KTH ROYAL INSTITUTE OF TECHNOLOGY

Degree Project in Technology

Second cycle, 30 credits

EMONAS: Evolutionary Multi-objective Neuron Architecture Search of Deep Neural Network

JIAYI FENG

Stockholm, Sweden, 2023

EMONAS: Evolutionary Multi-objective Neuron Architecture Search of Deep Neural Network

JIAYI FENG

Master's Programme, Embedded Systems, 120 credits
Date: December 14, 2023

Supervisor: Vahid Geraeinejad
Examiner: Masoumeh Ebrahimi
School of Electrical Engineering and Computer Science
Swedish title: EMONAS: Evolutionär multi-objektiv neuronarkitektursökning av djupa neurala nätverk för inbyggda system

Abstract

Customized Deep Neural Network (DNN) accelerators have been increasingly popular in various applications, from autonomous driving and natural language processing to healthcare and finance, etc. However, deploying them directly on embedded system peripherals within real-time operating systems (RTOS) is not easy due to the paradox of the complexity of DNNs and the simplicity of embedded system devices. As a result, DNN implementation on embedded system devices requires customized accelerators with tailored hardware due to their numerous computations, latency, power consumption, etc. Moreover, the computational capacity, provided by potent microprocessors or graphics processing units (GPUs), is necessary to unleash the full potential of DNN, but these computational resources are often not easily available in embedded system devices.

In this thesis, we propose an innovative method to evaluate and improve the efficiency of DNN implementation within the constraints of resource-limited embedded system devices. The Evolutionary Multi-Objective Neuron Architecture Search-Binary One Optimization (EMONAS-BOO) optimizes both the image classification accuracy and the innovative Binary One Optimization (BOO) objectives, with Multiple Objective Optimization (MOO) methods. The EMONAS-BOO automates neural network searching and training, and the neural network architectures' diversity is also guaranteed with the help of an evolutionary algorithm that consists of tournament selection, polynomial mutation, and point crossover mechanisms. Binary One Optimization (BOO) is used to evaluate the difficulty in implementing DNNs on resource-limited embedded system peripherals, employing a binary format for DNN weights.

A deeper implementation of the innovative Binary One Optimization will significantly boost not only computation efficiency but also memory storage, power dissipation, etc. It is based on the reduction of weights binary 1's that need to be computed and stored, where the reduction of binary 1 brings reduced arithmetic operations and thus simplified neural network structures. In addition, analyzed from a digital circuit waveform perspective, the embedded system, in interpreting the neural network, will register an increase in zero weights leading to a reduction in voltage transition frequency, which, in turn, benefits power efficiency improvement.

The proposed EMONAS employs the MOO method which optimizes two objectives. The first objective is image classification accuracy, and the second objective is Binary One Optimization (BOO). This approach enables

EMONAS to outperform manually constructed and randomly searched DNNs. Notably, 12 out of 100 distinct DNNs maintained their image classification accuracy. At the same time, they also exhibit superior BOO performance. Additionally, the proposed EMONAS ensures automated searching and training of DNNs. It achieved significant reductions in key performance metrics: Compared with random search, evolutionary-searched BOO was lowered by up to 85.1%, parameter size by 85.3%, and FLOPs by 83.3%. These improvements were accomplished without sacrificing the image classification accuracy, which saw an increase of 8.0%. These results demonstrate that the EMONAS is an excellent choice for optimizing innovative objects that did not exist before, and greater multi-objective optimization performance can be guaranteed simultaneously if computational resources are adequate.

Keywords

DNN (Deep Neural Network), NAS (Neural Architecture Search), EA (Evolutionary Algorithm), Multi-Objective Optimization, Binary One Optimization, Embedded Systems

Sammanfattning

Customized Deep Neural Network (DNN)-acceleratorer har blivit alltmer populära i olika applikationer, från autonom körning och naturlig språkbehandling till sjukvård och ekonomi, etc. Att distribuera dem direkt på kringutrustning för inbyggda system inom realtidsoperativsystem (RTOS) är dock inte lätt på grund av paradoxen med komplexiteten hos DNN och enkelheten hos inbyggda systemenheter. Som ett resultat kräver DNN-implementering på inbäddade systemenheter skräddarsydda acceleratorer med skräddarsydd hårdvara på grund av deras många beräkningar, latens, strömförbrukning, etc. Dessutom är beräkningskapaciteten, som tillhandahålls av potenta mikroprocessorer eller grafikprocessorer (GPU), nödvändig för att frigöra den fulla potentialen hos DNN, men dessa beräkningsresurser är ofta inte lätt tillgängliga i inbyggda systemenheter.

I den här avhandlingen föreslår vi en innovativ metod för att utvärdera och förbättra effektiviteten av DNN-implementering inom begränsningarna av resursbegränsade inbäddade systemenheter. Den evolutionära Multi-Objective Neuron Architecture Search-Binary One Optimization (EMONAS-BOO) optimerar både bildklassificeringsnoggrannheten och de innovativa Binary One Optimization (BOO) målen, med Multiple Objective Optimization (MOO) metoder. EMONAS-BOO automatiserar sökning och träning av neurala nätverk, och de neurala nätverksarkitekturernas mångfald garanteras också med hjälp av en evolutionär algoritm som består av turneringsval, polynommutation och punktövergångsmekanismer. Binary One Optimization (BOO) används för att utvärdera svårigheten att implementera DNN på resursbegränsade kringutrustning för inbäddade system, med ett binärt format för DNN-vikter.

En djupare implementering av den innovativa Binary One Optimization kommer att avsevärt öka inte bara beräkningseffektiviteten utan också minneslagring, effektförlust, etc. Den är baserad på minskningen av vikter binära 1:or som behöver beräknas och lagras, där minskningen av binär 1 ger minskade aritmetiska operationer och därmed förenklade neurala nätverksstrukturer. Dessutom, analyserat ur ett digitalt kretsvågformsperspektiv, kommer det inbäddade systemet, vid tolkning av det neurala nätverket, att registrera en ökning av nollvikter, vilket leder till en minskning av spänningsövergångsfrekvensen, vilket i sin tur gynnar en förbättring av effekteffektiviteten.

Den föreslagna EMONAS använder MOO-metoden som optimerar två mål. Det första målet är bildklassificeringsnoggrannhet och det andra målet är Binary One Optimization (BOO). Detta tillvägagångssätt gör det möjligt för EMONAS att överträffa manuellt konstruerade och slumpmässigt genomsökta DNN. Noterbart behöll 12 av 100 distinkta DNN:er sin bildklassificeringsnoggrannhet. Samtidigt uppvisar de också överlägsen BOO-prestanda. Dessutom säkerställer den föreslagna EMONAS automatisk sökning och utbildning av DNN. Den uppnådde betydande minskningar av nyckelprestandamått: BOO sänktes med upp till 85,1%, parameterstorleken med 85,3% och FLOP:s med 83,3%. Dessa förbättringar åstadkoms utan att offra bildklassificeringsnoggrannheten, som såg en ökning med 8,0%. Dessa resultat visar att EMONAS är ett utmärkt val för att optimera innovativa objekt som inte existerade tidigare, och större multi-objektiv optimeringsprestanda kan garanteras samtidigt om beräkningsresurserna är tillräckliga.

Nyckelord

DNN (Deep Neural Network), NAS (Neural Architecture Search), EA (Evolutionary Algorithm), Multi-Objective Optimization, Binary One Optimization, Inbyggda system

Acknowledgments

I would like to express my deepest gratitude to all those who have contributed to the completion of this academic work.

Firstly, I am immensely grateful to my examiner, professor Masoumeh Ebrahimi, for her guidance, support, and invaluable feedback throughout the research process. for the various kinds of paperwork signatures, the determination of the final decision about the evolutionary search, among reinforcement learning, SGD, etc.

I would also like to thank my supervisor, Vahid Geraeinejad, for his expertise, patience, and encouragement have been instrumental in shaping this work. for the tutorial in NAS setup, the paper review.

I would also like to thank my lecturer and embedded system director, prof Zhonghai Lu. His wonderful and vivid lecture has kindled my interested in the AI field. I am honored to attend Zhonghai's Hardware Acceleration for Deep Learning (IL2230) and Embedded Many-Core Architecture (IL2236). Zhonghai's wonderful and state-of-the-art lectures and tutorials have kickoff my career in the AI field. Furthermore, I was also recommended by him to prof. Masoumeh Ebrahimi and Vahid Geraeinejad group, for further development in the AI field.

I would also like to thank my colleagues and friends for their support and encouragement during this journey. Their constructive criticisms, insightful discussions, and encouragement have been invaluable in shaping my ideas and refining my arguments.

Finally, I would like to express my heartfelt gratitude to my family for their unwavering love, support, and encouragement throughout my academic career. Their patience and understanding have been essential in allowing me to pursue my academic aspirations.

Once again, thank you to everyone who has played a role in the completion of this academic work.

Stockholm, December 2023
Jiayi Feng

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an era where deep learning (DL) is famous for learning from a large amount of data and simulating the human brain's behavior by neural network weights' adjustment, update, and optimization, its integration into resource-constrained embedded systems presents a significant challenge. This project is motivated by conflicts between DL's growing resource demands and the limited resources of embedded systems. A novel numerical metric was introduced to quantify the implementation complexity of DL in embedded systems. The project successfully balances the novel numerical metric with image classification accuracy, ensuring efficiency in optimization without compromising performance. Compared with random search, the evolutionary-searched approach culminates in substantial reductions in the novel numerical metric by 85.1% while simultaneously enhancing image classification accuracy by 8.0%.

1.1 Motivation

Deep neural networks (DNNs), as one of the dominant learning models depicted in Figure 1.1, have demonstrated remarkable efficacy in a variety of challenging tasks across artificial intelligence and machine learning domains. These tasks span various areas, including image classification [1], speech recognition [2], and unsupervised learning tasks [3].

Among the various deep learning models, Convolutional Neural Networks (CNNs), a dominant subtype, have achieved significant success in advancing visual recognition tasks [4]. This has led to their extensive application across an array of computer vision tasks [5]. The success of CNNs extends beyond a particular platform and encompasses everything from workstations to rapidly

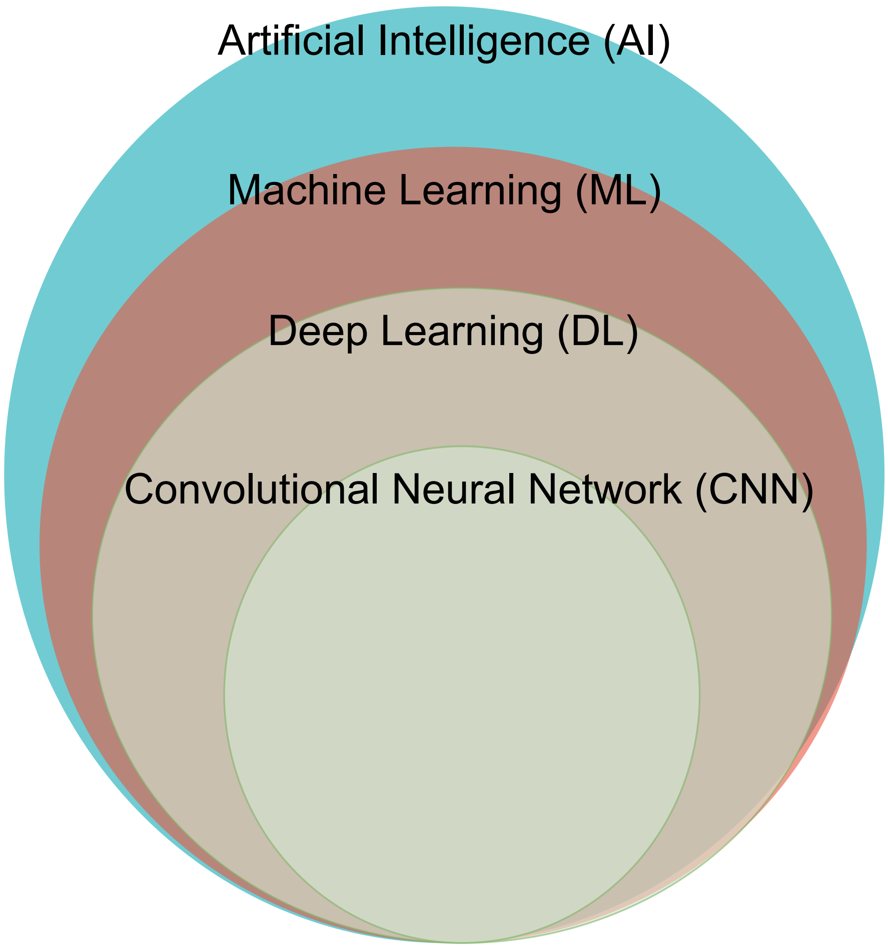

Artificial Intelligence (AI)
Machine Learning (ML)
Deep Learning (DL)
Convolutional Neural Network (CNN)

Figure 1.1: A glossary of Deep Learning

growing embedded system devices [6], shown in Figure 1.2.

However, capitalizing on the successes garnered by deep learning in industrial and embedded applications presents a conflict. The execution of DNNs in real-time, combined with high input bandwidth, consists of numerous floating-point operations. This necessitates a powerful microprocessor or graphics processing unit (GPU), which in turn results in substantial power consumption, unsuitable for an embedded device. Resolving this CNN implementation conflict is a key requirement in fully exploiting the benefits of deep learning in embedded systems.

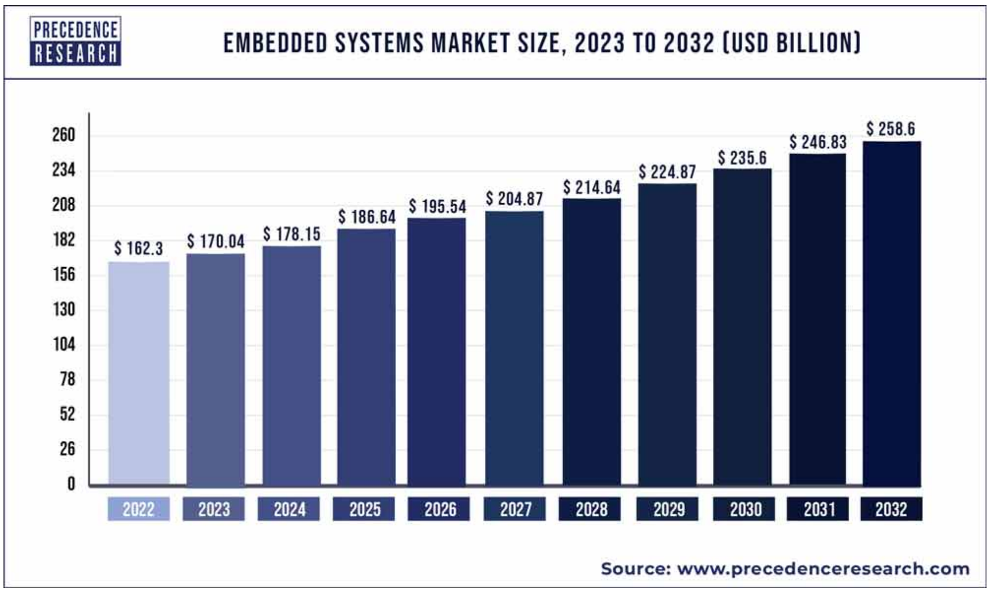


Figure 1.2: The global embedded systems market size from source [7]

1.2 Problem

In the realm of embedded systems of electrical engineering, the implementation of CNN stands as a challenging task. Embedded systems typically operate under stringent resource constraints, where computational resources, memory, and energy are limited. This limitation is further accentuated when deploying complex ML models like CNNs on Real-Time Operating Systems (RTOS). The intricate and resource-intensive computations inherent to CNNs present a significant conflict with the limited capabilities of embedded systems, leading to potential issues in performance, efficiency, and functionality, as elaborated in Section 1.1. In addition, CNN design objective tends to prioritize accuracy and overlook other crucial objectives such as latency [8], Floating-Point Operations (FLOPs) [9], and energy efficiency [10]. The aim is to optimize customized objectives while maintaining accuracy levels, and vice versa, which creates another point of conflict.

Deep learning has advanced significantly in various fields such as natural language processing, computer vision, and speech recognition. However, traditional deep learning methods have limitations regarding neural network architecture. The conventional approach primarily relies on hand-crafted architectures that depend on the designer's intuition, expertise, and creativity with hyperparameters manually tuned [11]. This method limits the scale and diversity of neural networks while also restricting the designer's ability to navigate through the vast design space to identify optimal solutions [11].

1.3 Goals

The goal of this project is to develop a customized DNN accelerator procedure to improve the performance of DNNs before the inference phase and this has been divided into the following sub-goals:

1. To quantify the performance of CNN implementations and optimization within embedded system devices.
2. To simultaneously optimize uncorrelated and even conflicting predefined objectives.
3. To incorporate diversity and automation into the process of network architecture search and validation, rather than inefficient neural network modeling.
4. To introduce a tailored algorithm into the automated framework that benefits models' modeling, search, and validation.

1.4 Research Methodology

This project introduces the Evolutionary Multi-objective Optimization Neural Architecture Search (EMONAS) framework, an innovative approach inspired by and building on the foundational principles of NSGA-Net. EMONAS is engineered to traverse the vast landscape of neural architecture possibilities, aiming to automatically generate Deep Neural Networks (DNNs) that are not only accurate but also optimized for BOO. This dual-objective strategy represents a paradigm shift from traditional neural architecture searches that prioritize accuracy. By incorporating BOO as an innovative but critical objective, EMONAS stands out for its potential to cultivate DNNs that are intrinsically tailored to the stringent resource limitations of embedded systems. To fulfill this ambitious goal, EMONAS leverages Multi-Objective Optimization (MOO) techniques to address the complex neural architecture search problem. It adeptly identifies a set of Pareto-optimal solutions, effectively balancing competing objectives to produce architectures that achieve both high accuracy and BOO efficiency. The contributions of this research are twofold: the integration of BOO into the NAS process, and the application of MOO for the systematic discovery of architectures that promise to enhance the performance of embedded systems. The EMONAS produced 12 high-potential neural networks that keep their image classification accuracy while achieving better performance in BOO. BOO was lowered by up to 85.1%, parameter size by 85.3%, and FLOPs by 83.3%, while the image classification accuracy saw an increase of 8.1%.

1.5 Delimitations

The scope of this study is limited to the optimization of neural network architectures using a multi-objective evolutionary algorithm for image classification tasks exclusively. Currently, the delimitations could be categorized into these areas, search space limitation, search strategy deployment, optimization objective definition, and others.

Starting with the limitation of the search space, the vast and multi-dimensional search space encapsulates plenty of parameters and hyperparameter configurations, and each of them contributes uniquely to the targeted objectives of the CNN models, such as efficiency and accuracy. Ideally speaking, navigating every expansive, multi-dimensional defined search space will yield the optimal results, but this ideal exhaustive exploration demands massive or even unacceptable computational resources and time. As a result, the acceptance of sub-optimal solutions and partial exploration is a compromise in daily research practice, which means our EMONAS-BOO's search space is bound by the practical constraints of computational capacity and execution time. The customized limited exploration of the search space is conducted to achieve a tradeoff between performance objectives such as accuracy, latency, FLOPs, etc., and the remaining unexplored hyperparameter combination of defined search space still can achieve optimal performance.

As for the search strategy deployment, EMONAS utilizes a multi-objective evolutionary strategy to automate the process of exploring the design space of deep neural architectures, instead of reinforcement learning (RL) and stochastic gradient descent (SGD), etc. The efficiency of the chosen evolutionary strategy, although well-suited to this application, may not always be the optimal solution in all circumstances or scenarios.

When it comes to the optimization objective definition, our EMONAS focuses on two objectives: network accuracy and Binary One Optimization (BOO), which means other potentially valuable objectives, such as FLOPs, may not be directly optimized. Moreover, EMONAS-BOO optimization has relatively better performance in two objectives, and more objectives bring worse optimization performance due to the increased computational complexity.

Last but not least, EMONAS-BOO operates within a limited set of datasets, primarily including CIFAR-10. Additionally, it has been restricted to hybrid computing resources comprising cloud computing and physical resources.

1.6 Structure of The Thesis

Chapter 2 presents relevant background information about deep learning, embedded systems, and their conflicting points. Chapter 3 presents the detailed background of EMONAS in terms of Neural Architecture Search (NAS), Evolutionary algorithm, and Multi-objective Optimization (MO) Chapter 4 presents the search results of the EMONAS, along with a search result discussion. Chapter 5 concludes the report and discusses future work about computational capability and database variety.

Chapter 2

Background

This chapter provides basic background information about Neural Architecture Search, a subfield of Automated Machine Learning, Machine Learning, and Artificial Intelligence. Additionally, this background chapter describes the evolutionary search algorithm, before stepping into multi-objective optimization. After that, the chapter also describes related work.

2.1 Automation Neural Architecture Search

To address the previously outlined constraints in machine learning that manually constructed neural networks are heavily inspired by the programmers' unstable personal expertise and talent, AutoML and NAS methodologies have been introduced to streamline the architectural search process, which facilitates a more proficient and robust design of architectures.

2.1.1 Parameter and Hyperparameter

In neural networks, parameters, often referred to as "weights," are integral to the model's functionality. For instance, weights in the fully connected layer or kernels in the convolutional layer require training. These parameters are optimized using training data, with techniques such as gradient descent guiding their updates. Once training is completed, the model's performance is evaluated on a specific dataset, with the aim being the highest possible accuracy for the neural network. In the following example [12], parameters are those intensive fully-connected lines between the nodes of each layer, in Figure 2.1.

Hyperparameters are distinct from the typical parameters or weights and

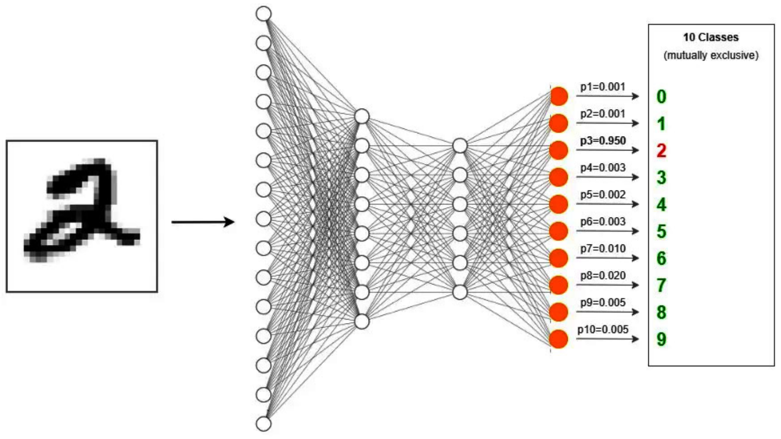


Figure 2.1: Creating a Multilayer Perceptron (MLP) Classifier Model to Identify Handwritten Digits [12]

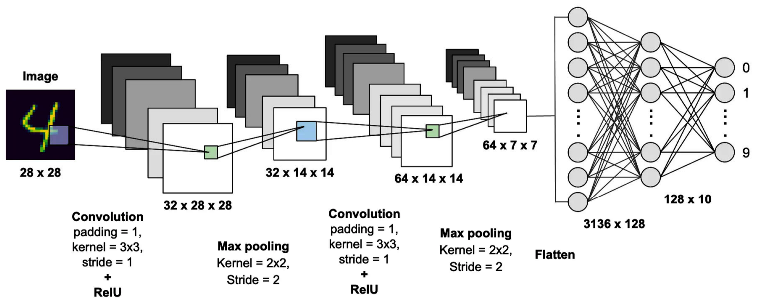


Figure 2.2: MNIST Handwritten Digits Classification using a Convolutional Neural Network (CNN) [13]

are set before the model's initialization and training. The parameters a neural network learns are influenced by the hyperparameters and the training data. Changing hyperparameters or training data can lead to variations in the resultant model parameters. In deep learning, hyperparameters generally fall into two main categories. The first pertains to the architecture of the neural network, including the number of convolutional layers, the number of kernels per layer, and the size of these kernels. The second relates to the optimization process and includes methods like Stochastic Gradient Descent (SGD), along with elements such as learning rate, batch size, and epoch count. These hyperparameters play a crucial role in determining the model's learned parameters, which in turn affects its accuracy on the dataset. Adjusting these hyperparameters correctly is crucial in deep learning, and the pursuit of automated adjustment methods is a prominent area of current research. In the example introduced in [13], Figure 2.2, all the numbers including padding, kernel, stride, conv size, and so on are parts of hyperparameter.

2.1.2 Convolutional Neural Network

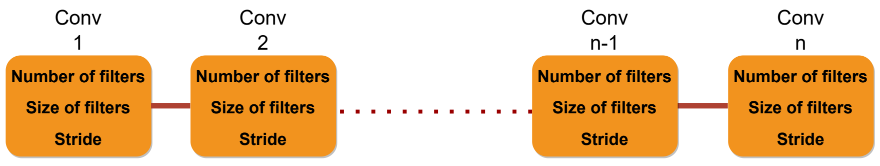


Figure 2.3: CNN architecture hyperparameters visualization

An artificial neural network is a computational system that consists of the basic unit of multiple interconnected artificial neurons, which could process and transmit information, inspired by the structure and function of biological neurons [15] in Figure 2.5. CNNs are a main branch of artificial neural networks for deep learning algorithms that deploy the convolution operation, specifically for computer vision tasks that involve the processing of pixels [16].

In this project, we explore techniques to automatically fine-tune hyperparameters that define the structure of neural networks, rather than layer-and-layer constructed manual neural networks. To illustrate this, we take the example of CNNs, since CNNs have dominant positions in the realm of deep

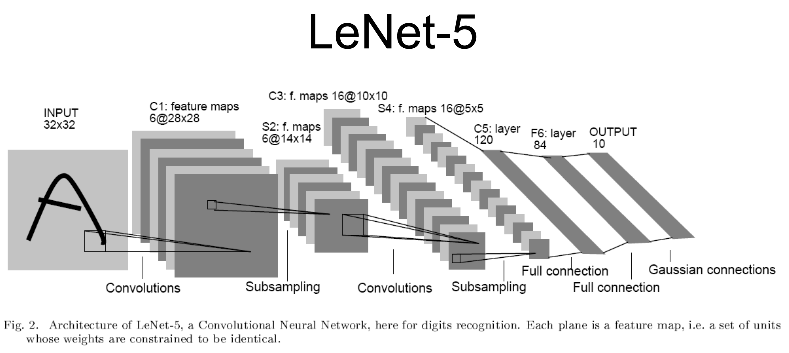


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Figure 2.4: LeNet-5 [14]

learning. The structural hyperparameters of CNNs encompass several aspects in Figure 2.3:

- The number of convolutional, dense, and fully connected layers
- The number of filters, size of filters, and stride in each conv layer
- The width of each dense layer, which is represented by the output tensor's size.

Famous neural network architectures, such as ResNet and LeNet [14] in Figure 2.4, predominantly rely on manual design. When designing a CNN, one needs to explicitly delineate the hyperparameters for every layer. This includes determining the configuration of filters (or kernels) from the first to the final layer. For instance, in a model with n convolutional layers, followed by the n-th layer (the last convolutional one) in Figure 2.3:

2.1.3 Neural Architecture Search

The introduction of ChatGPT [17] has reignited the public's interest in AI due to its vast knowledge base, interactive dialogue capabilities, and exceptional accuracy achieved through fine-tuned models. The creation of these large language models requires massive amounts of data and reinforcement learning, making manual construction or random generation infeasible [17]. To overcome these challenges, AutoML and NAS have been utilized in the setup stage of the training process.

AutoML is designed to facilitate the automation of model development and optimization by autonomously adjusting hyperparameters, as delineated

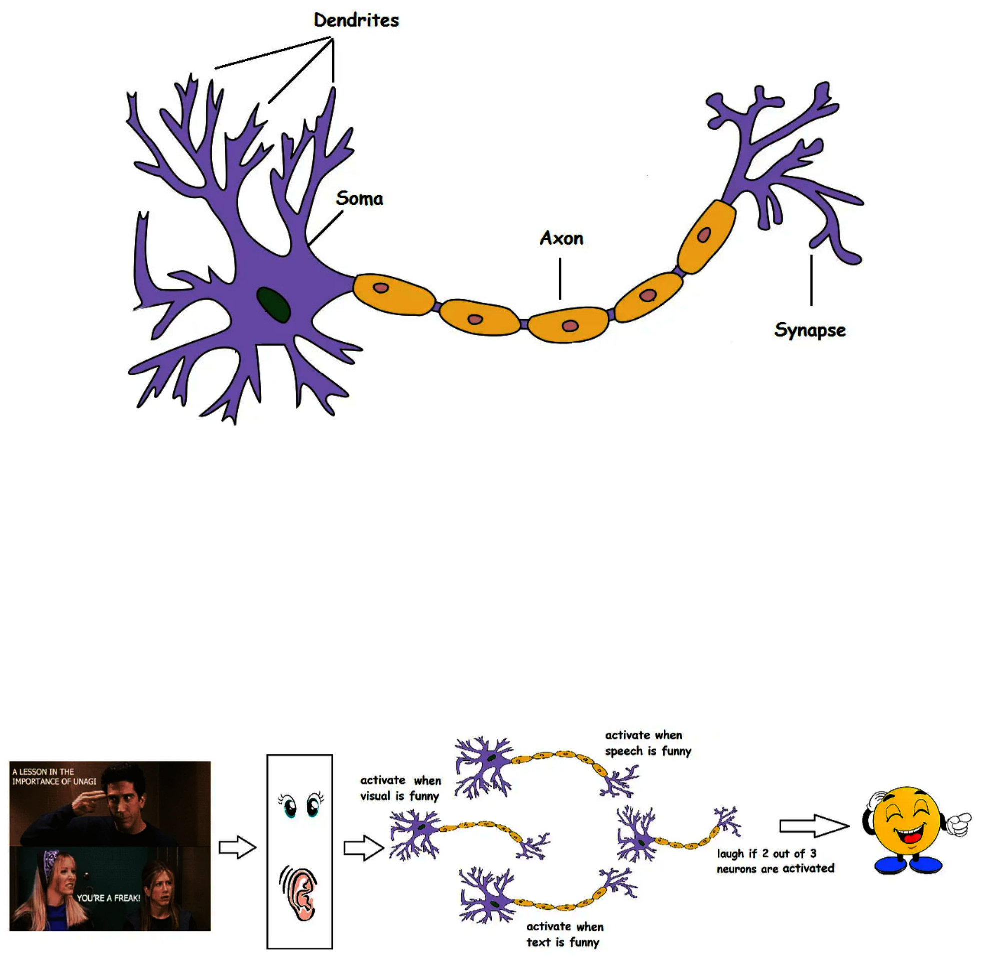

Dendrites
Soma
Axon
Synapse
A LESSON IN THE IMPORTANCE OF UNAGI
YOU'RE A FREAK!
activate when visual is funny
activate when speech is funny
activate when text is funny
laugh if 2 out of 3 neurons are activated

Figure 2.5: The Architecture of a Biological Neuron and How They Work [15]

in Subsection 2.1.1. Its primary objective is to simplify the intricate and labor-intensive aspects of machine learning model formulation, making it more user-friendly for non-specialists. A fundamental component of AutoML is NAS, which pertains to the systematic determination of optimal neural network configurations or the most suitable hyperparameter combinations set for specific tasks with reduced human intervention, increased complexity of deep learning models, and the need for maximizing validation efficiency and accuracy, rather than accuracy-oriented manual-tuned or random- searched small-scale neural network models. Consider, for example, that the accuracy of the ResNet architecture surpasses that of the VGG network. This could be interpreted as ResNet having a superior neural network structure compared to VGG. Nonetheless, during the process of optimizing for efficiency and accuracy, there are other factors to be taken into account, including computational capacity, memory utilization, power consumption, etc, rather than accuracy only. In practical applications, when deploying neural networks on mobile devices, it is crucial to consider both memory utilization and accuracy. The balance between these diverse objectives will be further explored in Section 2.3.

Table 2.1: Search Space Example

Hyper-parameter Types	Candidates
Number of filters	{16, 36, 64, 100}
Size of filters	{3×3, 5×5, 7×7}
Stride	{1, 2}

Table 2.2: Outcome of NAS Example

	Layer 1	Layer 2	···	Layer (N-1)	Layer N
Number of filters	16	36	···	100	64
Size of filters	3×3	3×3	···	7×7	5×5
Stride	1	2	···	1	1

Before initiating the NAS, it is necessary to define the search space to facilitate the search process of the optimal neural network architecture that maximizes objectives including deployment efficiency and image classification accuracy. The search space is essentially a set of hyperparameters, as elaborated in Subsection 2.1.1, encompassing potential neural network architectures defined by the researcher or user. The size of the search space is measured as the number of possible neural network architectures, which equals the number of a set of hyperparameters. For clarity, two illustrative examples, shown in Table 2.1 and 2.2, have been created. It is important to note that the figures within these tables are arbitrary, serving merely as exemplars to elucidate the concept of NAS's search space.

Consider the construction of a CNN comprising N convolutional layers. Each of these layers is hypothetically characterized by three hyperparameters for discussion: the number of filters, the size of the filters, and the stride. An example configuration is depicted on the left of Table 2.1, resulting in 3 hyperparameters for each layer. The objective is to search out the combination of hyperparameters that optimizes validation accuracy. However, the expansive range of potential hyperparameters presents a challenge. For instance, considering the stride alone, any positive integer is a potential hyperparameter. It is computationally infeasible to exhaustively explore all hyperparameter permutations due to the prohibitive computational cost. Consequently, it becomes essential to curate a subset of candidate hyperparameters, as showcased in the right column of Table 2.1. To illustrate in Equation 2.1, potential candidates for the number of filters might be limited to specific values like 16, 36, 64, and 100. Similarly, filter sizes might be constrained to three specific dimensions: 3×3, 5×5, 7×7. Strides could be limited to two values, namely 1 and 2. The search space is thus a Cartesian product of these candidates, encompassing all feasible hyperparameter combinations within the specified candidates' limits, shown in Equation 2.2.

- Build a CNN with N convolutional layers.
- Search space:

$$\begin{aligned}&\{16, 36, 64, 100\}^N\\ &\times \{3 \times 3, 5 \times 5, 7 \times 7\}^N\\ &\times \{1, 2\}^N\end{aligned} \tag{2.1}$$

- Size of the search space:

$$\begin{aligned}&\left\{\begin{matrix}\text{candidates of}\\ \text{number of filters}\end{matrix} \times \begin{matrix}\text{candidates of}\\ \text{size of filters}\end{matrix} \times \begin{matrix}\text{candidates of}\\ \text{stride}\end{matrix}\right\}^N\\ &= \{4 \times 3 \times 2\}^N\end{aligned} \tag{2.2}$$

During the neural architecture search, 3 hyperparameters, in this hypothetical scenario, are determined for each convolutional layer: the number of filters, the size of filters, and the size of the stride. After that, the optimal set of hyperparameters emerges. A representative outcome of this architecture search is delineated in Table 2.2. Equipped with this specific set of hyperparameters, the architecture for all N layers of the Convolutional Neural

Network is determined, contributing to the future construction of the neural network.

2.1.4 Random Search

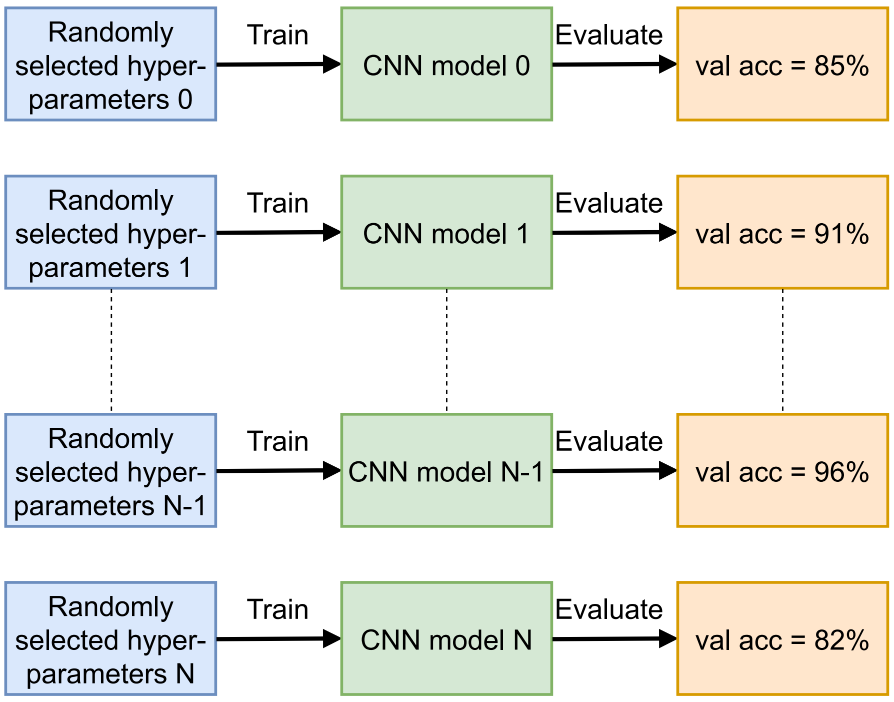


Figure 2.6: Random Search Baseline Examples

Within Neural Architecture Search (NAS), random search is predominantly utilized due to its ability to efficiently identify optimal models with minimal computational overhead, recognizing that not all hyperparameters necessitate fine-tuning [18]. Initially, a collection of hyperparameters is randomly chosen from a predefined search space of potential candidates. Leveraging these hyperparameters, various neural network architectures are systematically constructed. Following this, the formulated neural network is trained on the dataset, evolving its parameters from a random initialization state to a converged state. Once trained, the model is then applied to make predictions, with its validation accuracy serving as the metric for evaluating the convolutional neural network (CNN) model's efficacy.

For clearer comprehension, a visual representation with hypothetical accuracy figures of this process is presented in Figure 2.6. This random search procedure will be iteratively executed numerous times. When the random search is over, the architecture yielding the highest accuracy, as showcased in Figure 2.6 with the CNN model, in this hypothetical scenario, $N - 1$ achieving a validation accuracy of 96 percent, is selected. The corresponding hyperparameters from this optimal architecture are then adopted as the definitive neural network structure.

2.1.5 Hierarchical Structure of NAS

AI is a broad field in computer science that creates machines capable of performing human-intelligence-demanding tasks. These tasks encapsulate a wide variety of fields, including computer vision, natural language processing, robotics, cognitive science, and so on. To be more specific, these tasks include chess playing, mathematical theorem proving, poetic composition, autonomous vehicle navigation, and disease diagnosis [19].

ML is the specialized subfield of AI, which interacts with and learns from data, rather than data-independent AI frameworks without learning such as Symbolic AI/Good Old-Fashioned AI (GOFAI), Deterministic AI, Knowledge-based AI, Rule-based AI, Heuristic-Based AI, etc. Data points are often correlated and interacted with, including the previous result and current result, and it will be called learning. ML is the most popular solution to AI implementation, and is also primarily dedicated to the development of algorithms that empower computer systems to learn from data, thereby enabling them to make predictions or decisions based not on explicit programming, but the patterns learned [20].

Further along this hierarchy lies the realm of AutoML, a branch of ML. This domain is focused on automation spanning all components of the machine learning process, from data preprocessing and feature engineering to algorithm selection, hyperparameter tuning, and model selection [21].

At the heart of AutoML's expanse, we find the specialized subfield of NAS. This domain is primarily concerned with automating the process of devising optimal neural network architectures. Its core objective is to discover innovative network structures that can deliver state-of-the-art performance on a given task. NAS techniques have found extensive application in diverse areas such as image classification, object detection, and language modeling [22].

To better brief understanding, the hierarchical structure of the EMONAS can be simplified and visualized within the broader fields of AI, ML, AutoML, and NAS as follows:

- Artificial Intelligence
    - Machine Learning
        - Automated Machine Learning
            - Neural Architecture Search

2.2 Evolutionary Algorithm

The Evolutionary Algorithm (EA) of the EMONAS employs a population-based iterative approach, to progressively enhance the caliber of initial hyperparameter solutions. In each iteration of this procedure, a set of hyperparameters for CNNs, referred to as a 'population,' is derived from a selected group of existing CNNs, known as the parents. New CNN architectures, referred to as offspring, are then produced in quantities equal to that of the parent group. These offspring represent innovative network designs. Within this evolving population, each CNN, whether parent or offspring, competes based on performance metrics to be chosen for further iterations, ensuring that only the most promising architectures are carried forward in the search for optimal neural network configurations. The initial population can be formed randomly or steered by previous results.

Occupying a distinct place within the repertoire of heuristic search methods, the Evolutionary Algorithm is deeply rooted in the principles encapsulated in Charles Darwin's theory of evolution. Through a simulation of the natural selection mechanism, EA emphasizes the identification and selection of the fittest individuals based on their traits or fitness metrics. Such individuals are then designed for offspring reproduction, setting the stage for the emergence of subsequent generations.

The EA's systematic approach can be distilled into these fundamental phases:

- Initialization: Here, an initial population of potential solutions is generated, typically at random.
- Selection: The fittest individuals, based on their evaluated fitness, are chosen to contribute to the population's next generation.
- Crossover or Reproduction: Through techniques like evolutionary crossover, selected individuals are combined to produce offspring, thus creating a new generation.

- Mutation: Applying random changes to the members of the population happened in a certain probability.
- Termination: The algorithm concludes either when a satisfactory solution has been found or after a predetermined number of generations or time has elapsed.

Drawing parallels with the natural world, when confronted with diverse members within an ecosystem, nature instinctively opts for the fittest. These elite entities subsequently procreate, begetting progeny that often inherit enhanced or better-adapted attributes. The Evolutionary Algorithm emulates this insightful natural process, ensuring each subsequent generation inherits superior characteristics, optimizing towards a solution.

2.2.1 Selection

Selection plays a crucial role in determining the success of the optimization process. Among the various methods employed for selection, Roulette Wheel Selection, Tournament Selection, and Rank-Based Selection are the most commonly used techniques [23]. While rank-based selection proves simple and efficient in single-objective optimization – where basic algorithms can swiftly ensure optimal convergence – its application becomes more intricate with multiple or even conflicting objectives, where a straightforward sorting and corresponding maximization or minimization approach can suffice. Consequently, Tournament Selection emerges as the preferred choice in the EMONAS optimization, shown in Supporting Materials Section A.2. This method ensures both the diversity of the population pool and the quicker convergence towards an optimal solution, striking a balance between exploration and exploitation.

For instance, each time the selection function is invoked, designated parents are selected as specified. Assuming the specified parent number is N and the pressure is 2 (a measure of the likelihood of better individuals being selected as parents, where higher pressure means that only the very best individuals are selected, and lower pressure means that a wider range of individuals, including those that are not as fit, can be selected as parents.), then $2N$ individuals from the population pool are selected before binary comparison. The winners (for instance, the smaller performance the better) of these binary comparisons are then recorded by their index numbers as the parents of the generation. This process helps maintain a diverse genetic pool while gradually moving towards optimal solutions.

While Tournament Selection provides a mechanism for optimizing population diversity and expediting convergence, it simultaneously complements the principles of the Genetic Algorithm. Here, the Genetic Algorithm's initiation, involving the formation of an initial population comprising various CNN architectures, aligns seamlessly with the selection processes described in EMONAS. The adoption of Tournament Selection in EMONAS facilitates a strategic and methodical narrowing down of potential solutions – each represented as chromosomes in the GA terminology – based on their fitness values. This fitness, determined through a well-defined function, is crucial not only in assessing the immediate suitability of each CNN architecture but also in predicting its potential to contribute optimally to future generations. Thus, the selection process serves as a critical juncture in both methodologies, ensuring that the evolutionary journey of each neural network architecture within EMONAS is guided by a robust and strategic approach, in line with the foundational principles of genetic algorithms.

Both the Evolutionary Algorithm and its specialized subtype, the Genetic Algorithm, are computational optimization methods. GA represents the CNN model solutions in the form of encoded binary strings in Figure A.1. However, EA extends to a diverse set of representations, not only the binary format. Owing to the relative simplicity of binary analysis, binary representation might be introduced to illustrate the fundamental principles of EA.

The initiation of the Genetic Algorithm begins with the formation of the initial population, by the selection of a set of individuals that collectively contribute to the population.

For example, suppose we have a population comprising 21 different CNN architectures in Supporting Material Section A.1, labeled as

macro_network_architecture_0
macro_network_architecture_1
......
macro_network_architecture_19
macro_network_architecture_20

These entities jointly represent the initial population. Each member of this group symbolizes a distinct potential architecture for the designated objectives.

In the context of Genetic Algorithms, these individual solutions are denoted as ’chromosomes.’ Therefore, each single unit within the population is classified as a chromosome. Defining these chromosomes is a component known as a ’gene,’ which is an integral element of a chromosome. A gene usually represents a binary value, either 0 or 1, as shown in Supporting Material Section A.1.

Once the initial population is formulated, it is imperative to assign a fitness value to each solution or chromosome. This operation is conducted through a mechanism known as the fitness function. The fitness value, shown in Table 2.3, of each solution is discerned via the application of this fitness function. The BOO in Table 2.3 is short for Binary One optimization, an innovative metric, discussed in Section 3.1, defined as the effectiveness of DNNs implementation into the embedded systems. The candidates chosen for progression are typically those with higher fitness scores, which suggests they inherently possess attributes that bring them nearer to an optimal solution and are capable of yielding superior progeny as primary contributors.

In this phase, each individual’s "fitness" or suitability to solve the problem is assessed, but the assessed computation of the fitness function and score is not a straightforward process. To provide a simplified explanation, the fitness score of the CNNs discovered by the EMONAS-BOO is associated with image classification accuracy, in addition to other objectives such as floating point operations, power usage, parameter sizes, and so on. The trade-off between irrelevant or even contradictory objectives will be addressed in subsequent discussions in Section 2.3.

Table 2.3: Corresponding 5 Fitness Scores Derived from EMONAS-BOO Balancing Accuracy and an Innovative Objective in CNNs

Accuracy	**BOO**
86.550000	5702587
86.090000	5398006
86.160000	5721496
86.510000	4895424
86.440000	5731408

2.2.2 Crossover

Genetic algorithms are optimization algorithms inspired by the process of natural selection, used to find approximate solutions to optimization and search problems. Analogous to the natural selection process in biology, it involves a population of candidate solutions (individuals) to an optimization problem, where these individuals evolve iteratively through a process of selection, crossover, and mutation until a stopping criterion is met. One of the key steps in this process is crossover, also known as recombination, an important mechanism used to amalgamate the genetic information of two parents to produce one or more offspring. Analogous to biological crossover, the exchange of genetic material in mitosis brings diversity to the population pool and helps achieve convergence towards an optimum. In biological systems, mitosis and the subsequent exchange of genetic material during crossover ensures genetic diversity while retaining some of the characteristics of both the selected parents, which is crucial for the adaptation and survival of a species. Similarly, in genetic algorithms, crossover introduces diversity into the population, enabling the algorithm to explore a broader region of the solution space and, consequently, increasing the likelihood of converging to a global optimum. Here is a representative segment code of how the overall crossover works in the EMONAS in Supporting Material Section A.4.

Multi-point crossover is a variant of crossover wherein multiple crossover points are chosen along the length of the parents. The crossover_mask function is instrumental in performing the crossover operation and generating the offspring and can be found in Supporting Material Section A.4.

In the subsequent section, numerical examples will be provided to elucidate the concept of crossover. These examples have been simplified to input tensor, output tensor, and mask tensor. The results following the execution of this function will be presented subsequently. In the context of the output, 'T' stands for True, and 'F' stands for False. A 'True' indicates that a crossover has occurred between the selected parents.

Initially, the population before crossover consists of two sets of numeric tensors [[6, 6, 5, 5, 10], [8, 7, 10, 3, 5]]. The crossover operation is then applied, guided by a mask pattern of [False, False, True, True, True] for both tensors. This mask determines which elements from each set are exchanged with the other. The elements corresponding to 'True' in the mask are swapped between the two sets. As a result of applying this mask, the population after the crossover is as [[6, 6, 10, 3, 5], [8, 7, 5, 5, 10]]. Detailed code and corresponding execution result can be found in Supporting Material Section A.4.

2.2.3 Mutation

Maintaining genetic diversity within a population is paramount for the effectiveness of evolutionary algorithms in avoiding local optima and achieving global optimization. It is meticulously designed to foster diversity and bolster the algorithm's capability to avert local optima.

In simple binary encoding scenarios, a bit-flipping mutation operator is employed, a common practice in binary-coded genetic algorithms. Before that, the encoding will be simplified as a transformation between semantic neural network structure expression and numerical expressions such as tensors and strings. As a result, each neural network of the population pool is encoded from semantic and abstract expression into a numeric bit string. This approach entails a simple change in the binary encoding, also referred to as macro in the code, where a single bit in the genotype space is altered. For instance, a mutation might change a genotype from [0,1,0] to [0,0,0].

However, in scenarios where the genotypes are encoded in integers, the mutation operator adopts a different strategy. Instead of bit-flipping, a Polynomial mutation is leveraged, towards an integer genotype that looks like [3, 6, 5, 0, ..., 1] from initial genotype [3, 6, 4, 0, ..., 1].

2.2.4 Micro and Macro Encoding

Encoding is a method that helps computers comprehend customized neural network architectures using binary and integer strings. Many CNN architectures can be described as a composition of computational blocks that will be encoded using the method presented in [24]. A small change was made to this method, where a bit was added to represent a skip connection that forwards the input information directly to the output, bypassing the entire block. The introduction of encoding brings automation to the optimized CNN architecture search process and minimizes the need for professional expertise in CNN design, rather than inefficient and unstable manual CNN architecture design.

In this project, the macro encoding scheme and micro encoding scheme are defined to represent the CNN architecture as the genome, which can be used to apply evolutionary operators to produce new candidate CNN architectures. The difference between macro encoding and micro encoding is the Directed Acyclic Graph (DAG) between nodes in the cell. The cell in a neural network is a repeatable, modular block, a sub-network with a predefined structure of layers and operations. A node within a cell is a point where operations are applied and intermediate representations are formed, representing specific states or sets of feature maps.

Before the investigation of the macro and micro, some common points needed to be clarified. The node number of each cell is always 6, a node lesser than 6 means potential skip operations, and thus the nodes do not have any connections and are not displayed in the final plot.

In macro encoding, only connections are taken into consideration, and the cell number is fixed into 3. Let's go through it step by step. An example genome, visualized in Figure 2.7, of macro neural network architecture is:

$$\begin{aligned}\text{macro\_architecture\_example} =& [[[0],[1,0],[0,0,0],[1,0,1,0],[0,0,0,0,0],[0]],\\ & [[0],[0,1],[0,1,1],[0,0,0,0],[0,0,1,1,0],[1]],\\ & [[0],[0,0],[0,0,0],[1,0,0,0],[1,0,0,1,0],[1]]]\end{aligned}$$

Cell 1 $[[0],[1,0],[0,0,0],[1,0,1,0],[0,0,0,0,0],[0]]$ analysis:

- $[0]$: node 2 is not connected to node 1.
- $[1,0]$: node 3 is connected to node 1, and not connected to node 2.
- $[0,0,0]$: node 4 is not connected to nodes 1, 2, and 3.
- $[1,0,1,0]$: node 5 is connected to nodes 1 and 3, and not connected to nodes 2 and 4
- $[0,0,0,0,0]$: node 6 is not connected to nodes 1, 2, 3, 4, and 5.
- $[0]$: The input node and output node are not connected.

Cell 2 $[[0],[0,1],[0,1,1],[0,0,0,0],[0,0,1,1,0],[1]]$ analysis:

- $[0]$: node 2 is not connected to node 1.
- $[0,1]$: node 3 is connected to node 2, and not connected to node 1.
- $[0,1,1]$: node 4 is connected to nodes 2 and 3, and not connected to node 1.
- $[0,0,0,0]$: node 5 is not connected to nodes 1, 2, 3, and 4.
- $[0,0,1,1,0]$: node 6 is connected to nodes 3 and 4, and not connected to nodes 1, 2, and 5.
- $[1]$: The input node and output node are connected.

Cell 3 $[[0],[0,0],[0,0,0],[1,0,0,0],[1,0,0,1,0],[1]]$ analysis:

- $[0]$: node 2 is not connected to node 1.
- $[0,0]$: node 3 is not connected to nodes 1 and 2.
- $[0,0,0]$: node 4 is not connected to nodes 1, 2, and 3.
- $[1,0,0,0]$: node 5 is connected to nodes 1, and not connected to nodes 2, 3, and 4.
- $[1,0,0,1,0]$: node 6 is connected to nodes 1 and 4, and not connected to nodes 2, 3, and 5.
- $[1]$: The input node and output node are connected.

In micro encoding, both connections and operations are taken into consideration, with a focus on how these operations are structured into a neural network cell. Let's go through it step by step. An example genome, visualized in Figure 2.8, of micro neural network architecture is:

$$\begin{aligned}\text{micro\_architecture\_example} =&[('max\_pool\_3x3',0),('avg\_pool\_3x3',0),\\&('avg\_pool\_3x3',2),('dil\_conv\_5x5',2),\\&('avg\_pool\_3x3',3),('max\_pool\_3x3',3),\\&('skip\_connect',1),('dil\_conv\_5x5',3),\\&('avg\_pool\_3x3',3),('avg\_pool\_3x3',4)],\\&normal\_concat=[5,6]]\end{aligned}$$

- Operations are processed in pairs, s_0 and s_1 defined in the forward function, each pair contributing to a new intermediate node.
- Initial nodes $h[i-1]$ and $h[i]$: input states to the cell.
- First pair: both $('max\_pool\_3x3',0)$ and $('avg\_pool\_3x3',0)$ take input from node 0. These operations are combined to form the first intermediate node.
- Second Pair: both $('avg\_pool\_3x3',2)$ and $('dil\_conv\_5x5',2)$ take input from node 2. These form the second intermediate node.
- Third Pair: both $('avg\_pool\_3x3',3)$ and $('max\_pool\_3x3',3)$ take input from node 3. These form the third intermediate node.

- Fourth pair: $('skip\_connect', 1)$ takes input from node 1. The next operation $('dil\_conv\_5x5', 3)$ takes input from node 3. However, since $('dil\_conv\_5x5', 3)$ cannot pair with $('skip\_connect', 1)$ (as they have different input nodes), it pairs with the next operation $('avg\_pool\_3x3', 3)$, both taking input from node 3. These form the fourth intermediate node (node 5).

- Fifth Pair: The remaining operation $('avg\_pool\_3x3', 4)$ needs a pair. If an operation doesn't have a pair, it might be paired with a 'zero' operation or might directly influence the next node. This operation influences the fifth intermediate node (node 6).

- Concatenation $(normal\_concat)$: Based on $normal\_concat = [5, 6]$, the outputs of nodes 5 and 6 are concatenated to form the final output of the cell.

- Output Node $(h[i + 1])$: The concatenated output forms the final output node of the cell.

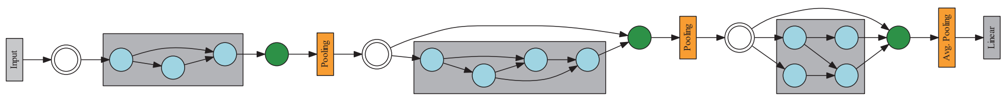


Figure 2.7: Macro Visual Representations of One Neural Network Discovered via the EMONAS-BOO

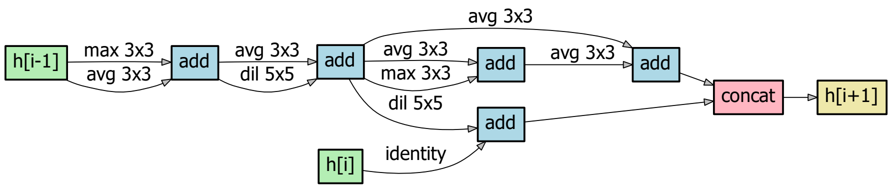


Figure 2.8: Micro Visual Representations of One Neural Network Discovered via the EMONAS-BOO

2.3 Multi-objective Optimization

Various multi-objective optimization methods have been established to optimize multiple conflicting objectives, which involve evolutionary algorithms, gradient-based algorithms, and methods rooted in machine learning. This section will provide a broad and visual overview of multi-objective optimization. Following this, the discussion will delve into Pareto front optimization, aiding in the interpretation of our concluding results plot, and pseudocode can be found in the Supporting Materials Section A.5.

2.3.1 General description

Multi-objective optimization is a distinct category within the broader field of optimization, characterized by its focus on dealing with multiple, and frequently conflicting, objectives. In the realm of machine learning, this type of optimization is applied to simultaneously improve diverse performance indicators such as accuracy, FLOPs, and power consumption, among others. Given the conflicting nature of these objectives, multi-objective optimization problems often yield multiple solutions, each optimal to different objectives. The primary aim of multi-objective optimization is to determine a set of non-dominated solutions, each representing a trade-off between conflicting objectives. Its application is widespread in machine learning, specifically in areas such as neural architecture search, hyperparameter optimization, and feature selection, facilitating the identification of optimal solutions that strike a balance between divergent objectives.

The significance of multi-objective optimization is underscored in numerous engineering applications where the simultaneous fulfillment of multiple, and often opposing, objectives is necessary [25]. For instance, customers in the computer manufacturing industry often demand systems possessing superior computational capabilities, lighter weight, and lower prices all at once. Similarly, in the automotive industry, a demand exists for vehicles that provide greater carrying capacity and higher speed concurrently. Given these demands, the optimization of multiple objectives becomes an essential component of engineering design.

2.3.2 Visualize the Multi-Objective Optimization

During the evolutionary search generations mentioned in Section 2.2, the output of the evolutionary section, and also the input of this Multi-Objective

Optimization in Section 2.3, will be the numerical value of distinct objectives' performance, in the form of the matrix:

$$\mathrm{ObjectivePerformance}[n] = \begin{bmatrix} \mathrm{objective1} \\ \mathrm{objective2} \\ \vdots \\ \mathrm{objective}_{\mathrm{n}-1} \\ \mathrm{objective}_{n} \end{bmatrix} \tag{2.3}$$

The result of $\mathrm{ObjectivePerformance}[n]$ could be expressed into n dimensional coordination for their data visualization. Since higher dimensions of the $\mathrm{ObjectivePerformance}[n]$, which means optimizing more conflicting or distinct objectives simultaneously, requires huge amounts of computational capacity, and the Return on Investment (ROI) is not linear, in other words, the border effect is obvious, lower dimensional $\mathrm{ObjectivePerformance}[n]$ is more acceptable in terms of simplicity in both architecture and visualization and computational-efficient. In the following content, the $\mathrm{ObjectivePerformance}[n]$ is limited to $n = 2$, which means the $\mathrm{ObjectivePerformance}[n]$ could be expressed, discussed, and visualized in the form of classic Cartesian coordinates.

The number of optimizing objectives can be arbitrary for any integers, and no priority assignment, which means all the defined objectives are treated equally during the optimization process. In this thesis, the number of optimizing objectives is set to 2 due to limited computational resources.

Multi-objective optimization is implemented in the evaluation phase of the search process, which will evaluate the performance of each CNN architecture in the population pool with a minimized optimization function. Due to the minimized function, the first objective classification accuracy has been transferred into error so that both the ideal classification error and BOO should be smaller and smaller, which perfectly aligned with the project's goals.

The numerical result of $\mathrm{ObjectivePerformance}[n]$, in Figure 2.9, has been visualized in the classic Cartesian coordinates. In Figure 2.9, Binary One Optimization (BOO) is presented as the secondary optimization objective of the EMONAS framework, and a detailed discussion of BOO is provided in Section 3.1. Secondary here only means numeric orders and it does not introduce any optimization priority to the multi-objective optimization among the EMONAS framework. However, there are some lines connected, and the connection and meaning of the connected curves in Figure 2.9 or surfaces will be discussed in subsection 2.3.3.

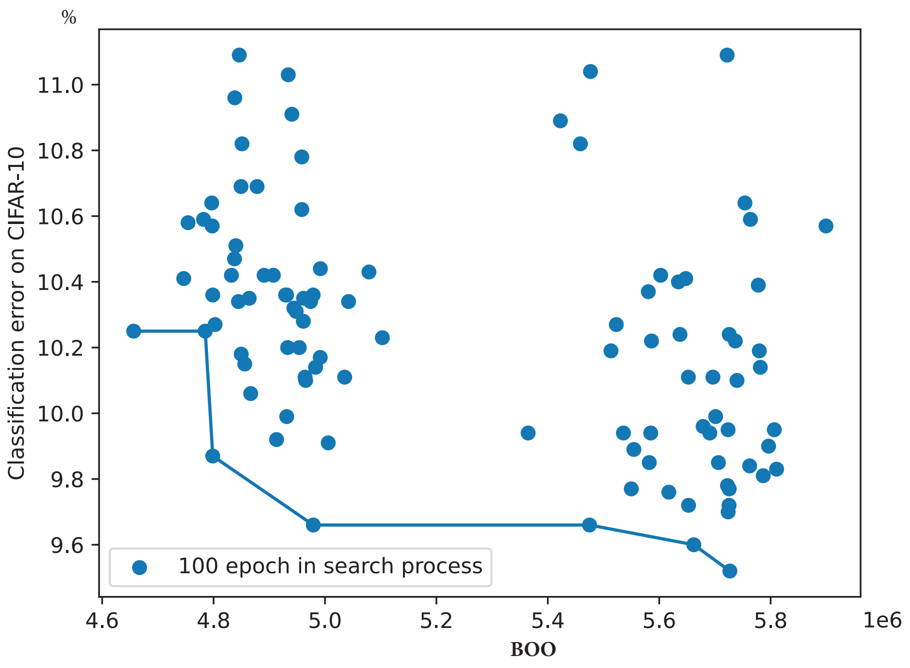


Figure 2.9: Visualization of Neural Network Architectures Post 100 Epochs of Training: Each Point Represents a Distinct Architecture

2.3.3 Pareto Front Optimization

Pareto Front Optimization means optimizing two or more conflicting objectives. Unlike single-objective optimization, where the optimal solution is a single point by simple $min()$, $max()$, or $sort()$ function, in multi-objective optimization, the optimal solution is a set of points connected by the lines, shown in Figure 2.9. These fronted optimal solutions consist of solutions set that it is not possible to improve any one objective without deteriorating the other objective.

In terms of the EMONAS, each individual in the population is evaluated based on objective 1, classification errors, and objective 2 the number of Binary Ones (BOO), and a fitness value is assigned to each individual based on its relative performance for the other individuals, CNN architectures genotypes, in the population. The CNNs then evolve through a process of selection, crossover, and mutation until the whole evolution after 10 generations has terminated.

Mathematically, let X be a set of solutions, and let $f_1, \ldots, f_m$ be the objective functions. Then, the Pareto front $P(X)$ is the set of all solutions $x \in X$ that are Pareto optimal, i.e., there does not exist a solution $x' \in X$ such that $f_i(x') \leq f_i(x)$ for all $i \in 1, \ldots, m$ and $f_i(x') < f_i(x)$ for at least one i.

The Pareto Front Optimization could be mathematically written as Equation 2.4:

$$
\begin{aligned}
P(X) = \Bigg\{ x \in X \mid x' \in X, x' \neq x, \\
(\forall_{i=1}^{m} f_i(x') \leq f_i(x)) \land \\
(\exists_{i=1}^{m} f_i(x') < f_i(x)) \Bigg\}
\end{aligned}
\tag{2.4}
$$

where denotes the existence of a variable x' such that the conditions inside the parentheses hold, and $\land$ denotes the logical conjunction (and).

2.4 Related Work

Related work mainly investigated existing projects that are related to customized DNN accelerators and the automated DNN models' construction and search frameworks. Related work's investigation aims to navigate the intricacies involved in deploying DNNs in embedded systems environments where computing resources are inherently limited. The related work utilizes my work in these aspects, the DNN accelerators, the NAS algorithm, and the corresponding framework.

The deployment of DNNs on embedded systems is challenging in comparison to conventional computing platforms such as CPUs, GPUs, and FPGAs, and the reason for this challenging paradox is the conflicting demands of embedded systems, resources-limited computing capacity, and the substantial computational power demanded for the effective implementation of DNNs [26]. Automated DNNs deployment techniques including quantization [27], pruning [26], and neural architecture search [28] have been used to optimize the computational capability for embedded system devices while maintaining accuracy.

NAS, which helps to automate the design of DNN architectures, methods can be broadly classified into three optimization algorithms: evolutionary algorithms [9, 29], reinforcement learning-based approaches, [25, 30] and others such as gradient-based optimization [31], to speed up the execution of the computational layers. Evolutionary algorithms, such as NSGA-Net [9] and AmoebaNet [29], treat network structure designs as a combination optimization problem and utilize population-based techniques like crossover, to explore different solution architecture space combination. RL-based approaches, such as MetaQNN [30], view network construction as a decision-making process and leverage strategies like Q-learning and policy gradients to build efficient architectures. As for gradient-based approaches DARTS [31], the idea is using differentiable relaxation of the architecture representation for efficient neural architecture search via gradient descent, which outperforms various conventional methods among various datasets but also narrows the selection of the search space to prevent breaking the continuation and differentiable relaxation.

In addition to NAS, the method of designing the architecture of DNNs, is the subset to the hyperparameter optimization. Efficient methods, including Grid Search [32], Random Search [33], Bayesian Optimization [34], and others have been proposed with system simplicity and interpretability [35], and also behave as strong competitors towards newly-introduced NAS-based frameworks.

2.5 Summary of Background

In this background chapter, EMONAS has been discussed into three sections, Neural Architecture Search, Evolutionary Algorithm, and Multi-objective Optimization, and their abbreviations combination are exactly our EMONAS. Neural Architecture Search enables the framework to automate the DL process, eliminating the manually repetitive demand of initialization, launching DL instances, searching, data recording, ranking, or sorting, and adjusting hyperparameters based on all the previous outcomes. Evolutionary algorithm brings efficient and rapid convergence to sub-optimal or optimal hyperparameter solution sets, under the situations of limited computational resources with acceptable implementation complexity. MOO helps to find the potential hyperparameter sets or DNN architectures that optimally balance our predefined objectives. After that, Related work investigated existing projects that are related to DNN implementation on embedded system devices and the DNN automation frameworks.

Chapter 3

Methods

In this chapter, the concept of BOO- Binary One Optimization, which optimizes the weights of deep neural networks (DNNs) for hardware implementation on embedded system devices, will be discussed, with theoretical analysis in both the hardware domain and neural network computing domain. The second topic will be different encoding and corresponding search method, micro, and macro, of the DNNs representation, and how the computer understand customized neural network architectures by binary and integer strings. Last but not least, random search will be discussed including how to set it based on the present framework.

3.1 BOO (Binary One Optimization)

To determine which Deep Neural Networks (DNN) implementation on embedded system devices outperforms others, clear and concise numerical criteria are essential in deciding which DNNs perform better than the others so that EMONAS can optimize its numerical result by backpropagation with automation. Traditional evaluators evaluate metrics or traits like FLOPs, latency, memory, power consumption, etc. However, in the EMONAS, we have introduced an innovative parameter originating from binary. This section delves into our innovative Binary One Optimization (BOO) method for evaluating DNNs implementation in embedded system devices.

BOO is defined as converting the DNNs' parameters into binary format and minimizing their number of binary ones, DNNs with optimized BOO are believed to have better implementation performance in embedded systems in various aspects, discussed in Section 3.2.

BOO's coding can be summarized as converting the numerical floating-point weights into binary type before calculating the total number of binary ones, and a detailed code snippet can be found in Supporting Material Section A.6. After converting every float-typed parameter into hexadecimal and then binary type, every float-typed parameter has been converted into a corresponding integer that records the total number of binary ones 1_2. Lastly, BOO equals the summation of every integer corresponding to the converted parameter.

After the BOO's code definition, this innovative method is successfully configured into the automated evolutionary-based framework by replacing the optimization objective where only the minimized value will be recorded as the best performance. During the search process, only the last epoch's BOO counting will be recorded after DNNs training because the same random seed will bring the same initial parameters and thus the same BOO counting value.

3.1.1 BOO examples

An example of this is related to CNN, and convolutional layers in CNNs can also benefit from BOO. During CNN's image processing, the convolution operation involves numerous kernels and patches of the input image multiplication.

Consider a 2×2 binary kernel matrix K and a 2×2 binary input patch P:

$$K = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.1}$$

$$P = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{3.2}$$

The convolution operation (ignoring stride and padding for simplicity) can be visualized as a matrix multiplication:

$$R = K \odot P = \begin{bmatrix} k_{11} \times p_{11} + k_{12} \times p_{12} \\ k_{21} \times p_{21} + k_{22} \times p_{22} \end{bmatrix} = \begin{bmatrix} 1 \times 0 + 0 \times 1 \\ 0 \times 1 + 1 \times 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.3}$$

If the matrix elements are uncertain, there should be in total of 4 additions and 2 multiplications for this single round. However, after invoking BOO the scenario might look like the example, once 0 were detected, output could be given directly, so time-consuming additions and multiplications in digital circuit design could be reserved for other more specific or important purposes.

In addition, equations presented here elegantly delineate the process of calculating the BOO for a hypothetical 2×2 weights matrix within a neural network model in Equation 3.4, consisting of weights' elements w_{ij}, representing individual weights of the network in floating-point format.

$$\text{Weights} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \tag{3.4}$$

The focus then shifts to Equation 3.5, which illustrates the transformation of each weight element from its floating-point representation, denoted as w_{ij}, to a binary format, denoted as bw_{ij}. Here, the BOO of each weight is computed, essentially quantifying the number of binary ones present in each weight's binary representation.

$$\text{BOO}(\text{Weights}) = \begin{bmatrix} \text{BOO}(w_{11}) & \text{BOO}(w_{12}) \\ \text{BOO}(w_{21}) & \text{BOO}(w_{22}) \end{bmatrix} = \begin{bmatrix} \text{BOO}(bw_{11}) & \text{BOO}(bw_{12}) \\ \text{BOO}(bw_{21}) & \text{BOO}(bw_{22}) \end{bmatrix} \tag{3.5}$$

The total BOO for the entire weight matrix is summarized in Equation 3.6, where the individual BOO values of each weight element are aggregated to yield the total BOO value for the weight matrix. This total BOO value serves as a crucial metric and a lower BOO indicates a potentially more efficient implementation.

$$\text{BOO}(\text{Weights}) = \text{BOO}(bw_{11}) + \text{BOO}(bw_{12}) + \text{BOO}(bw_{21}) + \text{BOO}(bw_{22}) \tag{3.6}$$

Finally, Equation 3.7 generalizes the BOO computation, extending it to an arbitrary number of weight elements. N equals the total number of weight elements of the chosen neural networks.

$$\text{BOO}(\text{Weights}) = \sum_{n=0}^{N} \text{BOO}(bw_n) \tag{3.7}$$

3.1.2 Parameter Size and FLOPs

To evaluate the impact of BOO, traditional CNN performance evaluator metrics Parameter Size and FLOPs have also been taken into consideration.

Parameter size is a fundamental metric in CNN models' complexity, calculated by the summation of trainable or gradient-required parameters, shown in the code snippet derived from the project. Firstly, for all the CNN parameters referred to by $model.parameters()$, the filter is needed and only the trainable or gradient-required parameters remain. Then, for each tensor v, we calculate the total number of parameters in this tensor by $np.prod(v.size()$, and then sum up the number of parameters for all the trainable tensors of the model.

FLOPs measure the computational load of the running CNN model, and it is the metric that helps in deploying CNN models into real-time operation systems. As a result, random data will be fed into the CNN model before flop counting is launched. The calculation of FLOPs is performed by feeding random data $torch.randn(1, 3, 32, 32)$ into the CNN model $model(torch.autograd.Variable(random_data).to(device))$ and tracking the number of floating-point operations executed during the model's forward pass. Each operation is recorded and the cumulative total represents the FLOPs of the CNN model.

3.2 BOO Benefits Analysis

The BOO (Binary One Optimization) in DNNs offers benefits, which could be summarized as arithmetic efficiency, hardware acceleration, memory efficiency, power dissipation, and others.

3.2.1 Arithmetic Efficiency

Started with computational efficiency with binary arithmetic, binary arithmetic inherently simplifies certain operations, such as multiplication and addition. A fewer number of 1's in neural network weights brings fewer arithmetic operations, such as multiplication and addition, which contribute to neural network complexity, memory occupation, and power dissipation. A notable example is the multiplication of any number with 0 yields 0. Moreover, the addition of any number with 0 retains the original number, which means more weights will interact with the redundant addition operation, not to mention redundant carry bits among many addition operations, whereas multiplication and carry bits require a more complex circuit in FPGA design. As a result, resources for both addition and carry bits can be saved to optimize the DNNs, particularly useful for arithmetic operation resources-limited embedded system peripherals and platforms.

3.2.2 Hardware Acceleration

Followed by the hardware acceleration, since the BOO leads to fewer and fewer weights being 1 in their binary form, then the weight matrix becomes sparse, enabling the use of customized algorithms and hardware accelerators designed for sparse matrix-vector and matrix-matrix multiplications [36]. Many modern hardware accelerators, like FPGAs and custom ASICs, can be optimized for fewer binary 1's or low-bit operations, in terms of circuit design, FPGAs, and System-on-Chips (SoCs). If fewer and fewer binary 1's are the optimization goals, these accelerators can be designed to skip some operations, leading to faster convergence towards feed-forward and backpropagation.

BOO can optimize the design of hardware platforms by skipping some operations or bits. FPGAs, have numerous logic blocks and interconnects, and fewer binary 1's by BOO can result in fewer logic blocks and processing unit utilization, such as fewer AND XOR logic gates, in FPGA design, not to mention reconfigurable FPGA's tailored optimization of binary data path and data operation (MAC, ALU), which improves resource utilization and the possibility to deploying of larger and more powerful models on the same resource-limited embedded system devices, whereas a similar situation happens to System-on-Chips (SoCs) as well.

3.2.3 Memory Efficiency

BOO can significantly reduce the memory requirements compared to the original DNNs' weights, the original binary 1's consumes more registers to store high voltage status information in circuit design, which, when freed up, can be reallocated to improve the computational power dissipation and efficiency of the system [37]. This reduction in register and buffer utilization can lead to faster register and buffer data fetch, load, and processing, and thus lower buffer occupation and then memory efficiency, making it particularly useful for embedded system devices [38]. By reducing both these factors, BOO is expected to have a lower energy footprint.

3.2.4 Power Dissipation Analysis

BOO can also help to reduce power consumption, mainly by reducing switching activities. In digital circuits where digital 0's represent low voltage and digital 1's high voltage, transitions between digital 0's and 1's, are switching activities, and the positive edge/rising edge of switching transitions, or low-to-high transition, happens when low voltage digital 0's jumps up to 1's, and vice versa negative edge/falling edge, or high-to-low transition. After knowing the basic knowledge and terms of digital circuits, we will kick off the power dissipation analysis for BOO.

Basic power and energy consumption could be calculated by:

- Power is drawn from a voltage source attached to the V_{DD} pin(s) of a chip.
- Instantaneous Power: $P(t) = i_{DD}(t)V_{DD}$
- Energy: $E = \int_0^T P(t)dt = \int_0^T i_{DD}(t)V_{DD}dt$
- Average Power: $P_{\text{avg}} = \frac{E}{T} = \frac{1}{T}\int_0^T i_{DD}(t)V_{DD}dt$

After that, with the help of calculus, Switching power can be:

$$\begin{aligned} P_{\text{switching}} &= \frac{1}{T}\int_0^T i_{DD}(t)V_{DD}dt \quad i(t) = C\frac{dV}{dt} \\ &= \frac{V_{DD}}{T}\int_0^T i_{DD}(t)dt \\ &= \frac{V_{DD}}{T}\left[Tf_{\text{sw}}CV_{DD}\right] \\ &= CV_{DD}^2 f_{\text{sw}} \end{aligned} \tag{3.8}$$

- $P_{\text{switching}}$: power consumed due to switching activities.
- C: capacitance being charged or discharged.
- f_{sw}: switching frequency.
- V_{DD}: supply voltage.

And sometimes switching power with activity factor is:

$$\begin{aligned} P_{\text{switching}} &= K_{\text{trans}}\, CV_{DD}^2 f_{\text{clock}} \\ \Rightarrow P_{\text{switching}} &= C_{\text{eff}}\, V_{DD}^2 f_{\text{clock}} \text{ where } C_{\text{eff}} = K_{\text{trans}}\, C \end{aligned} \tag{3.9}$$

Where:

- K_{trans}: activity factor or transition constant.
- C_{eff}: effective capacitance, which considers the activity factor.
- f_{clock}: clock frequency.

Dynamic power consumption in transistors:

$$P_{dyn} = C_{eff} \times V_{dd}^2 \times f_{\text{clock}} + t_{sc} \times V_{dd} \times I_{\text{peak}} \times f_{\text{clock}} \tag{3.10}$$

Where:

- P_{dyn}: dynamic power consumption.
- C_{eff}: effective capacitance.
- t_{sc}: short-circuit duration.
- I_{peak}: peak current during switching.

For the same embedded system devices that will implement power-consuming DNNs, parameters K_{trans} C_{eff}, t_{sc}, and I_{peak} are all determined before manufacture and treated as constant in this scenario. As a result, power dissipation is only dependent on the frequency (how fast the transition or switching goes), in the shape of $Power(frequency)$.

It is worth noting that the frequency of the transitions between high and low voltage states in the digital wave affects the dynamic power consumption. This is because every time a transition occurs, there is a small amount of energy dissipated due to the charging and discharging of the capacitive loads in the circuit. So, a waveform with frequent transitions will consume more dynamic power than a waveform with fewer transitions, even if they both have the same clock frequency.

As a result, some salient conclusions could be made about our innovative BOO:

- Power consumption in digital circuits is significantly influenced by the frequency of high-low voltage transitions, $P_{switching}$, where frequency refers to the switch rate between high and low-voltage states. The decrease in BOO performance helps to reduce the frequency of high-low voltage transitions, where fewer and fewer binary 1's over the same length of binary strings result in fewer and fewer ups and downs in the digital waveform.
- The BOO, by reducing the number of the digital 1's in neural network weights, inherently minimizes the number of transitions, or frequency f_{sw}, leading to decreased dynamic power consumption $P_{switching}$.

3.3 Random Search Setup

Unlike other frameworks that invoke additional untailored random search as the control group, which brings extra potential compatible issues, our simplified choice is EMONAS hyperparameter optimization. By assigning offspring and generations to 0, de facto disablement, and parent population size equal to the total population pool, de facto population pool, we can eliminate the need for complex frameworks and ensure efficient exploration of the solution space. The reason behind this is that parents in each generation are randomly selected through tournament selection. Consequently, if no offspring and generations are specified, the EMNOAS cannot evolve as intended, with only the initial randomly generated parents contributing to the search process, as shown in Table 3.1.

Table 3.1: Evolutionary Search and Random Search Hyperparameter Example

Hyperparameter	**Evolutionary Value**	**Random Value**
epoch	20	20
n_gen	10	0
n_offspring	10	0
pop_size	10	100
seed	0	0

Chapter 4

Results and Analysis

In this chapter, we delineate the results obtained from our experiments and provide a comprehensive discussion of the findings.

4.1 Simulation Setup

In the neural network study trained in dataset CIFAR-10, a set of default hyperparameters, as summarized in Table 4.1, was utilized to establish a framework baseline. CIFAR-10 is a widely recognized dataset used in the field of machine learning and computer vision, consisting of 60000 32x32 color images divided into 10 different classes, each representing a distinct category such as animals (dogs, cats, birds) and vehicles (trucks, cars, ships). The $epoch$ hyperparameter, which is set to 20 by default, signifies the number of complete feed-forward processes of the entire training dataset. The $init\_channels$, with a default value of 16, determines the number of filters present in the initial cell of the network. Further, the framework's depth is controlled by the $layers$ hyperparameter, which is set to 11. Each cell in the network, as depicted in Figure 2.7, contains grey boxes. These boxes represent computational blocks. Additionally, the network features blue circles, as shown in the same figure. These circles symbolize basic computational units. The units include operations such as convolution, pooling, and batch-normalization [39]. The number of blocks within each cell, and the maximum number of these blue circles or nodes, are determined by the hyperparameters n_block and n_cell. For this network, the default values for n_block and n_cell are 5 and 2, respectively. The evolution strategy is dominated by the n_gen hyperparameter, which sets the number of generations to 10, while n_node denotes the number of nodes in each phase, with a default of 6. To control the size of each DNN generation, the $n\_offspring$ and pop_size hyperparameters specify the number of offspring per generation and the population size, with values set to 10 each. The model's flexibility in terms of operation selection is facilitated by the "n_ops" hyperparameter, which allows for the consideration of 9 different operations. The encoding type for the network search is given by the "search_space" parameter, with 'macro' as its default value. Lastly, to ensure reproducibility, a "seed" value of 0 is used.

Table 4.1: Default Hyper-parameters in the Neural Network Argument

Description	**Hyperparameter Name**	**Default Value**
Epoch of training	epoch	20
Filters for the first cell	init_channels	16
Layer	layers	11
Block in a cell	n_block	5
Cell	n_cell	2
Generation value	n_gen	10
Node per phase	n_node	6
Offspring per generation	n_offspring	10
Operations considered	n_ops	9
Population size	pop_size	10
Micro or macro encoding	search_space	'macro'
Random seed	seed	0

In most cases, hyperparameters under scrutiny include the search methodology (with a distinction between evolutionary search and random search), the epoch size (comprising both 20-epoch training and 100-epoch training during the search phase), and the nature of the search (differentiated into micro search and macro search).

As for the search space choices, the convergence of the defined search space or the selected DNN models represents a tradeoff or balance between computational efficiency and search performance. Initially, a combination of trial-and-error and default search space data selection was employed and served as the initial search space choice reference to guide the subsequent search rounds. This foundational exploration provided the blueprint for subsequent search round iterations, culminating in the search space configurations presented in the default hyperparameter Table 4.1.

A significant portion of our exploration time was dedicated to the hyperparameters, shown in Table 4.1, $epoch, n\_gen, n\_offspring$, and pop_size. Before that, I commenced the investigation with a constant $seed$, utilizing Python's random seed functionality. Although the default seed was set to 0, subsequent trials with seeds 0, 1, 2, and 23 revealed consistent performance trends, albeit with varying performance values. Consequently, the seed is defined as a constant 0 for the remainder of our experiments. The remaining constants, namely $init\_channels, layers, n\_block, n\_cell, n\_ops$, and n_node are architectural hyperparameters that define the basic cell shape, shown in Figure 2.7. Given the constraints of my computational resources and the time-intensive nature of our research, I opted not to explore alternative basic cell configurations, rendering these parameters as constant hyperparameters.

With respect to the evolutionary-based hyperparameters, shown in Table 4.1, n_gen, and $n\_offspring$. My initial configuration, which proved to be computationally unacceptable, was

$$[n\_gen, n\_offspring] = [25, 25]$$

This necessitated a reduction in these figures, ranging from 24, 22, 20, and 16 to ultimately settling on

$$[n\_gen, n\_offspring] = [10, 10]$$

The $epoch$ also underwent rigorous evaluation. While the initial epoch was set at 20, I experimented with both larger and smaller values, and my findings indicated that smaller epochs, such as 5, 3, or even 0, were insufficient for effective training. Conversely, larger epochs, like 36 or 100, enhanced accuracy, but were computationally demanding and adversely impacted other optimization objectives, including FLOPs, parameter size, and our innovative objectives BOO during even the search phase.

In the ensuing graphical representations in Figure 4.1, 4.2, and 4.3, the vertical axis delineates our primary objective, which is the classification errors. Concurrently, the horizontal axis encapsulates our secondary objective, the innovative BOO, quantified by the number of ones. The epitome of DNN performance would gravitate toward the origin, in Figure 4.1, 4.2, and 4.3. This is because an ideal DNN would minimize both classification errors (thus being lower on the vertical axis) and the number of ones in BOO (thus being closer to the left on the horizontal axis), which mathematically means:

$$\lim_{\text{epoch}\to\text{ideal}} \text{Func\_Optimization}(\text{Objective\_1=Error}, \text{Objective\_2=BOO}) = (0,0) \tag{4.1}$$

The contiguous band that approaches the origin represents the Pareto front. This front demarcates the set of optimal DNN architectures, where any further improvement in one objective would result in a deterioration of the other.

4.2 Search A: Evolutionary versus Random, Macro

Before stepping into the search result analysis, some common things that appear in all these search result analysis sections will be discussed. Each dot in the figure shows one distinct neural network architecture from the population pool. The BOO in the horizontal axis depicts the total number of binary 1's while the vertical axis shows the image classification error. The Pareto-front lines are drawn for both the experiment group and the baseline group.

In this section, random searched neural architecture achieves the optimized image classification error of 12.68%. However, the evolutionary searched neural architecture in blue achieves the optimized BOO of 3757158.

- **Variable Element:** evolutionary and random search, demonstrated by hyperparameters n_gen, $n\_offspring$, and pop_size, shown in Table 4.2, discussed in Section 3.3.

- **Constant Element:** The experiment maintains consistency in several parameters across both search methods. This includes the macro search type, a fixed training duration of 20 epochs, and other parameters such as blocks and channels, shown in Table 4.2.

- **Observation:** The evolutionary search (represented in blue) exhibits superior performance in the second objective, BOO, and the random search (depicted in orange) demonstrates superior performance in the first objective, image classification error, shown in Figure 4.1.

In our experimental setup, we juxtapose the evolutionary search method against the random search method, in macro encoding. An observation from the results, shown in Figure 4.1, is the performance differential between the

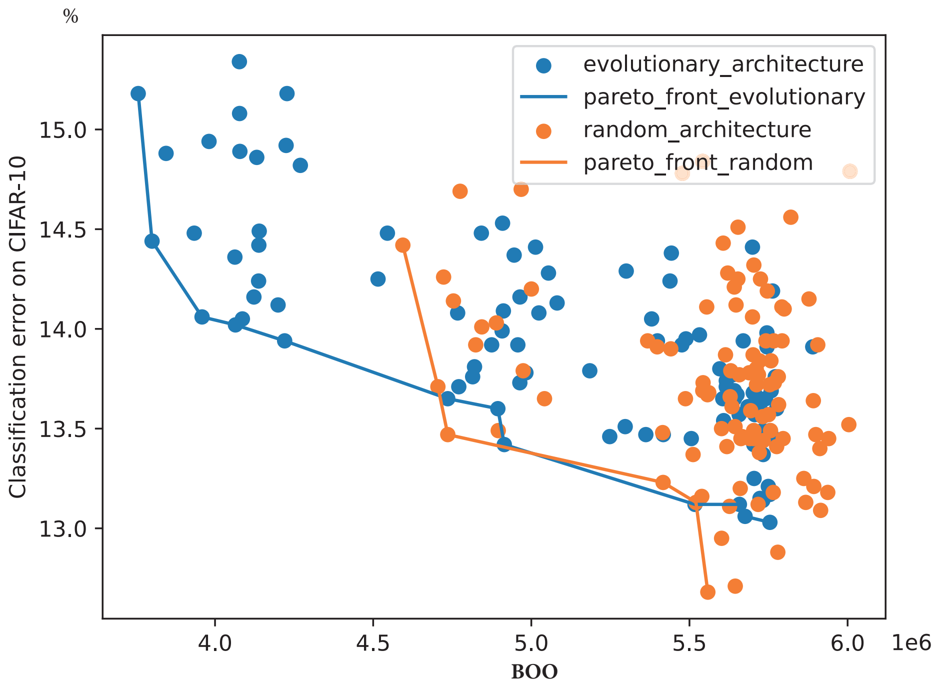


Figure 4.1: Trade-off Frontier Comparison Between Random Search and EMONAS-BOO, In Macro Encoding Scheme

two search methods. The evolutionary search, illustrated in blue, excels in the second objective of BOO. Contrarily, the random search, portrayed in orange, performs better in the first objective of image classification error. This superior performance of the evolutionary search can be ascribed to its better optimization towards our innovative second objective, BOO.

4.3 Search B: Comparing 20 and 100 Epoch, in Macro and Evolutionary

In this section, 100-epoch evolutionary searched neural architecture achieves the optimized image classification error of 9.52%. However, 20-epoch evolutionary searched neural architecture in blue achieves the optimized BOO of 3757158.

- **Variable Element:** training epoch size, shown in Table 4.3.

Table 4.2: A Detailed Comparison of Hyperparameter Configurations Between Blue EMONAS-BOO and Red Random Search within the Macro Encoding Scheme

Hyperparameter	**EMONAS-BOO**	**Random**
epoch	20	20
init_channels	16	16
layers	11	11
n_block	5	5
n_cell	2	2
n_gen	10	0
n_node	6	6
n_offspring	10	0
n_ops	9	9
pop_size	10	100
search_space	'macro'	'macro'
seed	0	0

Table 4.3: A Detailed Comparison of Hyperparameter Configurations, Focusing on Epoch Numbers 20 and 100, Within the EMONAS-BOO Micro Encoding Scheme

Hyperparameter	**Epoch-20**	**Epoch-100**
epoch	20	100
init_channels	16	16
layers	11	11
n_block	5	5
n_cell	2	2
n_gen	10	10
n_node	6	6
n_offspring	10	10
n_ops	9	9
pop_size	10	10
search_space	'macro'	'macro'
seed	0	0

- **Constant Element:** The experiment maintains consistency in several parameters across both search methods. This includes the macro search type, a fixed evolutionary search, and other parameters such as blocks and channels, shown in Table 4.3.

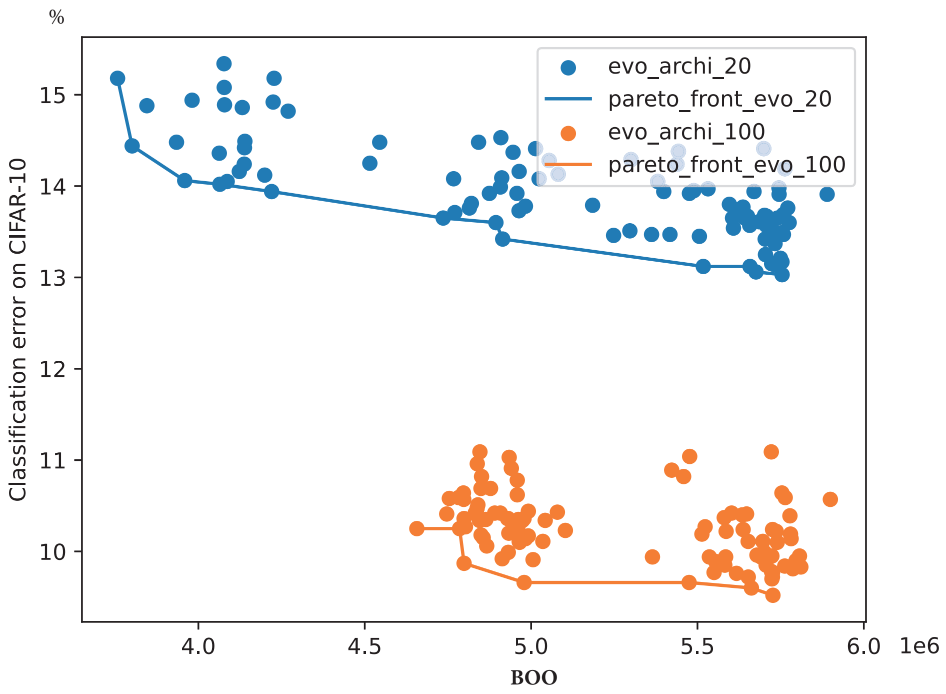


Figure 4.2: Comparative Analysis of Epoch Optimization Across Varied Numerical Values within the EMONAS-BOO Micro Encoding Framework

- **Observation:** While the Epoch-20 search (represented in blue) exhibits superior performance in the second objective, BOO (Binary One Optimization), the Epoch-100 search (depicted in orange) demonstrates superior performance in the first objective, image classification errors, with higher training epoch 100 during the search process, shown in Figure 4.2.

The difference between these two search methods can be potentially attributed to their excessive training in image classification, which exacerbates the optimization towards the second objective BOO. An increased epoch number can bring the image classification accuracy increases, but might not optimize or even exacerbate the innovative objective BOO.

4.4 Search C: Evolutionary Versus Random, Micro

In this section, evolutionary searched neural architecture in blue achieves the optimized image classification error of 11.66%. Similarly, evolutionary searched neural architecture in blue also achieves the optimized BOO of 3040227.

Table 4.4: A Detailed Comparison of Hyperparameter Configurations Between EMONAS-BOO and Random Search within the Micro Encoding Scheme

Hyperparameter	**EMONAS-BOO**	**Random**
epoch	20	20
init_channels	16	16
layers	11	11
n_block	5	5
n_cell	2	2
n_gen	10	0
n_node	6	6
n_offspring	10	0
n_ops	9	9
pop_size	10	100
search_space	'micro'	'micro'
seed	0	0

- **Variable Element:** The evolutionary search with the random search, demonstrated by hyperparameters n_gen, $n\_offspring$, and pop_size, shown in Table 4.4, discussed in Section 3.3.
- **Constant Element:** The experiment maintains a consistent framework, characterized by a micro search type, a training duration of 20 epochs, and specified blocks and channels, shown in Table 4.4.
- **Observation:** Evolutionary search exhibits obvious superiority in terms of achieving higher precision and optimizing both classification accuracy and BOO (Binary One Optimization) simultaneously, shown in Figure 4.3.

The Pareto front of the evolutionary search results encloses the random search's Pareto front results, suggesting that the evolutionary search is

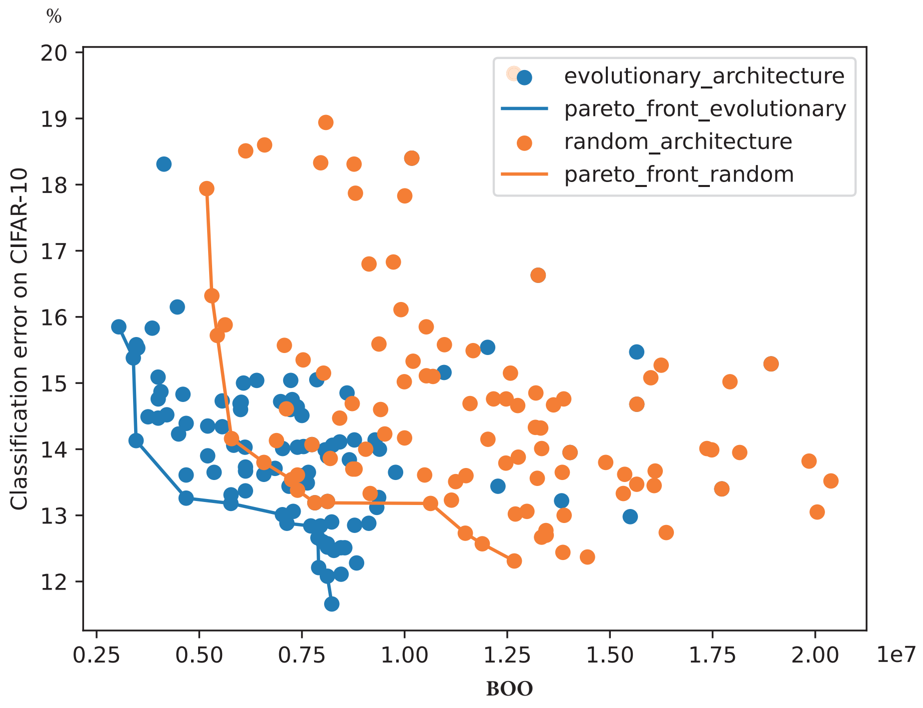


Figure 4.3: Trade-off Frontier Comparison Between Random Search and EMONAS-BOO, In Micro Encoding Scheme

capable of discovering more optimal neural networks' solutions towards both objectives than the random search.

4.5 Performance Analysis of Evolutionary Micro Search

Due to Micro Evolutionary Search's superior performance discussed in Section 4.4, its performance analysis of the innovative objective BOO's effectiveness is investigated by the introduction of some classical neural network performance evaluators, FLOPs, and parameter sizes. Neural networks, derived from the Evolutionary Micro Search configuration of Section 4.4, will be taken into the performance analysis.

Table 4.5, provides a comprehensive summary of performance metrics for Pareto front neural network architectures discovered through Evolutionary Micro Search (12 out of 100). Each row corresponds to a distinct neural network, as denoted by its unique Network ID, with corresponding metrics,

Table 4.5: Performance Metrics for Neural Networks in the Pareto Front of Evolutionary Micro Search

Network_ID	Acc %	Param Size (MB)	BOO	FLOPs (M)
97	84.15	0.177690	3040227	32.5235
80	84.62	0.198426	3394436	34.8705
96	85.87	0.202282	3461520	37.4305
74	86.74	0.273610	4680097	47.519
48	86.82	0.337210	5769896	61.7123
69	86.99	0.412586	7024962	72.6609
95	87.12	0.418762	7134078	73.4186
86	87.16	0.452794	7715063	80.4849
32	87.34	0.463034	7889428	81.644
59	87.79	0.464330	7907194	82.9957
29	87.92	0.476682	8119060	84.3269
85	88.34	0.482858	8227340	84.9987

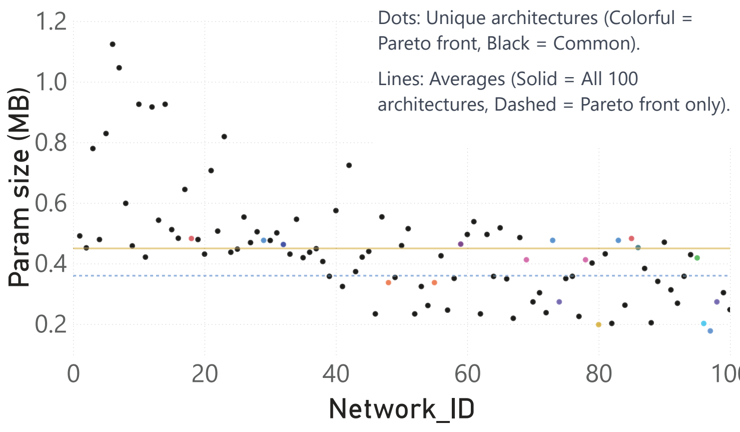


Figure 4.4: Parameter Sizes of Neural Network Architectures Discovered through Micro Evolutionary Search

including image classification accuracy (the first optimization objective), parameter size (in MB), BOO (the second optimization objective), and FLOPs.

To analyze the parameter size, FLOPs, and the innovative objective BOO, I have visualized and plotted Figures 4.4, 4.5 and 4.6. These figures visualize the performance metrics (MB means 10^6 level) of the parameter size, FLOPs, and

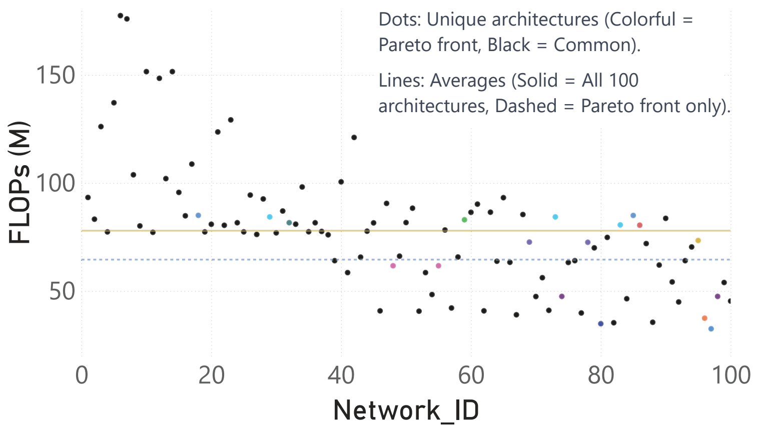


Figure 4.5: FLOPs of Neural Network Architectures Discovered through Micro Evolutionary Search

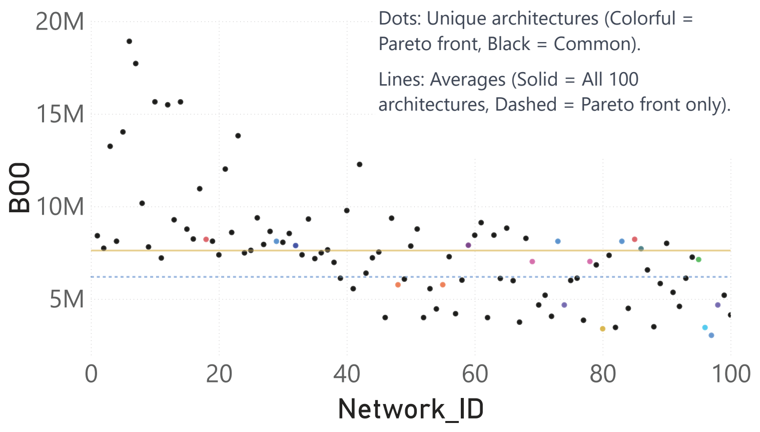


Figure 4.6: BOO of Neural Network Architectures Discovered through Micro Evolutionary Search

the innovative objective BOO, for all 100 distinct neural networks discovered through Micro Evolutionary Search. 100 distinct neural network architectures are denoted with distinct Network IDs in the horizontal axis, ranging from

Table 4.6: Comparative Analysis of DNNs Metrics Across Searched Architectures

Metric	Acc	Param Size (MB)	BOO	FLOPs (M)
Max	88.34%	1.211594	20382188	194.9271
Min	80.32%	0.17769	3040227	32.5235
Percentage	8.02%	85.3342%	85.0839%	83.3150%

Network_ID = 1 to Network_ID = 100, and each black dot represents a specific neural network in the figure. Among these 100 different neural networks, 12 out of 100 neural networks were Pareto front neural networks, which achieve the best multi-objective optimization in image classification accuracy and BOO, and these 12 Pareto front neural networks were represented with colorful dots in Figure 4.4, 4.5 and 4.6, which distinguishes them from other neural networks represented by black dots for comparison. In addition, the dashed line, in Figure 4.4, 4.5 and 4.6, is the averaged performance metrics value for the 12 Pareto front neural networks, and the solid line, in Figure 4.4, 4.5 and 4.6, is the averaged performance metrics value for all 100 neural networks.

Network IDs serve as unique markers for neural network architectures, sequentially generated during the evolutionary search process. These IDs range from 1 to 100, corresponding to generations 1 through 10. Consequently, a higher Network ID indicates that the respective network was generated in the later stages of the evolutionary search. Notably, the architectures discovered by EMONAS, particularly the 12 optimized on the Pareto front, possess larger Network IDs, shown in the first column of Table 4.5. These 12 Pareto front architectures' averaged Network IDs equals 71. This suggests that extensive mutation and crossover of evolutionary algorithm processes have significantly contributed to the network's optimization, thereby affirming the efficacy of the EMONAS framework in identifying and refining promising neural network architectures.

Last but not least, the optimized achievement will be quantified in Table 4.6. Table 4.6 investigated all 200 search architectures in Figure 4.3 and extracted their maximum and minimum values. BOO was lowered by up to 85.1%, parameter size by 85.3%, and FLOPs by 83.3%.

As a result, some conclusions could be drawn from these figure plots. Firstly, Parameter size, FLOPs, and BOO have similar trends through the Micro Evolutionary Search, the smaller BOO correlate to smaller FLOPs and Parameter Size, which exactly shows better optimization for the neural network deployment for embedded system devices. Secondly, The pareto front neural networks, in colorful dot format, have a relatively higher network ID as offspring, which means sufficient evolutionary search, in the format of iterative and efficient selection mutation, crossover, have a higher possibility to produce optimal neural networks for the defined multi-objectives. Last but not least, Pareto front neural networks selected by EMONAS-BOO have achieved excellent performance in terms of parameter size, FLOPs, and the innovative objective BOO, because the averages Pareto front performance metrics value, in the dashed line, is lower than the majority of the searched 100 neural networks in black dot, averaged in the solid line.

4.6 Summary of Results and Analysis

Macro Search Analysis in Section 4.2: When comparing evolutionary search with random search in resource-efficient macro networks, where binary strings illustrate the block connection, the evolutionary search demonstrated superior performance in optimizing the BOO. However, random search showed a generally enhanced performance in optimizing the image classification accuracy (reducing classification errors), suggesting that this classic search method still plays an important role in modern NAS, beating random search is not easy.

Epoch Size Influence in Section 4.3: In an experiment that varied the training epoch size while keeping other parameters constant, the evolutionary search with a higher epoch count (100 epochs) showed a pronounced improvement in image classification accuracy. This indicates the importance of appropriate training epoch size in achieving optimal accuracy of DNNs in the tailored dataset, cifar-10 for image classification training. That is the reason the overall accuracy will increase moderately after configuring the training epoch size in the training process. However, the increase in training epoch size cannot guarantee that DNN's image classification accuracy increases linearly, the tradeoff between invested computational resources and accuracy increase needs to be taken into consideration.

Micro Search Analysis in Section 4.4: In the context of micro search, evolutionary search is superior to random search, and micro search can be the preferred search method since micro evolutionary search also outperforms macro evolutionary search. This was evident in its superior performance in both primary objectives: classification accuracy and BOO. Furthermore, the Pareto front of the evolutionary search results encompassed that of the random search, underscoring the evolutionary search's capability to identify more and better potential DNN architectures easily. EMONAS takes advantage of the

NSGA-Net framework to optimize our innovative objective BOO and achieve excellent optimization performance.

Performance analysis in Section 4.5: Since Micro Evolutionary Search outperforms others, this section has investigated the effectiveness of the innovative objective BOO, with the help of parameter size and FLOPs. The Pareto front neural networks outperform the majority of normal neural networks, which proves the effectiveness of the BOO and its potential for embedded system devices.

In conclusion, this project focuses on developing neural network architectures for hardware through automation and multi-objective optimization, enabling the identification and implementation of optimal DNN hyperparameters in embedded systems without sacrificing accuracy. Fortunately, the introduction of the innovative method helps the automated framework successfully select 12 potential DNN candidates that reside on the Pareto front.

Chapter 5

Conclusions and Future work

In this chapter, the conclusion will be made before stepping into the limitations of the project. After that, potential future work will be followed.

5.1 Conclusions

In conclusion, with the development of machine learning and the spread of Deep Neural Networks, the market has raised a higher demand for deploying DNNs into embedded system devices, which is conflicting due to DNNs' inherent complexity and embedded systems' inherent simplicity. Especially, the challenges were mainly the inherent resource allocation paradox between resource-demanding DNNs and resource-limited embedded system devices. In this thesis, the innovative objective Binary One Optimization as a DL implementation performance assessment tool was introduced and successfully incorporated into a customized automated framework EMONAS-BOO to improve the efficiency of DNNs deployment on resource-limited embedded system devices while keeping the DNNs' classification accuracy by multi-objective optimization. As for the customized framework, the full automation originated from Neural Architecture Search, and the efficient search algorithm was evolutionary. Notably, with the help of a micro encoding scheme and evolutionary-based search algorithm, the EMONAS-BOO produced 12 out of 100 neural networks that keep their image classification accuracy while achieving better performance in BOO. Compared with random search, evolutionary-searched BOO was lowered by up to 85.1%, parameter size by 85.3%, and FLOPs by 83.3%. These improvements were accomplished without sacrificing the image classification accuracy, which saw an increase of 8.0%.

5.2 Limitations

The limitations could be summarized as limited and unstable computational resources and a limited variety of datasets.

As for the computational resources, conducting large-scale NAS experiments requires considerable computational resources to cope with extremely large search space. For each set of hyperparameters, we need to build a neural network and start training this neural network from the random initialization, so we can only try thousands or tens of thousands of sets of hyperparameters. The search results presented in this paper were derived from different platforms, from initial resource-constrained laptops and various cloud platforms such as Google Cloud, Amazon Web Services (AWS), and Oracle Cloud, and this hybrid setup introduced disruptions and potential inconsistencies in our search performance.

Turning our attention to the dataset diversity, temporal and computational constraints restricted our validation and testing to a singular image recognition dataset: CIFAR-10. This limitation raises potential concerns regarding inadvertent biases and coincidence findings, potentially undermining the robustness of the EMONAS framework. As a result, the comprehensive robustness and general applicability of EMONAS, consequently, remain subjects for further investigation.

5.3 Future work

Future work will be enhancing computational resources and broadening dataset diversity. Firstly, assessing the availability of the EMONAS with more efficient and stable computational resources by utilizing dedicated cloud computational resources enables more hyperparameter optimization choices or ranges.

Moreover, extend the EMONAS test and validate to multiple datasets. Validating and testing the EMONAS within CIFAR-10 only will raise concerns regarding biases and coincidence. Different kinds of datasets should be taken into consideration for further search and training of EMONAS.

Last but not least, validating the selected DNN candidates' real and physical performance in various ways will prove the selected DNN architectures by EMONAS-BOO are effective. Despite my excessive literature review of NAS, including published and mature models such as frameworks MONAS [40], DeepMaker [25], and DPPNet [41], a deeper performance analysis, emphasizing the performance improvement of these identified DNN architectures, is terminated at the search phase where potential DNN candidates are selected, and few discuss the further validation of the potential DNN candidates' physical performance, such as power consumption, latency, etc. Nevertheless, deeper performance analysis could start with the tangible performance assessment of candidate DNNs implementation on physically embedded system microcontrollers at the hardware level, and candidate DNNs FLOPs and parameter size, as BOO's reference, at the software level.

References

[1] E. Aptoula, M. C. Ozdemir, and B. Yanikoglu, "Deep learning with attribute profiles for hyperspectral image classification," *IEEE Geoscience and Remote Sensing Letters*, vol. 13, no. 12, pp. 1970–1974, 2016. doi: 10.1109/LGRS.2016.2619354 [Page 2.]

[2] K. Noda, Y. Yamaguchi, K. Nakadai, H. G. Okuno, and T. Ogata, "Audio-visual speech recognition using deep learning," *Applied intelligence (Dordrecht, Netherlands)*, vol. 42, no. 4, pp. 722–737, 2015. [Page 2.]

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature (London)*, vol. 518, no. 7540, pp. 529–533, 2015. [Page 2.]

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016. ISBN 9781467388511. ISSN 1063-6919 pp. 770–778. [Page 2.]

[5] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016. ISBN 978-3-319-46487-9 pp. 649–666. [Page 2.]

[6] 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 <0.5mb model size," 2016. [Page 3.]

[7] P. Research, "Embedded systems market size 2020 by product, semiconductor, technology, industry analysis, segments, region and

growth forecast to 2027," 2023, accessed: insert date here. [Online]. Available: https://www.precedenceresearch.com/embedded-systems-market [Pages viii and 4.]

[8] Y.-H. Kim, B. Reddy, S. Yun, and C. Seo, "Nemo: Neuro-evolution with multiobjective optimization of deep neural network for speed and accuracy." [Page 4.]

[9] Z. Lu, I. Whalen, V. Boddeti, Y. Dhebar, K. Deb, E. Goodman, and W. Banzhaf, "Nsga-net: neural architecture search using multi-objective genetic algorithm," in *GECCO 2019 - Proceedings of the 2019 Genetic and Evolutionary Computation Conference*, ser. GECCO '19. ACM, 2019. ISBN 9781450361118 pp. 419–427. [Pages 4, 29, and 30.]

[10] C.-H. Hsu, S.-H. Chang, J.-H. Liang, H.-P. Chou, C.-H. Liu, S.-C. Chang, J.-Y. Pan, Y.-T. Chen, W. Wei, and D.-C. Juan, "Monas: Multi-objective neural architecture search using reinforcement learning," 2018. [Page 4.]

[11] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016. [Pages 4 and 5.]

[12] R. Pramoditha. (2022) Creating a multilayer perceptron (mlp) classifier model to identify handwritten digits. Accessed: 20 07 2023. [Online]. Available: https://towardsdatascience.com/creating-a-multilayer-perceptron-mlp-classifier-model-to-identify-handwritten-digits-9bac1b16fe10 [Pages viii, 8, and 9.]

[13] K. Patel. (Sep 8, 2019) Mnist handwritten digits classification using a convolutional neural network (cnn). Accessed: 20 07 2023. [Online]. Available: https://towardsdatascience.com/mnist-handwritten-digits-classification-using-a-convolutional-neural-network-cnn-af5fafbc35e9 [Pages viii, 9, and 10.]

[14] 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. doi: 10.1109/5.726791 [Pages viii and 11.]

[15] A. L. Chandra, "Mcculloch-pitts neuron — mankind's first mathematical model of a biological neuron," *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1 [Pages viii, 10, and 12.]

[16] A. Geron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Sebastopol: O'Reilly Media, Incorporated, 2017. ISBN 9781491962299 [Page 10.]

[17] OpenAI, "Chatgpt," https://openai.com/blog/chatgpt, 2021, accessed April 19, 2023. [Page 11.]

[18] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012. [Page 15.]

[19] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. Pearson custom library. Pearson Education UK, 2013. ISBN 9781292024202 [Page 16.]

[20] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning series. Cambridge: MIT Press, 2009. ISBN 0262028182 [Page 16.]

[21] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-based systems*, vol. 212, pp. 106 622–, 2021. [Page 16.]

[22] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018. [Page 16.]

[23] J. Zhong, X. Hu, J. Zhang, and M. Gu, "Comparison of performance between different selection strategies on simple genetic algorithms." vol. 2, 01 2005. doi: 10.1109/CIMCA.2005.1631619 pp. 1115–1121. [Page 18.]

[24] L. Xie and A. Yuille, "Genetic cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2017. ISBN 9781538610329. ISSN 2380-7504 pp. 1388–1397. [Page 22.]

[25] M. Loni, S. Sinaei, A. Zoljodi, M. Daneshtalab, and M. Sjödin, "Deepmaker: A multi-objective optimization framework for deep neural networks in embedded systems," *Microprocessors and microsystems*, vol. 73, pp. 102 989–, 2020. [Pages 26, 29, and 54.]

[26] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Deploying deep neural networks in the embedded space," *arXiv.org*, 2018. [Page 29.]

[27] P.-E. Novac, G. Boukli Hacene, A. Pegatoquet, B. Miramond, and V. Gripon, "Quantization and deployment of deep neural networks on microcontrollers," *Sensors (Basel, Switzerland)*, vol. 21, no. 9, pp. 2984–, 2021. [Page 29.]

[28] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Computer Vision – ECCV 2018*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, pp. 289–304. ISBN 3030012484 [Page 29.]

[29] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," *arXiv.org*, 2018. [Pages 29 and 30.]

[30] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv.org*, 2017. [Pages 29 and 30.]

[31] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," *arXiv.org*, 2019. [Pages 29 and 30.]

[32] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *arXiv.org*, 2018. [Page 30.]

[33] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*. PMLR, 2013, pp. 115–123. [Page 30.]

[34] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *arXiv.org*, 2012. [Page 30.]

[35] 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 *ACM International Conference Proceeding Series; Vol. 227: Proceedings of the 24th international conference on Machine learning; 20-24 June 2007*. ACM, 2007. ISBN 1595937935 pp. 473–480. [Page 30.]

[36] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. [Page 35.]

[37] A. Asad, R. Kaur, and F. Mohammadi, “A survey on memory subsystems for deep neural network accelerators,” *Future internet*, vol. 14, no. 5, pp. 146–, 2022. [Page 35.]

[38] G. Raut, A. Biasizzo, N. Dhakad, N. Gupta, G. Papa, and S. K. Vishvakarma, “Data multiplexed and hardware reused architecture for deep neural network accelerator,” *Neurocomputing*, vol. 486, pp. 147–159, 2022. doi: https://doi.org/10.1016/j.neucom.2021.11.018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221016933 [Page 35.]

[39] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015. [Page 39.]

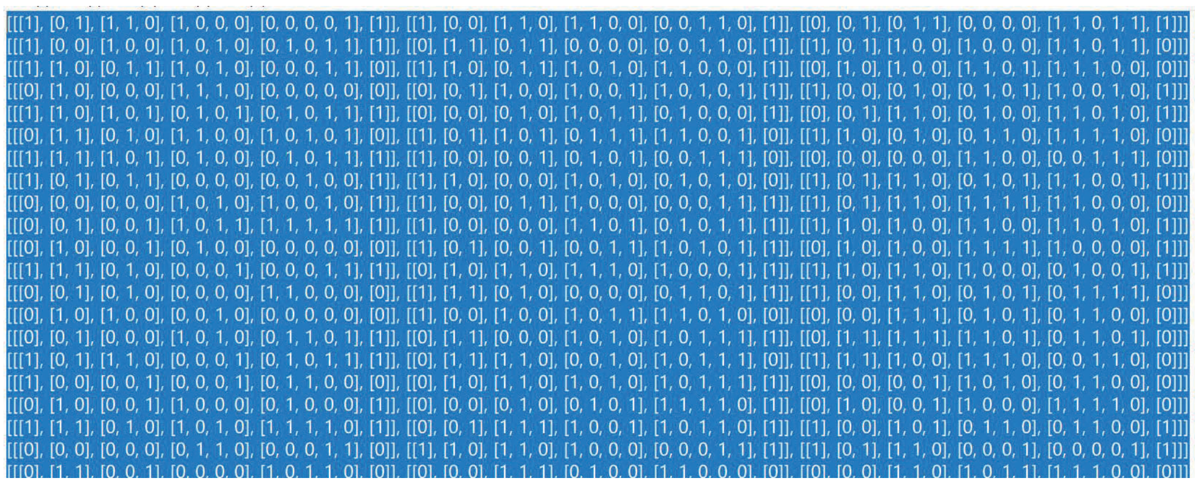
[40] C. Hsu, S. Chang, D. Juan, J. Pan, Y. Chen, W. Wei, and S. Chang, “MONAS: multi-objective neural architecture search using reinforcement learning,” *CoRR*, vol. abs/1806.10332, 2018. [Online]. Available: http://arxiv.org/abs/1806.10332 [Page 54.]

[41] J. Dong, A. Cheng, D. Juan, W. Wei, and M. Sun, “Dpp-net: Device-aware progressive search for pareto-optimal neural architectures,” *CoRR*, vol. abs/1806.08198, 2018. [Online]. Available: http://arxiv.org/abs/1806.08198 [Page 54.]

Appendix A

Supporting Material

A.1 Architecture Representation



```
[[[1], [0, 1], [1, 1, 0], [1, 0, 0, 0], [0, 0, 0, 0, 1], [1]], [[1], [0, 0], [1, 1, 0], [1, 1, 0, 0], [0, 0, 1, 1, 0], [1]], [[0], [0, 1], [0, 1, 1], [0, 0, 0, 0], [1, 1, 0, 1, 1], [1]]]
[[[1], [0, 0], [1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1, 1], [1]], [[0], [1, 1], [0, 1, 1], [0, 0, 0, 0], [0, 0, 1, 1, 0], [1]], [[1], [0, 1], [1, 0, 0], [1, 0, 0, 0], [1, 1, 0, 1, 1], [0]]]
[[[1], [1, 0], [0, 1, 1], [1, 0, 1, 0], [0, 0, 0, 1, 1], [0]], [[1], [1, 0], [0, 1, 1], [1, 0, 1, 0], [1, 1, 0, 0, 0], [1]], [[0], [1, 0], [1, 0, 0], [1, 1, 0, 1], [1, 1, 1, 0, 0], [0]]]
[[[0], [1, 0], [0, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0, 0], [0]], [[0], [0, 1], [1, 0, 0], [1, 0, 0, 1], [1, 0, 1, 0, 1], [1]], [[1], [0, 0], [0, 1, 0], [0, 1, 0, 1], [1, 0, 0, 1, 0], [1]]]
[[[1], [1, 0], [1, 0, 1], [0, 1, 0, 1], [0, 1, 0, 1, 1], [1]], [[0], [0, 0], [0, 1, 0], [1, 0, 1, 1], [0, 1, 0, 0, 0], [1]], [[0], [0, 1], [1, 1, 0], [0, 1, 0, 0], [1, 1, 0, 1, 0], [1]]]
[[[0], [1, 1], [0, 1, 0], [1, 1, 0, 0], [1, 0, 1, 0, 1], [0]], [[1], [0, 1], [1, 0, 1], [0, 1, 1, 1], [1, 1, 0, 0, 1], [0]], [[1], [1, 0], [0, 1, 0], [0, 1, 1, 0], [1, 1, 1, 1, 0], [0]]]
[[[1], [1, 1], [1, 0, 1], [0, 1, 0, 0], [0, 1, 0, 1, 1], [1]], [[1], [0, 0], [0, 0, 1], [0, 1, 0, 1], [0, 0, 1, 1, 1], [0]], [[0], [0, 0], [0, 0, 0], [1, 1, 0, 0], [0, 0, 1, 1, 1], [0]]]
[[[1], [0, 1], [0, 1, 1], [0, 0, 0, 0], [0, 0, 1, 0, 0], [1]], [[1], [1, 0], [0, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1, 0], [0]], [[1], [0, 1], [1, 1, 0], [0, 1, 0, 1], [1, 1, 0, 0, 1], [1]]]
[[[0], [0, 0], [0, 0, 0], [1, 0, 1, 0], [1, 0, 0, 1, 0], [1]], [[1], [0, 0], [0, 1, 1], [1, 0, 0, 0], [0, 0, 0, 1, 1], [1]], [[1], [0, 1], [1, 1, 0], [1, 1, 1, 1], [1, 1, 0, 0, 0], [0]]]
[[[0], [0, 1], [0, 0, 1], [1, 0, 1, 1], [1, 1, 1, 1, 1], [1]], [[1], [0, 0], [0, 0, 0], [1, 1, 0, 1], [0, 1, 0, 1, 1], [1]], [[1], [1, 0], [1, 0, 0], [1, 1, 0, 0], [1, 1, 0, 1, 0], [1]]]
[[[0], [1, 0], [0, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0, 0], [0]], [[1], [0, 1], [0, 0, 1], [0, 0, 1, 1], [1, 0, 1, 0, 1], [1]], [[0], [1, 0], [1, 0, 0], [1, 1, 1, 1], [1, 0, 0, 0, 0], [1]]]
[[[1], [1, 1], [0, 1, 0], [0, 0, 0, 1], [0, 0, 0, 1, 1], [1]], [[0], [1, 0], [1, 1, 0], [1, 1, 1, 0], [1, 0, 0, 0, 1], [1]], [[1], [1, 0], [1, 1, 0], [1, 0, 0, 0], [0, 1, 0, 0, 1], [1]]]
[[[0], [0, 1], [0, 1, 0], [0, 0, 0, 0], [1, 1, 0, 0, 0], [0]], [[1], [1, 1], [0, 1, 0], [0, 0, 0, 0], [0, 1, 1, 0, 1], [1]], [[1], [0, 0], [1, 1, 0], [0, 1, 0, 1], [0, 1, 1, 1, 1], [0]]]
[[[0], [1, 0], [1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0, 0], [0]], [[1], [0, 0], [1, 0, 0], [1, 0, 1, 1], [1, 1, 0, 1, 0], [0]], [[0], [0, 0], [1, 1, 1], [0, 1, 0, 1], [0, 1, 1, 0, 0], [0]]]
[[[0], [0, 1], [0, 0, 0], [1, 0, 1, 0], [0, 1, 1, 0, 1], [1]], [[0], [1, 1], [0, 0, 0], [1, 0, 1, 0], [1, 0, 1, 1, 1], [1]], [[0], [1, 1], [1, 1, 1], [1, 1, 0, 1], [0, 1, 1, 0, 1], [0]]]
[[[1], [0, 1], [1, 1, 0], [0, 0, 0, 1], [0, 1, 0, 1, 1], [1]], [[0], [1, 1], [1, 1, 0], [0, 0, 1, 0], [1, 0, 1, 1, 1], [0]], [[1], [1, 1], [1, 0, 0], [1, 1, 1, 0], [0, 0, 1, 1, 0], [0]]]
[[[1], [0, 0], [0, 0, 1], [0, 0, 0, 1], [0, 1, 1, 0, 0], [0]], [[0], [1, 0], [1, 1, 0], [1, 0, 1, 0], [1, 0, 1, 1, 1], [1]], [[0], [0, 0], [0, 0, 1], [1, 0, 1, 0], [0, 1, 1, 0, 0], [0]]]
[[[0], [1, 0], [0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0, 0], [1]], [[0], [0, 0], [0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 1, 0], [1]], [[0], [1, 0], [0, 0, 1], [1, 0, 0, 0], [1, 1, 1, 1, 0], [0]]]
[[[1], [1, 1], [0, 1, 0], [1, 0, 1, 0], [1, 1, 1, 1, 0], [1]], [[0], [0, 1], [1, 1, 1], [1, 0, 0, 1], [1, 0, 1, 1, 0], [1]], [[1], [0, 0], [1, 0, 1], [0, 1, 1, 0], [0, 1, 1, 0, 0], [1]]]
[[[0], [0, 0], [0, 0, 0], [0, 1, 1, 0], [0, 0, 0, 1, 1], [0]], [[1], [1, 0], [1, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1, 1], [1]], [[1], [0, 1], [1, 1, 0], [0, 0, 0, 1], [0, 0, 0, 0, 1], [1]]]
[[[0], [1, 1], [0, 0, 1], [0, 0, 0, 0], [1, 0, 1, 1, 0], [0]], [[0], [0, 0], [1, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0, 0], [0]], [[0], [0, 0], [1, 1, 0], [1, 0, 1, 1], [1, 1, 1, 0, 0], [0]]]
```

Figure A.1: Binary Representations of Convolutional Neural Network Architectures Discovered via the EMONAS-BOO

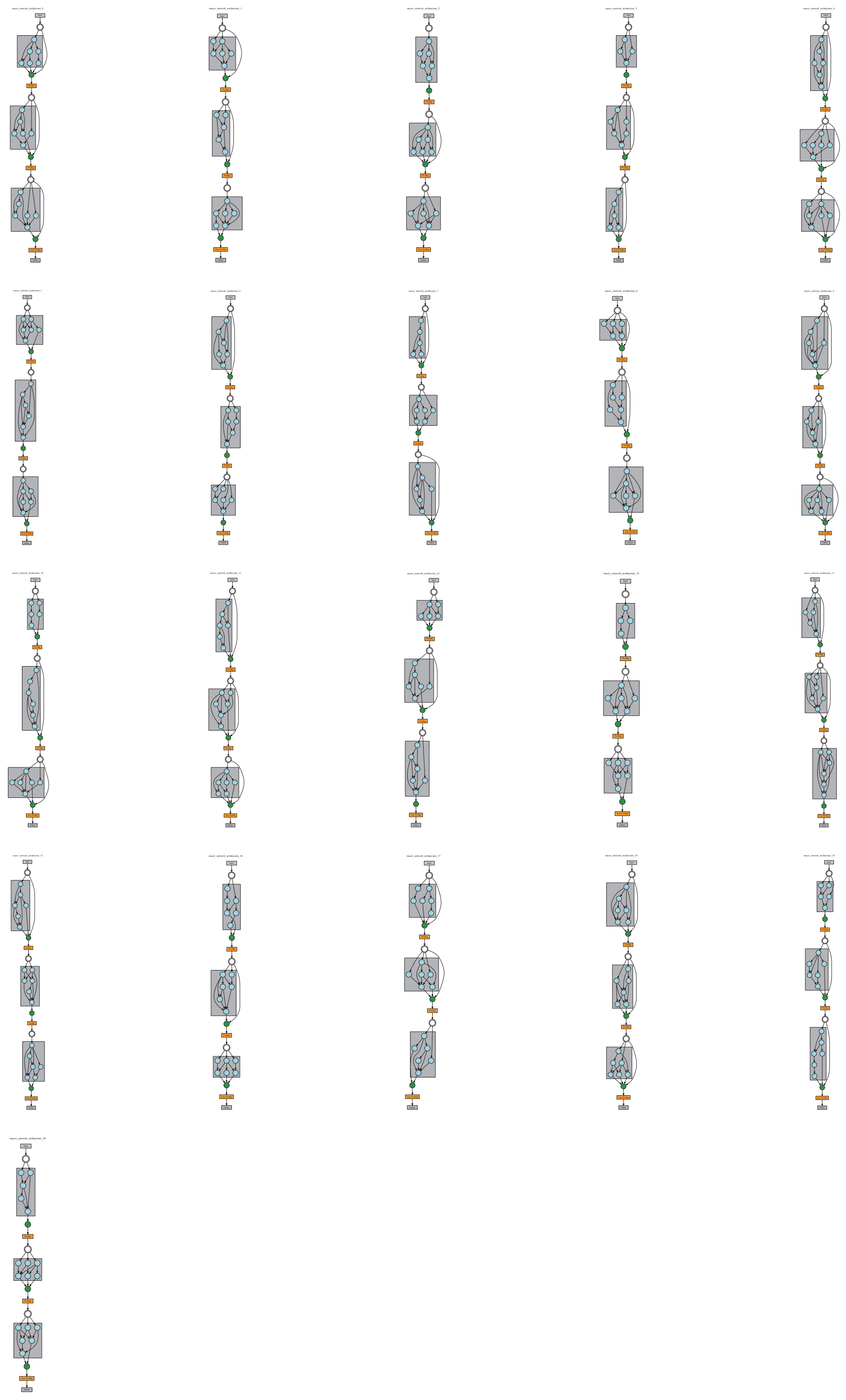


Figure A.2: Visual Representations of Convolutional Neural Network Architectures Discovered via the EMONAS-BOO

Table A.1: Fitness Scores Derived from EMONAS-BOO Balancing Accuracy and an Innovative Objective in CNNs

Accuracy	**BOO**
86.550000	5702587
86.090000	5398006
86.160000	5721496
86.510000	4895424
86.440000	5731408
85.940000	5699303
86.100000	5441000
85.810000	5746145
86.160000	5757590
86.130000	5614004
85.160000	5541817
85.750000	5724581
87.290000	5644822
86.230000	5657594
86.790000	5893162
86.060000	5766483
85.790000	5641731
87.320000	5558098
86.530000	5899494
86.770000	5416690
86.240000	5780909

The following 20 binary strings, ranging from architecture 0 to 20, are more clear binary representations of CNN architectures of the Figure A.1:

$$\begin{aligned}&\text{macro\_network\_architecture\_0} =\\&\quad [[[1],[0,1],[1,1,0],[1,0,0,0],[0,0,0,0,1],[1]],\\&\quad [[1],[0,0],[1,1,0],[1,1,0,0],[0,0,1,1,0],[1]],\\&\quad [[0],[0,1],[0,1,1],[0,0,0,0],[1,1,0,1,1],[1]]]\end{aligned}$$

$$\text{macro\_network\_architecture\_1} =$$
$$[[[1],[0,0],[1,0,0],[1,0,1,0],[0,1,0,1,1],[1]],$$
$$[[0],[1,1],[0,1,1],[0,0,0,0],[0,0,1,1,0],[1]],$$
$$[[1],[0,1],[1,0,0],[1,0,0,0],[1,1,0,1,1],[0]]]$$

$$\text{macro\_network\_architecture\_2} =$$
$$[[[1],[1,0],[0,1,1],[1,0,1,0],[0,0,0,1,1],[0]],$$
$$[[1],[1,0],[0,1,1],[1,0,1,0],[1,1,0,0,0],[1]],$$
$$[[0],[1,0],[1,0,0],[1,1,0,1],[1,1,1,0,0],[0]]]$$

$$\text{macro\_network\_architecture\_3} =$$
$$[[[0],[1,0],[0,0,0],[1,1,1,0],[0,0,0,0,0],[0]],$$
$$[[0],[0,1],[1,0,0],[1,0,0,1],[1,0,1,0,1],[1]],$$
$$[[1],[0,0],[0,1,0],[0,1,0,1],[1,0,0,1,0],[1]]]$$

$$\text{macro\_network\_architecture\_4} =$$
$$[[[1],[1,0],[1,0,1],[0,1,0,1],[0,1,0,1,1],[1]],$$
$$[[0],[0,0],[0,1,0],[1,0,1,1],[0,1,0,0,0],[1]],$$
$$[[0],[0,1],[1,1,0],[0,1,0,0],[1,1,0,1,0],[1]]]$$

$$\begin{aligned}&\text{macro\_network\_architecture\_5} =\\&\quad [[[0],[1,1],[0,1,0],[1,1,0,0],[1,0,1,0,1],[0]],\\&\quad [[1],[0,1],[1,0,1],[0,1,1,1],[1,1,0,0,1],[0]],\\&\quad [[1],[1,0],[0,1,0],[0,1,1,0],[1,1,1,1,0],[0]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_6} =\\&\quad [[[1],[1,1],[1,0,1],[0,1,0,0],[0,1,0,1,1],[1]],\\&\quad [[1],[0,0],[0,0,1],[0,1,0,1],[0,0,1,1,1],[0]],\\&\quad [[0],[0,0],[0,0,0],[1,1,0,0],[0,0,1,1,1],[0]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_7} =\\&\quad [[[1],[0,1],[0,1,1],[0,0,0,0],[0,0,1,0,0],[1]],\\&\quad [[1],[1,0],[0,0,0],[1,0,1,0],[0,1,0,1,0],[0]],\\&\quad [[1],[0,1],[1,1,0],[0,1,0,1],[1,1,0,0,1],[1]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_8} =\\&\quad [[[0],[0,0],[0,0,0],[1,0,1,0],[1,0,0,1,0],[1]],\\&\quad [[1],[0,0],[0,1,1],[1,0,0,0],[0,0,0,1,1],[1]],\\&\quad [[1],[0,1],[1,1,0],[1,1,1,1],[1,1,0,0,0],[0]]]\end{aligned}$$

$$\text{macro\_network\_architecture\_9} = \\ [[[0],[0,1],[0,0,1],[1,0,1,1],[1,1,1,1,1],[1]], \\ [[1],[0,0],[0,0,0],[1,1,0,1],[0,1,0,1,1],[1]], \\ [[1],[1,0],[1,0,0],[1,1,0,0],[1,1,0,1,0],[1]]]$$

$$\text{macro\_network\_architecture\_10} = \\ [[[0],[1,0],[0,0,1],[0,1,0,0],[0,0,0,0,0],[0]], \\ [[1],[0,1],[0,0,1],[0,0,1,1],[1,0,1,0,1],[1]], \\ [[0],[1,0],[1,0,0],[1,1,1,1],[1,0,0,0,0],[1]]]$$

$$\text{macro\_network\_architecture\_11} = \\ [[[1],[1,1],[0,1,0],[0,0,0,1],[0,0,0,1,1],[1]], \\ [[0],[1,0],[1,1,0],[1,1,1,0],[1,0,0,0,1],[1]], \\ [[1],[1,0],[1,1,0],[1,0,0,0],[0,1,0,0,1],[1]]]$$

$$\text{macro\_network\_architecture\_12} = \\ [[[0],[0,1],[0,1,0],[0,0,0,0],[1,1,0,0,0],[0]], \\ [[1],[1,1],[0,1,0],[0,0,0,0],[0,1,1,0,1],[1]], \\ [[1],[0,0],[1,1,0],[0,1,0,1],[0,1,1,1,1],[0]]]$$

$$\begin{aligned}&\text{macro\_network\_architecture\_13} = \\ &\quad [[[0], [1,0], [1,0,0], [0,0,1,0], [0,0,0,0,0], [0]], \\ &\quad [[1], [0,0], [1,0,0], [1,0,1,1], [1,1,0,1,0], [0]], \\ &\quad [[0], [0,0], [1,1,1], [0,1,0,1], [0,1,1,0,0], [0]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_14} = \\ &\quad [[[0], [0,1], [0,0,0], [1,0,1,0], [0,1,1,0,1], [1]], \\ &\quad [[0], [1,1], [0,0,0], [1,0,1,0], [1,0,1,1,1], [1]], \\ &\quad [[0], [1,1], [1,1,1], [1,1,0,1], [0,1,1,0,1], [0]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_15} = \\ &\quad [[[1], [0,1], [1,1,0], [0,0,0,1], [0,1,0,1,1], [1]], \\ &\quad [[0], [1,1], [1,1,0], [0,0,1,0], [1,0,1,1,1], [0]], \\ &\quad [[1], [1,1], [1,0,0], [1,1,1,0], [0,0,1,1,0], [0]]]\end{aligned}$$

$$\begin{aligned}&\text{macro\_network\_architecture\_16} = \\ &\quad [[[1], [0,0], [0,0,1], [0,0,0,1], [0,1,1,0,0], [0]], \\ &\quad [[0], [1,0], [1,1,0], [1,0,1,0], [1,0,1,1,1], [1]], \\ &\quad [[0], [0,0], [0,0,1], [1,0,1,0], [0,1,1,0,0], [0]]]\end{aligned}$$

$$\text{macro\_network\_architecture\_17} =$$
$$[[[0], [1, 0], [0, 0, 1], [1, 0, 0, 0], [0, 1, 0, 0, 0], [1]],$$
$$[[0], [0, 0], [0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 1, 0], [1]],$$
$$[[0], [1, 0], [0, 0, 1], [1, 0, 0, 0], [1, 1, 1, 1, 0], [0]]]$$

$$\text{macro\_network\_architecture\_18} =$$
$$[[[1], [1, 1], [0, 1, 0], [1, 0, 1, 0], [1, 1, 1, 1, 0], [1]],$$
$$[[0], [0, 1], [1, 1, 1], [1, 0, 0, 1], [1, 0, 1, 1, 0], [1]],$$
$$[[1], [0, 0], [1, 0, 1], [0, 1, 1, 0], [0, 1, 1, 0, 0], [1]]]$$

$$\text{macro\_network\_architecture\_19} =$$
$$[[[0], [0, 0], [0, 0, 0], [0, 1, 1, 0], [0, 0, 0, 1, 1], [0]],$$
$$[[1], [1, 0], [1, 1, 0], [1, 0, 0, 0], [0, 0, 0, 1, 1], [1]],$$
$$[[1], [0, 1], [1, 1, 0], [0, 0, 0, 1], [0, 0, 0, 0, 1], [1]]]$$

$$\text{macro\_network\_architecture\_20} =$$
$$[[[0], [1, 1], [0, 0, 1], [0, 0, 0, 0], [1, 0, 1, 1, 0], [0]],$$
$$[[0], [0, 0], [1, 1, 1], [0, 1, 0, 0], [1, 1, 0, 0, 0], [0]],$$
$$[[0], [0, 0], [1, 1, 0], [1, 0, 1, 1], [1, 1, 1, 0, 0], [0]]]$$

A.2 Selection Listing

Listing A.1: Python code of mutation

```
def emonas(
        pop_size=100,
        ...
        ...
        selection=TournamentSelection(
        func_comp=binary_tournament),
        crossover=PointCrossover(
        n_points=2),
        mutation=PolynomialMutation(
        eta=3, var_type=np.int),
        ...
        ...
        **kwargs):
        """
        eta: a parameter that controls the
        amount of perturbation, and
        a large value of eta
        results in a
        small perturbation,
        and vice versa.

        var_type: the data type of
        the variables. In this scenario,
        It is an integer because it
        represents the block or
        node connections
        or the number of them.

        """
```

A.3 Mutation Listing

Listing A.2: PolynomialMutation

```
class PolynomialMutation(Mutation):
    def __init__(self, eta, prob=None, var_type=np.double):
        ...
        ...
        ...
```

```
def _do(self, problem, pop, **kwargs):

    ...
    ...

    do_mutation = random.random(X.shape) < self.prob

    Y[:, :] = X

    xl = np.repeat(problem.xl[None, :],
    X.shape[0], axis=0)[do_mutation]
    xu = np.repeat(problem.xu[None, :],
    X.shape[0], axis=0)[do_mutation]

    if self.var_type == np.int:
        xl -= 0.5
        xu += (0.5 - 1e-16)

    X = X[do_mutation]
    # mutated
    delta1 = (X - xl) / (xu - xl)
    delta2 = (xu - X) / (xu - xl)

    mut_pow = 1.0 / (self.eta + 1.0)

    rand = random.random(X.shape)
    mask = rand <= 0.5
    mask_not = np.logical_not(mask)

    deltaq = np.zeros(X.shape)

    xy = 1.0 - delta1
    val = 2.0 * rand + (1.0 - 2.0 * rand)
    * (np.power(xy, (self.eta + 1.0)))
    d = np.power(val, mut_pow) - 1.0
    deltaq[mask] = d[mask]

    xy = 1.0 - delta2
```

```
        val = 2.0 * (1.0 - rand) + 2.0
        * (rand - 0.5) * (np.power(xy, (self.eta + 1.0)))
        d = 1.0 - (np.power(val, mut_pow))
        deltaq[mask_not] = d[mask_not]

        # mutated values
        _Y = X + deltaq * (xu - xl)

        ...
        ...

        return off
```

A.4 Crossover Listing

Listing A.3: crossover from pymoo packages

```
import numpy as np

from pymoo.model.crossover import Crossover
from pymoo.operators.crossover.util import crossover_mask
from pymoo.rand import random

class PointCrossover(Crossover):

    def __init__(self, n_points):
        super().__init__(2, 2)
        self.n_points = n_points

    def _do(self, problem, pop, parents, **kwargs):
        X = pop.get("X")[parents.T]
        _, n_matings, n_var = X.shape
        r = np.row_stack([random.perm(n_var-1) + 1
        for _ in range(n_matings)])[:, :self.n_points]
        r.sort(axis=1)
        M = np.full((n_matings, n_var), False)
        for i in range(n_matings):
            j = 0
            while j < r.shape[1] - 1:
                a, b = r[i, j], r[i, j + 1]
                M[i, a:b] = True
                j += 2

        _X = crossover_mask(X, M)
        return pop.new("X", _X)
```

Listing A.4: crossover_mask representation of crossover

```
M = np.full((n_matings, n_var), False)
for i in range(n_matings):

    j = 0
    while j < r.shape[1] - 1:
        a, b = r[i, j], r[i, j + 1]
        M[i, a:b] = True
        j += 2

_X = crossover_mask(X, M)
return pop.new("X", _X)
```

Listing A.5: simple crossover implementaion

```
def main():
    from pymoo.model.population import Population

    n_var = 5
    bounds = [(0, 10), (0, 10), (0, 10), (0, 10), (0, 10)]

    pop = Population(n_individuals=10)
    pop.set('X', np.round(np.random.uniform(0, 10, (10, n_var))))

    print(pop.get('X'))
    point_crossover = PointCrossover(1)
    parents = np.array([[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]])
    offspring = point_crossover.do(problem=None,
    pop=pop, parents=parents)
    print(offspring.get('X'))
```

```
if __name__ == '__main__':
    main()
```

Listing A.6: simple crossover implementation example

```
Population before crossover:
[[ 6.  6.  5.  5. 10.]
 [ 8.  7. 10.  3.  5.]
 [10.  1.  9.  3.  8.]
 [ 6.  1.  1.  1.  6.]
 [ 3.  6.  3.  5.  4.]
 [ 0.  4.  7.  3.  7.]
 [ 3.  1.  5.  6. 10.]
 [ 9.  5.  4.  4.  6.]
 [ 6.  1.  1.  7.  8.]
 [ 3.  8.  7.  9.  7.]]

Population after crossover:
[[ 6.  6. 10.  3.  5.]
 [ 8.  7.  5.  5. 10.]
 [10.  1.  1.  1.  6.]
 [ 6.  1.  9.  3.  8.]
 [ 3.  4.  7.  3.  7.]
 [ 0.  6.  3.  5.  4.]
 [ 3.  1.  5.  6.  6.]
 [ 9.  5.  4.  4. 10.]
 [ 6.  8.  7.  9.  7.]
 [ 3.  1.  1.  7.  8.]],

 where the masks looks like
 [[F, F, T, T, T]
 [F, F, T, T, T]
 [F, F, T, T, T]
 [F, F, T, T, T]
 [F, T, T, T, T]
 [F, T, T, T, T]
 [F, F, F, F, T]
 [F, F, F, F, T]
 [F, T, T, T, T]
 [F, T, T, T, T]]
```

A.5 Multi-objective Optimization Listing

Listing A.7: Python psudocode of MOO

```
def _evaluate(self, x, out, *args, **kwargs):
    objs = np.full((x.shape[0], self.n_obj), np.nan)
    for i in range(x.shape[0]):
        arch_id = self._n_evaluated + 1
        performance = train_search.main(......)
        objs[i, 0] = 100 - performance['valid_acc']
        objs[i, 1] = performance['BOO']
        self._n_evaluated += 1
    out["F"] = objs

def do_every_generations(algorithm):
    gen = algorithm.n_gen
    pop_var = algorithm.pop.get("X")
    pop_obj = algorithm.pop.get("F")


def main():
    ...
    ...
    problem = ...
    method = ...
    res = minimize(problem,
                   method,
                   callback=do_every_generations,
                   termination=('n_gen', args.n_gens),
                   verbose=True)

    logging.info("Architecture ,res.X={}".format(res.X))
    logging.info("MOO, res.F={}".format(res.F))
    ...
    ...
    return
```

A.6 BOO Listing

Listing A.8: BOO definition

```
def bin2float(b):
    h = int(b, 2).to_bytes(8, byteorder="big")
    return struct.unpack('>d', h)[0]

def float2bin(f):
    [d] = struct.unpack(">Q", struct.pack(">d", f))
    return f'{d:064b}'

def number_of_ones(n):
      one_count = 0
      for i in n:
          if i == "1":
              one_count+=1
      return one_count

def counter_ones(out):
    array_out= out.cpu().detach().numpy()
    array_out_tobytes= array_out.tobytes()
    hex_array_out_tobytes = bitstring.BitArray(array_out_tobytes)
    hex_array_out_tobytes_in_binary = hex_array_out_tobytes.bin
    ones_out2 = number_of_ones(hex_array_out_tobytes_in_binary)
    # number of ones: ones_out2
    efficiency2 = ones_out2 / len(hex_array_out_tobytes_in_binary)
    # ones / the total bits: efficiency
    return efficiency2


def counter_ones_for_params(out):
    random_tensor = torch.randn(len(out))
    for i in range(len(out)):
        random_tensor[i] = counter_ones(out[i])
    return random_tensor
```

```
def number_of_ones(n):
    one_count = 0
    for i in n:
        if i == "1":
            one_count+=1
    return one_count


def counter_ones_version_2(out):
    array_out= out.cpu().detach().numpy()
    # array_out= out.detach().numpy()
    array_out_tobytes= array_out.tobytes()
    hex_array_out_tobytes = bitstring.BitArray(array_out_tobytes)
    hex_array_out_tobytes_in_binary = hex_array_out_tobytes.bin
    ones_out2 = number_of_ones(hex_array_out_tobytes_in_binary)
    # number of ones: ones_out2
    length_out2 = len(hex_array_out_tobytes_in_binary)
    # number of the whole string: ones_out2
    ones_and_length = torch.tensor([ones_out2, length_out2])
    return ones_and_length



def counter_ones_version_3(out):
    array_out= out.cpu().detach().numpy()
    # array_out= out.detach().numpy()
    array_out_tobytes= array_out.tobytes()
    hex_array_out_tobytes = bitstring.BitArray(array_out_tobytes)
    hex_array_out_tobytes_in_binary = hex_array_out_tobytes.bin
    ones_out2 = number_of_ones(hex_array_out_tobytes_in_binary)
    return ones_out2

def counter_ones_for_params_version_3(out):
    random_tensor_for_ones = torch.randn(len(out))
    for i in range(len(out)):
        random_tensor_for_ones[i] = counter_ones_version_2(out[i]
        summation_of_ones = torch.sum(random_tensor_for_ones)
    return summation_of_ones
```