and thereby improving its ability to generalize to unseen data. These methods help strike a balance between fitting the training data well and maintaining the model's capacity to make accurate predictions on new data.

Regularization techniques for mitigating overfitting include:

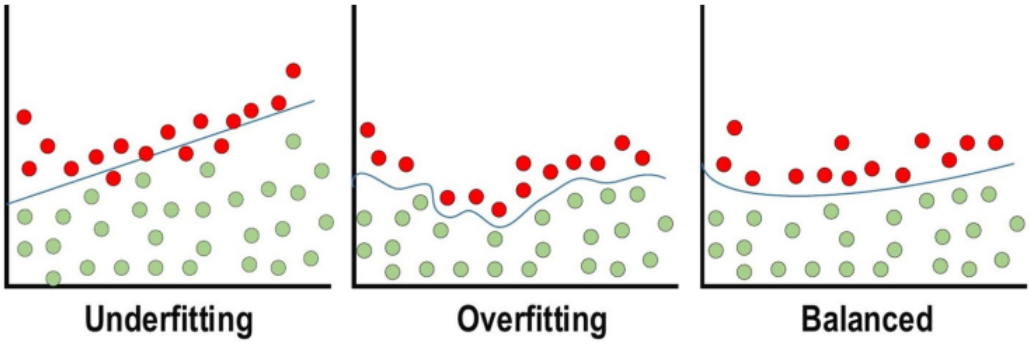

Figure 1.8: Over-ftting and under-ftting issues [37].

1. **L2 Regularization (Weight Decay)**: This technique adds a penalty term to the loss function, encouraging the model to minimize the squared magnitudes of the weights. It helps to prevent overly large weights that may lead to overfitting.

2. **L1 Regularization**: Similar to L2 regularization, L1 regularization adds a penalty term to the loss function. However, instead of penalizing the squared magnitudes of the weights, it penalizes the absolute magnitudes. This encourages sparsity in the weight matrix, leading to simpler models.

3. **Dropout**: Dropout is a regularization technique used in neural networks that randomly sets a fraction of input units to zero during the training phase. This approach helps to prevent the co-adaptation of neurons, thereby encouraging the model to learn more robust and generalized features. By temporarily omitting various neurons, dropout reduces the likelihood of overfitting, as the model becomes less reliant on specific pathways and instead learns to distribute its representations more evenly. This process improves the overall generalization capability of the neural network when applied to unseen data.

4. **Data Augmentation**: Data augmentation is a technique used to artificially increase the size of the training dataset by applying various transformations to the input data. These transformations can include rotation, translation, scaling, flipping, and other modifications that alter the appearance of the data while preserving its essential characteristics. By exposing the model to a wider range of variations, data augmentation helps to enhance the model's ability to generalize to unseen data. This method effectively reduces overfitting and improves the robustness and performance of the model when applied to new, unseen examples.

5. **Early Stopping**: Early stopping entails monitoring the model's performance on a val-

idation set during training and terminating the training process when the performance begins to deteriorate. This aids in mitigating the issue of overfitting, where the model excessively adapts to the training data.

6. **Batch Normalization**: Batch normalization normalizes the activations of each layer in a mini-batch, which helps to stabilize the training process and reduces the likelihood of overfitting.

7. **Model Simplification**: Simplifying the model architecture by reducing the number of parameters, layers, or neurons can help to prevent overfitting, especially when the training data is limited.

   These methods can be employed separately or in conjunction to successfully reduce overfitting and enhance the generalization capability of machine learning models.

1.2.5 Deep Learning Software Frameworks

Deep learning software frameworks are essential tools for building, training, and deploying deep neural networks. These frameworks provide a structured environment for developing artificial intelligence and machine learning models, especially in the domain of deep learning. Below are some of the prominent deep learning software frameworks:

(1) **Tensorflow**, backed by Google, currently stands as the most widely used deep learning framework, evident through its dominance in Google searches, numerous articles, books, and GitHub repositories. It boasts extensive capabilities, supporting a broad spectrum of deep learning and reinforcement learning algorithms. Moreover, proficiency in TensorFlow is highly sought-after in job listings, reflecting its importance in the field of machine learning. The framework's backing by Google, its capability to handle diverse machine learning tasks, and its prominence in industry applications contribute to its status as the leading deep learning framework. TensorFlow's enduring popularity reflects its significant impact on the field of artificial intelligence and machine learning.

(2) **Keras** is a software framework for building and training artificial neural networks. It is a high-level neural networks API, written in Python, that runs on top of lower-level deep learning frameworks like TensorFlow, Theano, and Microsoft Cognitive Toolkit (CNTK) [45]. Overall, Keras is a powerful tool for developing neural network models for various machine learning and deep learning applications.

Its user-friendly interface and versatility have contributed to its popularity in the deep learning community.

(3) **PyTorch**, developed by Facebook's AI group, has rapidly gained popularity in recent years as another widely used framework. It offers greater flexibility for creating, combining, and processing tensors, which are fundamental in neural networks. PyTorch utilizes a high-level API, making learning and development faster and more intuitive. One of its main advantages is its 'define-by-run' methodology, allowing for debugging and modification of the network during runtime. It supports both Python and C++ programming languages. While PyTorch is prevalent in research labs, it is not as commonly used in production compared to TensorFlow. However, PyTorch is steadily gaining popularity [46].

(4) **Caffe** has existed for six years. In Caffe, neural network models are constructed using configuration files instead of conventional hard-coded programming. It is primarily utilized for image processing and feedforward networks but does not support recurrent networks. Consequently, it is unsuitable for handling text, audio, and time series data. However, its usage has decreased over time due to the slow development of new features [47].

1.3 An overview of metaheuristics

1.3.1 Introduction

Over the previous three decades, there has been a growing interest in metaheuristics, which are techniques that often mimic natural processes such as evolution or animal behavior. Examples of such methods include GA, DE, SA, PSO, Artificial Bee Colony (ABC), ACO, FA, and Bat Swarm Optimization (BSO). These techniques are easy to design and implement and have proven to be versatile and effective in solving complex optimization problems in various fields such as image processing, autonomous learning, and vehicle routing. However, one major challenge is the difficulty in tuning the parameters of these algorithms, which can significantly impact their performance and require considerable computational resources. Additionally, choosing the most suitable metaheuristic for a specific problem can be a daunting task. In this part, we introduce the fundamental concepts of optimization and different types of bio-inspired metaheuristics.

1.3.2 Fundamental Background on Optimization

In the following, we provide a definition of some concepts used in the discipline of metaheuristic optimization.

1.3.2.1 Problem Definition

The optimization problem at hand must be well defined and modeled in which a particular objective /fitness/ cost function must be optimized within a given search space. Without loss of generality, the problem can be formulated as [48]:

$$\underset{x \in S}{\text{minimize}} f(x) = \{x^* \in S | f(x^*) \leqslant f(x) \forall x \in S\} \tag{1.9}$$

where $x \in S \subseteq R^D$ is a candidate solution to the problem, being D the problem dimensionality (usually referred to as design variables), and S the search space, typically defined in terms of bounding box constraints $lb_k \leqslant x_k \leqslant ub_k \forall\ k \in \{1,2,...D\}$, where lb_k and ub_k are the lower and upper bound, respectively, for each k^{th} variable; f(x) : $R^D \rightarrow R$ is the cost function and $x^* \in S$ is the global optimum solution. Relatively to a neighbor function N, which is a set of solutions generated by performing a perturbation using some search operator, a solution x is said to be local optimum if it has a better quality than all of its neighbors; that is, $f(x) \leqslant f(\acute{x})$, $\forall \acute{x} \in N(x)$ (Figure 1.9).

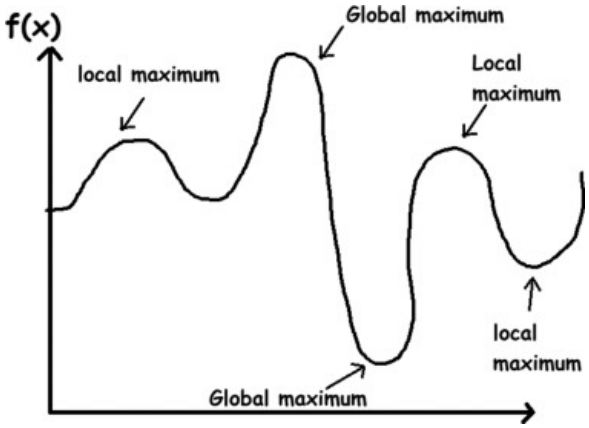

Figure 1.9: Local and global optimum in the search space S [49].

1.3.2.2 Exploration vs Exploitation

Exploration and exploitation, also known as diversification and intensification, represent two essential characteristics of every optimization metaheuristic [50]. Despite appearing

contradictory, they actually complement each other. Achieving an appropriate balance between them is crucial when designing any metaheuristic to effectively converge towards the global optimum, thus successfully solving the optimization problem at hand, while also avoiding getting trapped in a local optimum.

Specifically, exploration involves the ability to search for promising regions of the search space, typically at the beginning of the search process. These regions are where high-quality solutions are likely to be found. On the other hand, exploitation involves refining these high-quality solutions based on the search experience acquired, thereby promoting the convergence of the algorithm within the search space - (Figure 1.10).

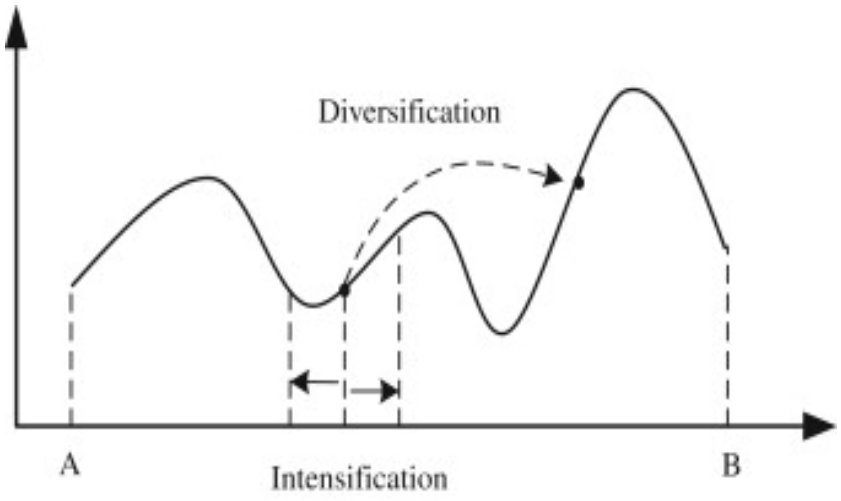

Figure 1.10: Exploration vs Exploitation [51].

1.3.2.3 Local Search vs Global Search

Local search optimization algorithms, in general, are more exploitative methods, often referred to as "search intensifiers." Examples include Tabu Search (TS) [52], Greedy Randomized Adaptive Search Procedure (GRASP) [53], and Iterated Local Search (ILS) [54]. These algorithms are adept at quickly and accurately reaching the local optimum by focusing on exploiting the immediate neighborhood of the current solution.

In contrast, global search methods are inherently more explorative. Examples include GA [55], PSO [56], and ACO [57]. These algorithms explore a wider range of solutions across the search space to identify the global optimum.

Additionally, there are several hybrid approaches that leverage the local search capability of local optimization algorithms to refine solutions within the global search framework of population-based metaheuristics.

1.3.2.4 Optimization Methods Classifcation

Depending on the complexity of the problem, it may be resolved using an exact or an approximate method. However, the precise terminology used varies slightly across authors (Figure 1.11).[58].

- **Exact methods**: These optimization methods aim to find the globally optimal solution by exhaustively evaluating all possible solutions within a given search space. Exact methods guarantee finding the best solution but may become computationally expensive or infeasible for large-scale problems due to the exponential growth of the search space. Examples of exact methods include branch and bound dynamic programming, and integer linear programming.
- **Approximate methods**: including the class of metaheuristics, focus on producing high-quality solutions within a reasonable computation time, rather than seeking the globally optimal solution. They prioritize efficiency over optimality, generating solutions that are of good quality without providing a guarantee of global optimality.

1.3.2.5 Metaheuristics

Numerous definitions of what a metaheuristic is have been proposed in the literature, but none have received universal approval. In general, the word "metaheuristics" refers to a class of stochastic, iterative algorithms that progress toward (a near approximation of) the global optimum of a black-box objective function. The majority of them are based on an evolving population of solutions. They frequently begin by initializing a population of solutions that will evolve by the use of certain search operators. The resulting new population is an improved one.

Many criteria can be used to classify metaheuristics:

Those inspired by natural processes vs those not inspired by them [58]: The most popular metaheuristics are often inspired by analogies with various natural phenomena. For instance, evolutionary algorithms draw inspiration from biology, simulated annealing from physics, and swarm intelligence from behaviors observed in ant colonies, bee

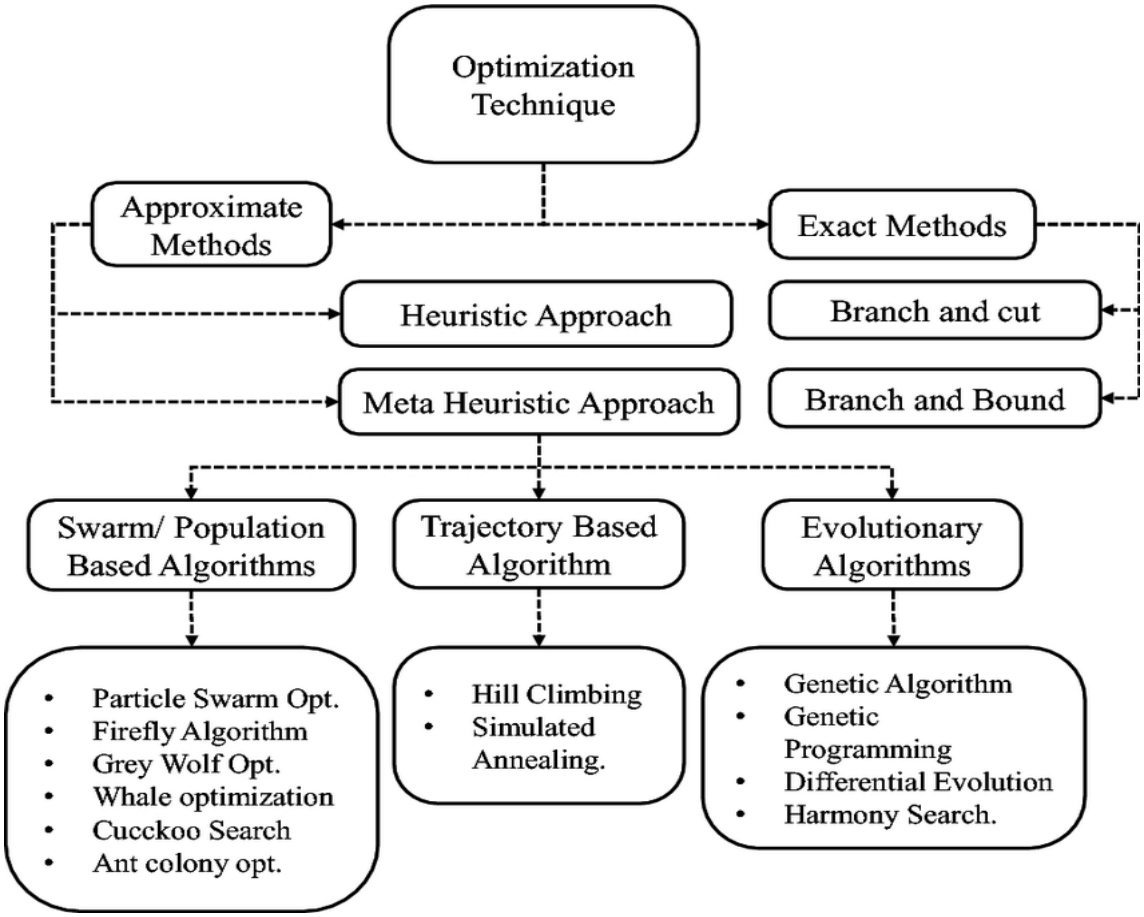

Figure 1.11: Classifcation of optimization methods [59].

colonies, and particle swarm optimization. Recently, in a comprehensive analysis conducted by Molina et al. [60], the authors examined over three hundred papers covering different types of metaheuristics. They classified these metaheuristics based on their natural or biological inspiration, allowing for quick and unambiguous identification of a specific algorithm's category. The goal of this taxonomy is to facilitate the grouping of the multitude of solvers published in the literature, making it easier for researchers to navigate and understand the landscape of metaheuristic algorithms.

With memory vs without memory [58]: Some metaheuristics operate without memory, meaning that no extracted information is utilized during the search process. Conversely, others make use of a memory that stores specific information obtained during the search. Tabu Search is an example of a metaheuristic that utilizes such a memory mechanism.

Deterministic vs Stochastic [58]: A deterministic metaheuristic solves an optimization problem by making fixed decisions, exemplified by local search and tabu search algo-

rithms. These algorithms consistently produce the same final solution when starting from the same initial solution. Conversely, stochastic metaheuristics incorporate random rules during the search process, as seen in algorithms like SA and evolutionary algorithms. This randomness allows stochastic metaheuristics to generate different final solutions from the same initial solution, highlighting their probabilistic nature.

Population-Based Search vs Single Solution-Based Search [58, 61]: Population-based search involves maintaining a population of candidate solutions throughout the optimization process. Algorithms following this approach, such as Genetic Algorithms (GA) and Particle Swarm Optimization (PSO), manipulate multiple solutions concurrently. These algorithms typically involve operations such as reproduction, mutation, and selection among the members of the population. Population-based search methods leverage the collective intelligence of the population to explore the search space more effectively and potentially discover better solutions. In contrast, a single solution-based search focuses on optimizing a single solution iteratively. Examples of algorithms following this approach include Local Search and Simulated Annealing. These algorithms iteratively improve a single solution by exploring its neighborhood or applying probabilistic transformations. Single solution-based search methods often exploit local information to refine the current solution iteratively.

1.3.3 Single Solution-Based Metaheuristics

They are also known as "trajectory methods". They handle only one solution at a time and gradually try, based on the notion of the neighborhood, to improve its quality during the different iterations. They construct a trajectory in the search space by trying to move towards optimal solutions. They have demonstrated their effectiveness in solving several optimization problems in different fields.

Many methods based on a single solution have been proposed in the literature. The most common ones are: Hill Clumbing (HC), SA, TS, ILS, Variable Neighborhood Search (VNS) and Guided Local Search (GLS) . . . etc. Each of them was the inspiration for other variants [62].

1.3.3.1 The Simulated Annealing

The SA algorithm traces its origins back to statistical mechanics, notably the Metropolis algorithm (Metropolis et al., 1953). It was initially introduced by Kirkpatrick et al. (1983) and independently by Cerny (1985). SA draws inspiration from the annealing process

utilized in metallurgy to attain a well-organized solid state with minimal energy while sidestepping metastable structures characteristic of local energy minima. This process involves gradually heating a material to high temperatures and cooling it down.

In optimization problems, SA employs the annealing concept to minimize the objective function. This function, akin to a material's energy, is minimized by introducing a controllable parameter referred to as temperature (T). As the temperature decreases, the algorithm explores the solution space more selectively, allowing it to escape local minima and converge towards the global minimum.

The algorithm starts by generating an initial solution, which can be done either randomly or using a heuristic approach, and then initializes the temperature parameter T. At each iteration, a solution $\acute{s}$ is randomly chosen from the neighborhood $N(s)$ of the current solution s. The solution becomes the new current solution based on T and the objective function values for and s, denoted by $f(\acute{s})$ and $f(s)$, respectively. If $f(\acute{s}) \leq f(s)$, is accepted and replaces s. Otherwise, if $f(\acute{s}) > f(s)$, may still be accepted with a probability $p(T, f(\acute{s}), f(s)) = \exp\left(-\frac{f(\acute{s})-f(s)}{T}\right)$. The temperature T decreases during the search process, causing the probability of accepting deteriorating moves to decrease gradually. The high-level SA algorithm is outlined in algorithm 1.

Algorithm 1: Simulated Annealing Algorithm

Input: Initial solution s, Initial temperature T

Output: Final solution S_{final}

```
while Stopping criterion not met do
    repeat
        Randomly select ś ∈ N(s);
        if f(ś) ≤ f(s) then
            Accept ś as the new current solution;
        else
            s ← ś with a probability p(T, f(ś), f(s));
        end
    until the "thermodynamic equilibrium" of the system is reached;;
    Decrease T;
end
```

Notably, the algorithm may converge to a solution s, even if a superior solution is encountered during the search process. As a basic improvement to SA, saving the best solution encountered during the search process can be implemented. SA has been

successfully applied to various discrete or continuous optimization problems, although it has been observed to be overly greedy or incapable of solving some combinatorial issues. The adaptation of SA to continuous optimization problems has been extensively studied [63]. A comprehensive bibliography can be found in [64–67].

1.3.4 Population-Based Metaheuristics

Population-Based Metaheuristics refer to a class of optimization algorithms that operate on a population of candidate solutions rather than single solutions. These algorithms explore the solution space by evolving and improving the population over multiple iterations. Examples of population-based metaheuristics include GA, PSO, and ACO. These algorithms leverage principles from natural systems such as evolution, swarm behavior, and ant colony foraging to efficiently search for high-quality solutions to optimization problems.

1.3.4.1 The genetic algorithm

The GA was initially devised by John Holland and his team at the University of Michigan to explore the adaptive processes found in natural systems and to create artificial systems that mimic the adaptive mechanisms of biological systems [55]. In a Genetic Algorithm, selection, crossover, mutation, and replacement are the four main operators employed to evolve the population towards better solutions.

- **Selection**: This operator selects the fittest individuals from the current population to become parents of the next generation. Selection methods can vary, such as roulette wheel selection or tournament selection, but the basic idea is to favor individuals with better fitness values.

- **Crossover**: This operator takes two selected parents and creates a new offspring by swapping parts of their genetic material. This process is called crossover, and it's usually performed by selecting a random point in the parent's chromosomes and exchanging genetic material beyond that point to create two new offspring.

- **Mutation**: This operator introduces random changes into the genetic material of individual solutions. The mutation is important because it provides diversity to the population and allows it to explore new regions of the search space. The mutation process is usually applied to each offspring with a low probability of occurrence.

- Replacement: This operator replaces the current population with the new offspring population created by applying selection, crossover, and mutation operators. The replacement process is necessary to ensure the population evolves over time and to prevent it from getting stuck in local optima.

The fundamental steps of a Genetic Algorithm (GA) include (illustrated in algorithm 2,Figure 1.12):

- Randomly initialize the population of solutions.
- Assess the fitness of each solution within the population.
- Employ the selection operator to choose parents for the subsequent generation.
- Apply crossover and mutation operators to generate new offspring.
- Evaluate the fitness of the newly created offspring.
- Substitute the old population with the newly formed offspring population.
- Iterate through steps 3-6 until a stopping criterion is met.
- The integration of selection, crossover, mutation, and replacement enables the GA to explore the search space effectively and move towards superior solutions.

Algorithm 2: Genetic Algorithm

```
1  Initialize population with random solutions.
2  Evaluate fitness of the population.
3  while termination criterion is not satisfied do
4      -Select parents.
5      -Perform crossover with probability.
6      -Apply mutation with probability.
7      -Decode and calculate fitness.
8      -Select survivors.
9      -Find the best solution.
10 return the best solution.
```

1.3.4.2 Particle Swarm Optimization

PSO is a population-based optimization algorithm inspired by the social behavior of bird flocks and fish schools. Introduced in 1995 by Russel Eberhart and James Kennedy [56], PSO mimics the collaborative behavior observed in nature, where individuals, called particles, collectively explore the search space to find optimal solutions. Each particle

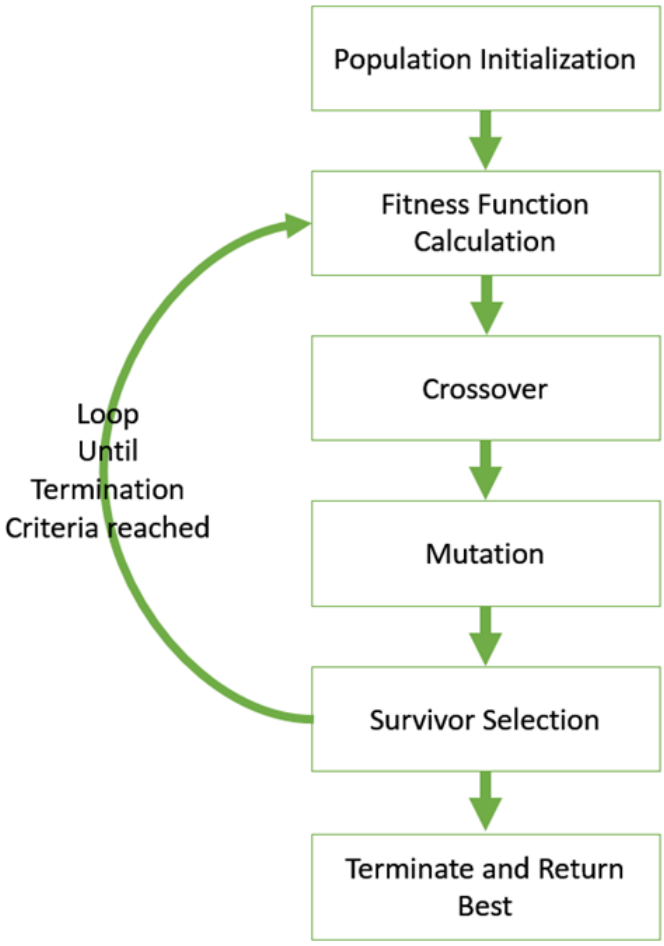

Figure 1.12: Basic structure of the genetic algorithm [68].

represents a potential solution and adjusts its position based on its own experience and that of its neighbors. Through iterative updates, particles converge towards promising regions of the search space, aiming to find the best solution to the optimization problem. The strategy for moving a particle is illustrated in Figure 1.13.

1.3.4.3 Formulation

In an n-dimensional search space, each particle i of the swarm is characterized by its position $X_i = (x_{i1}, x_{i2}, ..., x_{in})$ and a position change vector (called velocity) $V_i = (v_{i1}, v_{i2}, ..., v_{in})$.

The objective function's value decides the position's quality at that specific point. Each particle retains the memory of the best position it has encountered so far, referred to as $pBest_i$ (Personal Best). Additionally, the best position achieved by all particles in the swarm is denoted as $gBest$ (Global Best).

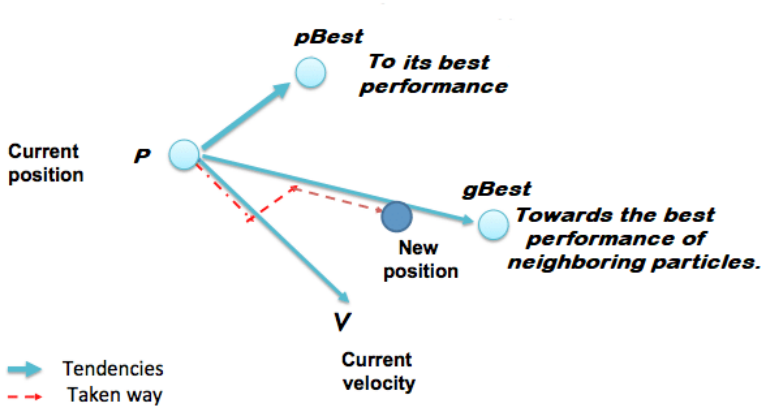

Figure 1.13: The motion of the particle in standard PSO [69].

PSO endeavors to discover the optimal solution to the problem by guiding the particles through movements and assessing the fitness of their new positions. Initially, the particles within the swarm are randomly initialized across the problem's search space. Throughout the algorithm's iterations, each particle seeks improved positions by adjusting its velocity and position. The velocity update depends on three factors: the particle's personal best position, the swarm's best position, and the particle's previous velocity. The velocity vector and position vector are calculated using equations Equation 1.10 and Equation 1.11, respectively.

$$v_{ij}(t+1) = wv_{ij}(t) + c_p r_p(pBest_{ij} - x_{ij}(t)) + c_g r_g(gBest_j - x_{ij}(t)) \tag{1.10}$$

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1) \tag{1.11}$$

Where w is the inertia coefficient that adjusts the balance between the exploration and exploitation phases of the research process, the c_p and c_g parameters control best-fit snapping and global best-snap snapping, respectively, finally, r_p and r_g are uniform random variables drawn from [0, 1].

Once the particles have moved, the new positions are evaluated, and the two vectors $pBesti$ and $gBest$ are updated at iteration $t+1$, according to the two equations Equation 1.12 (in the case of minimization) and Equation 1.13 (in a global version of PSO), respectively. This procedure is presented in algorithm 3, where N is the number of particles in the swarm.

$$pBest_i(t+1) = \begin{cases} x_i(t+1) & \text{if } f(x_i(t+1)) < f(pBest_i(t))) \\ pBest_i(t) & \text{else.} \end{cases} \tag{1.12}$$

$$gBest_i(t+1) = \arg minf(pBest_i(t+1)),\ 0 \leq i \leq N \tag{1.13}$$

Algorithm 3: Basic process of Particle Swarm Optimization (PSO) algorithm.

```
Initialize particles' positions x_i and velocities v_i randomly;
Initialize personal best positions pBest_i = x_i and global best position gBest_i
while maximum iteration not reached or satisfied solution not found do
    for each particle do
        Update particle's velocity according to Equation 1.10;
        Update particle's position according to Equation 1.11;
        Calculate pBest_i according to Equation 1.12
        Calculate gBest_i according to Equation 1.13
Return gBest
```

Improving particle swarms (PSO) is a typical improvement technique that has undergone many variations and adaptations since its inception. These variables aim to address specific challenges, improve performance, or adapt the algorithm to different problem areas. Among them, PSWV is a variant of the PSO algorithm where the velocity of particles is not explicitly computed or updated during optimization. In this approach, particles adjust their positions solely based on their personal best positions and the best global positions (Equation 1.14). By eliminating the velocity component, this variant simplifies the algorithm and reduces computational overhead, making it more efficient for specific optimization tasks. Despite the absence of velocity updates, PSWV can still effectively explore the search space and converge to optimal solutions by leveraging the collective intelligence of the particle swarm.

$$x_{ij}(t+1) = c_1 r_1 pBest_{ij} + c_2 r_2 gBest_j \tag{1.14}$$

r_1 and r_2 represent the combination weights, where r_1 and r_2 are within the range of [0,1], while c_1 and c_2 denote the own and social attraction coefficients, both of which also fall within the range of [0,1].

1.4 Hyperparameter Optimization

Automated hyperparameter optimization (HPO) is a critical aspect of machine learning, particularly in the context of automated machine learning (AutoML). It involves automatically adjusting the hyperparameters of machine learning models to optimize their performance. Hyperparameters are settings that govern the behavior and performance of machine learning algorithms, such as the learning rate in neural networks or the depth of decision trees [16].

The aim of automated HPO is to reduce the need for manual intervention in tuning hyperparameters, which can be time-consuming and require domain expertise. By automating this process, HPO can improve the efficiency and effectiveness of machine learning model development[16].

Automated HPO algorithms typically work by iteratively testing different combinations of hyperparameters and evaluating their performance on a validation dataset. The algorithm then uses this feedback to guide the search towards configurations that lead to better performance [16].

Overall, automated hyperparameter optimization is an important technique in machine learning that helps streamline the model development process and improve machine learning models' performance.

1.4.1 Traditional methods for hyperparameter optimization

Traditional methods for hyperparameter optimization typically refer to approaches not based on advanced optimization techniques or machine learning algorithms. These methods are often straightforward to implement but may not always yield the best results compared to more sophisticated approaches. Some traditional methods include:

1.4.1.1 Grid search

Traditionally, automated hyperparameter optimization has been achieved through methods such as grid search [70] or parameter sweep. Grid search involves exhaustively exploring a manually selected subset of a learning algorithm's hyperparameter space. This process relies on a performance metric, often assessed through cross-validation on the training set or evaluation on a held-out validation set. Figure 1.14 illustrates grid search using two hyperparameters with varying values, resulting in the testing of many combinations.

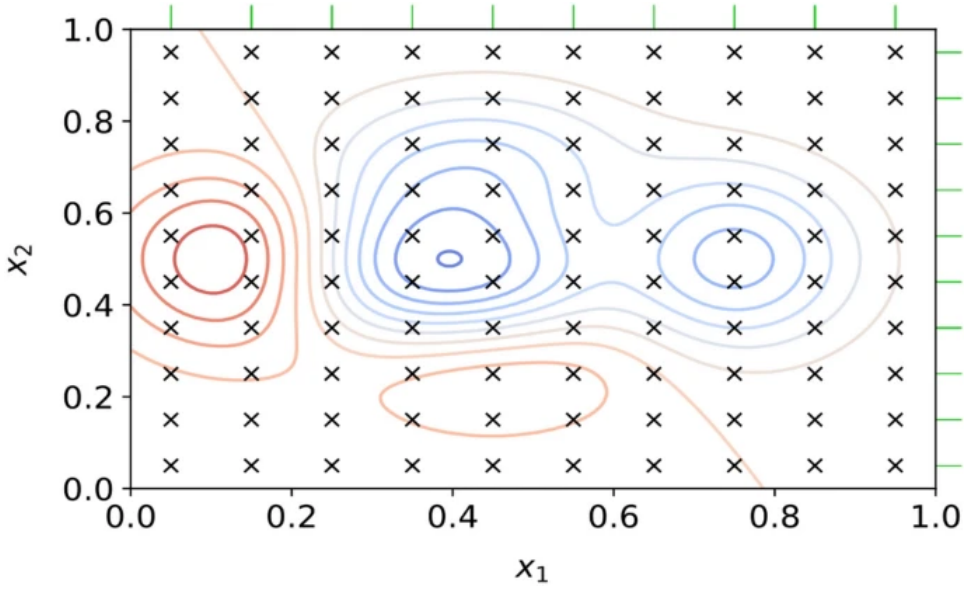

Figure 1.14: Grid search optimization [71].

1.4.1.2 Random Search

Random Search [72] is an alternative hyperparameter optimization approach that replaces the exhaustive enumeration of all combinations with a random selection of hyperparameter settings. It applies not only to discrete hyperparameters but also to continuous and mixed domains. Random Search can outperform Grid search, especially when only a few hyperparameters significantly impact the machine learning algorithm's performance. This method is particularly advantageous when the optimization problem has low inherent dimensionality. Additionally, Random Search is highly parallelizable and allows past information to influence the selection of the distribution from which to sample. Figure 1.15 illustrates a Random Search for two hyperparameters, exploring a diverse set of random combinations.

1.4.1.3 Bayesian optimization

In contrast, Bayesian optimization [73] is a global optimization strategy designed for noisy black-box functions. When applied to hyperparameter optimization, Bayesian optimization constructs a probabilistic model of the function mapping from hyperparameter values to the objective evaluated on a validation set. It seeks to gather observations that reveal as much information about this function as possible, including the location of the optimum. Bayesian optimization repeatedly assesses a potential hyperparameter configuration based on the current model and updates it accordingly. Figure 1.16 illustrates how

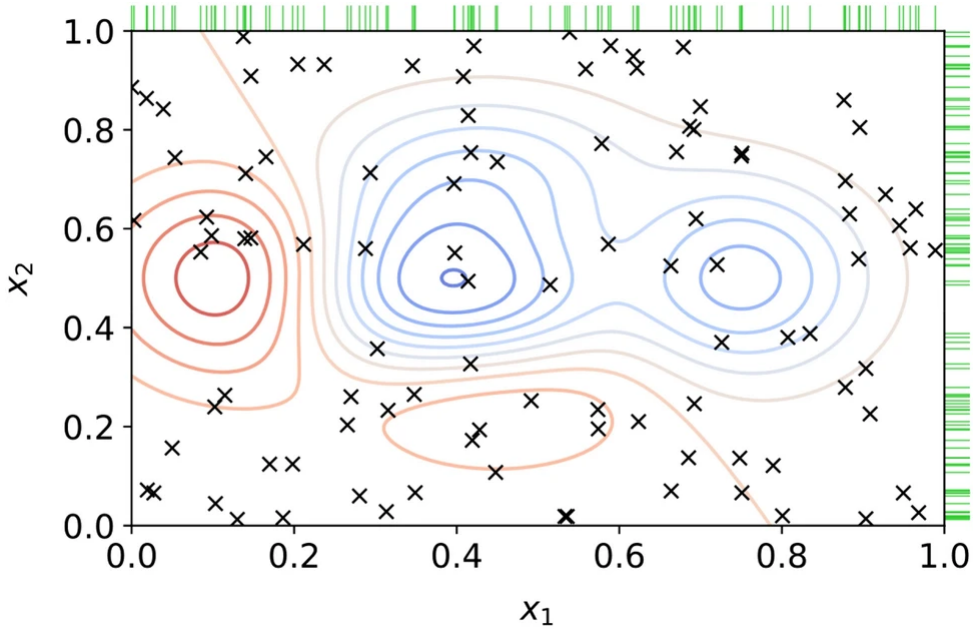

Figure 1.15: Random search optimization [71].

methods like Bayesian optimization intelligently explore the space of hyperparameter options by choosing which combination to investigate next based on past findings.

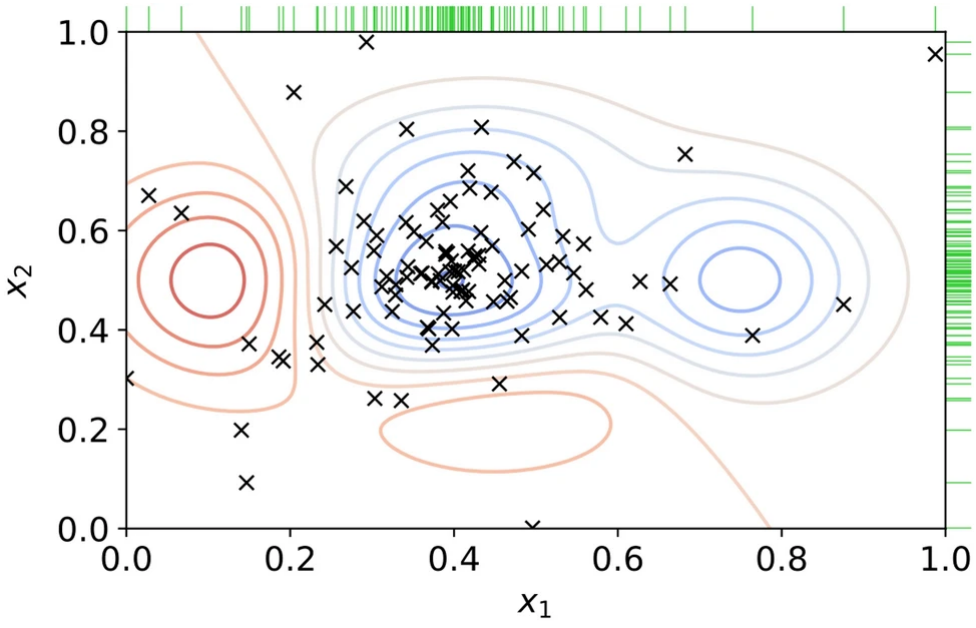

Figure 1.16: Bayesian optimization [71].

1.4.1.4

Search optimization methods are crucial in finding optimal solutions within a given search space. Hyperparameter optimization, in particular, is vital for enhancing the performance of machine learning models, and various approaches, including grid search, random search, Bayesian optimization, and reinforcement learning-based methods, are employed to automate and improve this process.

1.4.2 limitations of traditional optimization methods

Traditional optimization methods, such as grid search, random search, and Bayesian optimization, have several limitations:

1. Grid Search:

- Computational Inefficiency: Grid search systematically evaluates every combination of hyperparameters within predefined ranges, which can lead to a large number of evaluations, especially in high-dimensional hyperparameter spaces. This exhaustive search approach becomes computationally inefficient and time-consuming.

- Limited Exploration: Grid search may not efficiently explore the hyperparameter space, as it follows a fixed grid pattern. It may miss important regions of the space that could potentially yield optimal solutions, leading to suboptimal performance.

- Scalability Issues: As the dimensionality of the hyperparameter space increases, grid search becomes increasingly impractical due to the exponential increase in the number of configurations to evaluate. It may struggle to scale effectively to large-scale optimization tasks.

2. Random Search:

- Ineffective Exploration: While random search samples hyperparameter configurations randomly, it may fail to prioritize regions of the hyperparameter space that are likely to yield optimal solutions. It may overlook important areas or spend excessive resources exploring less promising regions, leading to inefficient exploration.

- Lack of Adaptability: Random search does not adapt its search strategy based on the feedback received from previous evaluations. It may continue to sample configurations randomly without leveraging information gained during the optimization process to guide future search directions.

- No Guarantee of Improvement: Random search does not guarantee improvement in performance with each iteration. It may require a large number of evaluations to discover optimal or near-optimal hyperparameter configurations, especially in complex or high-dimensional search spaces.

3. Bayesian Optimization:

- High Computational Cost: Bayesian optimization can be computationally expensive, especially in high-dimensional or complex hyperparameter spaces. The iterative process of building and optimizing surrogate models of the objective function requires significant computational resources and time.

- Dependency on Initial Data: Bayesian optimization relies on an initial set of observations of the objective function to build an accurate surrogate model. The quality of this initial data can significantly impact the performance of Bayesian optimization, and poor initial data may lead to suboptimal surrogate models and hyperparameter configurations.

- Limited Scalability: Bayesian optimization may struggle to scale effectively to large-scale optimization problems with a large number of hyperparameters or a large dataset size. As the dimensionality of the search space increases, the computational complexity of Bayesian optimization algorithms also increases, making them less practical for such scenarios.

Despite these limitations, each of these optimization methods has its advantages and is suitable for different scenarios. Researchers continue to explore techniques to address these limitations and improve the efficiency and effectiveness of hyperparameter optimization in machine learning.

1.4.3 Metaheuristics for hyperparameters optimization

The problem of finding an optimal CNN architecture can be viewed as a combinatorial optimization problem, where solutions are typically categorized into exact methods and heuristics. While exact methods are computationally expensive as they require considering all possible configurations, designing a heuristic that incorporates all existing knowledge about neural networks is challenging. As an alternative, the machine learning community often prefers metaheuristics. These methods are crafted to tackle a broad spectrum of combinatorial optimization problems without needing specific adjustments for each instance [61].

Metaheuristics, such as Simulated Annealing, Local Search, Particle Swarm Optimization, and Evolutionary Algorithms, among others, are representative approaches employed in this context [61]. A key characteristic of metaheuristics is their ability to strike a balance between exploring the search space (identifying promising regions) and exploiting it (intensifying the search in a promising region) [61]. Consequently, these methods vary in how they execute these two search strategies by adjusting their configuration parameters, often referred to as metaparameters.

It's important to note that due to the "No Free Lunch" theorems, there is no single metaheuristic that outperforms others across all optimization problem instances [61]. Instead, solving each instance independently is recommended to determine the most suitable metaheuristic.

1.4.3.1 Advantages of using metaheuristics for hyperparameters optimization

Metaheuristic algorithms offer several advantages for hyperparameter optimization in deep learning:

1. Global Optimization: Metaheuristic algorithms are specifically developed to effectively navigate and utilize the complete range of hyperparameters, allowing them to discover solutions that are globally optimal or very close to being optimal. Unlike traditional methods such as grid search or random search, metaheuristic algorithms can avoid getting trapped in local optima and instead explore for superior solutions.

2. Flexibility: Metaheuristic algorithms are highly flexible and adaptable to different optimization problems and search spaces. They can handle various types of hyperparameters, including continuous, discrete, and categorical variables, without requiring extensive modifications to the algorithm.

3. Efficient Exploration: Metaheuristic algorithms employ intelligent search strategies that balance exploration and exploitation of the hyperparameter space. They can efficiently explore promising regions of the search space while exploiting regions that show potential for optimal solutions. This efficient exploration helps in discovering better hyperparameter configurations with fewer evaluations.

4. Adaptability: Metaheuristic algorithms can dynamically adapt their search strategy based on the feedback received from previous evaluations. They can adjust parameters such as mutation rates, population sizes, or search operators to improve performance and convergence speed during the optimization process.

5. Handling Constraints: Metaheuristic algorithms can handle constraints on hyperparameters, such as bounds or dependencies between parameters, more effectively than traditional optimization methods. They can incorporate constraints into the optimization process and ensure that solutions meet all specified constraints while searching for optimal configurations.

6. Parallelizability: Many metaheuristic algorithms, such as genetic algorithms and particle swarm optimization, are inherently parallelizable. They can exploit parallel computing architectures to perform multiple evaluations simultaneously, speeding up the optimization process and enabling scalability to large-scale optimization problems.

7. Robustness: Metaheuristic algorithms are robust to noisy or stochastic objective functions and can handle uncertainty in the optimization process. They can effectively cope with noisy evaluations and converge to stable solutions even in the presence of randomness or fluctuations in the objective function.

Indeed, metaheuristic algorithms offer a powerful and versatile approach to hyperparameter optimization, providing efficient and effective solutions to various machine learning optimization problems. These optimization techniques can efficiently explore large search spaces, making them well-suited for hyperparameter optimization tasks where the search space is complex or high-dimensional.

1.5 Experimental Datasets

To comprehensively evaluate the effectiveness of our methods, we select all variants of the Mixed National Institute of Standards and Technology (MNIST) dataset: MNIST [74], MNIST-RD [75], MNIST-RB [75], MNIST-BI [75], and MNIST-RD+BI [75]. Additionally, to compare the robustness of our proposed `pswvCNN` algorithm with that of competitors, we conduct experiments on other datasets such as Rectangles [75], Rectangles-I [75], Convex [75], and MNIST-Fashion [76]. For a detailed description of these image datasets, refer to Table 1.3.

1.5.1 MNIST Dataset

The MNIST dataset, originally introduced by LeCun et al. in 1998 [74], is a widely recognized collection of images often used as a benchmark in machine learning. It comprises handwritten digits and serves as a fundamental resource for evaluating various machine learning models. The dataset consists of 70,000 images, with 60,000 images

allocated for training and an additional 10,000 reserved for testing purposes. These images are grayscale and have dimensions of 28 pixels by 28 pixels. To ensure compatibility across different machine learning applications, the MNIST images have been normalized. The MNIST dataset is commonly employed as an initial step in developing image recognition models and has become a standard benchmark for assessing the performance of such models. Figure 1.17 illustrates examples of images in the MNIST dataset.

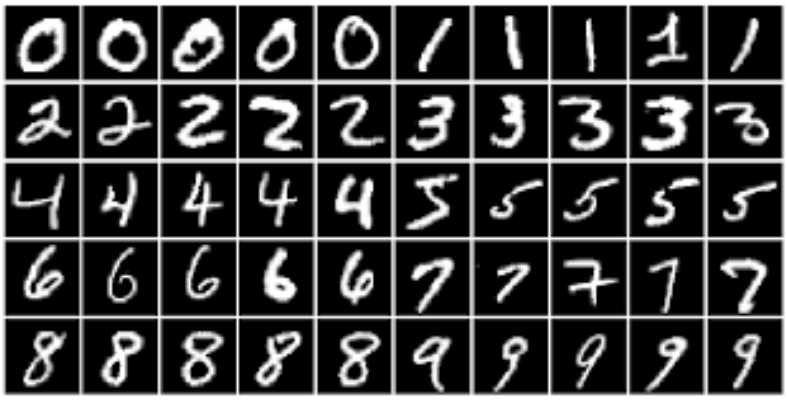

Figure 1.17: Sample images for the MNIST dataset.

1.5.2 Dataset Overview

Table 1.3: Overview of Datasets Utilized in the Experiments.

Dataset	Description	Input Size	Classes	Training-Test
MNIST	MNIST with basic digits	28 x 28 x 1	10	60,000-10,000
MNIST-BI	MNIST with background images	28 x 28 x 1	10	12,000-50,000
MNIST-RD+BI	MNIST with rotated digits and image background	28 x 28 x 1	10	12,000-50,000
MNIST-RD	MNIST with rotated digits	28 x 28 x 1	10	12,000-50,000
MNIST-RB	MNIST with random background	28 x 28 x 1	10	12,000-50,000
Convex	Distinguish between convex and concave shapes	28 x 28 x 1	2	8,000-50,000
Rectangle-I	Rectangles with image background	28 x 28 x 1	2	12,000-50,000
Rectangles	Distinguish between tall and wide rectangles	28 x 28 x 1	2	1,200-50,000
MNIST-Fashion	Zalando's article images	28 x 28 x 1	10	60,000-10,000

1.5.3 MNIST Variants

In our evaluation, we utilize a variety of datasets, including different variants of MNIST, to assess the performance of our models. These variant datasets pose additional challenges due to distraction factors.

1.5.3.1 MNIST-RD

By randomly rotating the digits at an angle uniformly selected from 0 to 2π radians, we introduce two key sources of variation. These sources are the rotation angle itself and the inherent factors of variation already present in MNIST, such as various handwriting styles. Figure 1.18 shows sample images from the MNIST-RD dataset.

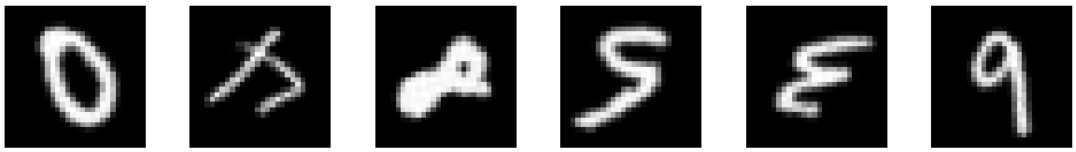

Figure 1.18: Sample images for the MNIST-RD dataset.

1.5.3.2 MNIST-RB

MNIST with random background. It includes handwritten digits with random background patterns. Like the other MNIST variants, it has 10 classes and a varying dataset size of 12,000 to 50,000. Refer to Figure 1.19 for visual examples.

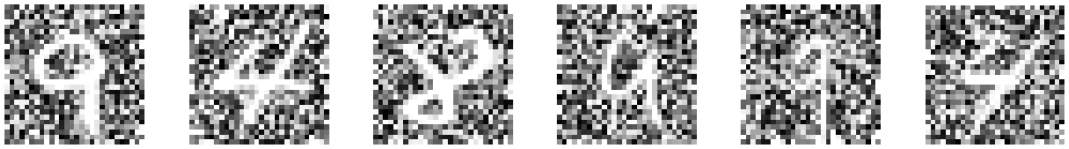

Figure 1.19: Sample images for the MNIST-RB dataset.

1.5.3.3 MNIST-BI

MNIST with background images. It includes handwritten digits with random background images. Similar to the original MNIST, it contains 10 classes with images of the same dimensions. The dataset size ranges from 12,000 to 50,000. Figure 1.20 shows sample images.

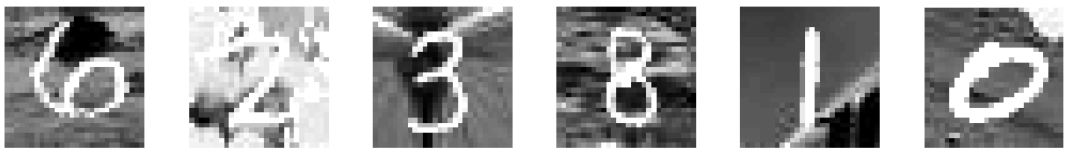

Figure 1.20: Sample images for the MNIST-BI dataset.

1.5.3.4 MNIST-RD+BI

The MNIST-RD+BI dataset extends the traditional MNIST dataset by incorporating rotated digits and random background images. This variation introduces additional complexity, making it suitable for evaluating more advanced machine learning models. The dataset consists of 10 classes, similar to the original MNIST, with the number of samples varying between 12,000 and 50,000. Refer to Figure 1.21 for visual examples.

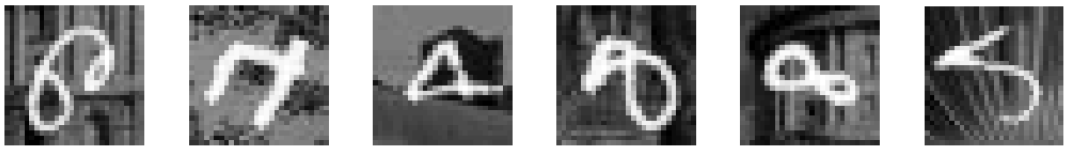

Figure 1.21: Sample images for the MNIST-RD+BI dataset.

1.5.4 Additional Datasets

In addition to the MNIST variants, we include other datasets for comparative analysis:

1.5.4.1 Convex

This dataset involves discriminating between convex and concave shapes. It contains grayscale images with dimensions of 28 pixels by 28 pixels and has 2 classes. The dataset size ranges from 8,000 to 50,000. Refer to Figure 1.22 for visual examples.

Figure 1.22: Sample images for the Convex dataset.

1.5.4.2 Rectangle-I

Rectangles with image background. It includes images of rectangles with random background images. The dataset comprises grayscale images with dimensions of 28 pixels by 28 pixels, with 2 classes. The dataset size ranges from 12,000 to 50,000. Refer to Figure 1.23 for visual examples.

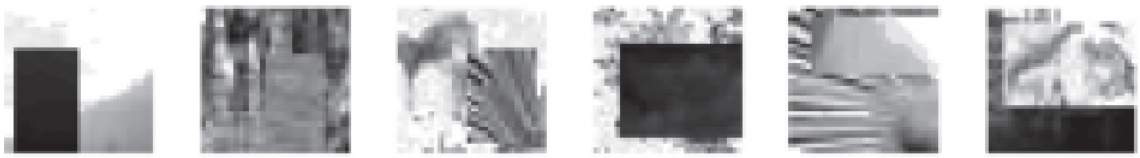

Figure 1.23: Sample images for the Rectangle-I dataset.

1.5.4.3 Rectangles

Discriminating between tall and wide rectangles. This dataset involves grayscale images of rectangles, where the task is to distinguish between tall and wide rectangles. It consists of 2 classes and varies in size from 1,200 to 50,000 images. Refer to Figure 1.24 for visual examples.

Figure 1.24: Sample images for the Rectangles dataset.

1.5.4.4 MNIST-Fashion

The MNIST-Fashion dataset, provided by Zalando, includes images of fashion items from their article collection. This dataset serves as a more challenging alternative to the original MNIST dataset, with the same format of grayscale images of 28x28 pixels. It consists of 10 classes, each representing a different category of fashion items. The dataset comprises 60,000 images for training and an additional 10,000 images for testing. Refer to Figure 1.25 for visual examples of the dataset.

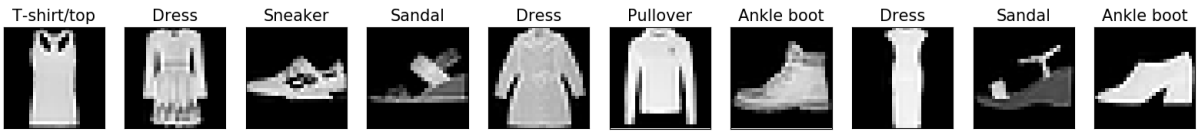

Figure 1.25: Sample images for the MNIST-Fashion dataset.

1.5.5 Conclusion

By evaluating our models on these diverse datasets, we can gain insights into their performance across various challenges and domains, enabling us to compare their effectiveness

and suitability for different tasks.

1.6 Evaluation metrics

The assessment of classification performance and the guidance for building a classifier heavily depend on the evaluation methods used. The classification process typically involves three main phases: training, validation, and testing.

Training Phase: During the training stage, a classification model undergoes training using a set of input patterns, also called training data. Throughout this phase, adjustments are made to the model's parameters to better align with the training data. In contrast, the training error is assessed to determine the model's fit with this data. It's important to note that the training error typically appears smaller compared to errors encountered in the validation and testing phases, as the model is specifically optimized to perform well on the same data it was trained on.

Testing Phase: The primary objective of a learning algorithm is to leverage insights gained from the training data to generate accurate predictions for new, unseen data. This is where the testing phase becomes crucial. However, accurately estimating the testing error or out-of-sample error can pose challenges due to the absence of class labels or outputs for the testing samples. Therefore, this phase assesses the model's ability to generalize to new, unseen data.

Validation Phase: The validation phase is a crucial step in evaluating the performance of a trained model. It serves the purpose of assessing how well the model is likely to perform on new, unseen data while also fine-tuning its hyperparameters. In this phase, a separate dataset called the validation dataset is used. The model's performance on this dataset provides an unbiased evaluation, helping to avoid overfitting (where the model fits the training data too closely and does not generalize well) and guiding the selection of hyperparameters that optimize the model's performance.

In summary, the classification process involves training a model on a dataset, using a separate validation dataset to assess performance and make adjustments, and then testing the model on a separate dataset to evaluate its ability to generalize to new, unseen data. The training phase tailors the model to fit the training data, while the validation and testing phases ensure that the model can perform well on data it has not seen before, with the validation phase also assisting in hyperparameter tuning.

Classification problems can be divided into two types based on the number of classes involved: binary classification, where there are only two classes, and multi-class classification, where the number of classes exceeds two. In binary classification, the classes are typically labeled as positive (P) and negative (N). An unknown sample is then categorized as either P or N. The classification model, developed during the training phase, is used to predict the true classes of unknown samples. The model produces either continuous or discrete outputs. The discrete output indicates the predicted class label for the unknown/test sample, while the continuous output provides an estimate of the sample's likelihood of belonging to a particular class.

Figure 1.26 illustrates four possible outcomes that correspond to the elements of a 2x2 confusion matrix or a contingency table. The green diagonal represents accurate predictions, while the pink diagonal indicates incorrect predictions. Here's how these outcomes are defined:

(1) **True Positive (TP)**: When a sample is positive and correctly classified as positive, it is counted as a true positive. In other words, it's a correct identification of a positive sample.

(2) **False Negative (FN)**: If a positive sample is classified as negative, it is considered a false negative, which is also known as a Type II error. This means the model failed to recognize a positive sample.

(3) **True Negative (TN)**: When a negative sample is accurately classified as negative, it's termed a true negative. It represents the correct identification of a negative sample.

(4) **False Positive (FP)**: If a negative sample is classified as positive, it is counted as a false positive, a false alarm, or a Type I error. This indicates that the model incorrectly labeled a negative sample as positive.

Figure 1.27 illustrates a confusion matrix for a multi-class classification problem involving three classes: A, B, and C. Let's break down the elements and their meanings:

* TP_A **(True Positives for Class A)**: This represents the count of samples from Class A that were correctly classified as Class A. In other words, it shows how many samples from Class A were accurately identified.

* E_{AB} **(Errors from Class A to Class B)**: This value indicates the number of samples from Class A that were mistakenly classified as Class B, meaning they were misclassified as belonging to Class B instead of Class A.

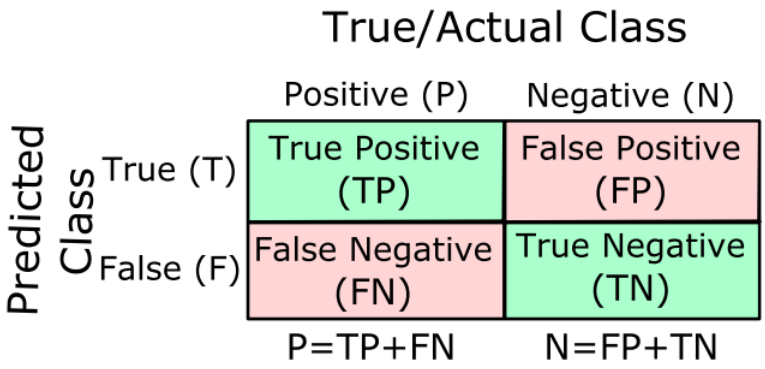

Figure 1.26: An example illustrating the 2x2 confusion matrix.

* FN_A **(False Negative in Class A)**: The false negatives for Class A can be calculated by adding E_{AB} and E_{AC}. In simpler terms, FN_A is equal to $E_{AB} + E_{AC}$. This quantity tells us how many Class A samples were incorrectly classified as either Class B or Class C.

* **False Negative for Any Class**: The concept of false negatives applies to other classes as well. The false negative (FN) for any class located in a specific column can be computed by summing up the errors within that column. This informs us about the number of samples from that class that were incorrectly classified as belonging to other classes.

* **False Positive for Any Predicted Class**: Conversely, the false positive (FP) for any predicted class located in a particular row represents the total of all errors within that row. It quantifies how many samples were predicted as belonging to that class but, in reality, belonged to other classes.

For instance, to calculate the false positive for Class A (FP_A), you would sum up $FP_A + E_{BA} + E_{CA}$. This accounts for all cases where samples were predicted as Class A but were actually from Class B or Class C.

In a confusion matrix with dimensions mxm (where m is the number of classes), there are m correct classifications along the diagonal (true positives), and there are a total of $m^2 - m$ possible errors. This matrix serves as a valuable tool for evaluating the performance of a multi-class classification model, as it quantifies both correct and incorrect predictions for each class[77].

Accuracy (Acc) is one of the most commonly used measures for the classification performance, and it is defined as a ratio between the correctly classified samples to the total number of samples as follows [78]:

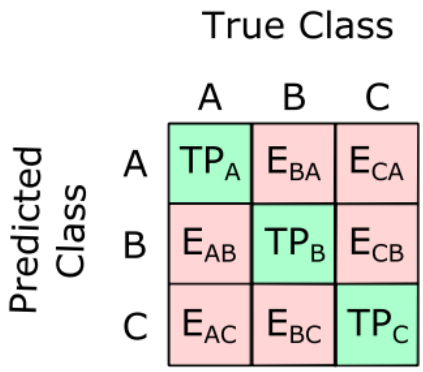

Figure 1.27: An illustrative demonstration of the multi-class confusion matrix.

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{1.15}$$

The error rate (ERR) or misclassification rate serves as the counterpart to the accuracy metric. It quantifies the number of misclassified samples across both positive and negative classes. This rate is determined using Equation 1.16 [79]:

$$ERR = 1 - Acc = \frac{FP + FN}{TP + TN + FP + FN} \tag{1.16}$$

Both accuracy and error rate metrics are affected by imbalanced data. Furthermore, accuracy poses a challenge as two classifiers may achieve the same accuracy while exhibiting different performance in terms of correct and incorrect decisions [80]. However, Takaya Saito and Marc Rehmsmeier noted that accuracy remains appropriate for imbalanced data, as they found identical accuracy values for both balanced and imbalanced datasets in their research [81]. This uniformity in accuracy values was observed because the combined sum of true positives (TP) and true negatives (TN) in both balanced and imbalanced datasets was consistent.

1.7 Conclusion

Overall, the literature review chapter comprehensively analysed the existing research on deep learning and metaheuristic algorithms. It highlighted the progress made in these fields, identified gaps and limitations in the current literature, and set the stage for the subsequent thesis chapters. The findings from this review will inform the development of

new approaches and methodologies to advance further integration of deep learning and metaheuristics in future research.

CHAPTER 2

OPTIMIZING CNN ARCHITECTURE USING A MODIFIED PSO

Contents

2.1 Introduction

In recent studies, metaheuristic algorithms have gained attention for addressing high-dimensional and nonconvex optimization problems, particularly in the context of optimizing hyperparameters in convolutional neural networks (CNNs). These modern optimization techniques, known as metaheuristics, have demonstrated success in solving optimization problems across various domains, including science, engineering, and industry. Their effectiveness in tackling CNN hyperparameter optimization problems has sparked considerable interest. Several researchers have successfully automated the evolution of Convolutional Neural Network (CNN) designs using: GA [17–20], GP [22], PSO [17, 24–27, 27–30], SOS [31], MBO [32], and FA [33].

This study focuses on the automated design of CNN architectures using metaheuristic algorithms. The effectiveness of these techniques is assessed by evaluating them on widely recognized benchmark datasets.

IPPSO [24] is a notable approach that utilizes Particle Swarm Optimization (PSO) to address the challenges associated with CNN architecture. The IPPSO model incorporates adaptive coding techniques that draw inspiration from IP addressing and subnet research in computer networks. In this particular situation, the parameters of a layer are expressed as a consecutive series of binary digits (0s and 1s), similar to an IP addressing strategy.

The IPPSO algorithm is implemented using a population of 20 particles, where each particle represents a potential CNN architecture. The maximum length of layers within a particle is set to 9, indicating the maximum number of layers that can be included. Additionally, the maximum number of fully-connected layers allowed in a particle is limited to 3.

To evaluate the performance of the trained CNN architectures, the IPPSO algorithm employs a training process involving a predefined number of training epochs, specifically 10. Once the training phase is completed, the trained CNN architectures are evaluated based on their learned representations and performance.

The IPPSO model offers a novel approach to optimizing CNN architectures for various computer vision tasks. It incorporates PSO and adaptive coding techniques inspired by IP addressing.

psoCNN [25] is an innovative method for generating advanced CNN architectures using PSO. It uses a test methodology with 20 particles, where each particle represents a potential CNN architecture. The particle layer length is limited to 3-20 layers, and

the kernel size ranges from 3x3 to 7x7. The maximum number of channels allowed is 256. psoCNN employs a traditional PSO algorithm to optimize the architecture by modifying the particle's position. This modification involves randomly selecting and copying layers from either the best personal or global solutions. With PSO and the defined test methodology, psoCNN aims to generate state-of-the-art CNN architectures that excel in various computer vision tasks. It achieves this by dynamically adapting the particle's architecture through the PSO process, efficiently exploring the search space to discover effective configurations for high-performance CNN models.

Li et al. proposed an evolution technique using quantum-behaved particle swarm optimization (BQPSO) in [26] to optimize the architecture of Convolutional Neural Networks (CNNs). Unlike traditional PSO, BQPSO does not utilize velocities and trajectories but focuses on position and distance. Each particle in the population, which represents a CNN architecture, is encoded using a fixed-length binary string. Different components of the CNN, such as convolutional layers, max-pooling, and fully-connected layers, are encoded with their respective parameters. These binary strings are concatenated to form the particle representation. The population is initialized with randomly generated binary strings. Fitness evaluation involves running the CNN model with the selected parameters, and the solution is evolved using PSO.

Lawrence et al. (2021) presented a new technique for autonomously improving convolutional neural networks (CNNs) for image categorization using particle swarm optimisation (PSO) [29]. Their methodology employs a group-based coding scheme, wherein each group consists of at least one convolutional layer and a pooling layer serving as the final layer. The authors provide a novel approach to update velocity and position by integrating weighted variables. This approach allows for exploration of the search space and the generation of varied solutions. The efficacy and competitiveness of the suggested approach were assessed on well-known image classification datasets. The experimental results have shown that it is superior to the most advanced approaches now available. This paper makes a substantial contribution to the field of automated deep learning by demonstrating the potential of Particle Swarm Optimisation (PSO) in evolving Convolutional Neural Networks (CNNs) for picture classification.

In [82], This paper introduces a parallel approach using Particle Swarm Optimization (PSO) for selecting hyperparameters in Deep Neural Networks (DNNs). The method evolves a population of particles and evaluates their fitness in parallel to identify the optimal hyperparameters. Experimental results demonstrate the scalability of the ap-

proach across different DNNs. The study showcases the effectiveness of parallel PSO in optimizing pre-existing expert-designed models within a feasible timeframe.

In the paper [31], The sosCNN algorithm presents an innovative approach to streamline the design process of convolutional neural network (CNN) architectures. Traditional methods of CNN design typically entail manual iterations, necessitating adjustments based on data characteristics and domain expertise. However, this manual approach is often time-consuming, labor-intensive, and may result in architectures lacking in generalizability. To overcome these challenges, sosCNN harnesses the power of a SOS algorithm, renowned for its robust global optimization capabilities. By integrating the SOS algorithm, sosCNN facilitates the automated generation of CNN architectures, thereby minimizing the necessity for manual intervention. Moreover, sosCNN introduces a distinctive integrated coding update technique to reduce loss within convolutional layers. This method enhances the feature extraction prowess of the generated architectures, ultimately leading to more efficacious CNN designs. Additionally, sosCNN incorporates three innovative non-numeric computational strategies—binary segmentation, slack gain, and dissimilar mutation—into the SOS algorithm. These strategies bolster the algorithm's capacity to explore the search space for optimal CNN architectures, enhancing its efficiency and effectiveness.

In their study, Singh et al. (2021) proposed the MPSO-CNN method, which combines a Multi-level Particle Swarm Optimisation (MPSO) algorithm with a Convolutional Neural Network (CNN) to simultaneously determine the architecture and hyperparameters of the network. This unique approach was introduced in their paper titled "Hybrid MPSO-CNN Algorithm" [27]. This approach employs numerous swarms operating at two distinct levels: the first level focuses on optimizing the architecture, while the second level focuses on optimizing the hyperparameters. A sigmoid-like inertia weight is utilised to improve both exploration and exploitation. The proposed strategy exhibits encouraging outcomes, showcasing a novel method for optimising CNNs.

The work [30] introduces the IntelliSwAS approach for optimizing deep neural network architectures, specifically for tasks involving classification or regression. While the approach focuses on image classification, it is versatile enough to be applied to other classification or regression problems. The proposed method utilizes a particle swarm-based optimization algorithm to automatically search for convolutional neural network architectures. To enhance the search process, a machine learning model called DAGRNN is introduced. DAGRNN is designed to predict the quality of network architectures, thereby improving the overall performance of the algorithm. Notably, the proposed

model has the capability to process data structured as directed acyclic graphs, making it applicable to network architectures in particular.

The paper [83] highlights the challenges novice users face in selecting appropriate neural network architectures and hyperparameters, often resulting in default settings. It emphasizes the potential for significant improvements in model accuracy by evaluating multiple architectures facilitated by Neural Architecture Search (NAS). The paper introduces OpenNAS, a system developed for image classification, which integrates open source tools to automatically generate Convolutional Neural Network (CNN) architectures using various metaheuristics such as AutoKeras, transfer learning, and Swarm Intelligence (SI) approaches like Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO). The research focuses on training and optimizing CNNs using the SI components of OpenNAS, particularly comparing PSO and ACO. Experimental results demonstrate the superiority of PSO over ACO in generating higher model accuracies, which is particularly evident with complex datasets. Additionally, the paper mentions the possibility of combining models developed through metaheuristics using stacking ensembles.

Therefore, the PSO algorithm has several benefits. Compared to other methods, population-based metaheuristics have a comparatively minimal number of parameters and are intuitive and easy to implement. Particle Swarm Optimisation (PSO) allows for the optimization of many types of objective functions, such as numerical and symbolic functions and algorithms, across any number of dimensions using variables of different types.

The PSO algorithm is disadvantaged by its tendency to rapidly converge to a local optimum, exhibiting a low convergence rate during the iterative process, and experiencing a loss of population diversity [84].

To tackle these concerns, a new adaptive PSWV is introduced. This variant incorporates improvements not just to the algorithm's variety but also to avoid particles from getting trapped in local optima. The simulation results clearly show that this option successfully addresses the issue of premature convergence. Furthermore, when compared to other algorithms, it demonstrates a higher rate of convergence. [85].

The objective of this project is to identify a CNN architecture for image classification that achieves a compromise between search speed for appropriate hyperparameters and classification accuracy. In order to tackle this issue, we suggest employing a unique variation of Particle Swarm Optimisation (PSO) termed PSWV, which does not use a velocity equation, in addition to a fresh particle update strategy. The approach is referred to as pswvCNN. pswvCNN is a specialized tool that aims to optimize the

hyperparameters of Convolutional Neural Networks (CNNs). These hyperparameters determine the structure of the CNN and, as a result, have a significant impact on the accuracy of classification.

2.2 The proposed algorithm

This section offers an in-depth review of the pswvCNN algorithm's main components.

2.2.1 Overview of the algorithm

The proposed pswvCNN method utilizes the PSWV (Particle Swarm Optimization without velocity equation) algorithm to design CNN architectures. The framework of this method is described in algorithm 4. The detailed flowcharts outlining the implementation of the pswvCNN Algorithm for optimizing CNN architecture are presented in Figure 2.1, which consists of three main processes:

(1) **itialization of Parameters**: Initially, all input parameters of the algorithm related to the specific problem are set. This includes defining the dataset used for training and configuring the parameters for the CNN structures to be generated.

(2) **Population Initialization**: In this step, a population of particles is created. Specifically, a total of pop_{size} particles are generated randomly. Each particle represents a potential CNN architecture. The number of layers in each particle is chosen randomly. Additionally, the positions of the particles, as well as their best personal positions ($pBest$) and the best global position ($gBest$), are initialized.

(3) **Optimization Process**: In each iteration of the optimization process, every particle updates its position based on its own best position ($pBest$) within the search space and the best position across the entire population ($gBest$). This update is done according to the strategy outlined in section 2.2.5. The iteration continues until a predefined stopping criterion is met (in this case, it is the number of iterations).

(4) **Termination or Continuing Optimization**: The algorithm checks if the maximum number of iterations has been reached. If the limit is reached, the solution represented by $gBest$ is considered the best CNN architecture for this particular dataset.

(5) **Iteration Continuation**: If the maximum iteration count has not been reached, the process returns to the third step and continues optimizing the CNN architectures.

Algorithm 4: Proposed pswvCNN

input : All parameters, Population size (pop_{size}), Maximum number of iterations nb_{itrs}, Training data D_{train}, epo_{eva};

output : The best CNN model with its architecture and selected hyperparameters;

- Initialize the swarm with a random combination of hyperparameters from the Table 2.1;
- Train the CNN model represented by each particle in the population with the number of epochs =epo_{eva} and compute its fitness score.
- Initialize $gBest = p_1$
- **for** *each particle* p_i *in the population* **do**
  - Initialize $pBest_i$ with a copy of p_i
  - Calculate $fitness(p_i.pBest)$, $fitness(p_i)$
  - **if** $fitness(gBest) \leq fitness(p_i)$ **then**
    - $gBest = p_i$
- **while** *Iteration criteria are not reached* nb_{itrs} **do**
  - **for** *each particle* p_i *in the population* **do**
    - updateParticle(p_i)
    - Calculate fitness(p_i)
    - **if** $fitness(p_i) \geq fitness(p_i.pBest)$ **then**
      - $p_i.pBest = p_i$
    - **if** $fitness(gBest) \leq fitness(p_i.pBest)$ **then**
      - $gBest = p_i.pBest$
- Save gBest and use it to reconfigure the best CNN architecture
- Train the best CNN architecture with the training set and epochs = epo_{train}
- Testing the best model with the test data set

In summary, this algorithm employs a population of particles, each representing a potential CNN architecture. Through iterative optimization, it seeks to discover the best CNN architecture for the given dataset and problem. The final solution is determined by the particle represented by $gBest$, which represents the optimal CNN architecture.

2.2.2 The strategy of particle encoding

Our method employs a comprehensive encoding scheme that incorporates multiple factors, such as the number of convolutional layers, pooling layers, and fully-connected layers, as well as their respective hyperparameters. This encoding accurately represents the complete structure of the network, including the sequence of layers and the precise values of hyperparameters for each layer. It plays a crucial role in our process, as it

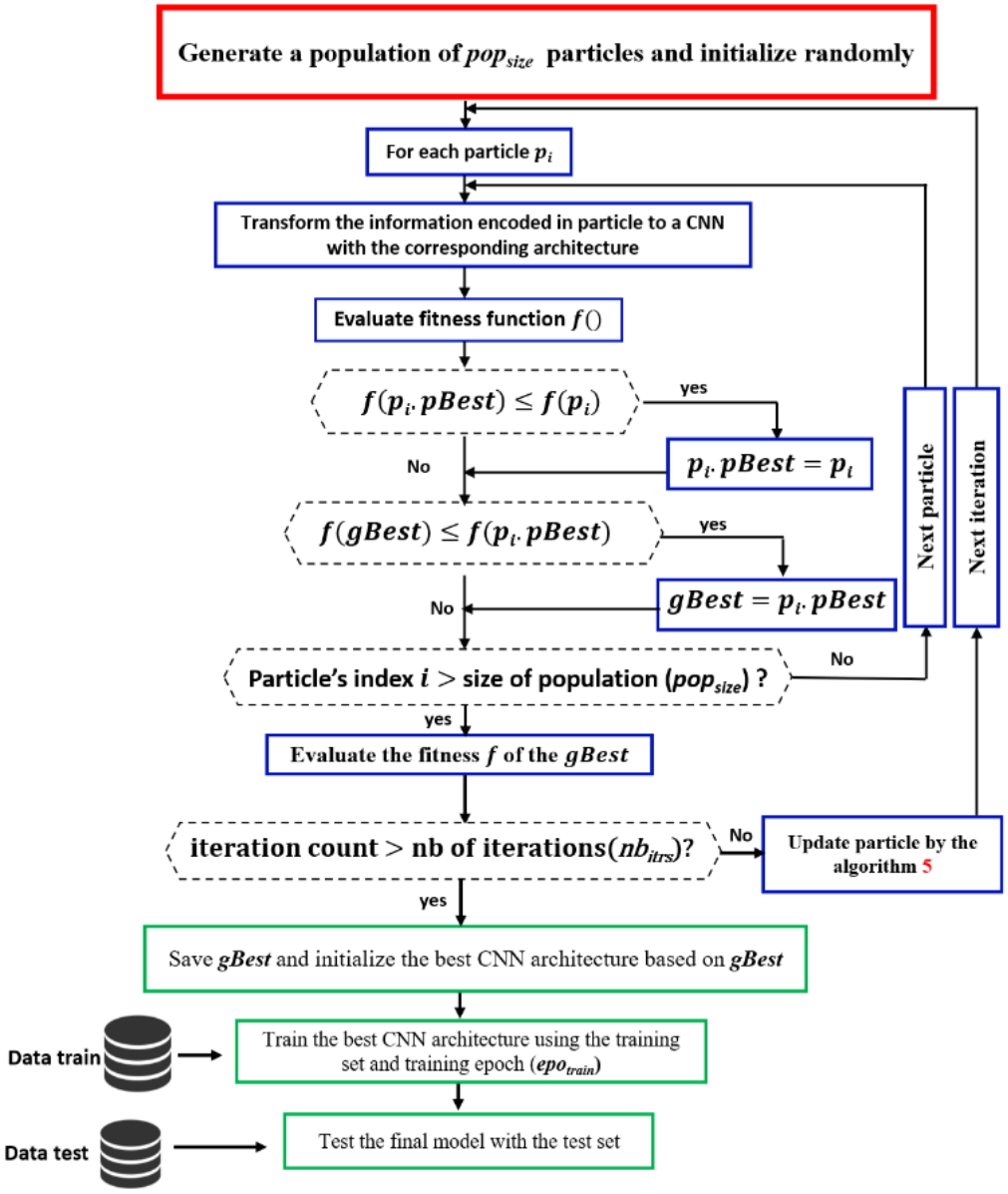

Figure 2.1: Flowchart representing the pswvCNN algorithm as proposed.

significantly impacts the efficiency and effectiveness of our approach. Moreover, this encoding establishes the range of possibilities in which the optimization problem is examined.

Within our method, we focus on adjusting specific hyperparameters. These adjustments primarily pertain to the network structure, which is characterized by parameters such as the number of layers, kernel size, number of feature maps, and number of neurons

in a fully-connected layer. Figure 2.2 illustrates Three different particles utilized in the pswvCNN algorithm, highlighting the random selection of the number of layers, which falls within a predefined range of three to len_{CNN}. Each particle incorporates three types of layers: convolutional, pooling, and fully-connected layers.

Overall, the encoding employed in our method plays a crucial role in capturing the architecture and hyperparameters of the convolutional neural network (CNN). By adjusting these parameters within the defined search space, we enhance the effectiveness of our approach and enable the exploration of various network configurations.

To construct CNN architectures, they all initiate with convolutional layers and conclude with fully-connected (FC) layers. Convolutional layers employ filters in a randomly assigned pattern, resulting in output feature maps ranging from 1 to $nb_{filters}$. The kernel size for these convolutions varies randomly from 33 to $Conv_{kernel}Conv_{kernel}$ (7 x 7), where $Conv_{kernel}$ denotes the maximum convolutional kernel size, with identical padding and stride parameters. Following each convolutional layer is a batch normalization operation [86] and the Rectified Linear Unit (ReLU) activation function [87]. With the exception of the initial layer, each convolutional layer in the CNN is preceded by a dropout layer. A dropout rate of 50% is considered near-optimal for preventing overfitting issues [13].

For the pooling layer, we assume a pooling layer with a kernel size of $Pool_{kernel}$ x $Pool_{kernel}$ and stride of $2x2$. The number of pooling layers, num_{pool}, must satisfy the rule $num_{pool} \leq log_2 dim$, where dim denotes the size of the input image.

It also indicates the correctness of CNN's structure when all the fully-connected layers (Fc) are present at the end of the structure; they cannot be placed between the layers of convolutional and pooling layers or at the beginning of the architecture.

2.2.3 Population Initialization

This research proposes the pswvCNN algorithm to initiate the swarm by generating a population of particles, each representing a CNN architecture, with a population size denoted as pop_{size}. These architectures are randomly created, with the number of layers varying from 3 layers to a maximum specified number, len_{CNN}. Certain constraints are enforced to ensure the practicality of the generated CNN architectures.

Primarily, each particle commences with a convolutional layer as its first layer and concludes with a fully-connected layer. This setup guarantees that the architecture

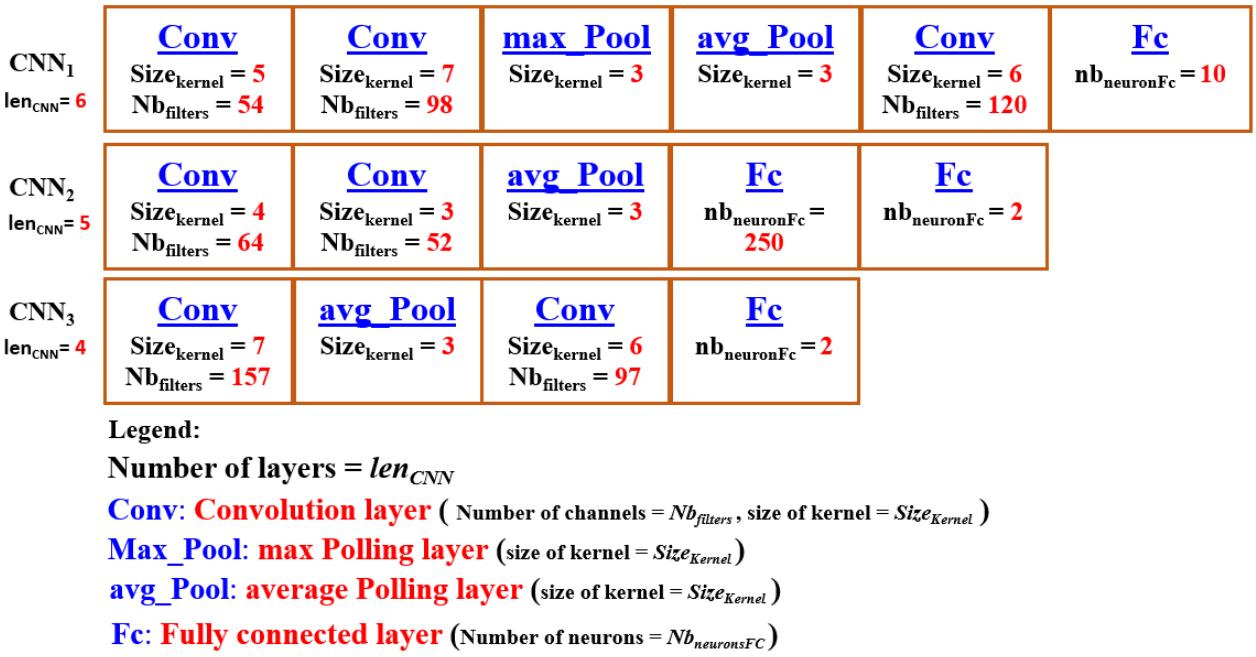

Figure 2.2: Particles in the proposed pswvCNN.

initiates with a convolutional operation and terminates with a layer suitable for regression or classification tasks. Additionally, fully-connected layers are prohibited from being situated between convolutional or pooling layers; they are only allowed towards the end of the architecture. This constraint ensures that once a fully-connected layer is introduced, all subsequent layers in the architecture are also fully-connected layers.

Furthermore, the algorithm considers the number of pooling layers employed in the architecture. As the outputs of pooling layers decrease in size, they can efficiently serve as inputs to the fully-connected layers without necessitating an excessive number of neurons. This facilitates more efficient network designs.

By incorporating these constraints and considerations, the pswvCNN algorithm ensures that the CNN architectures produced are practical and follow a coherent layer progression, thereby enhancing the effectiveness of the swarm optimization process.

Incorporate convolutional layers into each particle, with varying filters randomly chosen from a range of 1 to $nb_{filters}$. The kernel size for each convolutional layer is randomly set from $3x3$ to $Conv_{kernel}xConv_{kernel}$; the strides are always $1x1$; additionally, apply the Rectified Linear Unit (ReLU) activation function to all convolutional layers. Include either a max pooling layer or an average pooling layer with a kernel size of $Pool_{kernel}xPool_{kernel}$, where $Pool_{kernel} = 2$ or 3, and a stride of $2x2$ to the initial

particle configuration. Incorporate a fully-connected layer (Fc) into the particle with a variable number of neurons, ranging from 1 to $nbneuronFc$.

The number of neurons in the final fully-connected layer in our technique is dictated by the number of classes present in the dataset being utilized. The value is determined to be equal to the number of classes present in the dataset. For example, in the case of the MNIST dataset, which consists of ten different digits, the final fully-connected layer would have ten neurons. Similarly, for the Convex dataset, a binary classification problem with two classes (convex and non-convex), the final fully-connected layer would have two neurons. This ensures that the network can accurately predict each class in the dataset.

2.2.4 Fitness Evaluation

Like all metaheuristic algorithms, our approach aims to achieve the primary objective of minimizing errors within the shortest possible time. The fitness evaluation process of our proposed algorithm is detailed in algorithm 5. We extract the encoded architecture information from each particle within the population and utilize it to construct a corresponding CNN. Subsequently, this CNN undergoes training on the provided training dataset using a specified number of epochs to evaluate the particle's performance.

To assess the fitness of each particle, we utilize the categorical cross-entropy loss function as the fitness criterion, coupled with the Adam optimizer [88]. The network weights are initialized using Xavier initialization [89]. Moreover, we incorporate dropout, batch normalization, and regularization techniques to mitigate the risk of overfitting [13, 86]. These strategies aid in enhancing the generalization capability of the network and improving its performance on unseen data.

Algorithm 5: Fitness Evaluation

Input : The population, training data D_{train}, epo_{eva}
Output : The population and their respective fitness values.

for *each particle* p_i **do**
    CNN ← Convert the encoded information in the particle into a CNN with the corresponding architecture;
    Train CNN on D_{train} for a total of epo_{eva} epochs;
    p_i.fitness ← the classification accuracy of the trained CNN on D_{train};
Return the population.

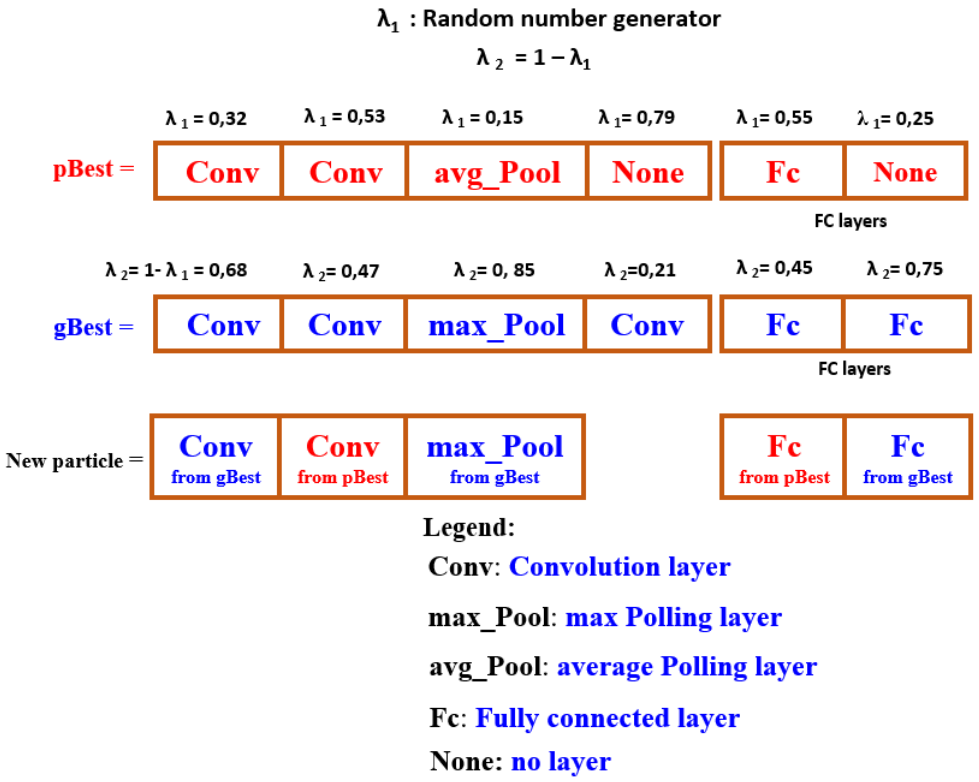

Figure 2.3: A showcase of how a particle is updated in the proposed pswvCNN.

2.2.5 The new strategy of particle update

In the conventional PSO algorithm, the updating of particles is typically done by updating their velocities and positions. However, in our approach, we have deviated from this conventional method by eliminating the velocity equation. Instead, we have devised an alternative algorithm that is tailored to address the specific requirements of our problem. In our method, the velocity update process has been discarded, and we have customized the algorithm to suit our needs and effectively solve the problem at hand.

In our approach, modifying a particle's configuration is a significant contribution to our proposed method. The procedure for updating the particle is executed within the $updateParticle()$ function as depicted in Algorithm 6. This process is visually depicted in Figure 2.3.

We partition the particle into two segments: the first segment comprises the convolutional and pooling layers, while the second segment encompasses the fully-connected (Fc) layers. Each segment is managed independently.

To update a particle using the pair $p_i.pBest$ and $gBest$, we iterate through the layers

of both particles, selecting one layer from $pBest$ and one layer from $gBest$ for each iteration. Then, we generate two random values λ_1 and λ_2 from the interval [0,1], where $c_1 \cdot r_1 = \lambda_1$ and $c_2 \cdot r_2 = \lambda_2$, ensuring that $\lambda_1 + \lambda_2 = 1$. Here, λ_1 and λ_2 represent the probabilities of choosing the layer from $pBest$ and $gBest$, respectively. If λ_1 is greater than λ_2, we add the layer from $pBest$ to the new particle; otherwise, we add the layer from $gBest$. This process continues until the longest particle is fully utilized.

We know that the lengths of the two particles, $p_i.pBest$ and $gBest$, may not always be equal. To handle this situation, we append $none$ to the end of either the Conv/Pool layers section or the Fc layers section to match the lengths for the particle.

The new particle does not include $none$ layers. In cases where the probability of adding a $none$ layer in one of the $gBest$ or $pBest$ particles exceeds the probability of the other layer, the iteration proceeds without incorporating the $none$ layer into the new particle.

The suggested particle update strategy facilitates the generation of new particles within the population, allowing for variations in their lengths and layers. However, there are instances where the number of pooling layers exceeds the specified limit. In such cases, any surplus pooling layers are eliminated from the particle's configuration.

Introduced this particle update method, which omits velocity updates and reduces the computational cost and the convergence rate (i.e., the number of iterations) involved in the optimization process.

2.3 Experimental setup

2.3.1 Peer Competitors

To appraise the efficacy of the pswvCNN approach, we conducted a comparative analysis against a set of advanced CNN models meticulously optimized for achieving high performance. These selected competing algorithms had previously presented their results on identical datasets utilized in evaluating the pswvCNN method.

It's worth noting that many of our competitors employed population-based algorithms to fine-tune their CNN architectures. This suggests that the complexity of these algorithms aligns closely with our approach. Therefore, we consider these algorithms as the primary contenders against our method, operating at a similar level of intricacy.

Algorithm 6: UpdateParticle

Input : $p_i.pBest$, $gBest$
Output : Updated particle

```
newParticle ← ∅
for p_layer, g_layer ∈ (p_i.pBest, gBest) do
    λ1 = Random number generator // λ1 < 1;
    λ2 = 1 − λ1 ;
    if p_layer ≠ no layer then
        if g_layer ≠ no layer then
            if λ1 ≥ λ2 then
                newParticle.Add(p_layer);
            else
                newParticle.Add(g_layer);
            end
        else
            if λ1 ≥ λ2 then
                newParticle.Add(p_layer);
            end
        end
    else
        if λ1 < λ2 then
            newParticle.Add(g_layer);
        end
    end
end
```

We employed population-based metaheuristic algorithms to benchmark and compare our results: evoCNN [17], IPPSO [24], psoCNN [25], BQPSO [26],MBO-CNN [32], sosCNN [31], PSO-based [29], MPSO-CNN [27], and IntelliSwAS [30].

Furthermore, we have gathered other recently published algorithms that have demonstrated promising classification errors for the specified criteria. These algorithms are as follows: LeNet-5 [4], this is a simple CNN designed for the digit recognition problem, LSVM+RBF [75], SVM+Poly [75], CAE-1 [90], CAE-2 [90], PCANet-2 [91], RandNet-2 [91] and LDANet-2 [91].

2.3.2 Parameters of the Algorithm

A detailed breakdown of the parameters used in the proposed `pswvCNN` approach is provided in Table 2.1. These parameters are categorized into three groups: PSO algorithm,

Table 2.1: Parameters employed for assessing the proposed pswvCNN algorithm.

Parameters	Parameters meaning	Value
	pso **algorithm**	
nb_{runs}	Number of execution	10
nb_{itrs}	Number of iterations	10
pop_{size}	Population size	25
	Initialization of particles	
len_{CNN}	Number of layers	[3 - 20]
$Conv_{kernel}$	Convolutional kernel size	[3 - 7]
$Pool_{kernel}$	Pooling kernel size	[2 - 3]
$nb_{filters}$	Filters number	[3 - 256]
$nb_{neuronFc}$	Neurons numbers in an Fc layer	[1 - 300]
	$CNNs$ **Training**	
epo_{eva}	Particle evaluation epochs	1
epo_{train}	Epochs for full training best CNN	100
$Dropout$	Dropout rate	0.5
$ActFun$	Activation Function	ReLu
$learningR$	Learning rate	0.001
$BatchNorm$	Batch normalize layer outputs	yes

initialization of particles, and CNN training.

2.3.2.1 PSO Algorithm Parameters

The first category, governing the behavior of the PSO algorithm, comprises three variables:

- The number of executions (nb_{runs}).
- The number of iterations (nb_{itrs}).
- The population size (pop_{size}).

The optimization process is conducted nb_{runs} times with different initializations to ensure reliable performance evaluation. The termination criterion is represented by the number of iterations (nb_{itrs}), signifying the total iterations completed by the algorithm. The population size (pop_{size}) determines the number of particles employed by the PSO algorithm.

2.3.2.2 Initialization of Particles Parameters

The second category pertains to the parameters defining the initial particles, representing the initial CNN architectures. Each parameter is randomly selected within the range specified in Table 2.1. These parameters include:

- The number of layers (len_{CNN}).
- Convolutional kernel size ($Conv_{kernel}$).
- Number of filters in a convolutional layer ($nb_{filters}$).
- Pooling kernel size ($Pool_{kernel}$).
- Number of neurons in a fully-connected layer ($nb_{neuronFc}$).

The output layer has a fixed number of neurons set to the number of classes.

2.3.2.3 CNN Training Parameters

The third category encompasses parameters that control the training process of each particle and the training associated with the CNN architecture discovered by the optimization process. These parameters include:

- The number of epochs for particle evaluation ($epo_{eva} = 1$).
- The number of epochs for evaluating the global best ($epo_{train} = 100$).
- The dropout rate ($Dropout = 0.5$).

Setting epo_{eva} to one was chosen for an efficient optimization algorithm, but experiments demonstrate that higher values can also be used at the cost of increased computational resources.

2.3.2.4 Implementation Details

The pswvCNN method is implemented using the Keras framework [92] with TensorFlow [93] and runs on a computer system equipped with an Nvidia GeForce GTX1070 GPU card.

2.3.3 Datasets

To evaluate the effectiveness of our method, we utilized nine datasets described in section 1.5, which were queried by competitors. These datasets include MNIST, MNIST-

RD, MNIST-RB, MNIST-BI, MNIST-RD+BI, Convex, Rectangles, Rectangles-I and MNIST-Fashion.

2.4 Results and analysis

2.4.1 Overall Results

In the following section, we describe and contrast the results of our method with those of other approaches, focusing on error rates across nine datasets. We also provide details about the best architectures resulting from our methods.

Given that our methodology relies on a metaheuristic algorithm, multiple trial runs and statistical analyses are necessary to draw valid conclusions and evaluate convergence towards the optimal solution. Due to limited computing resources, we conducted ten (10) experimental runs.

The results of the experiment of the proposed pswvCNN algorithm are shown in Table 2.2. The results include the average error rate, the lowest error rate, and the standard deviations (std dev) of the classification errors from 10 different runs. In order to facilitate comparison, the table utilizes symbols (+) and (-) to indicate if the error rate attained by pswvCNN is considerably superior or inferior, respectively, to the top outcome acquired from the corresponding competition. If there are no results available for the rival, (–), and if the competitor's best error rate is the same as pswvCNN's, (=) is shown.

As shown in Table 2.2, the proposed pswvCNN achieved optimal test errors of 0.30%, 2.27%, 11.47%, 2.74%, 1.67%, 1.35%, 1.33%, and 0.01% for each of the eight datasets: dataset that includes MNIST digits, four variants of MNIST (MNIST-BI, MNIST-RD+BI, MNIST-RD, MNIST-RB), convex, and two types of rectangles (rectangles-I and rectangles). The standard deviations for the corresponding values were 0.04%, 0.37%, 2.43%, 0.62%, 0.37%, 0.92%, 0.82%, and 0.92%. For these datasets, the average test errors across ten runs were 0.44%, 2.75%, 14.58%, 3.69%, 2.32%, 2.78%, 1.81%, and 0.78%. The boxplots Figure 2.4 illustrate the distribution of gBest test accuracies over 10 runs for each dataset.

Our suggested approach outperforms psoCNN and other PSO-based approaches, which have error rates of 0.32% and 0.35%, respectively, in achieving the best CNN model on the MNIST dataset, with an error rate of 0.30%. Our approach differs very little from MPSO-CNN, which has an error rate of 0.11%. Our results show good performance

on the MNIST dataset utilising basic CNN architectures. Remarkably, neural networks and the DropConnect approach were used to achieve the current best-reported test error of 0.21% [94].

Our suggested approach produces the best CNN model for the MNIST-BI dataset, with an error rate of 2.27% and a mean error rate of 2.75%. On the other hand, BQ-CNN is the best-performing model for this dataset, achieving an error rate and a mean error rate of 0.95%. By contrast, the error rates attained by psoCNN and other PSO-based techniques are 1.90% and 2.20%, respectively.

Similarly, BQ-CNN continues to be the best reference method for the MNIST-RD and MNIST-RB datasets, obtaining the best error rates of 0.98% and 0.97%, respectively. On the other hand, our suggested approach works better, yielding error rates of 1.67% for MNIST-RB and 2.74% for MNIST-RD. PSO-based, psoCNN, and other approaches, in contrast, do not result in reduced error rates.

We obtained a best-estimated error rate of 11.47% in the MNIST-RD+BI dataset, which is higher than the findings of other approaches like IPPSO (34.50%), psoCNN (14.28%), BQ-CNN (14.20%), PSO-based (11.61%), and IntelliSwAS (15.74%).

Our proposed solution outperforms all other competing methods with an amazing error rate of 0.72% for the Rectangles-I dataset. Furthermore, our method outperforms the PSO-based and IntelliSwAS methods, which reach error rates of 0.10%, in producing the best CNN model in the Rectangles dataset, with an incredibly low error rate of 0.01%.

With an error rate of 1.35%, our CNN model performs quite well when applied to the Convex dataset. This gives better results than the models produced by PSO-based and psoCNN approaches, which yield error rates of 1.70% and 1.36%, respectively. This specific dataset demonstrates the superior efficacy of our suggested approach.

We compared the number of trainable parameters between the models created by our suggested pswvCNN approach and those created by other competing methods in our examination of the MNIST-Fashion dataset. Table 2.3 presents the condensed findings. Our pswvCNN approach achieves a test error rate of 5.44%, which is higher than the performance of the top models among the fourteen rivals, clearly outperforming all other approaches.

The figure Figure 2.5 visually represents the progression of the highest particle scores (training accuracy) over 10 runs, providing a visual depiction of the advancement of our approach. A boxplot showing the test accuracies attained by the top particles is

also included in the figure. These visuals make it very evident how well our pswvCNN approach consistently performs on the MNIST-Fashion dataset.

Ultimately, our results provide compelling evidence for the pswvCNN method's supremacy on the MNIST-Fashion dataset. It surpasses all other approaches in test accuracy and exhibits exceptional stability over several runs.

Table 2.2: Comparison between the error rates of pswvCNN and various other models.

Model / Dataset	**MNIST**	**MNIST-BI**	**MNIST-RD+BI**	**MNIST-RD**	**MNIST-RB**	**Convex**	**Rectangles-I**	**Rectangles**
LeNet-5 [74]	0.95%(+)	–	–	–	–	–	–	–
SVM+RBF [75]	3.03%(+)	22.61%(+)	32.62%(+)	10.38%(+)	14.58%(+)	19.13%(+)	–	–
SVM+Poly [75]	3.69%(+)	24.01%(+)	37.59%(+)	13.61%(+)	16.62%(+)	19.82%(+)	–	–
CAE-1 [90]	2.83%(+)	16.7%(+)	48.10%(+)	11.59%(+)	13.57%(+)	–	21.86%(+)	1.48%(+)
CAE-2 [90]	2.48%(+)	15.5%(+)	45.23%(+)	9.66%(+)	10.90%(+)	–	21.54%(+)	1.21%(+)
PCANet-2 [91]	1.06%(+)	11.55%(+)	35.86%(+)	8.52%(+)	6.85%(+)	4.19%(+)	13.39%(+)	0.49%(+)
RandNet-2 [91]	1.27%(+)	11.65%(+)	43.69%(+)	8.47%(+)	13.47%(+)	5.45%(+)	17.00%(+)	0.09%(+)
LDANet-2 [91]	1.40%(+)	12.42%(+)	38.54%(+)	4.52%(+)	6.81%(+)	7.22%(+)	16.20%(+)	0.14%(+)
IPPSO [24]	1.13%(+)	–	34.50%(+)	–	–	8.48%(+)	–	–
evoCNN [17]	1.18%(+)	4.53%(+)	35.03%(+)	5.22%(+)	2.80%(+)	4.82%(+)	5.03%(+)	0.01%(=)
psoCNN [25]	0.32%(+)	1.90%(-)	14.28%(+)	3.58%(+)	1.79%(+)	1.70%(+)	2.22%(+)	0.03%(+)
BQ-CNN [26]	0.99%(+)	0.95%(-)	14.20%(+)	0.98%(-)	0.97%(-)	1.65%(+)	–	–
MBO-CNN [32]	0.36%(+)	–	–	–	–	–	–	–
sosCNN [31]	0.38%(+)	1.68%(-)	10.65%(-)	3.01%(+)	1.49%(-)	1.4%(+)	1.57%(+)	0%(-)
PSO-based [29]	0.35%(+)	2.2%(-)	11.61%(+)	3.23%(+)	1.8%(+)	1.36%(+)	1.01%(+)	0%(-)
MPSO-CNN [27]	0.11%(-)	–	31.9%(+)	–	–	7.27%(+)	–	–
IntelliSwAS [30]	0.38%(+)	3.53%(+)	15.74%(+)	4.46%(+)	2.55%(+)	1.20%(+)	2.22%(+)	0%(-)
pswvCNN (best)	**0.30%**	**2.27%**	**11.47%**	**2.74%**	**1.67%**	**1.35%**	**0.72%**	**0.01%**
pswvCNN (mean)	**0.44%**	**2.75%**	**14.58%**	**3.69%**	**2.32%**	**2.78%**	**1.81%**	**0.78%**
pswvCNN (std dev)	0.04%	0.37%	2.43%	0.62%	0.37%	0.92%	0.82%	0.92%

Table 2.3: Comparing various models with the MNIST-Fashion dataset's pswvCNN test errors.

Model	Test error(%)	#Parameters	#Epoch
2C1P2F+Drouout [17]	8.40(+)	3.27 M	300
2C1P [17]	7.50(+)	100 K	30
3C2F [17]	9.30(+)	–	–
3C1P2F+Dropout [17]	7.40(+)	7.14 M	150
GRU+SVM+Dropout [17]	10.30(+)	–	100
SqueezeNet-200 [95]	10.00(+)	500 K	200
MLP 256-128-64 [17]	10.00(+)	41 K	25
AlexNet [96]	10.1(+)	62.3 M	-
VGG16 [5]	6.5 (+)	26 M	-
GoogleNet [7]	6.3 (+)	23 M	200
evoCNN (best) [17]	5.47(+)	6.68 M	100
evoCNN (mean) [17]	7.28(+)	6.52 M	100
psoCNN (best) [25]	5.53(+)	2.32 M	200
psoCNN (mean) [25]	5.90(+)	2.5 M	200
sosCNN (best) [31]	5.68(+)	2.30 M	100
sosCNN (mean) [31]	6.17(+)	3.42 M	100
pswvCNN (best)	**5.44**	**5.15 M**	**100**
pswvCNN (mean)	**6.1**	**3.74 M**	**100**
pswvCNN (std dev)	**0.46**	**0.81 M**	**100**

2.4.2 Discussion

Our proposed method can efficiently detect CNN's designs that are shorter than other competing models. This is achieved by using a compact architecture, typically comprising 3 to 20 layers. Handcrafted networks often include unnecessary features, while smaller networks can effectively represent more complex models. Our method enhances the identification of optimal solutions by generating populations from networks of varying lengths. Recent findings indicate that smaller networks exhibit faster convergence compared to larger networks, enhancing the method's efficiency.

We provide the training curves of the top model identified by pswvCNN on the Convex dataset for 100 epochs in Figure 2.6 in order to diagnose the performance of the CNN model produced by our method. The validation set is the test dataset. The training loss curve declines until it reaches a stable point, as seen by the learning curve figure. The validation losses curve is close to the training losses and decreases until the end of stability. The suggested pswvCNN can identify appropriate models for a given image classification dataset.

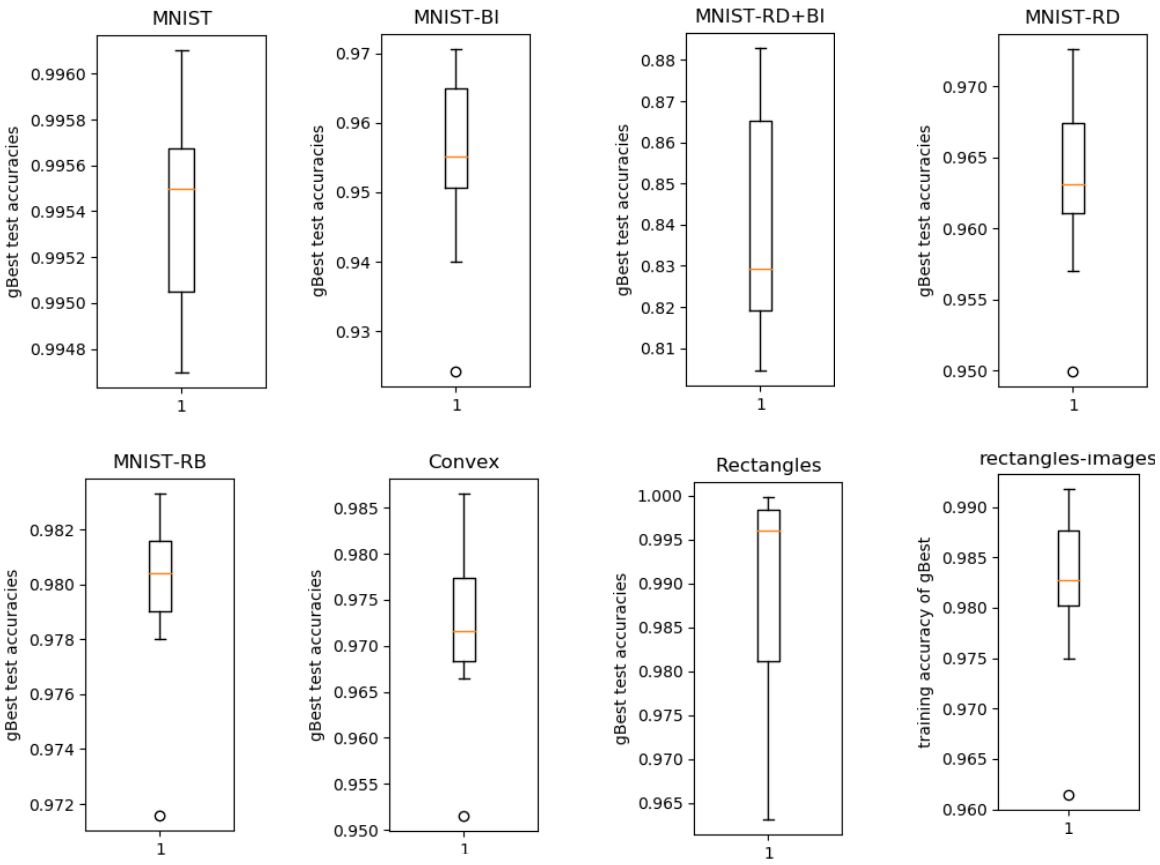

Figure 2.4: Boxplots showing the gBest test accuracy attained by the suggested pswvCNN for every dataset in 10 runs completed.

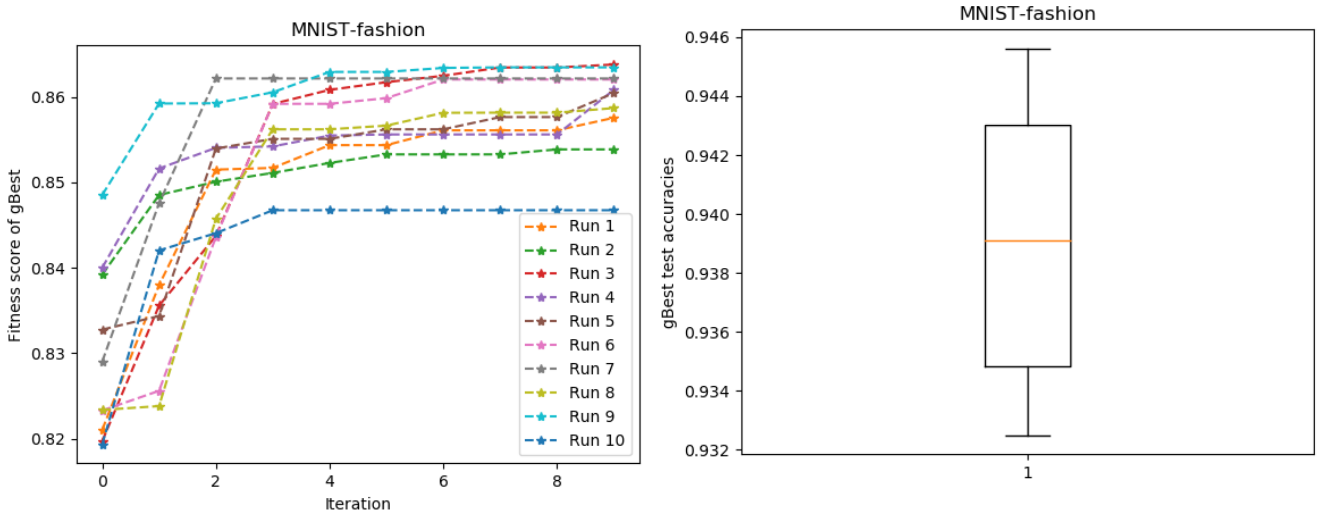

Figure 2.5: **Left**: Training accuracy for the gBest particle in each iteration on the MNIST-fashion dataset. **Right:** Boxplot shows the test accuracy of gBest discovered by pswvCNN.

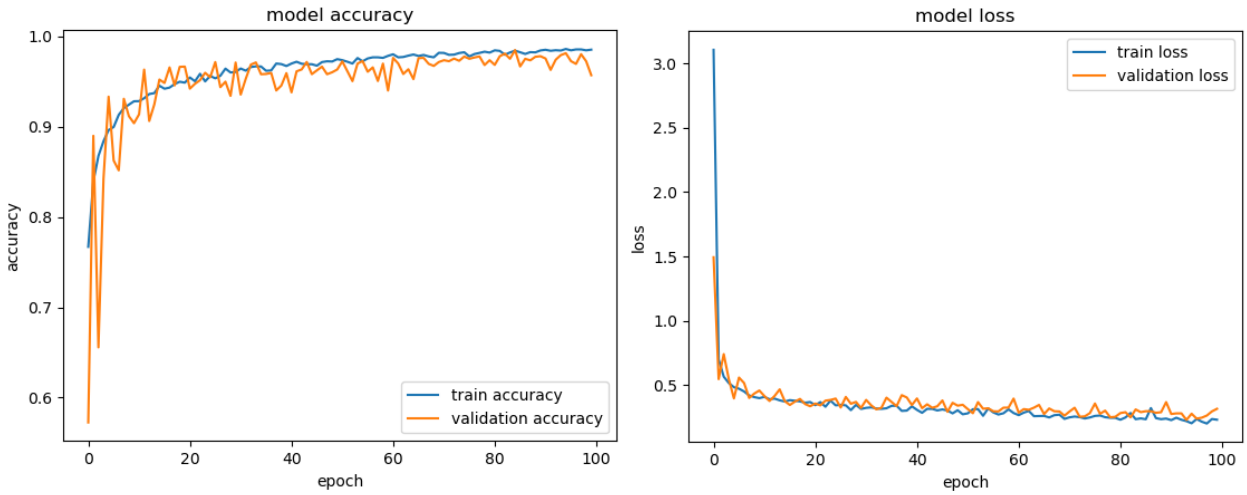

Figure 2.6: The training performance plot of the gBest model, discovered by the pswvCNN algorithm, on the Convex dataset.**Left**:model accuracy, **Right**:model loss

In the exploration of convolutional neural network (CNN) architectures, the focus on depth and performance across various datasets reveals intriguing contrasts among different methodologies. Particularly noteworthy is the comparative analysis between the pswvCNN algorithm and established benchmark models like IPPSO, psoCNN, PSO-based, and sosCNN. The findings underscore a compelling trend, pswvCNN consistently produces deeper networks, outshining its counterparts. Across datasets spanning MNIST, MNIST-BI, MNIST-RD+BI, MNIST-RD, MNIST-RB, Convex, Rectangles-I, Rectangles, and MNIST-fashion, as shown in Figure 2.7, the pswvCNN networks exhibit depths ranging from 7 to 11 layers. This depth advantage suggests a heightened capacity for the pswvCNN architecture to capture nuanced patterns and intricacies within the data, potentially translating into enhanced performance in classification tasks. Consequently, the experimental outcomes highlight the efficacy of the pswvCNN approach in crafting sophisticated network structures, positioning it as a promising candidate for diverse machine learning endeavors.

In six out of the nine datasets presented in Table 1.3, the models developed using the pswvCNN algorithm exhibit superior performance compared to the most recent models available. This indicates the effectiveness of the pswvCNN algorithm in achieving state-of-the-art results across various datasets.

Furthermore, it is important to mention that the models generated by the pswvCNN algorithm demonstrate a significant decrease in the number of parameters when compared to well-known architectures like as VGG16, GoogleNet, AlexNet, and evoCNN, as

illustrated in Table 2.5. The decrease in parameters suggests a more effective use of model complexity, leading to improved performance without adding too much computing burden. pswvCNN showcases its capacity to optimize model architecture and maintain efficacy in resource-limited situations or applications that prioritize computational efficiency, by obtaining equivalent or greater performance with fewer parameters. The importance of pswvCNN in the field of deep learning architectures is highlighted by its focus on efficiency.

For instance, when considering the MNIST fashion dataset, the most effective model identified through the pswvCNN algorithm contains only 5.15 million parameters. In contrast, the VGG16 model, renowned for its depth and complexity, comprises a significantly larger parameter count of 138 million. This stark contrast underscores the efficiency and effectiveness of the pswvCNN algorithm in achieving competitive performance with substantially fewer model parameters.

Overall, these findings highlight the potential of the pswvCNN algorithm as a practical and efficient approach for designing high-performing convolutional neural network architectures, particularly in scenarios where computational resources and model complexity need to be carefully managed.

In contrast, specific competing methods tend to generate shallower networks, meaning that the networks they create are limited in terms of their maximum depths due to the issue of vanishing gradients. The vanishing gradient problem significantly impacts a model's learning capacity. When gradients approach zero, backpropagation becomes ineffective in modifying weights, halting the learning process.

Conversely, across all datasets, the network architectures generated by the sosCNN method consistently exhibit greater depth. This phenomenon can be attributed to the effectiveness of the slack-gain strategy in facilitating the creation of deeper network structures. The slack-gain algorithm enables the construction of architectures with increased depth compared to alternative techniques.

The PSO algorithm is executed at each iteration to reevaluate the particles and calculate the fitness values to find the personal best ($pBest$) for each particle and the global best ($gBest$) for the entire population. These particles represent convolutional neural networks (CNNs), and their evaluation relies on the accuracy obtained by training them for a specific number of epochs, denoted as "epo_{eva}". It's essential to choose an appropriate number of epochs for this evaluation—not too few and not too many.

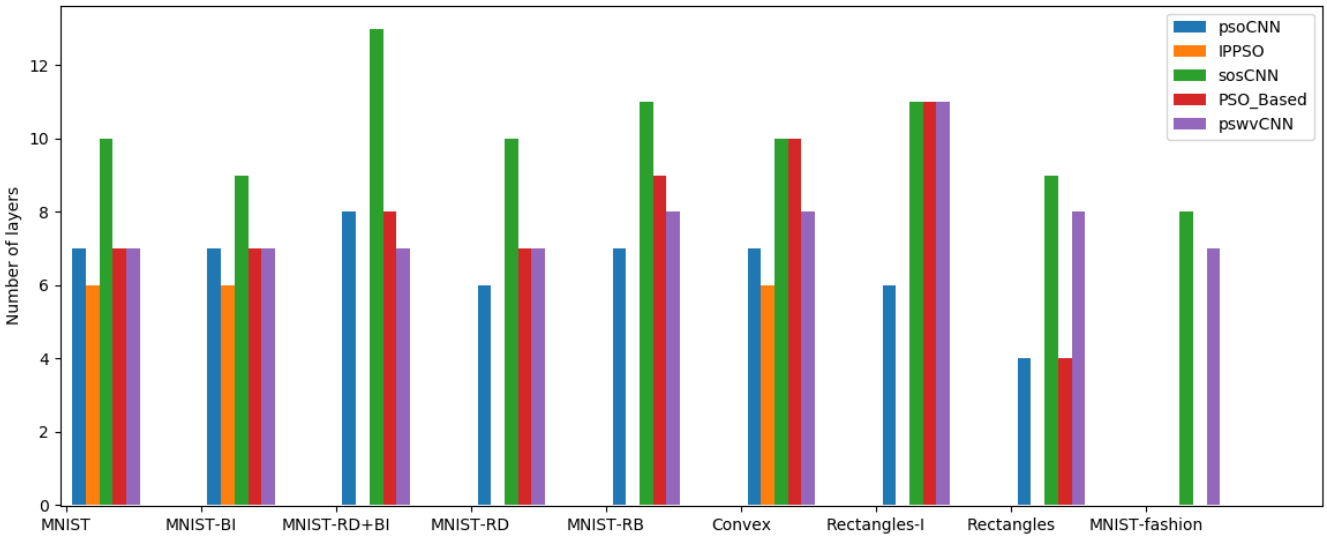

Figure 2.7: Model depth comparison between the models created by our approaches and the benchmark models on all datasets.

Using a high number of epochs to evaluate a CNN model leads to more accurate assessments, but it also extends the time required for the optimization process. This is because each particle must be trained for the same number of epochs before its performance can be determined.

Figure 2.8 illustrates the impact of using different values for "epo_{eva}," specifically 1, 3 and 7 epochs, during the evaluation of particles. As "epo_{eva}" increases, the evaluation becomes more accurate, but it also consumes more time. For instance, when working with the Convex dataset, setting "epo_{eva}" to 1 results in the best error rate for the best model after ten runs of the pswvCNN method, which is 1.35%, with a runtime of 0.49 hours. However, when "epo_{eva}" is set to 3, the best error rate after ten executions improves to 1.28%, but it takes significantly more time, approximately 4.33 hours.

Additionally, it's worth noting that when adjusting the "epo_{eva}" value while implementing the pswvCNN algorithm, it was observed that the depth of the resulting CNN models (measured by the number of layers) is directly proportional to the chosen value of "epo_{eva}". In other words, higher values of "epo_{eva}" tend to produce deeper CNN architectures.

The graph depicted in Figure 2.9 illustrates the progression of the fitness score of $gBest$ across ten different runs, each consisting of ten iterations, conducted on all datasets with 25 particles. All other pertinent parameters are detailed in Table 2.5. The data suggests a consistent improvement in accuracy over time, indicative of the

Table 2.4: Presents the best CNN architectures that were discovered by the pswvCNN method for all datasets.

Dataset	Layer	Parameters
MNIST	Conv	kernel size: 5 x 5; output filters: 184
	Conv	kernel size: 4 x 4; output filters: 251
	Conv	kernel size: 6 x 6; output filters: 214
	Conv	kernel size: 5 x 5; output filters: 204
	Max Pool	kernel size: 3 x 3;
	Conv	kernel size: 6 x 6; output filters: 102
	Fc	output neurons: 10
MNIST-BI	Conv	kernel size: 3 x 3; output filters: 166
	Conv	kernel size: 4 x 4; output filters: 116
	Conv	kernel size: 6 x 6; output filters: 185
	Conv	kernel size: 5 x 5; output filters: 132
	Conv	kernel size: 5 x 5; output filters: 132
	Conv	kernel size: 5 x 5; output filters: 132
	Fc	output neurons: 10
MNIST-RD+BI	Conv	kernel size: 3 x 3; output filters: 111
	Conv	kernel size: 5 x 5; output filters: 186
	Conv	kernel size: 3 x 3; output filters: 222
	Conv	kernel size: 5 x 5; output filters: 197
	Conv	kernel size: 6 x 6; output filters: 207
	Average Pool	kernel size: 3 x 3;
	Conv	kernel size: 6 x 6; output filters: 213
	Fc	output neurons: 10;
MNIST-RD	Conv	kernel size: 4 x 4; output filters: 250
	Conv	kernel size: 6 x 6; output filters: 65
	Average Pool	kernel size: 3 x 3;
	Conv	kernel size: 5 x 5; output filters: 215
	Conv	kernel size: 5 x 5; output filters: 215
	Average Pool	kernel size: 3 x 3
	Fc	output neurons: 10;
MNIST-RB	Conv	kernel size: 3 x 3; output filters: 191
	Conv	kernel size: 6 x 6; output filters: 107
	Conv	kernel size: 6 x 6; output filters: 157
	Conv	kernel size: 6 x 6; output filters: 223
	Average Pool	kernel size: 3 x 3
	Conv	kernel size: 5 x 5; output filters: 208
	Fc	output neurons: 10;
Convex	Conv	kernel size: 5 x 5; output filters: 222
	Conv	kernel size: 6 x 6; output filters: 143
	Conv	kernel size: 4 x 4; output filters: 149
	Conv	kernel size: 6 x 6; output filters: 220
	Average Pool	kernel size: 2 x 2;
	Max Pool	kernel size: 3 x 3
	Conv	kernel size: 6 x 6; output filters: 123
	Fc	output neurons: 2;
Rectangles-I	Conv	kernel size: 3 x 3; output filters: 239
	Conv	kernel size: 4 x 4; output filters: 147
	Conv	kernel size: 6 x 6; output filters: 182
	Conv	kernel size: 3 x 3; output filters: 130
	Conv	kernel size: 3 x 3; output filters: 97
	Conv	kernel size: 4 x 4; output filters: 249
	Conv	kernel size: 4 x 4; output filters: 66
	Average Pool	kernel size: 2 x 2
	Average Pool	kernel size: 2 x 2
	Conv	kernel size: 5 x 5; output filters: 150
	Fc	output neurons: 2;
Rectangles	Conv	kernel size: 5 x 5; output filters: 136
	Conv	kernel size: 5 x 5; output filters: 139
	Max Pool	kernel size: 3 x 3;
	Conv	kernel size: 5 x 5; output filters: 150
	Conv	kernel size: 6 x 6; output filters: 225
	Max Pool	kernel size: 3 x 3
	Conv	kernel size: 4 x 4; output filters: 244
	Fc	output neurons: 2;
MNIST-fashion	Conv	kernel size: 4 x 4; output filters: 150
	Conv	kernel size: 3 x 3; output filters: 245
	Conv	kernel size: 5 x 5; output filters: 254
	Conv	kernel size: 6 x 6; output filters: 162
	Conv	kernel size: 7 x 7; output filters: 188
	Max Pool	kernel size: 3 x 3
	Fc	output neurons: 10;

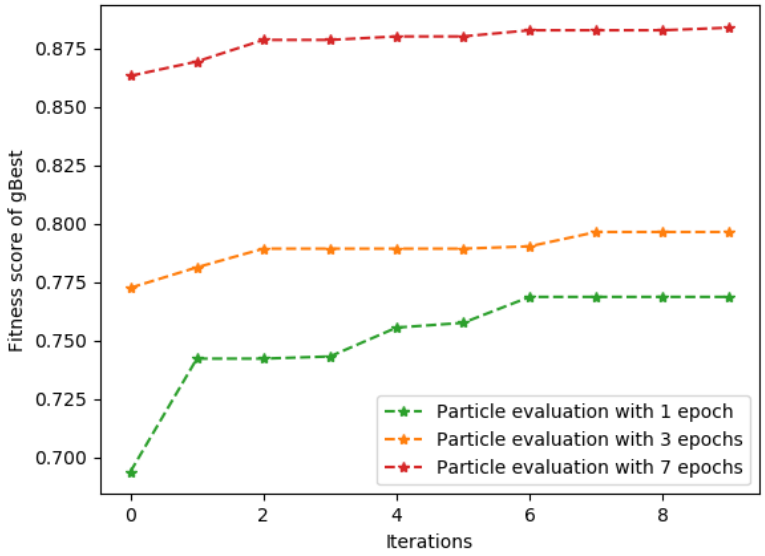

Figure 2.8: Impact of the value of the number of epochs utilized in the convex dataset particle evaluation.

effectiveness of the optimization process.

Observing the plot, it becomes evident that the convergence is swift. During the early stages of the evolutionary process, the accuracy of the $gBest$ particles rapidly ascents, peaking around the second iteration. Subsequently, the particles continue their search but encounter difficulties finding further enhancements, leading to a plateau in performance until the tenth iteration.

This rapid convergence can be attributed to the concerted efforts to minimize the search space facilitated by the proposed coding and particle update strategies. The algorithm efficiently navigates the solution space by leveraging these strategies, accelerating convergence toward optimal solutions.

Table 2.4 presents the optimal CNN architecture for each dataset, detailing its layers and respective parameters. Additionally, it provides the count of trainable parameters for each dataset. A noticeable trend is the algorithm's preference for straightforward structures with considerable performance potential. This observation is significant, as recent research confirmed by [97] underscores the effectiveness of such simpler architectures.

An important achievement we have made in this field of research is the reduction of the execution time of the pswvCNN algorithm. Our research shows that by removing the velocity equation and optimizing particle updates, pswvCNN can effectively identify CNN architectures that are highly competitive, and do it in a shorter amount of time. The

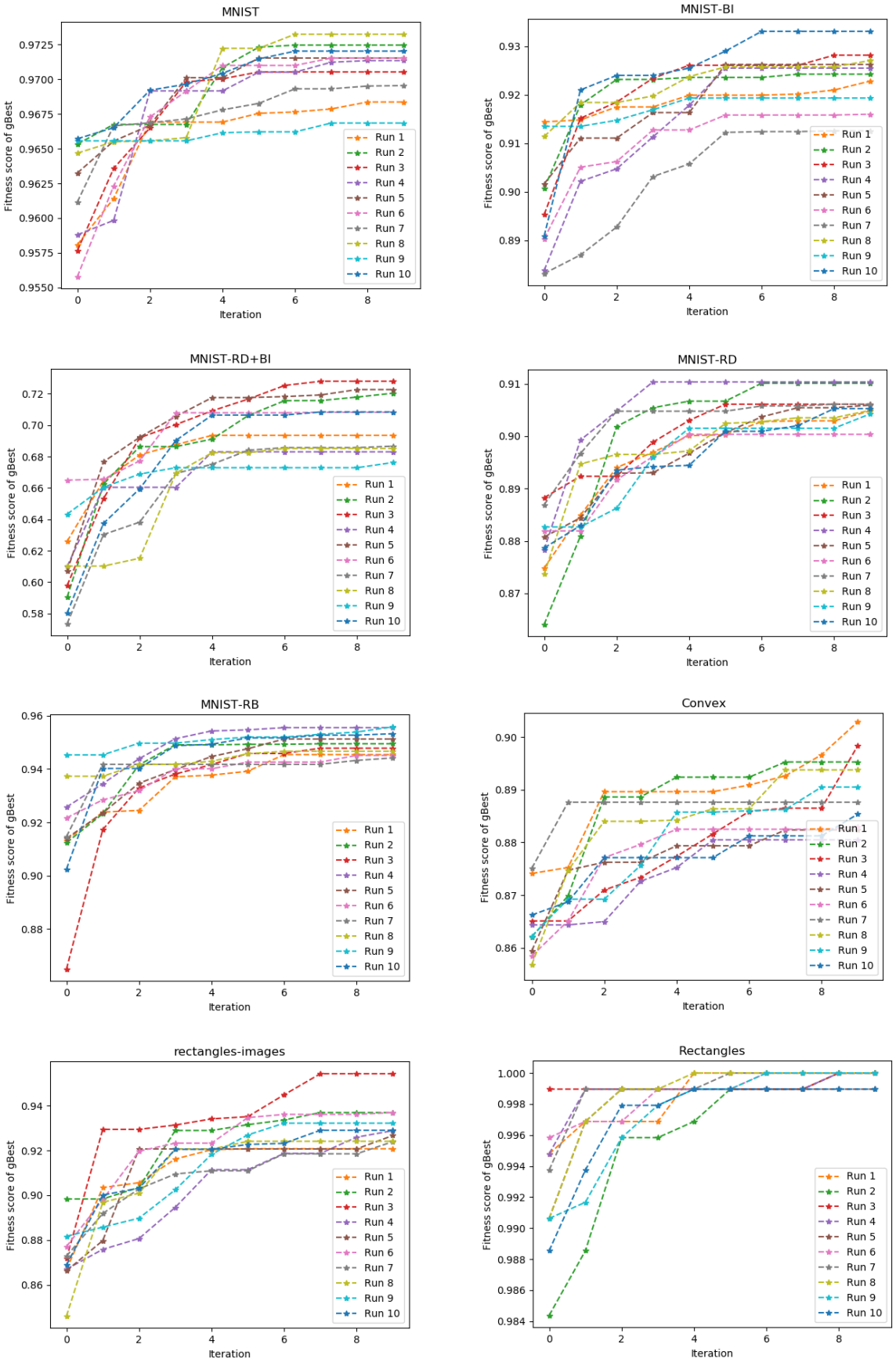

Figure 2.9: Plotting the training accuracy evolution for each dataset across ten iterations in ten runs using the proposed pswvCNN.

average execution times for different datasets, such as MNIST, MNIST-BI, MNIST-RD + BI, MNIST-RD, MNIST-RB, Convex, Rectangles-I, Rectangles, and MNIST-fashion, are 7 hours, 6 hours, 3.88 hours, 3.7 hours, 0.49 hours, 2.42 hours, 0.7 hours, and 10 hours, respectively. The aforementioned timings were obtained using a PC that was equipped with an Nvidia GeForce GTX 1070 GPU.

Although the datasets used in our studies were modest, our results demonstrate that pswvCNN is a powerful tool for searching and optimizing CNN designs. Therefore, pswvCNN shows potential as a beneficial tool for deep learning experts, assisting them in the automated creation of CNN models for various domain-specific uses.

Dataset	**Number of Trainable Parameters**
MNIST	3.64 M
MNIST-BI	3.6 M
MNIST-RD+BI	5.4 M
MNIST-RD	3.38 M
MNIST-RB	5.3 M
Convex	3.6 M
Rectangles-I	2.76 M
Rectangles	3.11 M
MNIST-Fashion	5.15 M

Table 2.5: Number of trainable parameters in the models optimized by the proposed *pswvCNN* method across various datasets.

2.5 Conclusion

This chapter presents a method for enhancing CNN architectures by utilizing a PSO algorithm that does not involve the velocity equation. The strategy is specifically tailored for image classification tasks. The main goal of our study was to determine the most effective hyperparameters for building a Convolutional Neural Network (CNN). This involved determining the ideal number of layers, convolutional layers, pooling layers, kernel size, number of filters, and neurons in the fully connected layers.

The experimental findings demonstrate that our proposed pswvCNN method has the ability to efficiently identify optimum CNN structures for different datasets. By utilizing a limited number of particles (25) and iterations (10), our technique is able to produce test errors that are comparable to those of more intricate models. Moreover, the

pswvCNN method demonstrates the ability to achieve superior topologies by leveraging supplementary computational resources.

Our research utilizes an Evolutionary Computation (EC) method to independently create CNN structures without any human involvement or prior knowledge. The pswvCNN algorithm demonstrates superior performance compared to both EC and non-EC approaches on various widely recognized benchmark datasets, such as MNIST, MNIST-BI, MNIST-RD + BI, MNIST-RD, MNIST-RB, Convex, Rectangles-I, Rectangles, and MNIST-fashion.

Through the use of an innovative approach, we tackle the constraints of conventional PSO search methods, leading to the proficient creation of CNN structures. By eliminating the velocity equation from the PSO algorithm, we improve the effectiveness and efficiency of our pswvCNN approach. This results in a higher level of accuracy, robustness, and greater performance compared to other rivals on standard datasets.

CHAPTER 3

OPTIMIZING CNN ARCHITECTURE USING A HYBRID METHOD

Contents

3.1 Introduction

The hybrid method combining Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) for CNN architecture design aims to leverage the strengths of both algorithms to enhance the optimization process. Genetic Algorithms utilize evolutionary principles such as selection, crossover, and mutation to evolve a population of candidate solutions, while Particle Swarm Optimization mimics the behavior of a swarm of particles moving through a search space to find optimal solutions.

In this hybrid approach, GA and PSO are integrated to collaboratively explore and exploit the search space of CNN architectures. GA provides the ability to explore a wide range of architectural configurations through genetic operations, enabling diversity and flexibility in the search process. PSO, on the other hand, contributes its swarm intelligence and local/global search capabilities to efficiently converge towards promising solutions.

By combining GA and PSO, the hybrid method aims to overcome the limitations of each algorithm individually and achieve better overall optimization results. This approach offers the potential for improved accuracy, computational efficiency, and convergence speed in finding optimal CNN architectures for various applications, including image recognition, computer vision, and deep learning tasks.

In this chapter, we present a detailed description of the Hybrid GAPSO Method, including the encoding scheme for representing CNN architectures, the fitness evaluation criteria, and the evolutionary and swarm operators used for architecture optimization. Experimental evaluations are conducted on benchmark datasets to assess the effectiveness and performance of the proposed approach, comparing it with other optimization methods and baseline architectures.

The contributions of this research include the introduction of a hybrid optimization approach that combines Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) for evolving CNN architectures. This approach presents a promising solution for automating the design and optimization process, addressing the limitations of the previous method, such as insufficient diversity and optimization intensification in the search strategies employed. In this chapter, we delve deeper into this issue by integrating an adaptive mutation strategy into the swarm particle update process. This combination aims to enhance the algorithm's exploration capabilities and accelerate the convergence rate, thereby mitigating the aforementioned limitation.

3.2 The proposed algorithm

As elaborated in Section 1.3.4.2, Particle Swarm Optimization (PSO) is a stochastic optimization method inspired by social behavior. This approach is guided by two equations, as described in Equations Equation 1.10 and Equation 1.11, which dictate the movement of particles. The choice of topology, representing the communication network between particles, significantly influences PSO's performance.

Initially, as summarized in algorithm 3, the PSO algorithm adopted a fully connected topology where each particle communicated with all others. Referred to as the best global position ($gBest$), this setup ensures that each particle receives information from the entire swarm. However, this approach has a notable drawback of insufficient exploration, which may result in convergence to local optima and hinder the system from adequately exploring the search space, especially in scenarios with excessive amplification or oscillations.

One popular hybridization approach involves integrating PSO into a Genetic Algorithm (GA) and replacing GA's mutation step with PSO mechanisms.

We propose a novel hybrid algorithm that integrates concepts from both GA and PSO by embedding GA within PSO. Instead of following PSO's standard procedure of updating velocities and positions, we incorporate GA's crossover and mutation steps. This hybrid algorithm operates in two main phases:

PSO Process:

- Initialize a population of particles for PSO, where each particle represents a potential solution.

- Exploring the neighborhood of the best position ($pBest_i$) and seeking the global best ($gBest$).

GA Process:

(1) **Select Crossover Point**:

- Randomly choose a crossover point to determine where the genetic material from $gBest$ and $pBest$ will be combined.

(2) **Create New Candidate Solution**:

- Generate a new position by blending elements from $gBest$ and $pBest$ around the selected crossover point. This involves selecting segments of genetic material from each position and merging them to create a new candidate solution.

(3) **Update Particle's Position**:

- Update the particle's position based on the new candidate solution generated through crossover.

(4) **Apply Mutation**:

- For each particle, apply mutation based on a predefined mutation probability.
- This mutation can involve perturbing the particle's position by a small random amount.

A flowchart in Figure 3.1 shows the suggested hybrid technique. This integration makes the integrative use of PSO and GA's strengths possible, which may improve exploration and convergence toward better solutions.

3.2.1 Population Initialization

The swarm is initialised in the proposed $GAPSO$ approach by creating pop_{size} particles with randomly generated Convolutional Neural Network (CNN) architectures. Each particle has a variable number of layers ranging from 3 to a maximum value of len_{CNN}. In order to guarantee the practicality of the created CNN structures, the initial and final layers of each particle are assigned as a convolutional layer and a fully connected layer, respectively. In addition, fully linked layers can only be placed at the end of the architecture and cannot be inserted between convolutional or pooling layers. Other studies commonly embrace this convention, as fully connected layers are typically used for feature categorization after feature extraction from convolutional and pooling layers.

Every particle is allocated a specific number of convolutional layers, chosen randomly, with a varying number of filters ranging from 1 to $nb_{filters}$. The kernel size for these layers is selected randomly from a range of 3 x 3 to $Conv_{kernel}$ x $Conv_{kernel}$. The kernel strides are set to 1 x 1, and the ReLU activation function is used. The particle's initial configuration contains either a pooling layer or an average pooling layer. The kernel size of the pooling layer is $Pool_{kernel}$ x $Pool_{kernel}$, where $Pool_{kernel}$ can be either 2 or 3. The pooling layer has a stride of 2 2. A particle is augmented with a fully connected layer (Fc) that consists of a variable number of neurons, ranging from 1 to $nbneuronFc$,

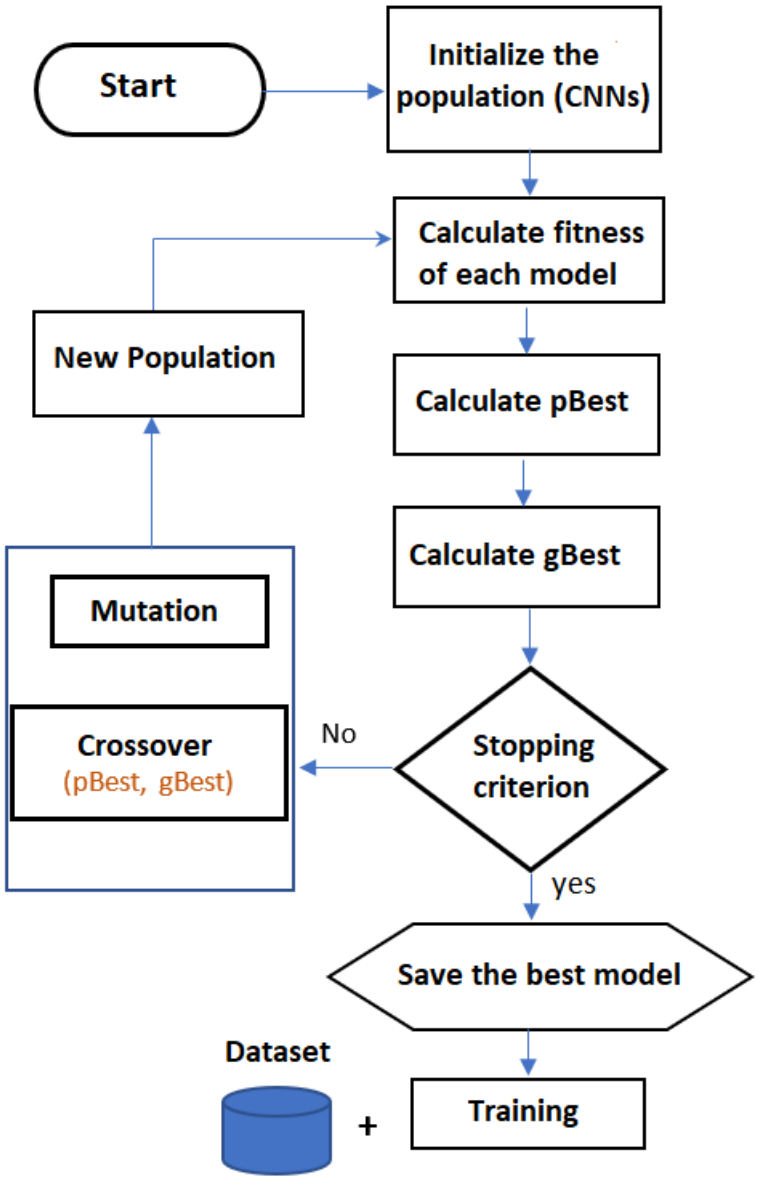

Figure 3.1: Proposed approach flowchart.

selected randomly. The number of neurons in the last fully connected layer is always equivalent to the number of classes in the dataset employed, such as ten for the MNIST dataset or two for the Convex dataset.

3.2.2 The encoding strategy

The encoding strategy utilized in the first contribution in (2.2.2) remains the one used in this method. This encoding approach is applied to represent the population particles

and their corresponding solutions. The encoding scheme plays a critical role in the effectiveness of the metaheuristic algorithm and is a fundamental aspect of its design. By employing a consistent encoding strategy, the search space associated with the problem is defined and maintained.

In our proposed algorithm, we adopt the same encoding strategy for the population particles, specifically for convolutional neural networks (CNN), as described in the first contribution. The population consists of $CNN_1; CNN_2; CNN_3; \ldots\ldots\ldots; CNN_i$, and so on, where i ranges from 1 to n. Each particle (CNN) is encoded using sequential layers, and the length of each particle may vary.

The encoding scheme employs a data structure to represent each layer within a particle, which includes the layer type (such as convolutional, max-pooling, average-pooling, fully connected) and the associated layer parameters (such as the number of filters, filter size, stride, and number of neurons in fully connected layers). This consistent encoding strategy enables the algorithm to manipulate and optimize the CNN architecture by modifying the layers and their parameters throughout the optimization process.

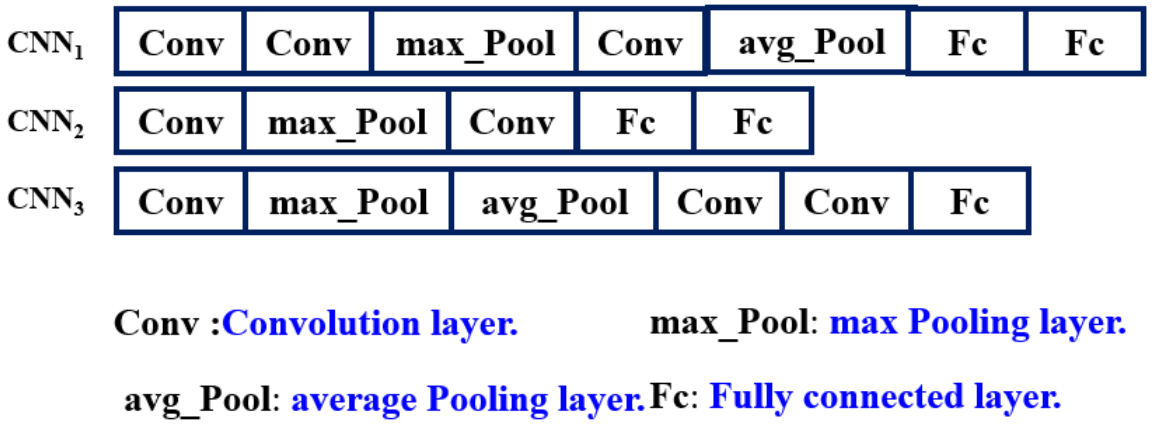

Figure 3.2: Presentation of the proposed $GAPSO$ particle

3.2.3 The selection operator

Our algorithm uses deterministic selection based on performance rather than stochastic methods like tournament or roulette selection. Each particle's parents are its personal best ($pBest_i$) and the global best ($gBest$). This ensures that offspring inherit both individualized improvements and global guidance, directing the search towards promising regions in the solution space for better convergence towards an optimal CNN architec-

ture. pendant quelques secondes Our algorithm uses deterministic selection based on performance. For each particle, we choose its personal best ($pBest_i$) and the global best ($gBest$) as parents. This approach ensures that offspring inherit both the particle's individual strengths and the best traits from the overall population, effectively guiding the search toward optimal CNN architectures.

3.2.4 The crossover operator

The suggested method incorporates a crossover operation between the population's personal best ($pBest_i$) and global best ($gBest$). This operation aims to create new offspring with improved attributes by combining the capabilities of the personal and global finest solutions. The crossover process improves the algorithm's capacity to explore and utilize the search area, possibly producing better solutions by employing the data from the best solutions found thus far. Each gene (layer) in our method is chosen randomly from either the $gBest$ or $pBest_i$ parent chromosomes using a uniform crossover technique. One particle is selected to be added to the population, as seen in Figure 3.3.

3.2.5 The mutation operator

A mutation is as simple as altering the characteristics of a layer, which is situated at a certain locus and is also randomly determined, as Figure 3.4 illustrates. Thus, the architecture of a solution is altered by the mutation operator in an entirely random manner, allowing us to add and preserve variation in our population of solutions. This operator is a "disturbing element" because it adds "noise" to the population. The algorithm's early convergence to a local maximum can be avoided with the help of this operator.

This operator has the following advantages:

(1) It guarantees the diversity of the population.

(2) It prevents a phenomenon known as genetic drift.

(3) It limits the risks of premature convergence caused, for example, by an elitist selection method imposing excessive selective pressure on the population.

A mutation strategy was incorporated into the particle updating process to achieve stronger exploration capability and a fast convergence rate.

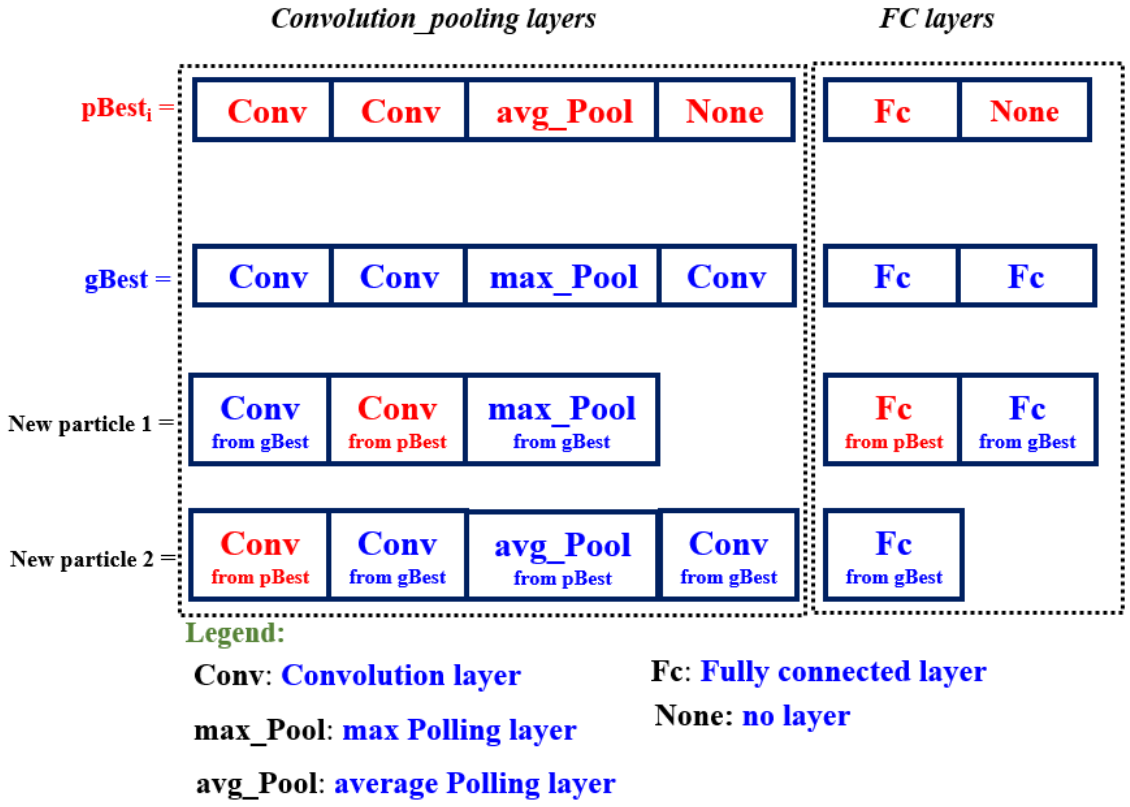

Figure 3.3: The crossover operator in the approach proposed

Algorithm 7: The proposed algorithm

Initialize randomly N model of CNN (population of PSO);
Evaluate particle positions
Initialize For each particle x_i,$pBest_i = x_i$
; **Calculate** $gBest$ according to Equation 1.13
; **while** *the stopping criteria is not satisfied* **do**
    Crossover between $pBest_i$ et $gBest$;
    Mutation of particles with probability p_m;
    Evaluate Particle Positions;
    Update $pBest_i$ and $gBest$ according to Equation 1.12 et Equation 1.13;
Return $gBest$ as the optimized solution.

3.2.6 fitness function

During this stage, the solutions within the population undergo assessment based on their objective function values. This process facilitates the categorization of solutions and the identification of those to be incorporated into a new population. For each particle within the initial population, a Convolutional Neural Network (CNN) is constructed and

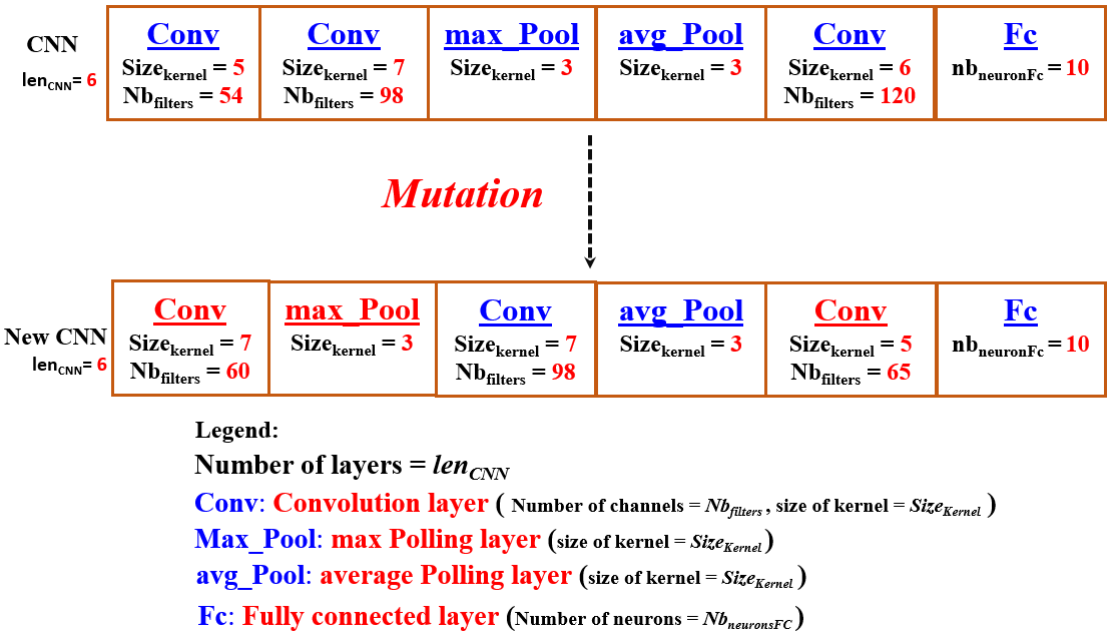

Figure 3.4: The mutation operator in the approach proposed

subsequently trained using a specific number of epochs on a designated training set. The accuracy of the CNN architecture is then computed to gauge the particle's performance. In evaluating the performance of each particle, their loss function is compared, utilizing the cross-entropy function in conjunction with the Adam optimizer [88]. Furthermore, techniques such as dropout, Batch Normalization, Regularization, and Xavier weight initialization [89] are used to mitigate the problem of overfitting.

Algorithm 8: Fitness Evaluation

Require: Particle architecture X, training dataset D_{train}, number of training epochs e_{train}

Ensure: Fitness value f (cross-entropy loss)

Compile the particle architecture X into a full-fledged CNN.
Initialize network weights using Xavier initialization.
Configure the CNN with dropout and batch normalization to mitigate overfitting.
Set the optimizer to Adam.
for $epoch = 1$ **to** e_{train} **do**
    Train the CNN on D_{train} for one epoch.
end for
Evaluate the trained CNN by computing the cross-entropy loss.
Return the computed loss f as the fitness value. =0

3.3 Experiment design

3.3.1 Peer Competitors

To evaluate the suggested GAPSO method's performance, we ran a comparative study using our rivals' most advanced optimized deep learning models. To be more precise, we chose rival algorithms that have published their findings on the same datasets used to assess the suggested GAPSO technique.

It is essential to draw attention to competitors that have optimized CNN designs using hybrid population-based methods, including genetic algorithms (GA) and particle swarm optimization (PSO). The complexity of these algorithms is similar to our suggested GAPSO approach.

We compared our results to many algorithms, including (HGAPSO [21], evoCNN [17], IPPSO [24], which relies on metaheuristics algorithms.

HGAPSO, a hybrid method of GA and PSO, is our most important competitor. Both GAPSO and HGAPSO share similarities in using hybrid evolutionary strategies for CNN optimization.

Also, other most recent algorithms with optimal parameters, which reported promising misclassifications in the selected criteria, are collected from recently published literature. Specifically, they are LeNet-5 [4],LSVM+RBF [75],SVM+Poly [75], CAE-1 [90], CAE- 2 [90],PCANet-2 [91],RandNet-2 [91], LDANet-2 [91].

3.3.2 Algorithm parameters

The parameters utilized in the proposed GAPSO algorithm can be categorized into three groups: PSO algorithm, Genetic algorithm, and CNN architecture, as outlined in Table 4.1.

The first category encompasses parameters that govern the behavior of both PSO and GA. These parameters include the number of executions, population size, number of iterations, and mutation rate.

The second category comprises parameters specific to CNN architectures, such as the number of layers, output dimensions of convolutional layers, neuron count in fully connected layers, convolutional filter size, dropout rate, learning rate, optimizer (Adam), and batch size.

The implementation of the proposed GAPSO method is carried out in Keras [92] with Tensorflow [93] as the primary backend. The execution of the GAPSO method is performed on a computer equipped with a GPU card featuring the Nvidia GeForce GTX1070 model.

Parameter	Value
PSO and Genetic algorithms	
Number of execution	10
Number of iterations	10
Population size	25
Mutation rate	0.25
Crossover rate	1.0
CNN architecture	
length of CNN layers	[3 - 20]
size of a Conv kernel	[3x3 -7x7]
number of filters	[1-256]
number of neurons in a Fc layer	[1-512]
epochs for particle evaluation	1
epochs for full training best CNN	100
Dropout rate	0.5
batch size	64
learning rate	0.001

Table 3.1: List of parameters used to evaluate the GAPSO.

3.3.3 Datasets

To assess the effectiveness of our approach, we employed five datasets outlined in section 1.5, which were also used by competitors. These datasets include MNIST,MNIST-RD+BI, MNIST-RD, Convex, Rectangles .

3.4 Results and analysis

3.4.1 Overall Results

Experimental results and comparison between GAPSO and state-of-the-art methods are shown in Table 3.2 (in the last three rows), including best error rate, average error rate and the standard deviations of the error rate obtained from the 10 runs.

Model / Dataset	MNIST	MNIST-RD+BI	MNIST-RD	Convex	Rectangles
LeNet-5 [74]	0.95(+)	-	-	-	-
SVM+RBF [75]	3.03(+)	32.62(+)	10.38(+)	19.13(+)	-
SVM+Poly [75]	3.69(+)	37.59(+)	13.61(+)	19.82(+)	-
CAE-1 [90]	2.83(+)	48.10(+)	11.59(+)	-	1.48(+)
CAE-2 [90]	2.48(+)	45.23(+)	9.66(+)	-	1.21(+)
PCANet-2 [91]	1.06(+)	35.86(+)	8.52(+)	4.19(+)	0.49(+)
RandNet-2 [91]	1.27(+)	43.69(+)	8.47(+)	5.45(+)	0.09(+)
LDANet-2 [91]	1.40(+)	38.54(+)	4.52(+)	7.22(+)	0.14(+)
evoCNN [17]	1.18(+)	35.03(+)	5.22(+)	4.82(+)	0.01(+)
IPPSO [24]	1.13(+)	34.50(+)	-	8.48(+)	-
HGAPSO [21]	0.74(+)	10.53(-)	-	1.03(-)	-
GAPSO (best)	**0.34**	**10.72**	**2.71**	**1.37**	**0.00**
GAPSO (mean)	**0.45**	**14.15**	**3.73**	**2.04**	**0.07**
GAPSO(std dev)	**0.05**	**1.90**	**0.53**	**0.44**	**0.08**

Table 3.2: Comparison of Test Errors for GAPSO Algorithm and Other Models Across Various Datasets.

The table presents the performance comparison of several algorithms, including LeNet-5, SVM+RBF, SVM+Poly, CAE-1, CAE-2, PCANet-2, RandNet-2, LDANet-2, evoCNN, IPPSO, and HGAPSO, alongside our proposed GAPSO method.

Each entry in the table indicates the misclassification rate achieved by the respective algorithm on different datasets. A lower misclassification rate signifies better performance. For example, a misclassification rate of 0.95 for LeNet-5 on the MNIST dataset indicates that 95

Comparing the performance of GAPSO with other algorithms, it is evident that GAPSO achieved the lowest misclassification rates across all datasets, indicated by the bold numbers. Additionally, the standard deviation values for GAPSO are relatively low, indicating consistency in performance across different executions.

To show clearly the results of the comparison, we use the terms (+), (-), and (=) as in Chapter 2. It can be observed that the proposed GAPSO method achieves a significant improvement in terms of the error rates shown in the Table 3.2. GAPSO significantly outperforms other competitors in the benchmark dataset.

To facilitate the presentation of comparison results, the terms (+) and (-) are employed to denote whether the GAPSO result is substantially superior or inferior to the corresponding competitor's best result, respectively. The symbol "=" indicates that the mean error rate of GAPSO is comparable to that of the competitor, indicating that the

difference is not statistically significant. The symbol (–) denotes the absence of reported results by the provider or their inability to be tallied. The results table indicates that the Table 3.2 presents a substantial enhancement in the error rates achieved by the proposed GAPSO method. GAPSO exhibits a markedly superior performance compared to its competitors in the benchmark dataset.

The proposed GAPSO algorithm exhibited outstanding performance across multiple datasets, achieving impressive error rates compared to other methods. Here are the detailed results:

- MNIST Dataset: GAPSO achieved the lowest error rate of 0.34, demonstrating its superior performance in accurately classifying handwritten digits.

GAPSO maintains low average error rates across all datasets, indicating reliable performance. For example, the error rate average for the Mnist dataset is 0.45, which is still better than the best value for HGAPSO.

GAPSO shows low standard deviation values, indicating stable and consistent performance across multiple runs. This low variation emphasizes the reliability of GAPSO.

- MNIST-RD + BI Dataset: GAPSO achieved an error rate of 10.72 on this dataset, surpassing the best performance of the HGAPSO method, which achieved an error rate of 10.53. Despite not achieving the lowest error rate, $GAPSO$ demonstrated competitive performance on this dataset.

- MNIST-RD Dataset: Our method outperformed all others on the MNIST-RD dataset, achieving the best performance with an error rate of 2.71. This indicates GAPSO's effectiveness in handling datasets with noise and background interference.

- Convex Dataset: GAPSO achieved an error rate of 1.37 on the convex dataset, slightly higher than the error rate of 1.03 obtained by the HGAPSO method. Although not the best, GAPSO still demonstrated strong performance on this dataset.

- Rectangles Dataset: Our method achieved the best error rate of 0.0 on the Rectangles dataset, indicating perfect classification performance. This underscores GAPSO's capability to handle datasets with well-defined geometric shapes effectively.

The detailed results highlight GAPSO's ability to achieve competitive or superior error rates across diverse datasets consistently, showcasing its effectiveness in optimizing CNN architectures for classification tasks.

Additionally, as observed in Table 3.3, our method obtained the optimal CNN architectures for each dataset with the fewest layers and parameters. This highlights the

algorithm's efficiency in designing compact yet practical models tailored to specific datasets.

Regarding the computational cost of the proposed GAPSO algorithm, each run on the reference dataset for 25 particles takes an average of two hours. This information provides insight into the time required for executing the algorithm and underscores the importance of computational resources in the optimization process.

3.4.2 Discussion

The proposed algorithm possesses a unique advantage in uncovering smaller CNN architectures compared to its competitors. This advantage stems from initializing the swarm with compact architectures, typically ranging from 3 to 20 layers. Unlike manually designed networks, which often contain redundant parameters, the algorithm emphasizes efficiency by favoring smaller network structures.

By leveraging populations generated from networks of diverse lengths, the algorithm explores a wide architectural spectrum. However, it tends to prioritize smaller networks during the optimization process. This preference for compact architectures allows the algorithm to converge more rapidly, as smaller networks generally require fewer computational resources and exhibit faster training times.

Furthermore, the algorithm's ability to efficiently navigate the search space of CNN architectures contributes to its effectiveness in identifying optimal solutions. By focusing on smaller architectures, the algorithm mitigates the risk of overfitting and promotes generalization, ultimately leading to superior performance on classification tasks.

Overall, the proposed algorithm's emphasis on smaller CNN architectures, coupled with its efficient optimization strategy, positions it as a robust and effective approach for architectural optimization in deep learning tasks.

The figure labeled "Evolution of the $gBest$ training accuracy GAPSO" shows how the fitness score of $gBest$ changes across ten runs, each consisting of ten iterations, on all datasets with 25 particles. All remaining parameters are listed in Table 3.1. The improvement in accuracy over time is clearly obvious. Convergence occurs rapidly. At first, the accuracy of the $gBest$ particles increases quickly until the second repetition. Afterwards, when the particles persist in their search, they encounter difficulties in finding superior solutions until the ninth repetition. The algorithm's swift convergence can be credited to endeavors focused on reducing the search space in the suggested coding approach and particle updating strategy.

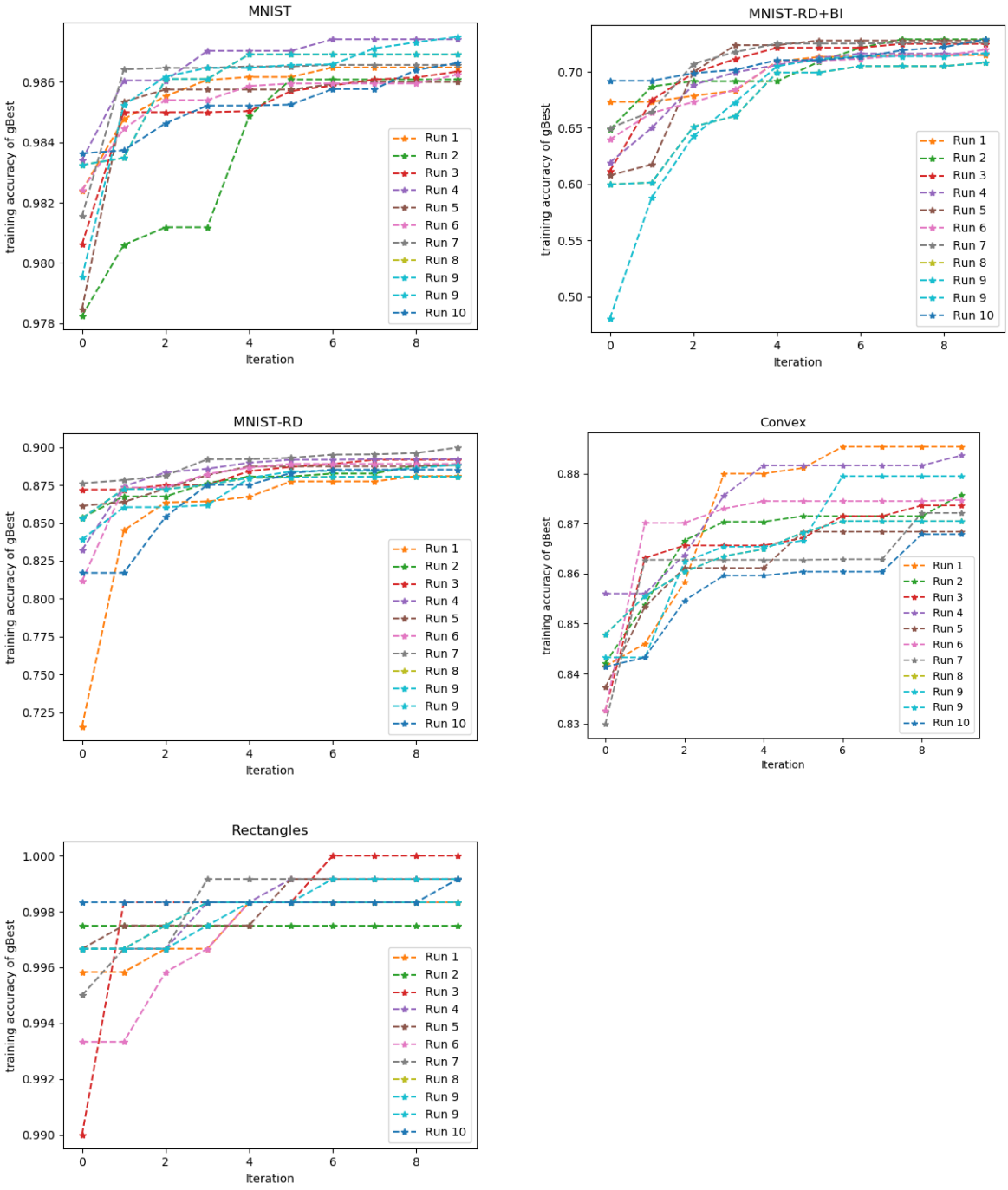

Figure 3.5: The evolution of the $gBest$ training accuracy for all datasets using the proposed GAPSO.

A crossover operator can be executed in this approach between the gBest and pBest solutions. The aim is to exchange information between the global best solution, representing the most promising solution found by any particle in the swarm, and the personal best solutions of individual particles. This exchange of information facilitates the propagation of beneficial features from the global best solution to individual particles, thereby enhancing exploration and convergence towards improved solutions.

The mutation operator in PSO can be applied to the particle positions to introduce random perturbations. These perturbations aid the particles in exploring new regions of the search space, potentially leading to the discovery of better solutions.

Note that most CNNs architecture (Table 3.3) are without much layer pooling, and to compensate for this, a larger step is used in the Convolutional layer from time to time.

The models crafted by our GAPSO algorithm surpass state-of-the-art models in three of the five datasets. Additionally, GAPSO derived models exhibit significantly fewer parameters compared to well-known architectures such as VGG16, GoogleNet, AlexNet, and evoCNN, as depicted in Table 3.4. For instance, the optimal model identified for the MNIST dataset contains 9.97 million parameters. This highlights the efficiency and effectiveness of GAPSO in designing compact yet high-performing CNN architectures tailored to specific datasets.

3.5 Conclusion

This study introduces a pioneering methodology named GAPSO, which combines Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) to optimize Convolutional Neural Network (CNN) architectures autonomously. Notably, this optimization process requires no human expertise in the target problem or the CNN designs. Remarkably, GAPSO outperforms Evolutionary Computing (EC) and non-EC competitors across various benchmarks, notably excelling on the widely recognized MNIST dataset.

GAPSO's primary strength lies in its capacity to streamline the CNN architecture design process, eliminating the need for human intervention, a characteristic often associated with advanced competitors in the field.

GAPSO employs a crossover operation between the global best (gBest) and personal best (pBest) solutions, facilitating the exchange of invaluable insights. This mechanism enables the dissemination of beneficial features by sharing the most promising solution attained by any particle with the individual best solutions. Consequently, this knowledge

Dataset	Layer	parameters
Mnist	Conv	kernel size: 5 x 5; output filters: 184
	Conv	kernel size: 4 x 4; output filters: 217
	Conv	kernel size: 6 x 6; output filters: 252
	Conv	kernel size: 6 x 6; output filters: 236
	Conv	kernel size: 6 x 6; output filters: 236
	Conv	kernel size: 6 x 6; output filters: 236
	Fully-Connected	output neurons: 10
MNIST-RD-BI	Conv	kernel size: 5 x 5; output filters: 184
	Conv	kernel size: 3 x 3; output filters: 34
	Conv	kernel size: 5 x 5; output filters: 239
	Conv	kernel size: 4 x 4; output filters: 108
	max Pool	Kernel size: 2 x 2; strides: 2 x 2
	Conv	kernel size: 5 x 5; output filters: 180
	Conv	kernel size: 6 x 6; output filters: 231
	Conv	kernel size: 6 x 6; output filters: 231
	Fully-Connected	output neurons: 10
MNIST-RD	Conv	kernel size: 5 x 5; output filters: 208
	Conv	kernel size: 5 x 5; output filters: 156
	Conv	kernel size: 5 x 5; output filters: 183
	Conv	kernel size: 6 x 6; output filters: 158
	Conv	kernel size: 5 x 5; output filters: 54
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Conv	kernel size: 5 x 5; output filters: 230
	Conv	kernel size: 5 x 5; output filters: 252
	Conv	kernel size: 5 x 5; output filters: 252
	Fully-Connected	output neurons: 10
Convex	Conv	kernel size: 4 x 4; output filters: 22
	Conv	kernel size: 5 x 5; output filters: 95
	Conv	kernel size: 5 x 5; output filters: 228
	Conv	kernel size: 5 x 5; output filters: 38
	Conv	kernel size: 5 x 5; output filters: 151
	Conv	kernel size: 6 x 6; output filters: 254
	Fully-Connected	output neurons: 13
	Fully-Connected	output neurons: 2
Rectangles	Conv	kernel size: 4 x 4; output filters: 155
	max Pool	Kernel size: 2 x 2; strides: 2 x 2
	Conv	kernel size: 6 x 6; output filters: 244
	Conv	kernel size: 6 x 6; output filters: 244
	Fully-Connected	output neurons: 127
	Fully-Connected	output neurons: 2

Table 3.3: Best CNN architecture found by GAPSO.

transfer fosters enhanced exploration and expedites the convergence toward superior solutions.

Dataset	**number of trainable parameters**
Mnist	9.97M
MNIST-RD-BI	4.91M
MNIST-RD	6.56M
Convex	4.92M
Rectangles	8.74M

Table 3.4: Trainable parameter counts for models designed using our proposed GAPSO method across various datasets.

CHAPTER 4

"MPSO: MUTATION-ENHANCED PARTICLE SWARM OPTIMIZATION FOR CNN ARCHITECTURE OPTIMIZATION"

Contents

4.1 Introduction

In our previous works, pswCNN and GAPSO, we utilized modified versions of the PSO algorithm. However, a notable limitation was the inability of particles to fully explore the search space, as they directly copied layers from personal and global best solutions. This reduced search diversity and increased the risk of getting stuck in local optima.

To overcome these drawbacks, this study introduces a novel variant of PSO that integrates mutation mechanisms from genetic algorithms. This hybrid approach, termed MPSO, aims to enhance search space exploration diversity and prevent premature convergence. By incorporating mutation into PSO, MPSO dynamically adjusts particle trajectories, enabling a more comprehensive exploration of potential DNN architectures.

This chapter outlines the theoretical framework for MPSO, elucidating its algorithmic structure and the employed mutation strategy. We evaluate MPSO's performance on various benchmark deep learning tasks and compare it with traditional PSO and other state-of-the-art neural architecture search techniques. Our findings demonstrate that MPSO expedites convergence towards optimal DNN architectures and outperforms competitors on complex tasks. This underscores the potential of mutation-enhanced swarm algorithms in automating DNN design.

4.2 The proposed algorithm

The training data and parameters for the CNN structures to be built, including the maximum number of layers for particles during initialization, are accepted as input parameters by the suggested technique, which is relevant to the particular problem. algorithm 9 provides a general sketch of the proposed MPSO. This algorithm comprises a PSO framework with six procedures that allow the search for optimal CNN architectures: velocity computation, particle update, mutation operator, initialization of a swarm of particles, fitness evaluation of individual particles, measurement of the difference between two particles, and efficient CNN representation. The ensuing subsections provide more information about these procedures.

For CNN representation, initialization of a swarm of particles, and fitness evaluation of individual particles, the procedures remain the same as in the previous two proposed works.

Algorithm 9: The proposed algorithm.

```
Initialize randomly N model of CNN (a swarm of PSO)
Initialize particles' positions and velocities
while termination condition not met do
  for each particle do
    Update particle's velocity
    Update particle's position
    Apply mutation operator to particle's position
    Evaluate particle's fitness
    if particle's fitness is better than pBest then
      Update pBest
    end if
    if particle's fitness is better than gBest then
      Update gBest
    end if
  end for
end while=0
```

4.2.1 Difference calculation between two particles

To compute a single particle's velocity, we first must define how we measure the difference between two particles. This procedure is outlined in Figure 4.1, where particles P1 and P2 are compared. To prevent the creation of fully-connected layers between convolution and pooling (Conv/Pool) layers and thus avoid an invalid CNN architecture, we independently compare each particle's Conv/Pool and FC layers. This comparison considers only the layer type of each particle. For example, suppose both particles have a convolution layer as their first layer. In that case, the difference will be zero, indicating to the velocity operator that this layer position should remain unchanged during updates to a particle's architecture. Additionally, the comparison is always relative to the first particle. If both particles have different layer types, the first particle's layer with its corresponding hyperparameters is retained. If the first particle has fewer layers than the second one, those additional layers are marked for removal. Conversely, if the first particle has more layers than the second one, layers will be added to the final difference.

4.2.2 Velocity computation

In the proposed MPSO algorithm, a particle's velocity (P) is determined by computing the differences between the global best (gBest) and the particle's best (pBest). This process is illustrated in Figure 4.2, where differences (gBest P) and (pBest P) are

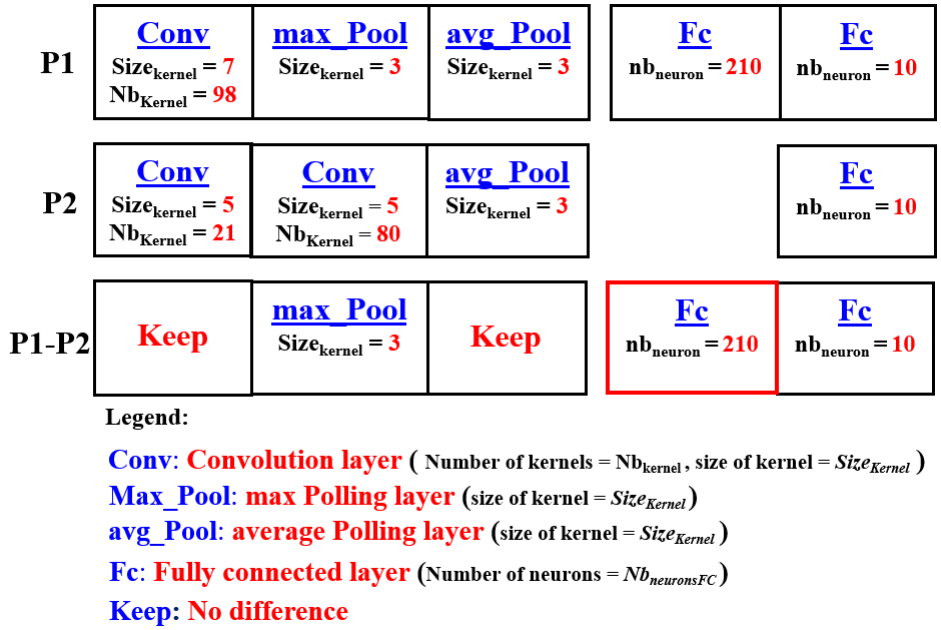

Figure 4.1: Difference calculation between two particles.

calculated. The final velocity is chosen from these differences based on a decision factor, Cg. If a randomly drawn number (r) falls within [0, 1) and is less than or equal to Cg, a layer from (gBest P) is selected; otherwise, a layer from (pBest P) is chosen. This decision factor controls the convergence rate towards gBest. Additionally, the velocity computation separates Conv/Pool layers from FC layers to ensure a precise adjustment of the CNN architecture.

4.2.3 The mutation operator

In the MPSO algorithm, mutations randomly alter parameters of a specific layer, as shown in Figure 4.3. This random mutation introduces diversity and acts as a "disturbing element" by injecting "noise" into the solution population. This process prevents the algorithm from prematurely converging to a local maximum, ensuring broader solution space exploration.

4.2.4 Particle update

Updating a particle's architecture is the most straightforward procedure in the proposed MPSO algorithm, as demonstrated in algorithm 10. The algorithm adjusts the particle's architecture based on its velocity, identifying which layers to modify. Layers

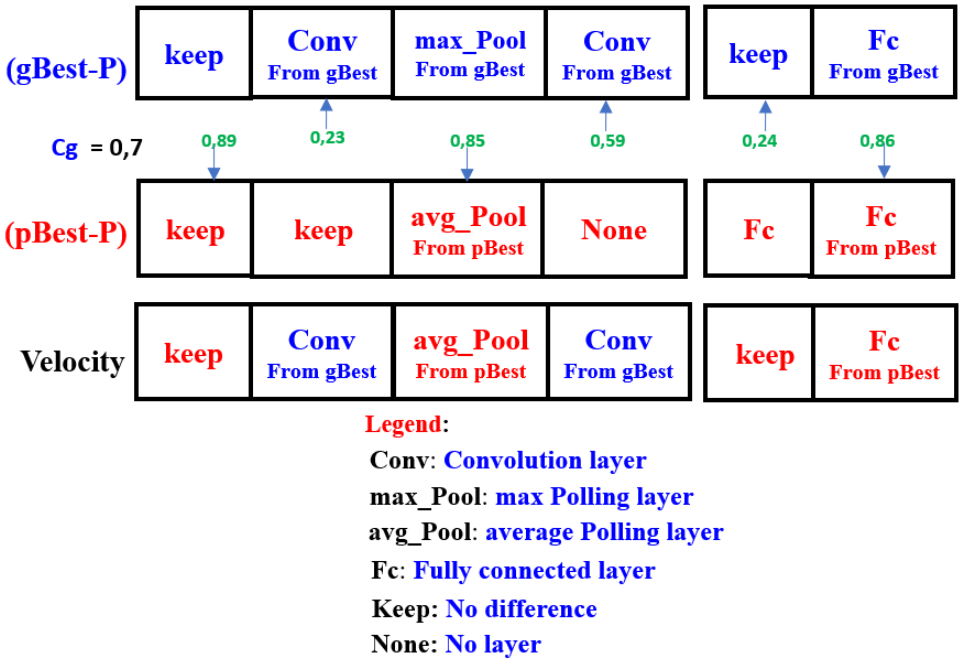

Figure 4.2: Example of the velocity computation.

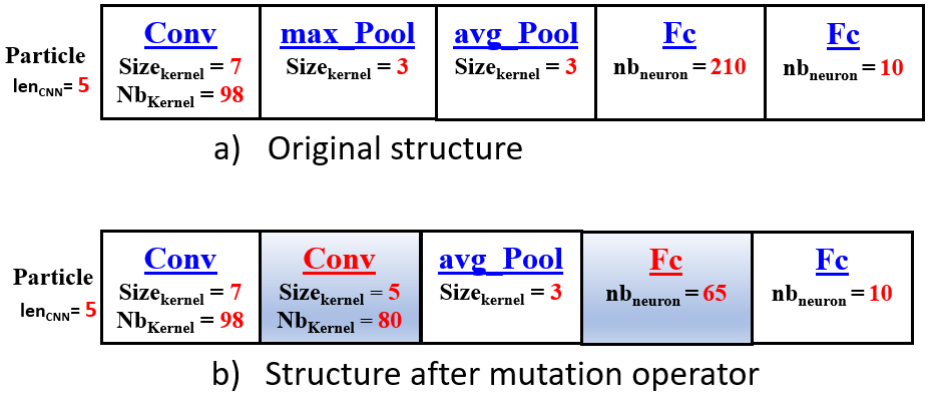

Figure 4.3: The mutation operator in the approach proposed.

are either added or removed according to the particle's velocity. The algorithm must also monitor the number of pooling layers in the architecture. Given the size of the training inputs, only a limited number of pooling layers is permissible. If a particle ends up with more pooling layers than allowed after the update, the excess layers are removed sequentially from the last layer to the first. This process is illustrated in Figure 4.4.

Algorithm 10: UpdateParticle

Input : $p, p.velocity$
Output : Updated particle

```
import deepcopy from copy
newParticle ← deepcopy(p)
for P_layer, V_layer ∈ zip(p, p.velocity) do
    if V_layer = "keep" then
        newParticle[P_layer] ← P_layer
    else
        newParticle[P_layer] ← V_layer
    end
end
```

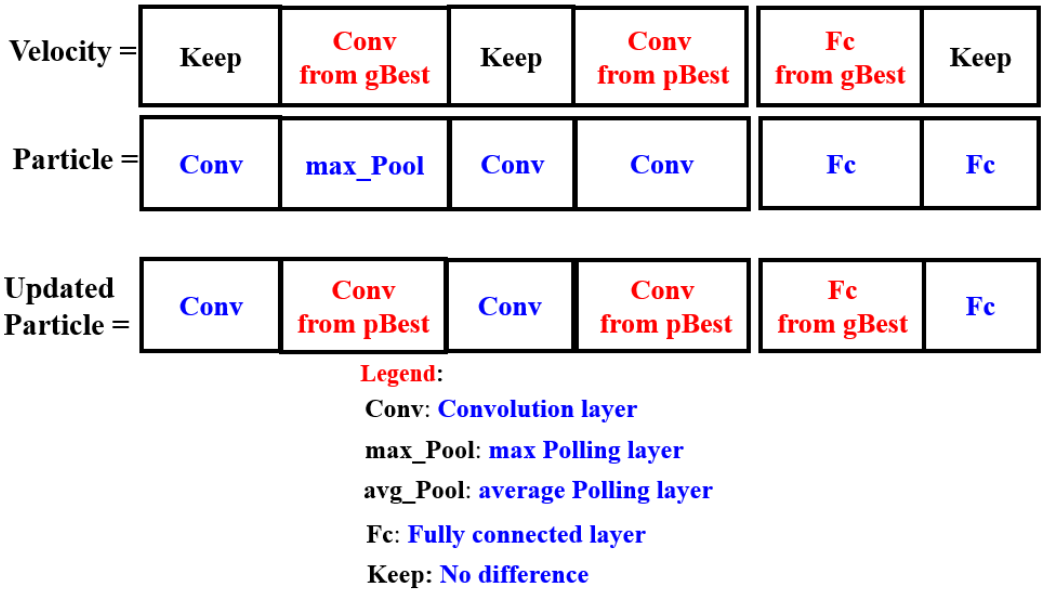

Figure 4.4: Example of a particle architecture update.

4.3 Experiment design

4.3.1 Peer Competitors

We compared the MPSO algorithm's performance to state-of-the-art optimized deep learning models. By evaluating results from the same datasets, we aimed to highlight the effectiveness and competitiveness of MPSO. This analysis provides insights into its advantages, such as improved accuracy or convergence speed. Our comparison included algorithms like HGAPSO [21], evoCNN [17], IPPSO [24], along with recent models like

LeNet-5 [4],LSVM+RBF [75],SVM+Poly [75], CAE-1 [90], CAE- 2 [90],PCANet-2 [91],RandNet-2 [91], LDANet-2 [91].

The provided information introduces a new competitor method, presented in a paper by Munsarif et al. [23], which relies on the Particle Swarm Optimization (PSO) algorithm. This method, termed LDRW-PSO, utilizes randomized weights within the PSO framework to optimize Convolutional Neural Networks' hyperparameters (CNNs). The experiments conducted on the MNIST dataset reveal that although the proposed LDRW-PSO method achieves superior accuracy compared to random search, its execution time is slower. This insight suggests that while LDRW-PSO may offer advantages in terms of optimization performance, such as achieving higher accuracy, it may come at the cost of increased computational time. This trade-off highlights the importance of considering accuracy and computational efficiency when evaluating optimization methods, especially in scenarios where time constraints are crucial.

Overall, the introduction of LDRW-PSO as a competitor method enriches the comparative analysis and provides valuable insights into the strengths and weaknesses of different optimization approaches for CNN hyperparameter optimization.

4.3.2 Algorithm parameters

The MPSO method categorizes parameters into three groups: PSO algorithm, Mutation operator, and CNN architecture. PSO algorithm parameters include execution number, population size, iteration count, and mutation rate. CNN architecture parameters cover layer count, convolutional layer output size, fully connected layer neuron count, and filter size. Additional parameters include dropout rate, learning rate, optimizer (Adam), and batch size. The method is built using the Keras framework [92] in conjunction with Tensorflow [93] as the primary backend, executed on an Nvidia GeForce GTX1070 GPU card.

4.3.3 Datasets

To evaluate the effectiveness of our method, we used six datasets from section 1.5, which were also utilized by competitors. These datasets include MNIST, MNIST-RD+BI, MNIST-BI, MNIST-RB, Rectangles, and Convex.

Parameter	Value
PSO and Mutation operator	
Number of execution	10
Number of iterations	10
Population size	25
Mutation rate	0.25
CNN architecture	
Length of CNN layers	[3 - 20]
Size of a Conv kernel	[3x3 -7x7]
Feature maps	[1-256]
Conv stride size	1
Pool stride size	2
Pool type	average or max
Pool kernel size	3
Number of neurons in a FC layer	[1-512]
Training of CNN	
Activation function	ReLu
Weight initialization	Xavier
Optimizer	Adam
Learning rate	0.001
Batch size	32
Dropout rate	0.5
No. of epochs for single particle evaluation	1
No. of epochs for final particle	100

Table 4.1: List of parameters used to evaluate the MPSO.

4.4 Results and analysis

4.4.1 results

The passage describes the content of the final three rows of a Table 3.2. These rows present experimental results and a comparative analysis of MPSO with state-of-the-art techniques. Specifically, they showcase the best error rate, average error rate, and standard deviations of the error rate obtained from ten independent runs of the experiment. This information allows readers to assess MPSO's performance in comparison to other cutting-edge techniques, providing insights into its effectiveness and reliability across multiple runs. The inclusion of standard deviations offers additional context by indicating the variability of results observed during the experimental runs, thus providing a more comprehensive understanding of the algorithm's performance stability.

The **??** comparison findings unequivocally show that the MPSO approach outperforms

its rivals on the benchmark dataset. The MPSO method proposed achieved its best error rate of 0.37 for the: MNIST dataset. Across ten runs on this dataset, the average error rate was 0.47, with a standard deviation of 0.06.

Our method recorded an error of 14.65 on the MNIST-RD + BI dataset, which did not outperform the HGAPSO method, which achieved a lower error of 10.53.

Our method excelled on the MNIST-BI dataset, achieving the lowest error rate of 2.71, outperforming all competing methods. Similarly, our method topped the charts on the MNIST-RB dataset with an error rate of 1.74, surpassing all other methods.

Our method registered an error of 1.50 for the convex dataset, slightly higher than the 1.03 error achieved by the HGAPSO method. However, our method achieved the best error rate of 0.01 on the Rectangles dataset.

The best CNN architectures optimized using Mutation-Enhanced Particle Swarm Optimization (MPSO) are presented in Table 4.3.

This table Table 4.4 presents the number of trainable parameters for models developed using the MPSO method across various datasets. Understanding the number of trainable parameters helps assess the model's complexity and resource requirements.

Figure 4.5 shows the training accuracy of the global best (gBest) solutions over successive iterations for ten different runs. Each line represents a distinct run, illustrating the convergence behavior and improvements in accuracy achieved by the MPSO algorithm across multiple trials. The figure highlights consistent early gains in accuracy, with most runs reaching a high level of performance within the first few iterations, demonstrating the effectiveness of MPSO in optimizing CNN architectures for all datasets.

On average, running our proposed MPSO algorithm on the reference dataset with 25 particles typically takes about fourteen hours to complete.

4.4.2 Discussion

The MPSO algorithm demonstrates a remarkable ability to identify compact CNN architectures, outperforming traditional methods by initializing with a broad range of layer counts from 3 to 20. This initial diversity helps avoid the redundancy typical in manual setups and leads to the discovery of advanced, streamlined models. Additionally, the variation in network sizes within the population promotes quicker convergence for smaller networks, enhancing overall efficiency.

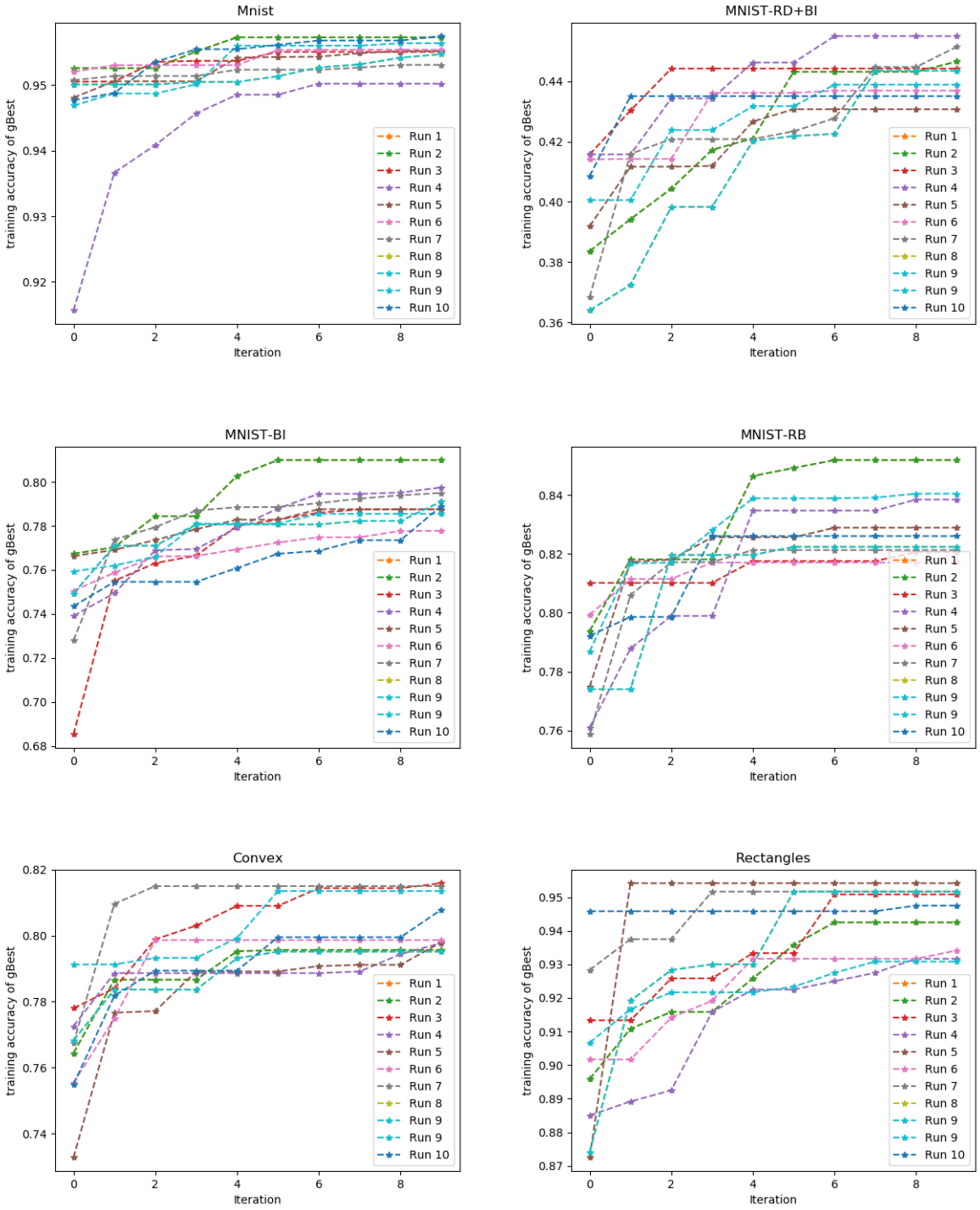

Figure 4.5: The evolution of the $gBest$ training accuracy for all datasets using the proposed MPSO.

Model / Dataset	MNIST	MNIST-RD+BI	MNIST-BI	MNIST-RB	Convex	Rectangles
LeNet-5 [74]	0.95(+)	-	-	-	-	-
SVM+RBF [75]	3.03(+)	32.62(+)	10.38(+)	14.58(+)	19.13(+)	-
SVM+Poly [75]	3.69(+)	37.59(+)	13.61(+)	16.62(+)	19.82(+)	-
CAE-1 [90]	2.83(+)	48.10(+)	11.59(+)	13.57(+)	-	1.48(+)
CAE-2 [90]	2.48(+)	45.23(+)	9.66(+)	10.90(+)	-	1.21(+)
PCANet-2 [91]	1.06(+)	35.86(+)	8.52(+)	6.85(+)	4.19(+)	0.49(+)
RandNet-2 [91]	1.27(+)	43.69(+)	8.47(+)	13.47(+)	5.45(+)	0.09(+)
LDANet-2 [91]	1.40(+)	38.54(+)	4.52(+)	6.81(+)	7.22(+)	0.14(+)
evoCNN [17]	1.18(+)	35.03(+)	5.22(+)	2.80(+)	4.82(+)	0.01(=)
IPPSO [24]	1.13(+)	34.50(+)	-	-	8.48(+)	-
HGAPSO [21]	0.74(+)	10.53(-)	-	-	1.03(-)	-
LDRW-PSO [23]	0.91(+)	-	-	-	-	-
MPSO (best)	**0.37**	**14.65**	**2.72**	**1.74**	**1.50**	**0.01**
MPSO (mean)	**0.47**	**20.07**	**3.73**	**2.20**	**2.82**	**0.23**
MPSO(std dev)	**0.06**	**3.08**	**0.53**	**0.36**	**0.88**	**0.28**

Table 4.2: Comparison of Test Errors for MPSO Algorithm and Other Models Across Various Datasets

The architectures presented in Table 4.3 illustrate the effectiveness of MPSO in optimizing CNN layers for various datasets. Each configuration, from kernel size to the number of output filters and the use of pooling layers, reflects a tailored approach to maximize performance for the specific characteristics of each dataset. This diversity in design underscores the flexibility and robustness of MPSO in neural network architecture optimization.

In performance comparisons, MPSO optimized models surpass leading models on half of the datasets tested. These models are particularly efficient, featuring significantly fewer parameters than well-known networks like VGG16, GoogleNet, AlexNet, and evoCNN. Notably, the most effective MPSO model for the MNIST dataset consists of only 2.6 million parameters Table 4.4. The variance in the number of parameters suggests that the MPSO method adapts the model complexity based on the dataset's nature and requirements. For example, the MNIST-BI dataset requires a more complex model than the simpler Rectangles dataset.

The evolutionary dynamics of MPSO, as illustrated in Figure 4.5, demonstrate the algorithm's ability to quickly and consistently improve the training accuracy of CNN architectures for all datasets. The rapid convergence in the early iterations and the consistent final accuracies across different runs underline MPSO's effectiveness in

Dataset	Layer	parameters
MNIST	Conv	kernel size: 6 x 6; output filters: 240
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Conv	kernel size: 4 x 4; output filters: 245
	Conv	kernel size: 5 x 5; output filters: 223
	Fully-Connected	output neurons: 10
MNIST-RD-BI	Conv	kernel size: 3 x 3; output filters: 243
	Conv	kernel size: 4 x 4; output filters: 107
	Conv	kernel size: 6 x 6; output filters: 132
	Conv	kernel size: 6 x 6; output filters: 210
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Fully-Connected	output neurons: 10
MNIST-BI	Conv	kernel size: 4 x 4; output filters: 244
	Conv	kernel size: 6 x 6; output filters: 232
	Conv	kernel size: 6 x 6; output filters: 255
	Fully-Connected	output neurons: 10
MNIST-RB	Conv	kernel size: 4 x 4; output filters: 201
	Conv	kernel size: 6 x 6; output filters: 105
	Conv	kernel size: 6 x 6; output filters: 213
	Conv	kernel size: 6 x 6; output filters: 225
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Fully-Connected	output neurons: 10
Convex	Conv	kernel size: 4 x 4; output filters: 214
	Conv	kernel size: 5 x 5; output filters: 248
	Conv	kernel size: 3 x 3; output filters: 117
	Conv	kernel size: 4 x 4; output filters: 236
	Conv	kernel size: 4 x 4; output filters: 82
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Fully-Connected	output neurons: 2
Rectangles	Conv	kernel size: 6 x 6; output filters: 213
	Conv	kernel size: 6 x 6; output filters: 86
	max Pool	Kernel size: 2 x 2; strides: 2 x 2
	avg Pool	Kernel size: 2 x 2; strides: 2 x 2
	Conv	kernel size: 6 x 6; output filters: 226
	Fully-Connected	output neurons: 2

Table 4.3: Best CNN architecture found by MPSO.

optimizing neural network parameters efficiently.

MPSO favors CNN architectures with fewer pooling layers, compensating by increasing convolutional layers to capture complex features. This strategy ensures sustained performance by facilitating detailed feature learning. By merging mutation operator and particle swarm optimization, MPSO autonomously refines architectures for optimal performance, showcasing its capacity to adapt and optimize within specific domains.

Dataset	number of trainable parameters
Mnist	2.6M
MNIST-RD-BI	2.2M
MNIST-BI	4.1M
MNIST-RB	3.6M
Convex	2.3M
Rectangles	1.3M

Table 4.4: Trainable parameter counts for models designed using our proposed MPSO method across various datasets.

4.5 Conclusion

In conclusion, the Mutation-Enhanced Particle Swarm Optimization (MPSO) algorithm presents a novel approach for autonomously optimizing convolutional neural network (CNN) architectures. By integrating the Mutation operator with a traditional particle swarm optimization algorithm, MPSO performs better than traditional methods, particularly in identifying compact CNN architectures with fewer pooling layers. This strategic approach and effective convergence mechanisms enable MPSO to efficiently explore the search space and discover advanced, streamlined models. Moreover, MPSO's ability to autonomously refine network designs highlights its potential for enhancing CNN optimization methodologies. Future research directions may include validating MPSO on larger datasets and exploring its adaptability to newer CNN architectures, further advancing autonomous CNN optimization techniques.

GENERAL CONCLUSION

This part serves as the conclusion of the study, bringing together the key findings in relation to the research goals and questions, as well as highlighting their significance and contribution. It also addresses the limitations encountered during the research and provides recommendations for future studies.

Conclusions

In this thesis, we have tackled the challenging task of designing optimal CNN architectures for image classification by introducing three novel optimization approaches: PSO without Velocity Equation (pswvCNN), a hybrid PSO with Genetic Algorithms (GAPSO), and Mutation-Enhanced Particle Swarm Optimization (MPSO). Each of these methods aims to minimize human intervention, achieve quick convergence, and significantly reduce the time required to discover the best CNN designs.

pswvCNN: pswvCNN offers a streamlined approach to optimizing CNN architectures by focusing solely on position updates and eliminating the velocity component of traditional PSO. This simplification makes the algorithm more accessible and reduces computational complexity while maintaining effective solution space exploration. pswvCNN provides a robust method for automating and improving the design of CNN architectures, making it a valuable tool in the field of deep learning.

GAPSO: GAPSO represents a robust and effective method for optimizing CNN architectures by leveraging the complementary strengths of PSO and GA. By enhancing exploration through PSO and introducing genetic diversity through GA operations, GAPSO provides a comprehensive approach to discovering high-performance CNN models. This hybrid optimization technique is valuable for automating and improving the design of deep learning architectures, making it an essential tool in the field of machine learning.

MPSO: MPSO offers a powerful approach to optimizing CNN architectures by leveraging mutation operations to enhance the exploration capabilities of standard PSO. This results in more effective and efficient discovery of high-performance CNN models, making MPSO a valuable tool for automating and improving the design of deep learning architectures..

Each of these PSO variants enhances the basic PSO algorithm by addressing specific limitations. pswvCNN simplifies the update mechanism for potentially better precision,

GAPSO leverages genetic diversity for improved exploration, and MPSO uses mutation to avoid local optima and enhance global search capabilities.

Our extensive evaluation of these approaches on nine benchmark datasets has demonstrated their effectiveness in identifying CNN structures that perform comparably to, or even better than, standard designs. By employing a varying-length encoding strategy and integrating innovative particle updating methods, these algorithms have shown substantial promise in navigating the complex parameter space of CNN architectures.

The comparative analysis with 27 contemporary algorithms from the literature further underscores the competitiveness of our proposed approaches. Empirical findings validate that PSWV, GAPSO, and MPSO not only reduce the need for expert intervention but also expedite the convergence process, making them valuable tools for the efficient and effective design of deep neural networks.

In summary, pswvCNN, GAPSO, and MPSO represent significant advancements in the field of neural architecture search, offering practical solutions to the intricate problem of CNN design. Future work can explore further enhancements to these algorithms, including adaptive mechanisms for parameter tuning and the application of these methods to other deep learning tasks beyond image classification.

Future perspectives

It is important to consider conducting further research that includes larger datasets like ImageNet and CIFAR-100 to evaluate the method thoroughly. Additionally, we intend to examine and analyze several hyperparameters of Convolutional Neural Networks (CNNs), such as the number of epochs, learning rate, activation function, optimizer, dropout rate, convolutional layer stride, padding type, and other related factors. This investigation will provide a more profound understanding of the influence of these hyperparameters on CNNs' performance.

Furthermore, our intention is to investigate evolutionary methods specifically designed for recurrent neural networks (RNNs) to address the challenges presented by time-varying inputs such as speech and video data. We are also interested in exploring the real-time processing of input images or extracting them from video sources.

Another area of focus will be the advancement of the pswvCNN algorithm to enable the creation of parallel CNN architectures capable of efficiently handling more complex

networks. This development will enhance the algorithm's scalability and applicability to diverse and demanding CNN architectures.

Additionally, we aim to investigate the design of a non-PSO metaheuristic algorithm or a hybrid approach that can generate optimized solutions within a shorter time frame. This research direction will explore alternative optimization strategies and their potential to efficiently produce high-quality results.

Overall, these future endeavors will contribute to the broader understanding and improvement of CNN architecture optimization, enabling more robust and efficient solutions for various applications.

BIBLIOGRAPHY

[1] D. Elhani, A. C. Megherbi, A. Zitouni, F. Dornaika, S. Sbaa, and A. Taleb-Ahmed, "Optimizing convolutional neural networks architecture using a modified particle swarm optimization for image classification," *Expert Systems with Applications*, p. 120411, 2023.

[2] D. Elhani and A. C. Megherbi, "Mpso: Mutation-enhanced particle swarm optimization for cnn architecture optimization," in *2024 International Conference on Advances in Electrical and Communication Technologies (ICAECOT)*, pp. 1–6, IEEE, 2024.

[3] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, p. 106, 1962.

[4] X. Liu, Z. Deng, and Y. Yang, "Recent progress in semantic image segmentation," *Artificial Intelligence Review*, vol. 52, no. 2, pp. 1089–1106, 2019.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition (2015). cite," *arXiv preprint arxiv:1512.03385*.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[8] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

[9] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

[10] S. Shekhar, A. Bansode, and A. Salim, "A comparative study of hyper-parameter optimization tools," in *2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pp. 1–6, 2021.

[11] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization.," *Journal of machine learning research*, vol. 13, no. 2, 2012.

[12] R. Martinez-Cantin, "Bayesopt: a bayesian optimization library for nonlinear optimization, experimental design and bandits.," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3735–3739, 2014.

[13] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[14] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[15] F. Hutter, L. Kotthoff, and J. Vanschoren, *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[16] M. Feurer and F. Hutter, "Hyperparameter optimization," *Automated machine learning: Methods, systems, challenges*, pp. 3–33, 2019.

[17] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Evolving deep convolutional neural networks for image classification," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 394–407, 2019.

[18] Y. Sun, B. Xue, M. Zhang, G. G. Yen, and J. Lv, "Automatically designing cnn architectures using the genetic algorithm for image classification," *IEEE transactions on cybernetics*, vol. 50, no. 9, pp. 3840–3854, 2020.

[19] S. Loussaief and A. Abdelkrim, "Convolutional neural network hyper-parameters optimization based on genetic algorithms," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, pp. 252–266, 2018.

[20] N. M. Aszemi and P. Dominic, "Hyperparameter optimization in convolutional neural network using genetic algorithms," *Int. J. Adv. Comput. Sci. Appl*, vol. 10, no. 6, pp. 269–278, 2019.

[21] B. Wang, Y. Sun, B. Xue, and M. Zhang, "A hybrid ga-pso method for evolving architecture and short connections of deep convolutional neural networks," in *pacific rim international conference on artificial intelligence*, pp. 650–663, Springer, 2019.

[22] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the genetic and evolutionary computation conference*, pp. 497–504, 2017.

[23] M. Munsarif, M. Sam'an, and A. Fahrezi, "Convolution neural network hyperparameter optimization using modified particle swarm optimization," *Bulletin of Electrical Engineering and Informatics*, vol. 13, no. 2, pp. 1268–1275, 2024.

[24] B. Wang, Y. Sun, B. Xue, and M. Zhang, "Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE, 2018.

[25] F. E. F. Junior and G. G. Yen, "Particle swarm optimization of deep neural networks architectures for image classification," *Swarm and Evolutionary Computation*, vol. 49, pp. 62–74, 2019.

[26] Y. Li, J. Xiao, Y. Chen, and L. Jiao, "Evolving deep convolutional neural networks by quantum behaved particle swarm optimization with binary encoding for image classification," *Neurocomputing*, vol. 362, pp. 156–165, 2019.

[27] P. Singh, S. Chaudhury, and B. K. Panigrahi, "Hybrid mpso-cnn: Multi-level particle swarm optimized hyperparameters of convolutional neural network," *Swarm and Evolutionary Computation*, vol. 63, p. 100863, 2021.

[28] J. Fregoso, C. I. Gonzalez, and G. E. Martinez, "Optimization of convolutional neural networks architectures using pso for sign language recognition," *Axioms*, vol. 10, no. 3, p. 139, 2021.

[29] T. Lawrence, L. Zhang, C. P. Lim, and E.-J. Phillips, "Particle swarm optimization for automatically evolving convolutional neural networks for image classification," *IEEE Access*, vol. 9, pp. 14369–14386, 2021.

[30] S. C. Nistor and G. Czibula, "Intelliswas: Optimizing deep neural network architectures using a particle swarm-based approach," *Expert Systems with Applications*, vol. 187, p. 115945, 2022.

[31] F. Miao, L. Yao, and X. Zhao, "Evolving convolutional neural networks by symbiotic organisms search algorithm for image classification," *Applied Soft Computing*, vol. 109, p. 107537, 2021.

[32] N. Bacanin, T. Bezdan, E. Tuba, I. Strumberger, and M. Tuba, "Monarch butterfly optimization based convolutional neural network design," *Mathematics*, vol. 8, no. 6, p. 936, 2020.

[33] I. Strumberger, E. Tuba, N. Bacanin, M. Zivkovic, M. Beko, and M. Tuba, "Designing convolutional neural network architecture by the firefly algorithm," in *2019 International Young Engineers Forum (YEF-ECE)*, pp. 59–65, IEEE, 2019.

[34] B. Wang, Y. Sun, B. Xue, and M. Zhang, "A hybrid differential evolution approach to designing deep convolutional neural networks for image classification," in *Australasian Joint Conference on Artificial Intelligence*, pp. 237–250, Springer, 2018.

[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[36] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 249–270, 2020.

[37] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[38] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE access*, vol. 7, pp. 53040–53065, 2019.

[39] C. Gulcehre, K. Cho, R. Pascanu, and Y. Bengio, "Learned-norm pooling for deep feedforward and recurrent neural networks," in *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part I 14*, pp. 530–546, Springer, 2014.

[40] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[41] D.-X. Zhou, "Theory of deep convolutional neural networks: Downsampling," *Neural Networks*, vol. 124, pp. 319–327, 2020.

[42] G. Li, M. Zhang, J. Li, F. Lv, and G. Tong, "Efficient densely connected convolutional neural networks," *Pattern Recognition*, vol. 109, p. 107610, 2021.

[43] W. Fang, P. E. Love, H. Luo, and L. Ding, "Computer vision for behaviour-based safety in construction: A review and future directions," *Advanced Engineering Informatics*, vol. 43, p. 100980, 2020.

[44] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[45] J. Moolayil, J. Moolayil, and S. John, *Learn Keras for deep neural networks*. Springer, 2019.

[46] N. Ketkar, J. Moolayil, N. Ketkar, and J. Moolayil, "Introduction to pytorch," *Deep Learning with Python: Learn Best Practices of Deep Learning Models with PyTorch*, pp. 27–91, 2021.

[47] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

[48] M. N. Omidvar, X. Li, and X. Yao, "A review of population-based metaheuristics for large-scale black-box global optimization—part i," *IEEE Transactions on Evolutionary Computation*, vol. 26, no. 5, pp. 802–822, 2021.

[49] B. Rabbouch, H. Rabbouch, F. Saâdaoui, and R. Mraihi, "Chapter 22 - foundations of combinatorial optimization, heuristics, and metaheuristics," in *Comprehensive Metaheuristics* (S. Mirjalili and A. H. Gandomi, eds.), pp. 407–438, Academic Press, 2023.

[50] J. Xu and J. Zhang, "Exploration-exploitation tradeoffs in metaheuristics: Survey and analysis," in *Proceedings of the 33rd Chinese control conference*, pp. 8633–8638, IEEE, 2014.

[51] H. D. Purnomo and H.-M. Wee, "Soccer game optimization with substitute players," *Journal of Computational and Applied Mathematics*, vol. 283, pp. 79–90, 2015.

[52] F. Glover, M. Laguna, and R. Marti, "Principles of tabu search," *Approximation algorithms and metaheuristics*, vol. 23, pp. 1–12, 2007.

[53] T. A. Feo and M. G. Resende, "Greedy randomized adaptive search procedures," *Journal of global optimization*, vol. 6, pp. 109–133, 1995.

[54] H. R. Lourenço, O. C. Martin, and T. Stützle, *Iterated local search*. Springer, 2003.

[55] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[56] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.

[57] M. Dorigo and G. Di Caro, "Ant colony optimization: a new meta-heuristic," in *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, vol. 2, pp. 1470–1477, IEEE, 1999.

[58] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009.

[59] M. H. K. Roni, M. Rana, H. Pota, M. M. Hasan, and M. S. Hussain, "Recent trends in bio-inspired meta-heuristic optimization techniques in control applications for electrical systems: A review," *International Journal of Dynamics and Control*, pp. 1–13, 2022.

[60] D. Molina, J. Poyatos, J. D. Ser, S. García, A. Hussain, and F. Herrera, "Comprehensive taxonomies of nature-and bio-inspired optimization: Inspiration versus algorithmic behavior, critical analysis recommendations," *Cognitive Computation*, vol. 12, pp. 897–939, 2020.

[61] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, 2013. Prediction, Control and Diagnosis using Advanced Neural Computations.

[62] K. Danach, *Hyperheuristics in logistics*. PhD thesis, Ecole centrale de Lille, 2016.

[63] J.-P. Courat, G. Raynaud, I. Mrad, and P. Siarry, "Electronic component model minimization based on log simulated annealing," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 41, no. 12, pp. 790–795, 1994.

[64] N. Collins, R. Eglese, and B. Golden, "Simulated annealing–an annotated bibliography," *American Journal of Mathematical and Management Sciences*, vol. 8, no. 3-4, pp. 209–307, 1988.

[65] M. Fleischer, "Simulated annealing: past, present, and future," in *Proceedings of the 27th conference on Winter simulation*, pp. 155–161, 1995.

[66] C. Koulamas, S. Antony, and R. Jaen, "A survey of simulated annealing applications to operations research problems," *Omega*, vol. 22, no. 1, pp. 41–56, 1994.

[67] P. J. Van Laarhoven, E. H. Aarts, P. J. van Laarhoven, and E. H. Aarts, *Simulated annealing*. Springer, 1987.

[68] L. Haldurai, T. Madhubala, and R. Rajalakshmi, "A study on genetic algorithm and its applications," *International Journal of computer sciences and Engineering*, vol. 4, no. 10, p. 139, 2016.

[69] M. Zemzami, N. El Hami, M. Itmi, and N. Hmina, "A comparative study of three new parallel models based on the pso algorithm," *International Journal for Simulation and Multidisciplinary Design Optimization*, vol. 11, p. 5, 2020.

[70] D. Chicco, "Ten quick tips for machine learning in computational biology," *BioData mining*, vol. 10, no. 1, p. 35, 2017.

[71] F. M. Talaat and S. A. Gamel, "Rl based hyper-parameters optimization algorithm (roa) for convolutional neural network," *Journal of Ambient Intelligence and Humanized Computing*, vol. 14, no. 10, pp. 13349–13359, 2023.

[72] N. de Freitas, "Bayesian optimization in a billion dimensions via random embeddings," 2016.

[73] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 847–855, 2013.

[74] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[75] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning*, pp. 473–480, 2007.

[76] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[77] A. Srinivasan, "Note on the location of optimal classifiers in n-dimensional roc space," tech. rep., Technical Report PRG-TR-2-99, Oxford University Computing Laboratory, Oxford . . . , 1999.

[78] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond accuracy, f-score and roc: a family of discriminant measures for performance evaluation," in *AI 2006: Advances in Artificial Intelligence: 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006. Proceedings 19*, pp. 1015–1021, Springer, 2006.

[79] A. P. Bradley, "The use of the area under the roc curve in the evaluation of machine learning algorithms," *Pattern recognition*, vol. 30, no. 7, pp. 1145–1159, 1997.

[80] V. García, R. A. Mollineda, and J. S. Sánchez, "Theoretical analysis of a performance measure for imbalanced data," in *2010 20th International Conference on Pattern Recognition*, pp. 617–620, IEEE, 2010.

[81] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets," *PloS one*, vol. 10, no. 3, p. e0118432, 2015.

[82] P. R. Lorenzo, J. Nalepa, L. S. Ramos, and J. R. Pastor, "Hyper-parameter selection in deep neural networks using parallel particle swarm optimization," in *Proceedings of the genetic and evolutionary computation conference companion*, pp. 1864–1871, 2017.

[83] S. Lankford and D. Grimes, "Neural architecture search using particle swarm and ant colony optimization," *arXiv preprint arXiv:2403.03781*, 2024.

[84] M. Li, W. Du, and F. Nian, "An adaptive particle swarm optimization algorithm based on directed weighted complex network," *Mathematical problems in engineering*, vol. 2014, pp. 1–7, 2014.

[85] M. M. El-Sherbiny, "Particle swarm inspired optimization algorithm without velocity equation," *Egyptian Informatics Journal*, vol. 12, no. 1, pp. 1–8, 2011.

[86] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[87] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[88] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv: 1412.6980*, 2014.

[89] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[90] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *ICML*, 2011.

[91] T.-H. Chan, K. Jia, S. Gao, J. Lu, Z. Zeng, and Y. Ma, "Pcanet: A simple deep learning baseline for image classification?," *IEEE transactions on image processing*, vol. 24, no. 12, pp. 5017–5032, 2015.

[92] F. Chollet, "Keras. https://github.com/fchollet/keras," 2015.

[93] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[94] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *International conference on machine learning*, pp. 1058–1066, PMLR, 2013.

[95] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[96] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, p. 84–90, 2012.

[97] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.