AUTOMATIC DESIGN OF DEEP NEURAL NETWORK ARCHITECTURES WITH EVOLUTIONARY COMPUTATION

By

FRANCISCO ERIVALDO FERNANDES JUNIOR

Technologist in Industrial Mechatronics
Federal Institute of Education, Science and Technology
of Ceará (IFCE)
Fortaleza, Ceará, Brazil
2012

Master of Science in Electrical Engineering
Campinas State University (UNICAMP)
Campinas, São Paulo, Brazil
2014

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 2020

AUTOMATIC DESIGN OF DEEP NEURAL NETWORK ARCHITECTURES WITH EVOLUTIONARY COMPUTATION

Dissertation Approved:

Dr. Gary G. Yen

Dissertation Advisor

Dr. Martin Thomas Hagan

Dr. Sabit Ekin

Dr. Christopher John Crick

ACKNOWLEDGMENTS

I want to express my gratitude to my advisor, Dr. Gary G. Yen, for helping me grow as a researcher. Dr. Yen's advice helped improve my problem solving and communication skills through the course of my Ph.D. He was also available to help during the most problematic and challenging stages of my research; thus, I cannot thank him enough for his support.

I also want to thank Drs. Martin M. Hagan and Cristopher J. Crick for teaching me everything I know about Machine Learning and Deep Neural Networks which were used extensively in my research. Additionally, I want to thank Dr. Sabit Ekin for providing essential feedback and Dr. Jean Van Delinder for helping me solve bureaucratic problems at the administration level.

I would not be able to come to the U.S. and perform this research without the money invested by the Brazilian government through the Brazilian National Council for Scientific and Technological Development (CNPq) agency, which paid for my stipends, tuition, and fees during my Ph.D. degree (CNPq grant 203076/2015-0). Thank you for investing in the education of our country. The quality of life of our population can only improve with free public and universal access to education.

For last but not least, I want to express my deepest gratitude to my wife, Mariana de Carvalho, for providing me with all kinds of support during this stage of my life. You are my greatest accomplishment. Thank you!

Francisco Erivaldo Fernandes Junior[1]

[1]Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: FRANCISCO ERIVALDO FERNANDES JUNIOR
Date of Degree: MAY, 2020
Title of Study: AUTOMATIC DESIGN OF DEEP NEURAL NETWORK ARCHITECTURES WITH EVOLUTIONARY COMPUTATION

Major Field: ELECTRICAL ENGINEERING

Abstract: Deep Neural Networks (DNNs) are algorithms with widespread use in the extraction of knowledge from raw data. DNNs are used to solve problems in the fields of computer vision, natural language understanding, signal processing, and others. DNNs are state-of-the-art machine learning models capable of achieving better results than humans in many tasks. However, their application in fields outside computer science and engineering has been hindered due to the tedious process of trial and error multiple computationally intensive models. Thus, the development of algorithms that could allow for the automatic development of DNNs would further advance the field. Two central problems need to be addressed to allow the automatic design of DNN models: generation and pruning. The automatic generation of DNN architectures would allow for the creation of state-of-the-art models without relying on knowledge from human experts. In contrast, the automatic pruning of DNN architectures would reduce the computational complexity of such models for use in less powerful hardware. The generation and pruning of DNN models can be seen as a combinatorial optimization problem, which can be solved with the tools from the Evolutionary Computation (EC) field. This Ph.D. work proposes the use of Particle Swarm Optimization (PSO) for DNN architecture searching with competitive results and fast convergence, called psoCNN. Another algorithm based on Evolution Strategy (ES) is used for the pruning of DNN architectures, called DeepPruningES. The proposed psoCNN algorithm is capable of finding CNN architectures, a particular type of DNN, for image classification tasks with comparable results to human-crafted DNN models. Likewise, the DeepPruningES algorithm is capable of reducing the number of floating operations of a given DNN model up to 80%, and it uses the principles of Multi-Criteria Decision Making (MCDM) to output three pruned model with different trade-offs between computational complexity and classification accuracy. These ideas are then applied to the creation of a unified framework for searching highly accurate, and compact DNN applied for Medical Imaging Diagnostics, and the pruning of Generative Adversarial Networks (GANs) for Medical Imaging Synthesis with competitive results.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

Deep Neural Networks (DNNs) are powerful models used to solve a variety of real-world problems, such as computer vision, natural language understanding, and signal processing. However, their design and deployment are not trivial undertakings. In the present work, the task of designing accurate and compact DNN architectures is considered as a combinatorial optimization problem, and the developments of specific algorithms to perform such task automatically are proposed. These algorithms are inspired by the field of Evolutionary Computation (EC), which has been used successfully for many years to solve a variety of optimization problems employing little to no prior knowledge about the problem in question. The motivation, problem statement, contributions, and organization of the proposed work are presented in this chapter.

1.1 Motivation

Humanity has been dreaming of building intelligent machines for many decades or even centuries now. The development of robots and computers are the materialization of such dreams. Even before the advent of the discipline of Artificial Intelligence (AI), Alan Turing, in the 1950s, designed a test to verify if a machine could be called intelligent or not, commonly known as the *Turing Test* [1]. Russell et al. [2] argue that a machine needs to be proficient in four areas to be able to pass the *Turing Test*: *Natural Language Processing*, *Knowledge Representation*, *Automated Reasoning*, and

Table 1.1: Artificial intelligence sub-disciplines, according to [2].

Sub-discipline	Description
Natural Language Processing	Studies the human communication through the use of language, both oral and written
Knowledge Representation	Studies how to store and retrieve previously learned knowledge
Automated Reasoning	Studies how a machine should act based on its internal knowledge and its sensors
Machine Learning	Studies how to use the results from past data to extrapolate results on new data

Machine Learning. Interestingly enough, these are all parts of the AI discipline, which has been studying how intelligence works and how to replicate it on machines since its inception in 1956 [2]. Table 1.1 sums up what each of these disciplines studies — the field of *Natural Language Processing* studies how humans use language to communicate and transmit knowledge; the field of *Knowledge Representation* studies how to store and represent learned knowledge efficiently; the field of *Automated Reasoning* studies how to reasoning using stored knowledge. For last, the field of *Machine Learning* studies how to use data to construct and extrapolate knowledge to unseen situations.

No machine or algorithm developed so far has ever passed the *Turing Test*. However, many advancements were made in the Machine Learning (ML) field along the way. The objective of an ML model is to find some structure or pattern in a given set of data [3]. This process is also known as *data fitting*, where the model tries to find a function that explains the data, which can be later used to extrapolate results from new data. This data fitting procedure is also called *adaptation*, *learning*, or *training*. A Machine Learning model can learn through the use of three learning paradigms:

Supervised, *Unsupervised*, and *Reinforced*. In Supervised Learning, the model learns by observing data containing pairs of inputs and outputs, known as labeled data. While in Unsupervised Learning, the model needs to separate the data in clusters, and the training data contain only inputs, known as unlabeled data. For last, Reinforced Learning works as a mix of Supervised and Unsupervised Learning, and the model is awarded for every correct prediction or action executed on a set of inputs. Table 1.2 sums up the types of data used in each learning paradigm.

Currently, the most popular family of ML models used to extract meaningful information from raw data are Deep Neural Networks (DNNs), which are part of the Deep Learning (DL) field. There are two categories of DNNs: feed-forward and recurrent. In feed-forward DNNs, the information flows in one direction from the beginning to the end of the DNN architecture. While in recurrent DNNs, the flow of information from one part of the network can be fed back to previous nodes creating a network with memory capabilities. There are multiples and different DNN architectures in each of these categories used to solve specific problems. For example, feed-forward DNNs are suitable to solve most of the problems related to the field of computer vision, while recurrent DNNs are suitable for use with natural language understanding and dynamics problems. Fully-Connected, Convolutional [4], Residual [5], and Dense [6] Neural Networks are typical examples of feed-forward DNN architectures, while Long-Short-Term Memory (LSTM) [7], Gated Recurrent Unit (GRU) [8], and Hopfield [9] Neural Networks are typical examples of recurrent DNNs. Because DNNs are highly adaptable models, among other tasks, they can be used to detect and classify objects in images and videos [10, 11], to translate texts and voice between different languages automatically [12, 13], and to generate text and speech [14, 15, 16]. The types of DNNs and their uses are summarized in Table 1.3. However, to craft a DNN model, one must take into account a series of unclear guidelines.

Table 1.2: Types of data used by ML algorithms, according to [3].

Learning type	Data description
Supervised Learning	Data contains both inputs and outputs
Unsupervised Learning	Data contains only inputs
Reinforced Learning	Data contains rewards for correct executed actions

First, the type and architecture of a DNN model are dependent on the type of data available for training. If the data available are time-dependent, then recurrent DNNs are the most suitable for the task. If the data are composed of images, then feed-forward DNNs with convolutional layers, to be detailed in Chapter II, are the best to exploit. Moreover, the number of parameters of a DNN model is also dependent on the complexity of the data available, with small DNN models being able to learn only simple data.

Second, the adaptation procedure of a DNN depends on the data being labeled or unlabeled. Similar to other ML model, a DNN needs to adjust its internal parameters interactively through a training process. Thus, DNNs can be supervised trained if the available data is labeled, or unsupervised trained if the available data is unlabeled.

Third, a massive amount of data needs to be gathered. Given a sufficiently large model and amounts of data, DNNs can learn any function, and, because of this, they are known as universal approximators [17]. However, if the available training data is not large enough, this also means that a DNN can easily overfit and memorize it instead of learning from it. Thus, DNNs must be trained in databases containing tens of thousands or even millions of samples.

Fourth, in order to train, or even deploy a trained DNN model, a large amount of computing power is needed. For example, the most successful DNN models used

Table 1.3: Deep neural networks types.

DNN Category	Architectures	Most used for
Feed-forward	Fully-connected, Convolutional [4], Residual [5], Dense [6], etc.	Computer vision problems [10, 11]
Recurrent	LSTMs [7], GRUs [8], Hopfield [9], etc.	Natural language understanding and dynamics problems [12, 13, 14, 15, 16]

in image classification tasks can take months to train on Central Processing Units (CPUs). Hence, in recent years, the development of General Purpose Graphical Processing Units (GPGPUs) was key to the revitalization of the Artificial Neural Networks field in the form of Deep Learning (DL). Although the basic theory used in today's DNNs was first developed in the 1980s [18, 4], only recently the amount of data and computational power, due to the advent of Big Data [19] and GPGPUs [20], were finally enough to allow the development of very deep neural networks capable of surpassing human beings in image classification tasks [21, 22].

For last, there are still other factors that one must take into consideration when crafting a DNN model. For example, the creators of the VGG networks [23], a very successful DNN model, showed that the use of small convolutional filters in DNNs for image classification achieved better results than DNNs with large filters. Likewise, Szegedy *et al.* [24] observed that the use of small networks inside a large one, a concept called *network-in-network* [25], produced better results in image classification tasks than those with simple DNN architectures. He *et al.* [5] showed that a DNN model could increase its classification performance if it was trained to learning a residual mapping instead of a direct mapping of the network's inputs and outputs. Huang *et*

al. [6] also increased the classification performance of DNN models by connecting any given convolution layer to all of its previous layers. Thus, even though DNNs can be used to solve a myriad of real-world problems, their development and deployment are based on trial and error with no standardized methodology available, as illustrated in Figure 1.1.

On the other hand, Evolutionary Computation (EC) algorithms are mostly used to solve optimization problems. There are two types of approaches when dealing with optimization problems: derivative-based and derivative-free. Derivative-based algorithms use the gradient of a loss function to direct the search. By contrast, derivative-free algorithms are ideal for problems where the derivatives are too expensive to obtain, or they cannot be computed, such as discrete and combinatorial optimization problems. Table 1.4 shows some examples of derivative-based and derivative-free algorithms. Steepest descent, Newton's method, and Levenberg-Marquardt are some examples of derivative-based algorithms. On the other hand, EC algorithms, simulated annealing, random search, and downhill simplex search are some examples of derivative-free algorithms [26]. EC algorithms are also referred to as meta-heuristic

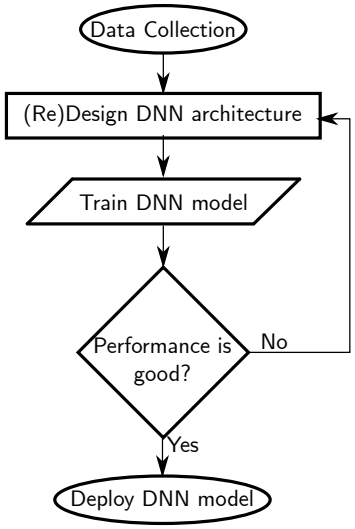


Figure 1.1: Process of creation and deployment of a DNN model.

Table 1.4: Types of optimization algorithms.

Type	Algorithm
Derivative-based	Steepest Descent [26]
	Newton's Method [26]
	Levenberg-Marquardt [27]
	Etc.
Derivative-free	Genetic Algorithms [28]
	Evolutionary Programming [28]
	Evolutionary Strategies [28]
	Simulated Annealing [26]
	Random search [26]
	Downhill Simplex Search [29]
	Etc.

algorithms because most of them are derivative-free algorithms created based on natural metaphors, such as evolution and animal behavior, instead of purely mathematical analysis. Similar to DNNs, EC algorithms were also created many years ago, but they still see substantial interest at present.

Three algorithms are considered the most known representatives of the field [28]: the Genetic Algorithm (GA) created by John Holland in the 1960s, the Evolutionary Programming (EP) created by Fogel, Owens, and Walsh in 1966, and the Evolutionary Strategies (ES) created by Rechenberg and Schwefel in 1965. The authors of EP and ES created these algorithms to solve specific design problems, while GAs were created to study the evolutionary and adaptation processes. More recently, algorithms based more on the behavior of animals than in evolution were also developed. Some examples are the Particle Swarm Optimization (PSO) algorithm created by Kennedy and Eberhart in 1995 [30, 31], the Ant Colony System (ACS) created by Dorigo *et al.*

[32], the Firefly algorithm created by Yang [33], the Fish School Search created by Bastos Filho *et al.* [34] and the Cuckoo Search created by Yang [35], as exemplified in Table 1.5.

As EC algorithms are used mostly to solve optimization problems, they can also be placed into two categories: algorithms to solve single-objective optimization problems (SOPs), and algorithms to solve multi-objective optimization problems (MOPs). Most of the previously cited algorithms are used to solve SOPs where the maximum or minimum value of only one single objective function is searched. In MOPs, two or more conflicting objective functions, where the improvement in one of the functions deteriorates some of the others, are optimized at the same time. Most of the previous algorithms can be enhanced to also work with MOPs.

EC algorithms have also been used to improve DNNs for a very long time now. In the 1990s, GAs were mainly used for training DNNs at a time when the available data were not large enough to avoid overfitting problems caused by the traditional use of gradient descent [36, 37, 38]. In the 2000s, EC algorithms were mainly used for DNN training and architecture searching, when the field of *NeuroEvolution* was established. One of the most important works from that time was the development of the *NeuroEvolution of Augmenting Topologies* (NEAT) by Stanley and Miikkulainen in 2002 [39, 40]. However, NEAT and others' algorithms from the 2000s cannot find massive DNN models with the same level of performance as the ones hand-crafted by human experts.

Although many improvements happened from the 1980s until now in the field of *Deep Learning* and *NeuroEvolution*, the development of DNN architectures to solve any given problem is still a complicated endeavor. Most of the algorithms for searching for DNN architectures as part of *NeuroEvolution* are still convoluted and

Table 1.5: Popular algorithms used in evolutionary computation.

Inspired by	Algorithm
Natural selection and evolution	Genetic Algorithms [28]
	Evolutionary Programming [28]
	Evolutionary Strategies [28]
	Differential Evolution [41]
	Etc.
Animal behavior	Particle Swarm Optimization [30, 31]
	Ant Colony System [32]
	Firefly Algorithm [33]
	Fish School Search [34]
	Cuckoo Search [35]
	Etc.

have a prohibitively computational cost for use by a person outside the field. Thus, in the present work, a series of algorithms are proposed to address the problem of automatically searching for DNN architectures for image classification tasks.

1.2 Problem Statement

Currently, data are everywhere and being continuously generated [19]. Ideally, researchers, experts, and professionals from all academic disciplines would like to extract meaningful knowledge from all these data automatically. As stated before, DNNs are the most prominent tool used for such tasks because they can find correlations in any data used to train them. One would expect that data with similar characteristics, such as images from dogs or cats, could be easily analyzed with a single DNN model or very similar models. In reality, various DNN models need to be devised even when dealing with similar data coming from different sources. For example, in the field of

image classification, there are many databases used to benchmark models. The most famous being the ImageNet [42], CIFAR10 [43], CIFAR100 [43], and MNIST [44] datasets. Some of them are easier to classify than others, but, even though they are all composed of digital images, to get useful results, one needs to use different DNN models for each of them. One can use a technique called transfer learning [45, 46] to avoid the need to design a new DNN from scratch for use with a different database than the one used to train the original DNN. With transfer learning, a DNN trained in a large dataset can be repurposed to work with a smaller and simpler dataset by retraining only the last few layers of the network in this new dataset.

However, transfer learning does not address the fact that the most successful DNN models found in the literature have a high computational complexity for both training and deployment. Indeed, model and data complexity are considered one of the biggest challenges of the DL field [47, 48]. For example, the VGG16, VGG19, and Inception, three DNNs with good results on ImageNet, contain a total of 138 million, 143 million, and 23 million parameters[1], respectively. These DNNs need to have millions of parameters because they were designed to classify the 3.2 million images found in the ImageNet dataset [42], and they can take weeks to finish one entire training session. Even though they can be repurposed to work with other datasets with the use of transfer learning, they will still contain roughly the same number of parameters. One approach that could be used to reduce the computational complexity of DNNs is the elimination of unnecessary or redundant parameters from their architectures. This elimination of redundant parameters is known as compression or pruning of DNN architectures [49, 50, 51, 52, 53, 54, 55, 56]. Nonetheless, the process of pruning a DNN architecture is usually an afterthought, and it can also increase the time required to implement DNN-based solutions.

[1]The number of parameters can be found in https://keras.io/applications/

Therefore, this Ph.D. work proposes the development of algorithms capable of searching for DNN architectures for image classification tasks that can find solutions with a good trade-off between performance and computational complexity for a given dataset automatically. The proposed algorithms empower researchers and professionals from all disciplines to unlock the potential of DL and DNNs in their future works allowing for reduced development time and higher research output. In this sense, the tasks of searching and pruning DNN architectures are seen as optimization problems, where the DNN architecture searching algorithm tries to find models with the best performance in a chosen dataset, while the DNN architecture pruning tries to find models with the smallest number of parameters [53, 56]. However, the performance of a DNN model and its computational complexity are two conflicting objectives. It is not possible to maintain a model's performance if too many parameters are removed. For this reason, the proposed work considers that the task of pruning DNN architectures is a multi-objective optimization problem, and also allows the possibility of using user preference to guide the pruning procedure.

Furthermore, at the time of writing, the present work is one of the first to propose such a unified framework for searching and pruning DNN architectures. Many works in the literature deal with network architecture searching and pruning separately, but there is no work where both are considered part of the same problem.

1.3 Contributions

The contributions of the present work are three-fold:

- First, the development of a novel DNN architecture search algorithm based on Particle Swarm Optimization is presented, called *psoCNN*. This algorithm can automatically find Convolutional Neural Networks (CNNs) architectures given

an image classification dataset. A novel encoding strategy and velocity computation are devised to deal with variable length individuals in the population. The results obtained are competitive with peer competitors.

- Second, the development of a novel DNN pruning algorithm based on Evolution Strategies (ES) is presented, called *DeepPruningES*, which can prune pre-trained CNNs, Residual Neural Networks, and Densely Connected Neural Networks architectures automatically. The algorithm uses the Minimum Manhattan Distance (MMD) approach to find the solution with the best trade-off between classification accuracy and computational complexity. It also finds two boundary solutions: one with the best classification accuracy and the other with the smallest computational complexity. With a total of three pruned solutions, the algorithm can help decision-makers to choose which one is the best to be used based on his or her needs.

- Third, a unified framework for DNN architecture searching and pruning is proposed, called *DNNDeepeningPruning*, where very deep DNN architectures are built from scratch as quickly as possible. Then, the newly found DNN is pruned to create a compact model. DNNs are built by adding residual blocks, one on the bottom of another. During pruning, models are selected based on the user preference of a small computational complexity or higher classification performance. The proposed framework is applied to search for meaningful DNN models used in medical imaging diagnostics applications with competitive results.

1.4 Dissertation Organization

The present document is organized into the following chapters:

- The basic background theories about Deep Neural Networks, Evolutionary Computation, and related DNN architecture searching and pruning algorithms are presented in Chapter II.

- The first contribution of this work is presented in detail in Chapter III, where the Particle Swarm Optimization algorithm is used for CNN architecture searching. It is also shown the results obtained when the algorithm is applied in a series of image classification databases.

- In Chapter IV, the proposed DNN architecture pruning algorithm based on Evolution Strategies is presented, as well as the results obtained in a series of state-of-the-art DNN models.

- In Chapter V, the proposed unified framework for DNN architecture searching and pruning is presented. This chapter also presents the medical imaging databases used to test the framework.

- In Chapter VI, Generative Adversarial Networks (GANs) used to generate x-ray images are pruned using the proposed DNN architecture pruning algorithm.

- Finally, the conclusions of the present work are drawn in Chapter VII.

CHAPTER II

RELATED WORKS

The basic theory behind deep neural networks (DNNs) and evolutionary computation (EC), as well as related works on DNN architecture searching and pruning algorithms, are presented in this chapter.

2.1 Deep Neural Networks

Most of the Deep Neural Networks (DNNs) designs used at present are the same ones used by the very first Artificial Neural Networks (ANNs). Thus, an overview of ANNs will be presented first and, then, it will be used to explain further how DNNs work.

2.1.1 Overview of Artificial Neural Networks

ANNs are composed of multiple computational nodes known as artificial neurons. The purpose of an artificial neuron is to compute a simple function of its inputs with its weights. In general, each neuron in an ANN produces only one single output. Neurons have two sets of parameters: weights and biases.

The basic artificial neuron has one single input and output, as illustrated in Figure 2.1. First, this single input neuron will perform a multiplication between its weight, w, and its input, x. Then, the bias, b, is added to produce n. For last, the neuron's output, a, is produced by inputting n to an activation function, $f(.)$. Usually, non-

linear functions are used as activation function, such as the hyperbolic tangent and the sigmoid. In DNNs, the rectified linear unit (ReLU) is the commonly employed activation function, and it is written as $f(n) = max(n, 0)$ [57]. Typically, a neuron's output can also be used as inputs to other neurons allowing multiple layers of neurons to be used in the building of a single ANN.

Although it is easier to understand the inner working of an artificial neuron by using a single input example, in general, most neurons used to solve real-world problems have multiple inputs, as illustrated in Figure 2.2. To facilitate computations when dealing with multiple inputs and a single neuron, the inputs, $\mathbf{x}$, and the neuron's weights, $\mathbf{w}$, can be seen as row and column vectors with the same dimension, respectively, and the scalar multiplication from before will become a vector multiplication.

In the present work, feed-forward ANNs with no feedback connections are used; one example is shown in Figure 2.3. In the field of Deep Learning (DL), this type

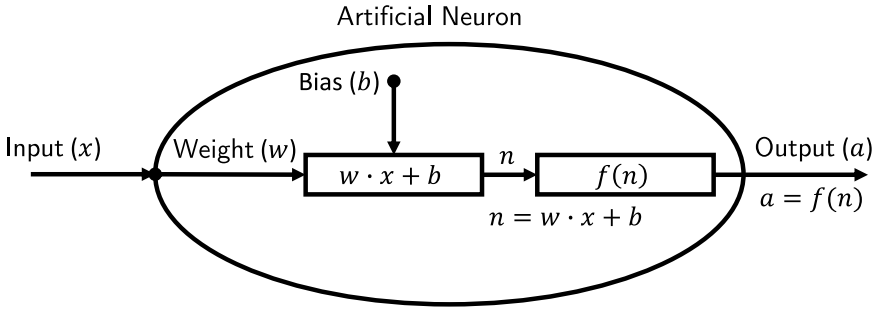


Figure 2.1: Artificial neuron with one input.

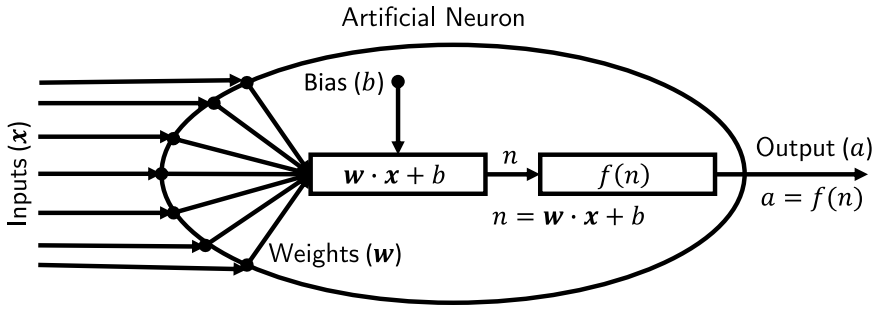


Figure 2.2: Artificial neuron with multiple inputs.

of ANN can also be used as a layer to DNNs, and it is commonly known as a *fully-connected* layer because each neuron from any given layer is connected to all neurons in the next layer. The ANN shown in Figure 2.3 has two layers: one hidden layer, the middle line of black nodes, and one output layer, the right-most black node. The left-most line of black nodes is called the input layer, and it is not used when counting the total number of layers in an ANN because its nodes do not perform any computation. The input layer only forwards the inputs to the first hidden layer. Moreover, an ANN can have an arbitrary number of hidden layers and output nodes.

The crucial advantage of stacking neurons is the fact that the weights of all neurons from a given layer can be combined into a single matrix, and the outputs of each layer can be treated as a vector. In Figure 2.3, the weights $\mathbf{W}_1$ and $\mathbf{W}_2$ are two different matrices. Thus, the output, y, of the ANN can be computed as follows:

$$y = f_2(f_1(\mathbf{x} \cdot \mathbf{W}_1 + \mathbf{b}_1) \cdot \mathbf{W}_2 + \mathbf{b}_2), \tag{2.1}$$

where $f_1(.)$ and $f_2(.)$ are the activation functions of the first and second layers, respectively, $\mathbf{b}_1$ and $\mathbf{b}_2$ are the bias vectors used by the neurons in the first and second layers, respectively, and $\mathbf{x}$ is the inputs of the network. The process of computing the output of an ANN or DNN from a set of inputs is called inference, and it is mainly performed once the network parameters are fully trained [58].

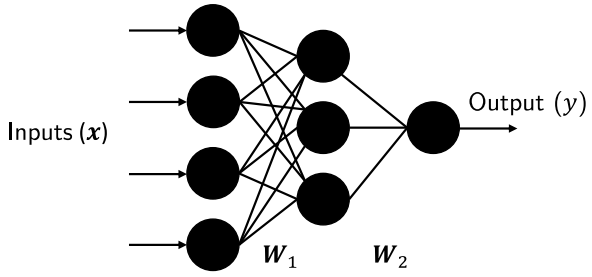


Figure 2.3: Feed-forward artificial neural network.

2.1.2 Training Procedure for ANNs and DNNs

The process of training unlocks one of the most compelling aspects of ANNs and DNNs, which is their high adaptability to data. The backpropagation algorithm is the most used one to train ANNs and DNNs because it can compute the gradients of the network parameters with respect to the network's output error.

The process of training is, in fact, an optimization problem. In supervised training, one must find a set of parameters that minimizes the error between the network predicted output and the correct one. Thus, it is natural that before the training can begin, one must choose a loss function meaningful to the problem at hand. In modern DNNs used for image/object classification, the activation function used in the output layer is the softmax, which outputs a probability distribution of all possible classes from a problem. The softmax is shown in Equation 2.2, and it uses the output of all nodes, n_j, in the last layer, from j equal to 1 to k, where k is the total number of classes in the problem, to determine the probability of the current output node, n_i. The output of a DNN used for classification is its confidence that a given input comes from a specific class. Thus, because the cross-entropy loss is the most suitable loss used when comparing two probability distributions, it is also the most suitable one for use in DNNs. This loss function is shown in Equation 2.3, where $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, ..., \hat{y}_k]^T$ is the ground truth, $\mathbf{y} = [y_1, y_2, ..., y_k]^T$ is the predicted output, and k is the number of classes.

$$y_i = f_i(\mathbf{n}) = \frac{e^{n_i}}{\sum_{j=1}^{k} e^{n_j}} \tag{2.2}$$

$$CrossEntropyLoss(\mathbf{y}) = -\sum_{i=1}^{k} \hat{y}_i \ln(y_i) \tag{2.3}$$

With the loss function defined, one can train an ANN or DNN by iteratively performing two steps: a forward step and a backward step. The forward step is performed by giving inputs and computing the network's losses, and no parameter is adjusted in this step. The backward step is performed by first computing gradients using backpropagation and, then, updating the network's parameters. The backpropagation algorithm uses the information from the forward step and the chain rule of Calculus to compute the gradients for all parameters in the network with respect to the loss function. The gradients are then used to update the network's parameters using some optimization algorithm. For example, one could use the gradient descent algorithm to update the parameters as follows:

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \alpha \nabla F(\mathbf{z}_{t-1}), \tag{2.4}$$

where $\mathbf{z}$ represents any parameter in the network: weights and biases; t represents the current time step, $t-1$ represents the previous time step; α is the learning rate, and $\nabla F(\mathbf{z}_{t-1})$ is the gradient of the current parameter $\mathbf{z}$ with respect to the network's loss. Other optimization algorithms can also be used with backpropagation, such as the *Adam* [59] and *AdaGrad* [60] optimizers.

2.1.3 DNN Architectures for Image Classification

In this section, first, the most common layers' type used in DNNs for computer vision tasks are presented, and, then, some specific examples of DNNs follows.

Modern DNNs are mainly composed of three types of layers: *convolutional*, *pooling*, and *fully-connected*. As the name suggests, convolutional layers perform a convolution operation between its weights, also called filters or kernels, and its inputs. The output of a convolutional layer is produced by sliding the filters over the input. The

filter will move over the input according to a parameter called *stride*, which controls how many positions the filter will move to produce the next output. The output of a convolution layer is shown in Equation 2.5, where i represents the current layer, $f(.)$ is the activation function of the current layer; $\mathbf{A}_i$, $\mathbf{W}_i$, and $\mathbf{X}$ are multi-dimensional matrices representing the output, the weights, and the input of the current layer, respectively, while $\mathbf{b}_i$ is a vector representing the biases of each filter in the current layer; $\circledast$ represents a convolutional operation. A simple convolutional operation is shown in Figure 2.4, where a filter of size 3×3 and strides 1×1 is convolved with a 6×6 input to produce a 4×4 output. Convolutional layers are frequently used due to their *weight sharing* nature, which allows the use of more compact networks when dealing with images [4]. They also extract features from images that are used to facilitate the work of classifying them by the final layers. Thus, convolutional layers are always used before any fully-connected layer in a DNN architecture.

$$\mathbf{A}_i = f(\mathbf{W}_i \circledast \mathbf{X} + \mathbf{b}_i) \tag{2.5}$$

Pooling layers are used for downsampling their inputs. This downsampling is done by using a sliding window over its input to produce an output by performing an average or a maximum operation with the elements inside the window. This procedure

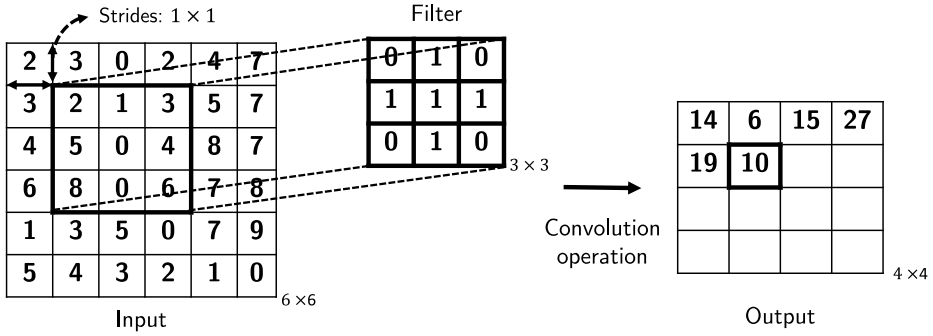


Figure 2.4: Example of a convolutional operation with a 3×3 filter and strides equal to 1×1.

is illustrated in Figure 2.5, where an input of size 8×8 is downsampled to 4×4 by using a maximum and an average pooling. This type of layer is commonly placed between convolutional layers and before the first fully-connected layer. This downsampling process also contributes to the creation of compact DNNs. When building a DNN architecture, the first few layers have higher output dimensions than the ones from the final layers.

Fully-connected layers used in DNNs are just simple ANNs similar to the ones explained before. They are also called classifier layers because they take the features extracted by a sequence of convolutional and pooling layers and determine in which class the input belongs.

Modern DNN architectures used for image classification were created to compete in the ImageNet challenge, where the network needs to be able to correctly classify three million color images with an average resolution of 400×350 pixels in one of the one thousand possible classes [42]. The entire ImageNet database has a size of 140 gigabytes. To solve such a complex problem, researchers have designed many different DNN architectures. Currently, the most successful DNNs in the ImageNet challenge are the VGGs, Residual (ResNets), and Densely Connected (DenseNets)

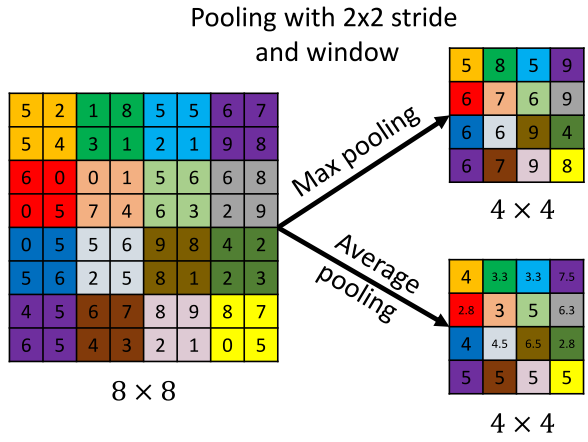


Figure 2.5: Example of pooling with a 2×2 window size and stride.

Neural Networks.

The VGG neural networks were created by Simonyan and Zisserman and won the ImageNet challenge in 2014 [23]. They showed that convolutional layers with small convolutional kernels were able to achieve better classification performance than networks using large convolutional kernels. There is a total of four VGG networks ranging from 11 to 19 convolutional layers. These VGG networks are shown in Figure 2.6, where *Conv* stands for convolutional layer, *MaxPool* stands for max-pooling layer, *FC* stands for fully-connected layer, the number in each convolutional layer represents how many 3×3 filters are in that layer, and the number in each fully-connected layer represents how many neurons is in that layer. The input size used in all four VGG networks is equal to $3 \times 244 \times 244$, where the first dimension represents the number of channels in a color image (red, green, and blue channels).

Residual neural networks (ResNets) were created by He *et al.* to improve the gradient computation in very deep neural networks [5]. Feed-forward DNNs, such as the VGGs, are trained to learn a direct mapping between its inputs and outputs. However, when DNN architectures contain too many layers, the gradient of the first layers will become close to zero in a phenomenon called *vanishing gradient*. He *et al.* showed that, instead of computing a direct mapping between inputs and outputs, a DNN could be trained to learn the residual mapping. In practice, this can be achieved by adding shortcut connections between multiple layers. This new connectivity pattern between layers improves the DNN performance without adding any extra parameters to the network. It also allows for even deeper DNNs than the ones seen before. The basic ResNet building block is the so-called *residual block*, shown on the left side of Figure 2.7. A *residual block* contains two convolutional layers with 3×3 filters and one shortcut connection from the inputs to the output of the second convolutional layer. ResNets can also use *bottleneck residual blocks*, shown on the

right side of Figure 2.7. These *bottleneck residual blocks* are composed of three convolutional layers with the middle layer having 3×3 filters, and the others with 1×1

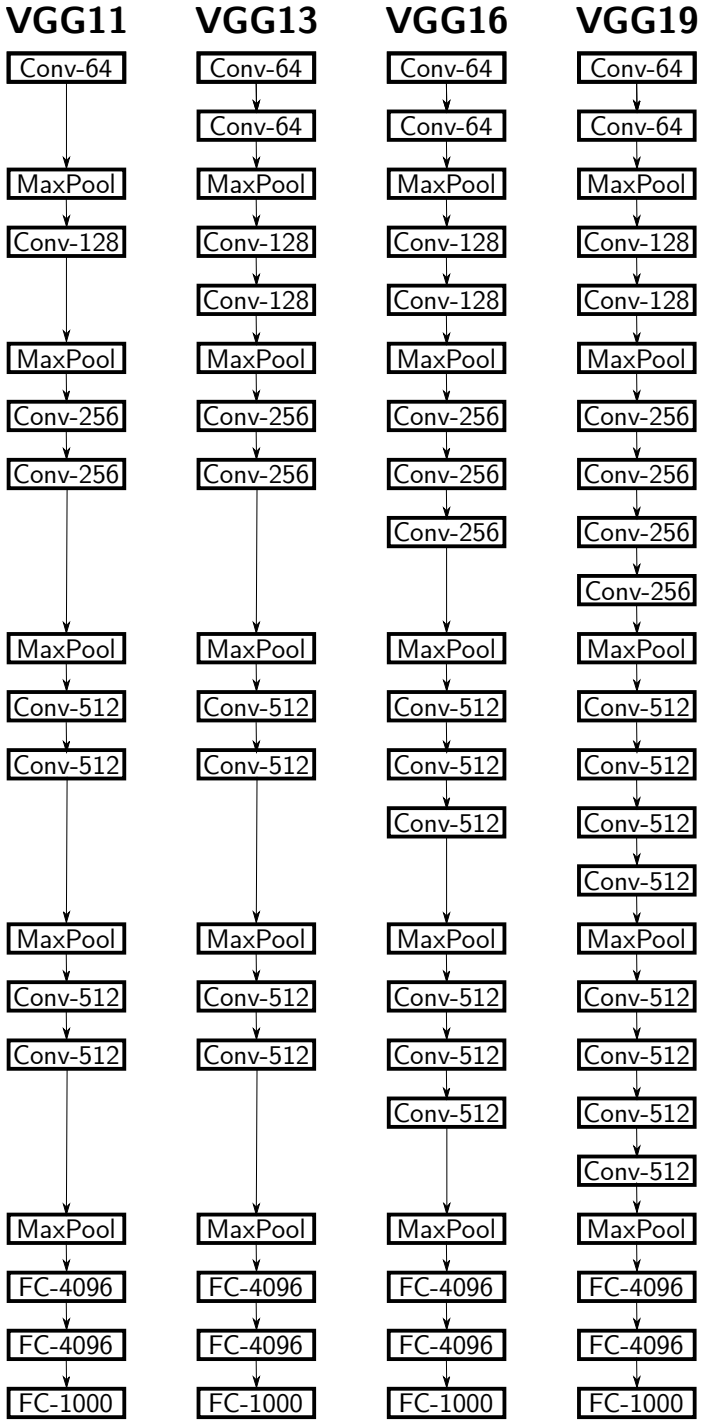


Figure 2.6: VGG neural network architecture. Adapted from [23].

filters. He *et al.* claim that the bottleneck design is more computationally efficient than the non-bottleneck one. However, the bottleneck design does not show any signs of performance improvement in the network when compared with the non-bottleneck design. For last, the shortcut connection is joined with the output of the last layer in a block by performing an addition operation followed by a rectified linear activation function (ReLU). Thus, the shape of the inputs needs to be equal to the shape of the last layer's output.

Densely connected neural networks (DenseNets) created by Huang *et al.* [6] can be seen as an improvement from ResNets. Where in a *residual block* only the inputs are connected to the outputs through a shortcut connection, the so-called *dense block* can have an arbitrary number of layers, and each layer's outputs within the same *dense*

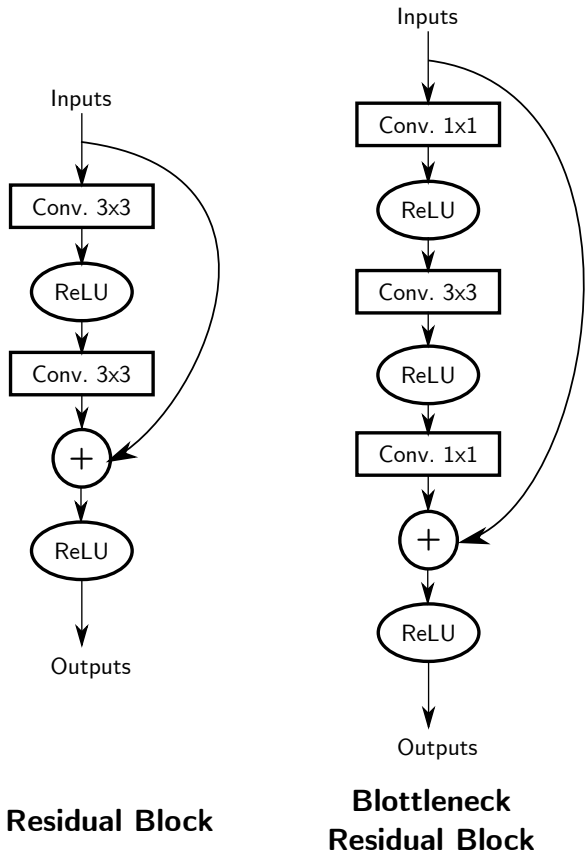


Figure 2.7: Residual blocks used by ResNets. Adapted from [5].

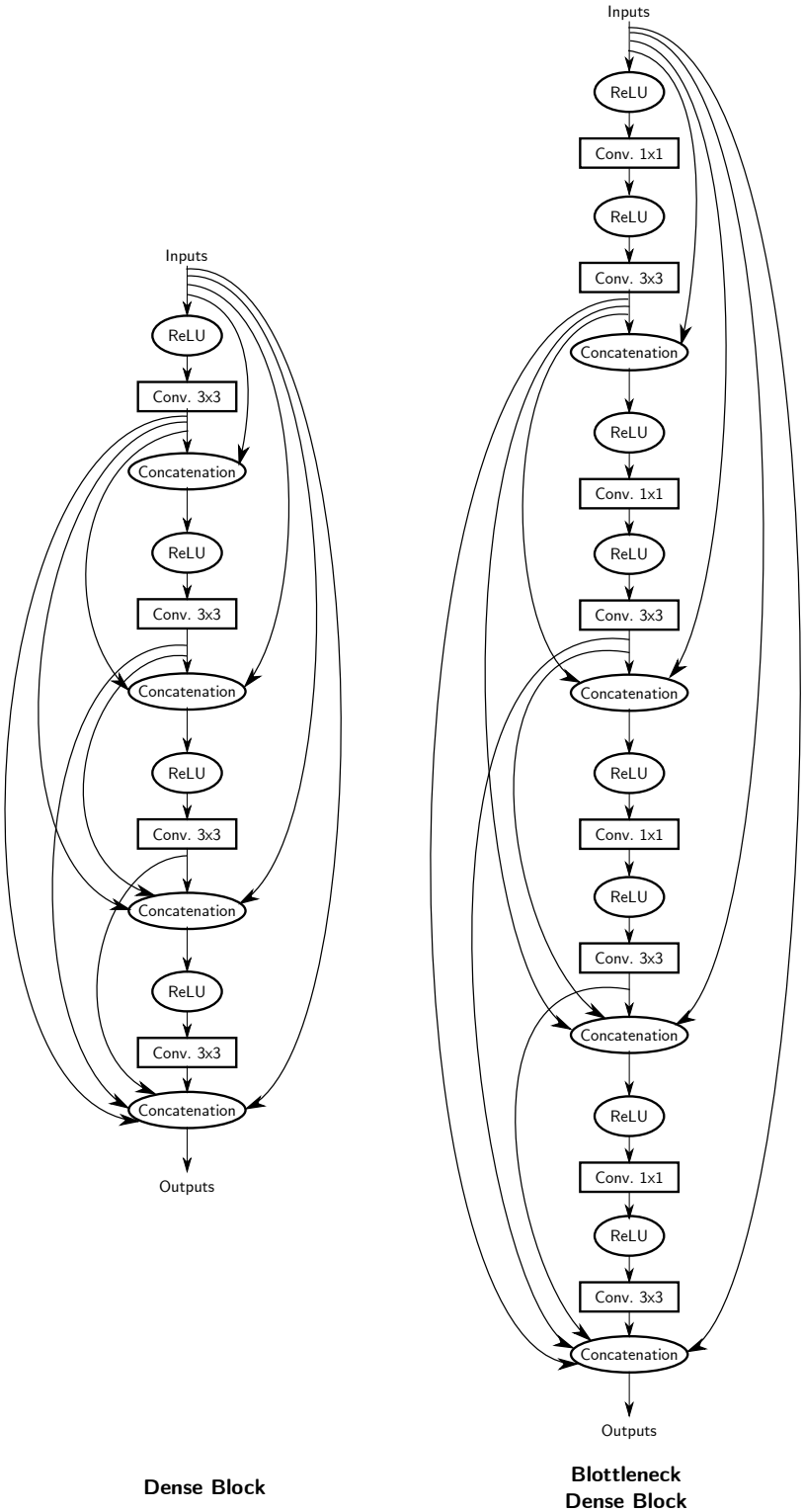


Figure 2.8: Example of Dense blocks with 4-layers. Adapted from [6].

block are connected to the input of all subsequent layers in a pattern that resembles the neurons' connectivity in fully-connected layers. Instead of using addition, like in the *residual blocks*, the *dense blocks* use concatenation. Thus, the number of inputs for each layer is not constant and increases from one layer to the next, illustrated in the left side of Figure 2.8. Similar to the *residual blocks*, it is also possible to use a bottleneck version of a *dense block*. The bottleneck is built by adding convolutional layers with 1×1 filters before the convolutional layers with 3×3 filters. The 1×1 convolutional layers help to stabilize the number of inputs that each 3×3 convolutional layer receives, which creates more computationally efficient networks, illustrated in the right side of Figure 2.8. After each *dense block*, a transition layer is used to avoid that the number of inputs at the end of the network explodes exponentially. The connectivity pattern found in DenseNets allows the use of fewer filters in each convolutional layer; this produces DNNs with higher classification performance and fewer parameters than any other DNN architectures.

2.1.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a particular type of DNN used to generate data instead of classifying it. GANs were first developed by Goodfellow *et al.* [61] in 2014, which consisted of two ANNs playing a minimax game: one called the *generator*, and the other called the *discriminator*. The *generator* learns the probability distribution of real data, while the *discriminator* learns to distinguish between data coming from the real distribution or the distribution generated by the *generator*. This vanilla GAN was later enhanced to allow the generation of images by using convolutional layers in the *discriminator* and transposed convolutional layers in the *generator*, and it is called Deep Convolutional Generative Adversarial Network (DCGAN) [62]. Figure 2.9 illustrates a GAN in which DCGANs share the same

structure. In general, $\mathbf{z}$ represents a vector of random values, which is used as the input of the *generator*, and data coming from the real probability distribution is represented by $\mathbf{x}$. Finally, the training of GANs and DCGANs is performed by using the following loss function [61]:

$$\min_{G}\max_{D} L(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[log(D(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[log(1-D(G(\mathbf{z})))], \qquad (2.6)$$

where the *generator* (G) tries to minimize $log(1-D(G(\mathbf{z})))$, and the *discriminator* tries to maximize $log(D(\mathbf{x}))$.

The Wasserstein Generative Adversarial Network (WGAN) developed by Arjovsky *et al.* in 2017 [63] is another important GAN type. The WGAN architecture is essentially the same as the ones found in DCGANs. The main difference is that the WGAN's *discriminator* has a linear activation function as output, which is not limited to 0.0 and 1.0 as in the traditional DCGANs. The loss function in WGAN is not a minimax game, but the Wasserstein distance or Earth-Mover distance, and it

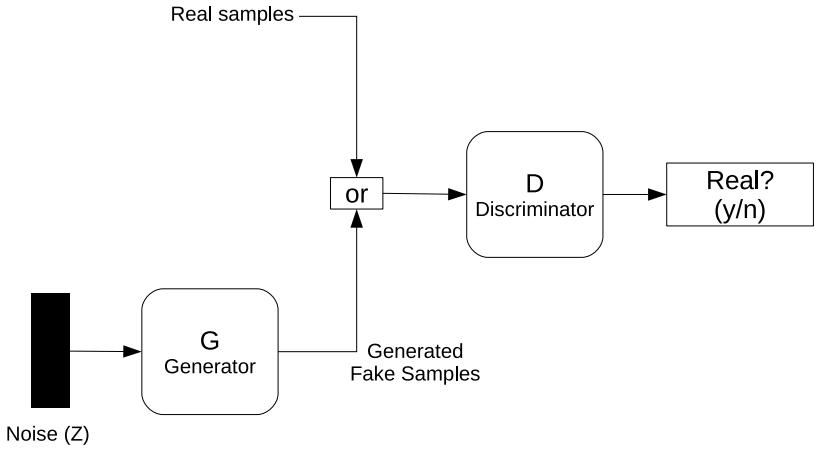


Figure 2.9: Example of a Generative Adversarial Network.

is defined as follows [63]:

$$L(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \prod(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[||x - y||], \tag{2.7}$$

where $\mathbb{P}_r$ the probability distribution of the real data and $\mathbb{P}_g$ the probability distribution of the *generator*'s parameters. The Wasserstein distance measures the distance between two probability distribution in the horizontal axis instead of the vertical axis. This loss function also allows for a more stable training avoiding problems with *gradient vanishing* or *mode collapse* in the generator.

2.2 Evolutionary Computation

Evolutionary computation (EC) algorithms are a class of derivative-free algorithms used to solve optimization problems. In this section, the definitions of single-objective and multi-objective optimization problems are presented and, then, two EC algorithms related to the present work, particle swarm optimization (PSO), and evolution strategy (ES), are explained in detail. PSO is a heuristic algorithm based on the flying pattern of bird flocks capable of finding a suitable solution faster than other EC algorithms. While ES is one of the first EC algorithms created back in the 1960s. It takes advantage of randomized heuristics to find solutions to discrete and combinatorial optimization problems easily. These two algorithms contain important characteristics that can be exploited during the generation and pruning of deep neural network architectures.

2.2.1 Single-objective and Multi-objective Optimization

In an optimization problem, one would like to find the best possible solution from a vast pool of candidate solutions [64]. Commonly, the best possible solution is the

one that gives the minimum or maximum of an *objective*, *loss*, *cost*, or *fitness* function. Note that EC algorithms also present a stochastic behavior and, therefore, it is not guaranteed that the solution found is the global best possible solution, also known as the *optimal* solution [64, 65]. It is also possible to place constraints on the solutions being searched, reducing the size of the search space considerably. One key aspect of EC algorithms that is not easily controllable is the level of exploration and exploitation [66]. Exploration refers to the algorithm's ability to find solutions far away from the current ones, while the exploitation is the algorithm's ability to find solutions close to existing ones. The so-called operators that exist in all EC algorithms are one way of achieving exploration and exploitation. All EC algorithms have a *selection* operator that is used to choose the best individuals and *variation* operators that ensure individuals will change over time. Moreover, optimization problems can be divided accordingly to the number of functions being optimized.

If only one function is being optimized, the problem is called a Single-Objective Optimization (SOP). A simple example is shown in Figure 2.10, where the black dot represents the optimal minimum of a single input function, which also has some local maximum and minimum points. An algorithm using information from this function's derivative, such as steepest descent, would become trapped in any of the local minimum points because the derivatives in these points are equal to zero. Once a derivative-based algorithm is trapped in a local minimum, the optimal minimum becomes impossible to find. Because of their derivative-free nature, EC algorithms can escape from local minimum points and are also ideal for use in problems where the objective function is no guaranteed to be continuous.

If two or more functions are being optimized at the same time, the problem is then called a Multi-Objective Optimization (MOP) problem. Furthermore, in these types of problems, the functions being optimized conflict with each other. In this sense, a

good solution in one of the objectives may be a bad one in the other objectives. In MOPs, in general, it is not possible to find a solution that is the best in all objective functions. Thus, algorithms dealing with MOPs are developed to find multiple solutions from an N-dimensional space formed by the N objective functions used in the problem, called the objective space. The selection of solutions in MOPs uses the concept of *domination* in the objective space. A solution $\mathbf{x}^1$ is said to dominate a solution $\mathbf{x}^2$ ($\mathbf{x}^1 \succ \mathbf{x}^2$), if $f_n(\mathbf{x}^1)$ is no worse than $f_n(\mathbf{x}^2)$, for all objective functions, $f_n(.)$, $n = 1, ..., N$, and $\mathbf{x}^1$ performs better than $\mathbf{x}^2$ in at least one objective function [65, 67]. The group of solutions in the objective space that dominates all other solutions is called the *non-dominated front*, shown in Figure 2.11, and it is frequently used by MOP algorithms to choose the best solutions in the population.

2.2.2 Particle Swarm Optimization

Kennedy and Eberhart created the Particle Swarm Optimization (PSO) algorithm in 1995 [30, 31], and, in its simplest form, it is used to solve nonlinear Single-Objective

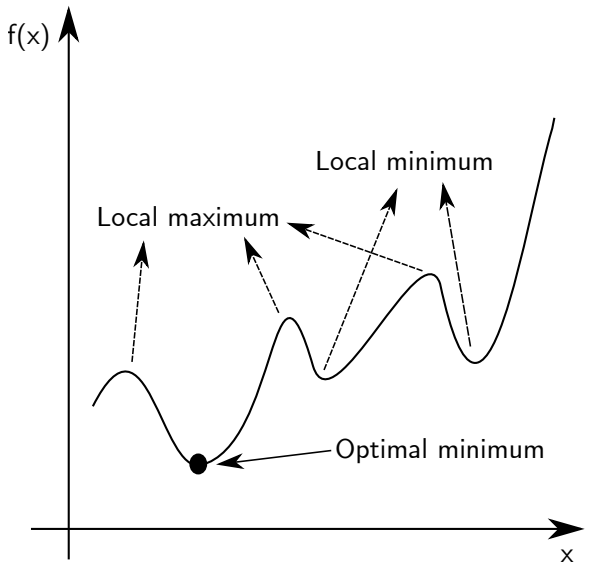


Figure 2.10: Example of a continuous Single-Objective Optimization problem.

Optimization problems. The flying pattern of a flock of birds was the inspiration for the creation of PSO. The idea is that birds can travel together knowing only the location of its neighbors and its current location. An individual bird does not know the final objective of the flight, and, at any moment, if any bird finds something of interest, it can change the flight direction of the flock. The PSO authors called this *social* or *swarm* intelligence, and it is nature's solution for optimization tasks. [30]. One key advantage of PSO over other EC algorithms is that PSO converges to good solutions faster than other algorithms [68, 69].

In PSO, each *particle* or *individual* knows the position of the current best particle in the population, the global best (**gBest**), and its own previous best position, the personal best (**pBest**). At any given iteration, k, a velocity vector, $\mathbf{v}_i$, is computed for each particle, i, which is used to compute the particle's position vector, $\mathbf{x}_i$, for the next iteration, $k+1$. The velocity of each particle is computed according to Equation 2.8, where $v_{i,j}(k+1)$ represents the j-th component of the i-th particle's velocity for the next iteration $k+1$; $pBest_{i,j}$ represents the j-th component of the

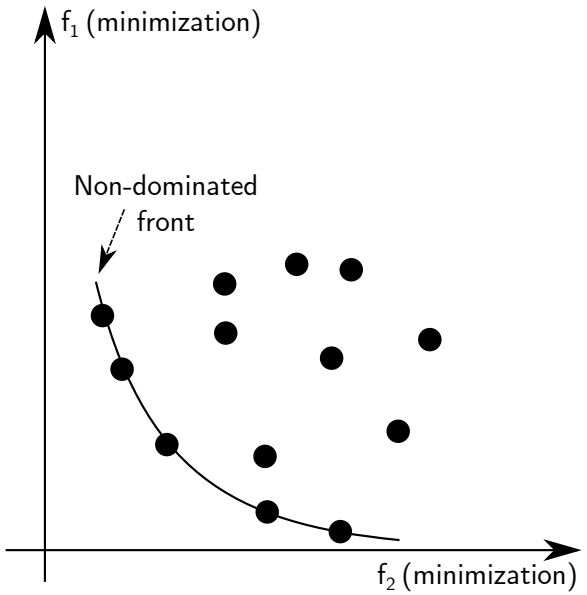


Figure 2.11: Example of a non-dominated set in a two-objective space.

best position of particle i so far; $gBest_j$ represents the j-th component of the current global best particle; w is a constant defined beforehand called momentum; c_p and c_g are also constants defined beforehand; r_p and r_g are random numbers generated in the interval of 0 and 1. Once a particle's velocity is computed, its position can be updated following Equation 2.9.

$$v_{i,j}(k+1) = w \times v_{i,j}(k) + c_p \times r_p \times (pBest_{i,j} - x_{i,j}(k)) + c_g \times r_g \times (gBest_j - x_{i,j}(k)) \quad (2.8)$$

$$x_{i,j}(k+1) = x_{i,j}(k) + v_{i,j}(k+1) \quad (2.9)$$

The general PSO algorithm is shown in Algorithm 1, where P_i represents a particle i in the population, and $P_i.obj$ represents the value of the particle i in the objective function used for optimization, also known as the particle's fitness value. A population of N particles is first initialized at random, and their objective function values are evaluated. The $pBest$ of all particles are initialized to its initial position, and the global best particle, $gBest$, is chosen within the population. Then, the velocity of each particle and their positions are iteratively computed and updated according to Equations 2.8 and 2.9. After each position update, the quality of the particle is evaluated on the objective function and compared with its $pBest$ and $gBest$. If the updated particle is better than $pBest$ and $gBest$, the particle's position is replaced. Not shown in Algorithm 1 is how to handle particle's positional constraints, which can be quickly implemented by limiting the positional values to upper and lower bounds when updating the particle's position in line 20 of Algorithm 1.

2.2.3 Evolution Strategies

Evolution Strategies (ES) were first developed back in the 1960s to solve the problem of designing successive experiments to study turbulent airflow [70]. The ES cre-

Algorithm 1: Particle Swarm Optimization

Input : Objective function ($f(.)$), population size (N), maximum number of iterations ($iter_{max}$), momentum (w), constants c_p, and c_r.
Output: The global best particle ($gbest$).

```
// Population initialization
for i = 0 to N do
    P_i.x ← Initialize position values at random;
    P_i.obj ← Evaluate the objective function value (f(P_i.x));
    // Make the particle's pBest equal to itself
    P_i.pBest ← P_i;
    // gBest initialization
    if i = 0 then
        gBest ← P_i;
    else
        if P_i.obj is better than gBest.obj then
            gBest ← P_i;
        end
    end
end
// Objective function optimization
for k = 1 to iter_max do
    for i = 1 to N do
        P_i.velocity ← Update velocity according to Equation 2.8;
        P_i.x ← Update position according to Equation 2.9;
        P_i.obj ← Evaluate the objective function value (f(P_i.x));
        if P_i.obj is better than P_i.pBest.obj then
            P_i.pBest ← P_i;
            if P_i.pBest.obj is better than gBest.obj then
                gBest ← P_i.pBest;
            end
        end
    end
end
// Return global best individual
return gBest;
```

ators observed that a well thought randomized heuristic produced better results than the gradient-based one they had been using back then. The idea of using randomized heuristics to solve optimization problems creates an elegantly simple algorithm with only two essential actions: variation and selection.

The most basic variation operator used by ES algorithms is random mutations of individuals' parameters. Optionally, a recombination operator can also be used to produce variation in the population. The discrete recombination is the most commonly used one. It is performed by randomly selecting ρ parents from the population, and one new individual is generated by randomly picking one parameter of each ρ parents to be transferred to the new individual [70]. An example of discrete recombination with four individuals is shown in Figure 2.12. The selection procedure in ES is very straightforward. At each generation, the best μ individuals are selected to produce new offspring. Commonly, the best individuals are the ones with the highest, in a maximization case, or the lowest, in a minimization case, fitness values.

Most of ES algorithms configurations can be summed up using the following

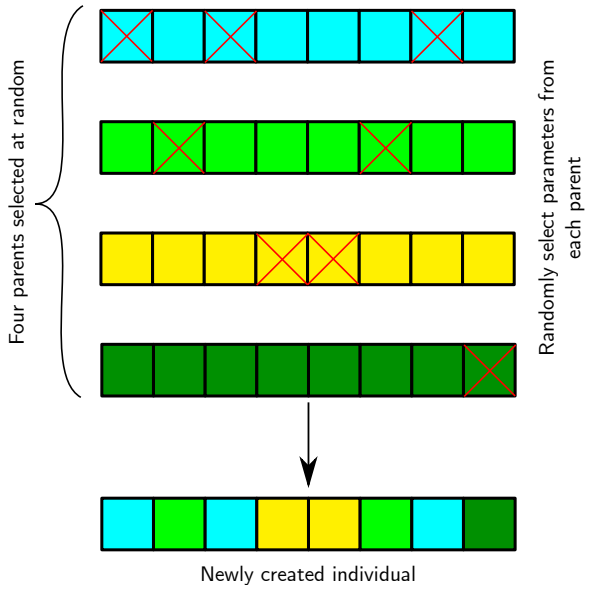


Figure 2.12: Discrete recombination with four individuals. Adapted from [70].

nomenclatures: $(\mu/\rho + \lambda) - ES$ or $(\mu/\rho, \lambda) - ES$ [70], where μ represents the number of parents selected at every generation, λ is the number of offspring that will be produced at every generation from the μ parents selected, and ρ is the number of individuals selected for use in the recombination process. Moreover, there are two types of ES algorithms: a *plus* (+) and a *comma* (,) versions. Their main difference is in the selection procedure. In the *plus* version, the selection is performed by using the combination of μ parents **plus** λ offspring, while, in the *comma* version, the selection is performed by using **only** λ offspring. The *plus* version is an elitist version of ES because good solutions will be kept in the population until better offsprings are found. On the other hand, in the *comma* version, only the offsprings are kept, and all parents are eliminated regardless of how good they are. The entire ES pseudo-code is presented in Algorithm 2.

2.3 DNN Architecture Searching and Pruning

In this section, related works about DNN architecture searching are first discussed, followed by works on DNN architecture pruning.

2.3.1 DNN Architecture Searching

In the early 1990s, most evolutionary approaches (EAs) were used as a replacement for backpropagation during the training phase of ANNs [36, 37, 38]. However, EAs started been used to design ANN architectures soon after. Such algorithms are currently known as NeuroEvolution, Evolutionary Artificial Neural Networks (EANNs) or Topology and Weight Evolving Artificial Neural Networks (TWEANNs) [71, 39], and, as suggested by their names, they are capable of evolving only plain ANNs because they work by evolving the connections between individual neurons instead of

groups of neurons.

Similar to EAs used to solve simple optimization problems, TWEANNs are also population-based algorithms where each candidate solution represents a different ANN architecture. Then, the objective of these algorithms is to use the EC's selection and variation operators to find better architectures over time. The definition of better architectures will depend on the problem at hand; it can be architectures with better accuracy in a given dataset or architectures with a small number of parameters [71].

The main challenge faced by researchers developing TWEANNs was how to rep-

Algorithm 2: Evolution Strategy

Input : Objective function ($f(.)$), maximum number of generations (gen_{max}), number of parents (μ), number of individuals for recombination (ρ), number of offspring (λ), and algorithm version (ver).

Output: The best individual in the final generation.

```
// Initialize population
P_μ, P_λ ← Initialize individuals at random;
f(P_μ), f(P_λ) ← Evaluate the fitness of the population;
// Objective function optimization
for g = 1 to gen_max do
    if ver = plus then
        P_μ ← Select best μ individuals from P_μ + P_λ;
    else if ver = comma then
        P_μ ← Select best μ individuals from P_λ;
    end
    for i = 0 to λ do
        P_ρ ← Randomly select ρ individuals from P_μ;
        P_λi ← Generate one offspring using recombination from P_ρ;
        P_λi ← Randomly mutate each of the individual's parameters;
        f(P_λi) ← Evaluate the fitness of the offspring;
    end
end
return the best individual in P_μ;
```

resent ANN architectures in a meaningful way for use by EC's operators. Early TWEANNs used *direct encoding* schemes to represent an ANN architecture. In this kind of scheme, each neuron and each connection between them have a directly representation in the algorithm. It was common to use binary connectivity matrices to direct represent ANNs, where each matrix element, $a_{ij} = 1$, represents if a connection from neuron i to j exists [71]. Some examples of works that use this type of representation can be found in [36, 72, 73]. Weighted graphs are another *direct encoding* representation used in [74], where real numbers are used to encode the weights of an ANN. However, *direct encoding* is not efficient to use when the network contains hundreds of millions of neurons, even when using real numbers to represent the ANN architecture instead of binary numbers [71]. To overcome this problem, researchers developed indirect means to represent an ANN architecture, called *indirect encoding* schemes. Parametric representation is one conventional approach to represent ANNs indirectly in which a candidate solution is constructed from a set of parameters, such as the number of neurons in a given layer, the number of connections between layers, and others [71, 75].

Perhaps the most influential TWEANN is the NeuroEvolution of Augmenting Topologies (NEAT) developed by Stanley and Miikkulainen in 2002 [39, 40]. NEAT uses a direct encoding scheme, where the representation or genome of an individual is a list of neurons and their connectivity. NEAT works by growing simple networks with only inputs and outputs neurons into more complex ones over the generations. Embedded in the individual's representation is also a history of changes that are used as a speciation mechanism. Hence, the population is composed of individuals from different species. The idea of using speciation is to avoid that individuals been eliminated before they can begin to perform better, which also avoids premature convergence of the population. An efficient crossover and mutation (variation) operators

were also developed to deal with variable length genomes, and it always produces valid architectures. At the time, NEAT was considered the best algorithm to evolve neural networks with outstanding results. However, it was only capable of finding feed-forward and recurrent ANN architectures, which does not scale well for use in deep learning applications.

The Evolutionary Acquisition of Neural Topologies (EANT) [76, 77] is another NeuroEvolution algorithm for searching feed-forward and recurrent ANN architectures. EANT uses a hybrid representation scheme where the ANN weights are directly encoded in each of the genome position, and the ANN architecture can be decoded by reading the genome positions from left to right. In contrast with NEAT, EANT only uses mutations to modify individuals in the population. It contains two mechanisms for such a task: structural exploration and structural exploitation. Structural exploration performs mutations in the ANN architecture, while structural exploitation uses Evolution Strategy to optimize the ANN weights. EANT achieves comparable results to NEAT.

Particle swarm optimization (PSO) algorithms have also being used to evolve ANNs weights and architectures similar to the purely evolutionary approaches cited previously. The work presented by Gudise and Venayagamoorthy in [78] is one of the firsts to use PSO to train ANNs replacing the backpropagation algorithm. Similarly, the works developed by Carvalho and Ludermir in [79, 80] were the first ones to use PSO to search ANN architectures and weights at the same time. However, these and other works using PSO suffers from the same problem of classic TWEANNs as that they are only capable of searching for classic ANN architectures [81, 82].

Stanley *et al.* in 2009 [83] tried to overcome the limitation in the size of ANNs created by TWEANNs at the time by developing an indirect representation to use together with the NEAT algorithm, called HyperNEAT. An individual is represented

using hypercubes, and the algorithm uses NEAT to evolve connective compositional pattern-producing networks (CPPNs) that can be later decoded into ANNs architectures. The advantage of HyperNEAT is that it can generate ANNs taking into account the geometry of the problem at hand. Although HyperNEAT can create more complex ANNs than the ones from NEAT, it still cannot create award-winning ANNs for use in deep learning.

Recently, researchers from Google showed that it was possible to search for DNN architectures, instead of simple ANNs, using evolutionary approaches given enough computational power is available [84]. They developed a custom evolutionary algorithm, called Large-Scale Evolution of Image Classifiers (LSEIC), where candidate solutions are modified using only random mutations. LSEIC is one of the first algorithms to find DNN architectures with competitive results in challenging image classification datasets. However, its main downside is the amount of computational power required. The authors used a server with 250 graphical processing units (GPUs), taking a total of 4×10^{20} floating operations (FLOPs), and 11 days to achieve good results. Unfortunately, not all interested researchers have this amount of computational power at their disposal. Thus, there is still substantial efforts to develop DNN architecture searching algorithms that can be executed in consumer-grade GPUs in a reasonable amount of time.

Liu *et al.* [85] also developed an evolutionary algorithm for DNN architecture searching by performing only random mutations on candidate solutions. Different from LSEIC that builds the DNN layer by layers, Liu *et al.* build the DNN architecture hierarchically, where each node in the structure can be further developed into a small network. This algorithm also obtained competitive results in challenging classification datasets. However, it still suffers from a high computational burden. The authors reported that with 200 GPUs, it still takes 1.5 days to find reasonable

solutions.

Xie and Yuille [86] developed the *genetic CNN* algorithm to search for DNN architectures. Specifically, this algorithm is capable of finding convolutional neural networks (CNNs), residual neural networks (ResNets), and densely connected neural networks (DenseNets) architectures. It also uses a complete EA framework with crossover and mutations operators to perform variation in candidate solutions. Moreover, it has a smaller computational burden than those of LSEIC and Liu *et al.*'s algorithms, with the authors reporting two days using 10 GPUs.

Sun *et al.* developed an evolutionary algorithm to search specifically for CNN architectures, called *evoCNN* [87]. It uses novel crossover and mutation operators to deal with a variable number of layers on a candidate solution. Due to its reduced search space, it can obtain good solutions with significantly less computational power, and the authors report a running time of three days using two GPUs. Sun *et al.* further developed their algorithm in [88] to use blocks of ResNets and DenseNets to construct a DNN architecture. Although it can obtain better results than the original *evoCNN* algorithm, this new algorithm does require more computational resources, with the authors reporting a running time of nine days using three GPUs. Evolutionary approaches can also be used to search for unsupervised DNN architectures. Sun *et al.* in [89] developed the evolving unsupervised DNNs (EUDNN) algorithm, where the solutions are encoded using a set of basis vectors. Unfortunately, no information about the computational complexity of EUDNN was provided, but the algorithm is capable of finding solutions with competitive results to other unsupervised machine learning approaches.

2.3.2 DNN Architecture Pruning

DNN architecture pruning, also called by some as DNN compression, is used to reduce the computational complexity of DNN models. Most DNNs contain millions of parameters, requiring a lot of computational power to train and deploy them. The main hypothesis used for pruning is that DNN architectures created by human experts may contain too many unnecessary or redundant parameters. Thus, it may be possible to created DNNs with fewer parameters, and without performance degradation. This procedure can be formulated as an optimization problem where one wants to find the right combination of filters in each DNN layer that reduces its number of parameters as much as possible while maintaining the original DNN performance.

Pruning can only be performed on convolutional and fully-connected layers, but not on pooling layers because they do not contain any parameter. Fully-connected layers can be pruned by eliminating individual neurons, while convolutional layers can be pruned by eliminating entire filters. Most human-crafted DNN architectures contain a series of convolutional and pooling layers, and just a few fully-connected layers at the end. Thus, the convolutional layers could contain almost ten times the computational complexity of the fully-connected ones [55]. Because of this, most researchers are interested in pruning only the convolutional layers of a DNN architecture.

The parameters or weights of a single convolutional layer are a four-dimensional tensor, $\mathbf{W}$, with dimensions equal to $(K \times D \times W \times H)$, where K represents the number of individual filters $(D \times W \times H)$, and D the depth, W the width, and H the height of each filter. The pruning procedure will consist of the elimination of up to $K - 1$ filters in each convolutional layer. This procedure is exemplified in Figure 2.13, where, in the top half of the figure, the first filter from a total of four filters of size $(3 \times 3 \times 3)$ is selected to be pruned, and, in the bottom half, is the resulting

pruned model. Hence, most researchers developed algorithms to select which filters to eliminate and which ones to keep in the architecture, in a process called *filter selection*.

Usually, researchers use some statistical information from the current layer to decide which filters to eliminate. In the literature [90], the most used criteria for filter selection are the following:

- The *mean activation* of each filter is used by Molchanov *et al.* in [53], where the filters with the lowest mean activation are considered the least important

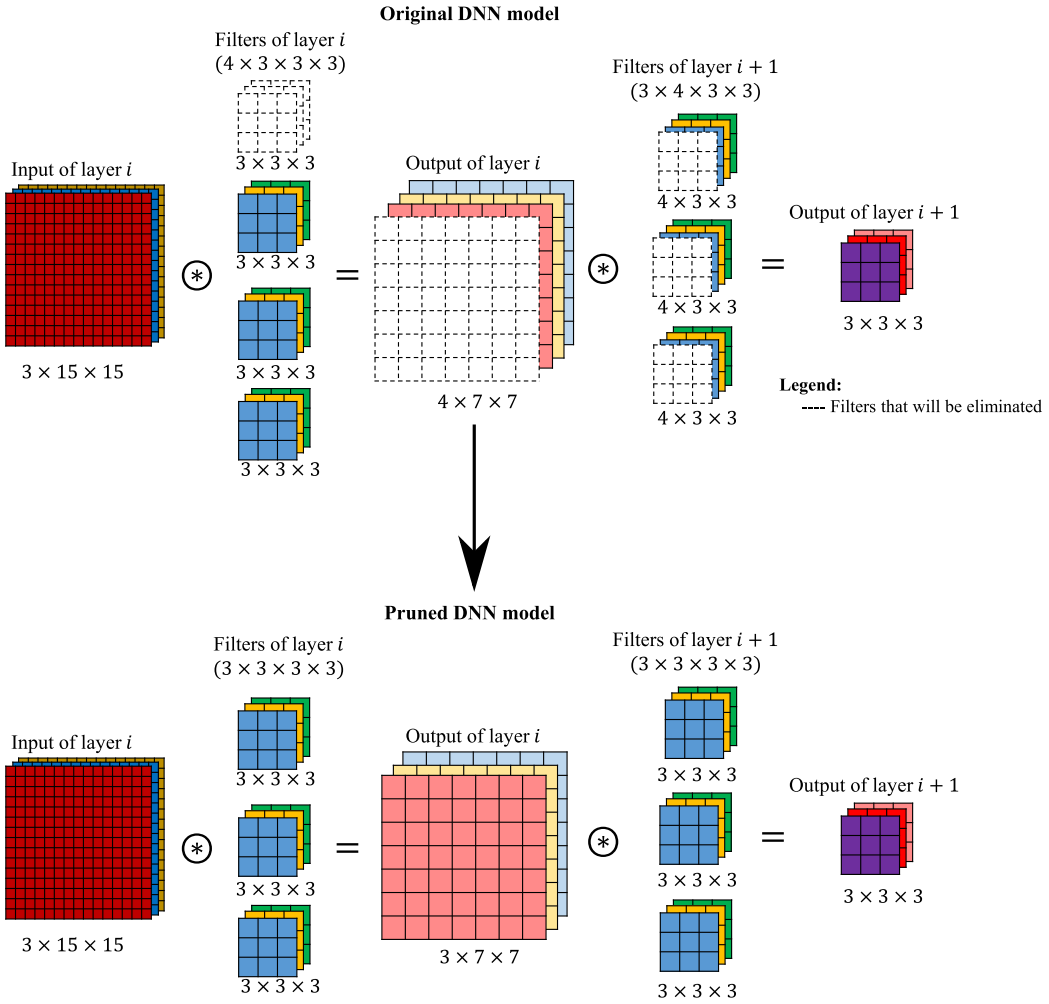


Figure 2.13: Pruning of a single filter in a convolutional layer.

ones and are selected for elimination.

- The filter's $l1$-norm is used by Li *et al.* in [55], where the filters considered least important are the ones with the smallest $l1$-norm, i.e., the filters with the smallest sum of its absolute weights and are usually eliminated.

- The *Average Percentage of Zeros* (APoZ) of each filter can also be used as selection criteria [50]. Filters containing many zeros are considered less important for the DNN model.

- Lu and Wu [49] used the *entropy* of each filter, given a set of inputs and outputs. They argued that unimportant filters will have similar *entropy* even for different sets of inputs and outputs.

- Luo *et al.* [56] used the output of the next layer to decide which filter to eliminate in the current layer.

- Mittal *et al.* [90] argued that the filter selection criteria is not essential and that one could select filters at *random*.

Other works use less popular pruning techniques instead of *filter pruning*. For example, Han *et al.* and Hubara *et al.* use quantization to reduce the number of bits needed to represent the parameters of a DNN model [51, 52]. While the approach from Ding *et al.* [54] is to not modify the DNN architecture at all but to change the way convolutions are computed by using block-circulant matrices. However, these pruning technique needs specialized software and hardware for implementation, which makes the *filter pruning* technique the most desired one for further development.

CHAPTER III

CONVOLUTIONAL NEURAL NETWORK ARCHITECTURE SEARCHING BASED ON PARTICLE SWARM OPTIMIZATION

The most used models for image and object classification tasks are Convolutional Neural Networks (CNNs). However, finding meaningful CNN architectures is not trivial and needs human experts to design them by trial and error. Thus, in this chapter, a Particle Swarm Optimization based algorithm to design CNNs automatically is proposed and evaluated in challenging image classification datasets[1].

3.1 Introduction

In image or object classification tasks, one would like to build a model capable of classifying in which class a given image belongs to. Historically, researchers have been using a plethora of algorithms and models to solve this type of tasks, such as support vector machines [92], k-nearest neighbors [93], and Fisher vectors [94], to name a few. However, only after the introduction of Deep Learning (DL) techniques is when models began to be able to classify images with human-level accuracy.

The most successful models used in image classification are Convolutional Neural Networks (CNNs), in which DL has allowed the creation of CNNs with dozens of layers. These networks are a particular type of Deep Neural Networks (DNNs), where a feed-forward architecture, without shortcut or skip connections, and with multiple

[1]The present chapter is partially based on my previously published work found in [91]. Tables and Figures are reproduced here with permission of the publisher of [91].

layers of convolutional operations, is used. Although CNN architectures are known to be the most successful models for use in such tasks, finding the ideal architecture for a given problem is still a trial and error undertaking. For example, Krizhevsky *et al.* [57] showed that the use of the rectified linear unit (ReLU) as the convolutional layer's activation function, where $f(n) = max(0, n)$, yielded better results than the use of the classical hyperbolic tangent function, $f(n) = tanh(n)$. Likewise, Simonyan and Zisserman [23] found out that the use of small 3×3 convolutional filters decreases the overall number of parameters of a CNN which allowed for deeper networks with higher classification capabilities. Researchers have developed other approaches to improve DNNs classification capabilities, but these approaches change the network architecture so much that they are not considered simple CNNs anymore.

The ability to design a CNN architecture automatically would disseminate the use of DL techniques to researchers and experts outside the field and reduce the development time of CNN applications for consumer uses. In the last 30 years, as presented in Section 2.3.1, many researchers have been developing methods for automatically searching for CNN architectures. However, most of them were developed using the concepts from genetic algorithms, which require massive amounts of computational power to search for a single CNN model. Therefore, the development of an algorithm to search CNN architectures without requiring the computational power only available in data-centers is proposed in this chapter.

The Particle Swarm Optimization (PSO) algorithm is considered an excellent alternative for Genetic Algorithms because of its fast convergence speed [68, 69]. Furthermore, as presented in Section 2.3.1, researchers have used PSO algorithms for artificial neural network architecture search with varying degrees of success in the past. For example, Sun *et al.* [95] successfully developed a PSO algorithm to design convolutional autoencoders (CAEs) automatically instead of direct CNN architec-

tures, and they obtained competitive results when compared with other methods. However, the closest related work to the proposed algorithm presented in this chapter is the one done by Wang *et al.* [96] called IPPSO. In that work, each layer of a CNN is encoded using real numbers based on IP addresses from computer networks. The use of real number encoding allows for the implementation of an almost standard PSO algorithm with no need for the development of specific operators, such as particle's velocity and position updating. Although IPPSO can deal with particles with different lengths, it cannot produce CNNs with more layers than a specified maximum length. Furthermore, the results presented by the IPPSO authors are limited to only three image classification datasets.

Thus, a novel PSO algorithm capable of searching for CNN architectures with a good balance between searching speed and CNN classification accuracy, called *psoCNN* [91], is presented in this chapter. The main contributions of the proposed *psoCNN* algorithm are the following:

- A novel encoding scheme is developed where each position of a single particle's parameters encodes a single layer together with the layer's hyperparameters, such as the number and size of its convolutional filters.

- A novel operator to compute the difference between two particles is developed, which also allows the comparison of particles with different lengths.

- A novel velocity operator with only a single turning parameter is developed, allowing a particle to resemble the global best particle in the population or its own personal best.

These contributions together facilitate the development of a novel PSO-based algorithm that allows particles representing CNNs to grow or shrink in size, which

is extremely important when developing algorithms for CNN architecture searching. In the next section, the proposed *psoCNN* algorithm is discussed in detail. After that, the results obtained with the proposed algorithm are presented as well, showing that *psoCNN* is capable of obtaining competitive results when compared with peer competitors' algorithms.

3.2 Proposed Algorithm

The proposed *psoCNN* is shown in Algorithm 3. It receives as inputs the size of the population or swarm (N); the total number of iterations that the algorithm will run ($iter_{max}$); the image classification dataset used during the searching process ($\mathbf{X}$); the probability of updating a particle's parameters using the global best over the personal best when computing the particle's velocity (C_g); the maximum number of layers that a particle can be initialized (l_{max}); the maximum number of filters and their size that a convolutional layer can be initialized ($filters_{max}$ and k_{max}); the maximum number of neurons that a fully-connected layer can be initialized (n_{max}); the number of outputs (n_{out}); the number of training epochs used for particle evaluation (e_{train}), and the number of training epochs to fine-tuning the best CNN architecture found (e_{test}). The algorithm's output is the best CNN architecture found ($gBest$).

The algorithm uses the standard framework found in most PSO algorithms due to the proposed encoding scheme, and contains a total of four main components:

- The initialization of the population or swarm (Algorithm 3, line 1);
- The velocity computation of each particle (Algorithm 3, line 15);
- The parameter updating of each particle (Algorithm 3, line 16).
- The fitness or loss computation of each particle (Algorithm 3, lines 3, 8, 17, and

28).

The following subsections explain the encoding scheme used to represent a single CNN architecture and each of the above components. After these, more details about the influence of each of the algorithm's inputs are given.

3.2.1 Particle Encoding Scheme

The main challenge in adapting a PSO algorithm to work with CNN architecture searching is how to develop an encoding scheme that can be used to produce meaningful particle velocities. In the proposed encoding scheme, each particle's parameter encodes a single layer directly without using any conversion to a numbering system. The encoding scheme can be thought of as a nested list where each position of the list represents one layer, and, for each position, there is another list representing the hyperparameters of that layer. This encoding scheme is exemplified in Figure 3.1, where the illustrated particle has four parameters representing a four-layer CNN architecture.

There is a total of four types of layers allowed: convolutional, average pooling, max pooling, and fully-connected. The hyperparameters used by the convolutional layers are the number of filters, the two-dimensional size of each filter, and the two-dimensional strides used by the moving window. The pooling layers only have the type of pooling used by the layer: max or average. The fully-connected layers only have one hyperparameter that is the number of neurons in the layer.

Thus, to construct a CNN architecture from the proposed encoding scheme, the algorithm reads the particle's parameter list from left to right and creates a CNN by adding each layer represented by the parameter. The activation function used for every layer is the rectified linear unit (ReLU), and the strides used for the pooling

Algorithm 3: Proposed psoCNN [91]

Input : Swarm size (N), number of iterations ($iter_{max}$), training data (X), probability of choosing $gBest$ over $pBest$ when computing velocities (Cg), maximum number of layers used in the initialization (l_{max}), maximum number of convolutional filters used in the initialization ($filters_{max}$), maximum convolutional filter size used in the initialization (k_{max}), maximum number of neurons in a FC layer used in the initialization (n_{max}), number of outputs (n_{out}), number of training epochs during particle evaluation (e_{train}), number of training epochs for the best CNN found (e_{test}).

Output: The best CNN architecture found.

$S = \{P_1, ..., P_N\} \leftarrow InitializeSwarm(N, l_{max}, filter_{max}, k_{max}, n_{max}, n_{out})$;
$P_1.pBest \leftarrow P_1$;
$P_1.loss, P_1.pBest.loss \leftarrow ComputeLoss(P_1, X, e_{train})$;
Initialize $gBest \leftarrow P_1$;
$gBest.loss \leftarrow P_1.loss$;
for $i = 2$ *to* N **do**
    $P_i.pBest \leftarrow P_i$;
    $P_i.loss, P_i.pBest.loss \leftarrow ComputeLoss(P_i, X, e_{train})$;
    if $P_i.loss \leq gBest.loss$ **then**
        $gBest \leftarrow P_i$;
    end
end
for $iter = 1$ *to* $iter_{max}$ **do**
    for $i = 1$ *to* N **do**
        $P_i.velocity \leftarrow UpdateVelocity(P_i, Cg)$;
        $P_i \leftarrow UpdateParticle(P_i)$;
        $P_i.loss \leftarrow ComputeLoss(P_i, X, e_{train})$;
        if $P_i.loss \leq P_i.pBest.loss$ **then**
            $P_i.pBest \leftarrow P_i$;
            $P_i.pBest.loss \leftarrow P_i.loss$;
            if $P_i.pBest.loss \leq gBest.loss$ **then**
                $gBest \leftarrow P_i.pBest$;
                $gBest.loss \leftarrow P_i.pBest.loss$;
            end
        end
    end
end
$gBest \leftarrow ComputeLoss(gBest, X, e_{test})$;
return $gBest, gBest.loss$;

layers are always 2×2. Furthermore, the proposed encoding scheme does not encode the weights of a CNN model, needing a small retraining phase when the algorithm is computing the fitness or loss of a given particle.

3.2.2 Particle Evaluation

The main objective of the proposed *psoCNN* algorithm is to find the best CNN architecture for a given image classification dataset. Hence, it does not search for CNN weights for a given CNN architecture. When the algorithm needs to evaluate the quality of a given particle, performed by the function $ComputeLoss()$ found in Algorithm 3, lines 3, 8, 17, and 28, it first decodes the particle's parameters into a CNN model with weights initialized using the Glorot approach [97]. Then, the decoded CNN model is trained for a certain number of epochs using Adam optimizer [59] over the whole dataset. When the algorithm is only searching for the best CNN architecture, the models are trained using a smaller number of epochs (e_{train}) than the number of epochs used to retrain the global best particle (e_{test}) at the end of the searching process. This particle evaluation procedure is also the main bottleneck of the proposed algorithm because, at every iteration, the whole population needs to be retrained.

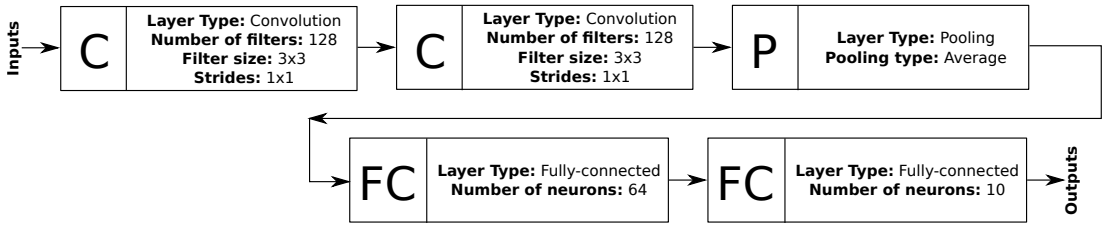


Figure 3.1: Proposed representation of a CNN architecture in *psoCNN*.

3.2.3 Swarm Initialization

The first step in the proposed *psoCNN* is to initialize the population or swarm of particles, which are initialized at random by following a series of input parameters given to the function $InitializeSwarm()$ in Algorithm 3, line 1. The parameters this function receives as inputs are the following: the total number of particles in the swarm N; the maximum number of layers that a single particle can be initialized (l_{max}); the maximum number of convolutional filters of any convolutional layer ($filters_{max}$); the maximum size of a convolutional filter of any given layer (k_{max}); the maximum number of neurons of any given fully-connected (FC) layer (n_{max}), and the number of outputs of the CNN being represented (n_{out}).

The proposed initialization scheme is shown in Algorithm 4. First, the number of layers of a particle is randomly chosen to be between three and l_{max} (Algorithm 4, line 2). Second, the first layer of a particle is always chosen to be a convolutional layer with a randomly assigned number of filters and filters size (Algorithm 4, line 5), and the last layer is always chosen to be a fully-connected (FC) layer with its number of neurons equal to the number of outputs, n_{out} (Algorithm 4, line 7). Third, the remaining layers are chosen at random to be convolutional, average or max pooling, or fully-connected (Algorithm 4, lines 11 to 17). For last, if a fully-connected layer is added to the particle, all subsequent layers need to be fully-connected as well to produce a valid CNN architecture (Algorithm 4, lines 8, and 9).

Not shown in Algorithm 4 is a mechanism that eliminates pooling layers in excess. At the end of the initialization process, each particle goes into a *correction phase* in which the sizes of each layer's outputs are computed, and any pooling layer that produces an output size smaller than 4×4 is eliminated from the particle. This mechanism ensures that every particle always represents a valid CNN architecture.

There are two hyperparameters found in the convolutional layers that have fixed values for every layer of every particle. The first one is the *padding of inputs*, which is done by using zero-padding to make the size of the outputs equal to the size of the inputs, and the second one is the *convolutional filters' strides* being always equal to 1×1.

Algorithm 4: Swarm initialization in the proposed *psoCNN* ($InitializeSwarm()$) [91].

Input : Swarm size (N), maximum number of layers (l_{max}), maximum number of convolutional filters ($filters_{max}$), maximum convolutional filter size (k_{max}), maximum number of neurons in a FC layer (n_{max}), number of outputs (n_{out}).

Output: A set of N particles, $\mathbf{S} = \{P_1, ..., P_N\}$.

for $i = 1$ *to* N **do**
  $P_i.depth = rand(3, l_{max})$;
  for $j = 1$ *to* $P_i.depth$ **do**
    if $j == 1$ **then**
      $list\_layers[j] \leftarrow addConv(k_{max}, filters_{max})$;
    else if $j == P_i.depth$ **then**
      $list\_layers[j] \leftarrow addFC(n_{out})$;
    else if $list\_layers[j-1].type ==$ "$fully - connected$" **then**
      $list\_layers[j] \leftarrow addFC(n_{max})$;
    else
      $layer\_type \leftarrow rand(1, 3)$;
      if $layer\_type == 1$ **then**
        $list\_layers[j] \leftarrow addConv(k_{max}, filters_{max})$;
      else if $layer\_type == 2$ **then**
        $list\_layers[j] \leftarrow addPool()$;
      else
        $list\_layers[j] \leftarrow addFC(n_{max})$;
      end
    end
  end
  $P_i.list\_layers \leftarrow list\_layers$;
end
return $\mathbf{S} = \{P_1, ..., P_N\}$;

3.2.4 Computing the Difference Between Two Particles

To be able to compute the velocity and, subsequently, update the parameters of a particle, a novel operator is proposed to compute the difference between two particles symbolically. This procedure is illustrated in Figure 3.2, where two particles ($P1$ and $P2$) are compared. The first step is to separate the particles' convolutional and pooling layers (Conv/Pool) from their fully-connected (FC) layers, as illustrated in the top-right side of Figure 3.2, which are going to be compared independently of each other. The difference between particles is always computed with respect to the first one ($P1$), and it is computed by comparing the blocks of Conv/Pool and FC layers. For the Conv/Pool blocks, the difference is computed from left to right, while in the FC blocks, the difference is computed from right to left. Thus, if the layer type of $P1$ is equal to the layer type of $P2$, the difference is equal to zero. If the layer type of $P1$ is different from $P2$, the difference is equal to the layer from P1. If $P1$ has fewer layers than $P2$, the difference is equal to -1, which indicates to the velocity operator that any layer in that position should be eliminated. For last, if $P1$ has more layers than $P2$, the difference is equal to $+L$, where L represents the layer from $P1$ and indicates to the velocity operator that a layer should be added to this position. This process is summed up in the bottom part of Figure 3.2.

3.2.5 Particle's Velocity Computation

The velocity operator ($UpdateVelocity()$) makes use of the difference explained in the previous subsection. First, the difference between the global best particle ($gBest$) and the current particle (P) is computed ($gBest - P$). Then, the difference between the particle's personal best ($pBest$) and the current particle configuration is computed ($pBest - P$).

The output of the velocity operator is a list of layers chosen from $gBest - P$ or $pBest - P$ accordingly to a threshold value (C_g). A random number generator is used to perform this task. If the random number generated is smaller than C_g, the velocity operator will choose the difference from $gBest - P$. On the other hand, if the random number generated is greater than or equal to C_g, the velocity operator will choose the difference from $pBest - P$. The process is repeated for each layer represented by these two differences, as illustrated in Figure 3.3. Thus, the C_g parameters will control how much a particle will be similar to the global best or the personal best. If C_g is close to 1.0, the particle will be more similar to the $gBest$ after its updating

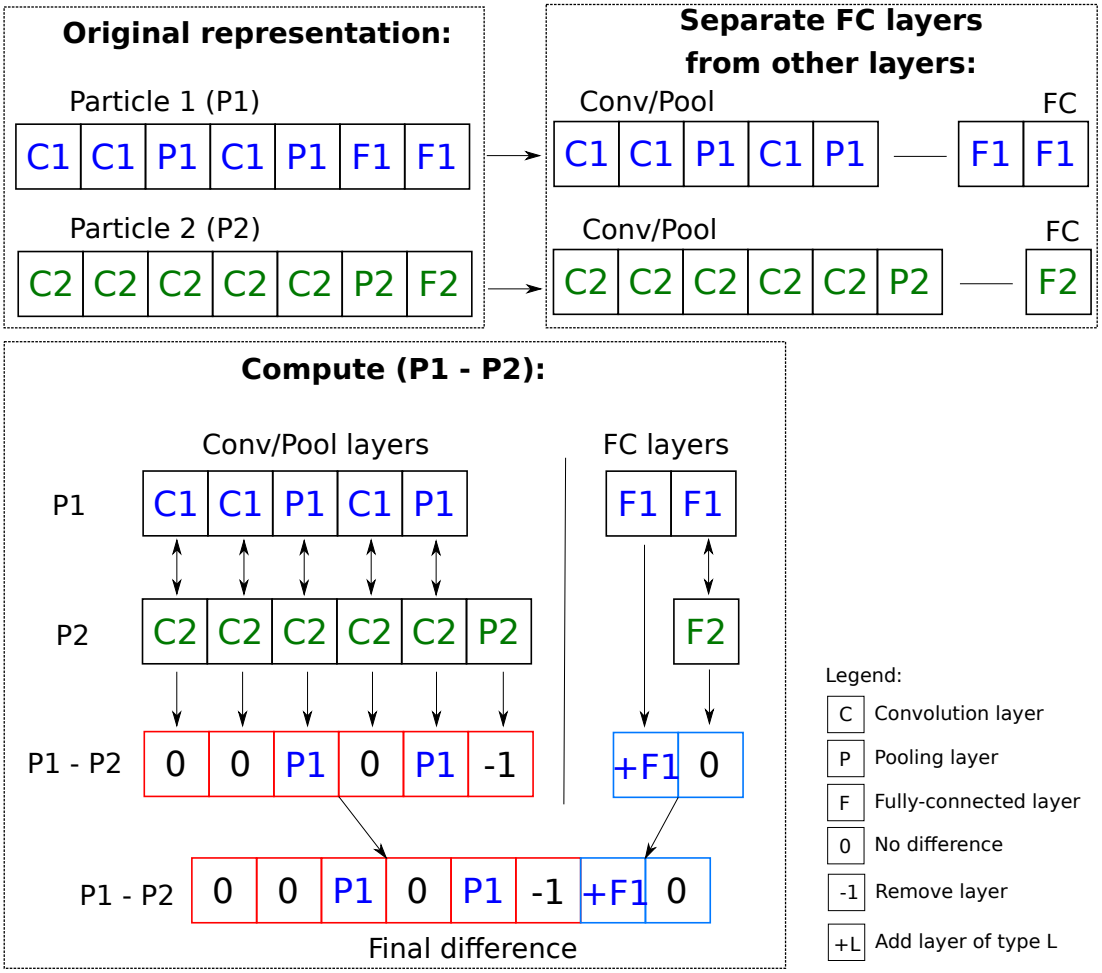


Figure 3.2: Computing the difference between two particles in the *psoCNN* [91].

process. The output from the velocity operator is then used to update the particle's parameters.

There is a special case when computing a particle's velocity that needs to be addressed independently. Close to the final iterations, the layers' types of a given particle can become similar to its $gBest$ and $pBest$, making the difference $gBest - P$ equal to $pBest - P$. In this case, the velocity operator will choose to use the hyperparameters from $gBest$ or $pBest$ accordingly to the parameter C_g, as illustrated in Figure 3.4.

3.2.6 Particle's Updating

After computing a particle's velocity, the proposed *psoCNN* will update it by using the $UpdateParticle()$ function. The updating procedure is illustrated in Figure 3.5, and it is performed by comparing the particle's velocity with its current configuration. The updating algorithm also treats blocks of Conv/Pool and FC layers separately, and it will replace the positions that are different from zero, as indicated by the velocity operator. Thus, the proposed difference and velocity operators allow particles to shrink, by eliminating layers or grow, by adding layers to a particle's architecture.

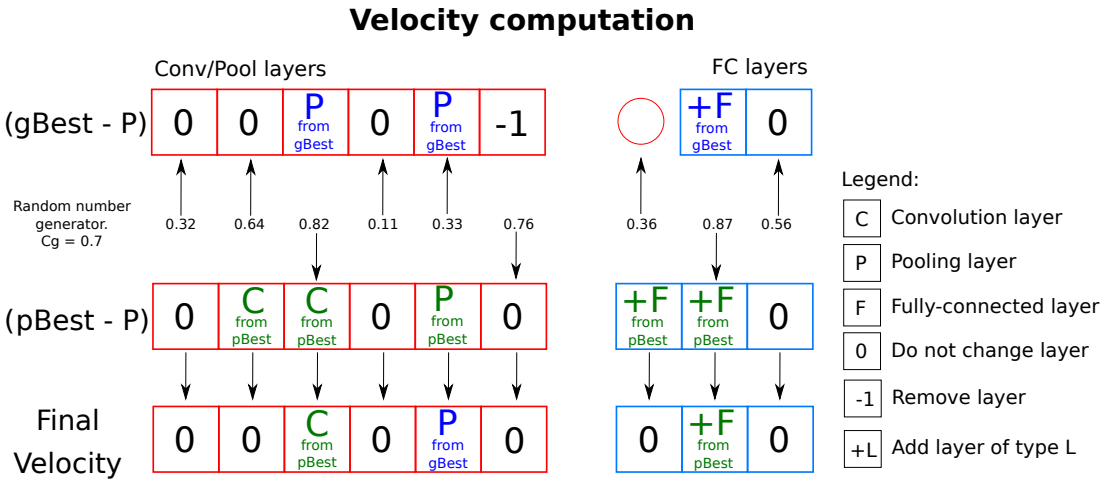


Figure 3.3: Computing the velocity of a single particle in *psoCNN* [91].

3.3 Experimental Design

In this section, the chosen datasets and peer competitors algorithms used to evaluate the proposed *psoCNN* algorithm, as well as a discussion about the algorithm's parameters are presented.

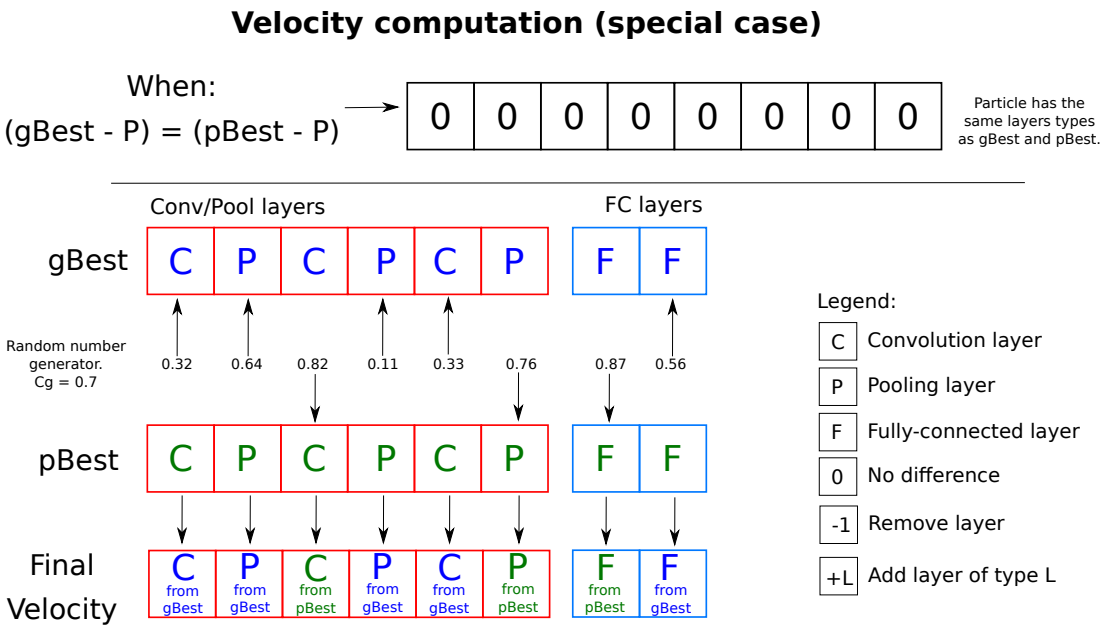


Figure 3.4: Computing the velocity of a single particle in *psoCNN* when *gBest* and *pBest* have the same layers types [91].

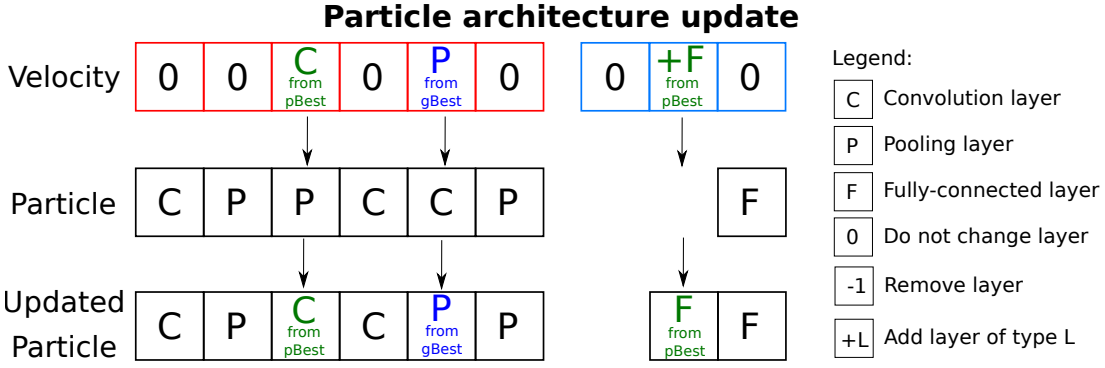


Figure 3.5: Updating a particle's architecture in *psoCNN* [91].

3.3.1 Datasets

Nine image classification datasets with publicly available results were chosen to evaluate the proposed *psoCNN* algorithm. The chosen datasets were also used by other researchers, which makes it easier to compare the *psoCNN* results with the ones obtained by peer competitors. They are illustrated in Figure 3.6 and explained in detail in the following.

The first five datasets shown in Figure 3.6, from top to bottom, are based on the Modified National Institute of Standards and Technology (*MNIST*) dataset created by LeCun *et al.* [4]. They are the original *MNIST*, the *MNIST* with rotated digits (*MNIST-RD*), the *MNIST* with random noise as background (*MNIST-RB*), the *MNIST* with background images (*MNIST-BI*), and the *MNIST* with rotated digits and background images (*MNIST-RD+BI*) datasets. They all have grayscale images with sizes of 28×28 pixels, and a single-digit handwritten numeral between zero and nine (ten classes) in the center of the image. The original *MNIST* dataset only contains images with a black background with white numbers in the foreground, with 50,000 images used as the training set and 10,000 used as the test set. The other four datasets based on *MNIST* were created by Larochelle *et al.* [98]. They use rotated digits with a variety of backgrounds, and are considered more challenging than the original one because models need to learn to ignore the irrelevant information presented in the images. These *MNIST* variations contain only 12,000 images in the training set and 50,000 images in the test set.

The *Rectangles*, *Rectangles-I*, and *Convex* datasets were also created by Larochelle *et al.* [98], and they are used to test models' ability to learn polygonal shapes. The *Rectangles* and *Rectangles-I* contain grayscale images with 28×28 pixels in size with a single rectangle located somewhere in the image. A CNN model would need to learn

Table 3.1: Overview of the datasets used to evaluate the *psoCNN* [91].

Dataset	**Input size**	**# classes**	**# training**	**# test**
MNIST	$28 \times 28 \times 1$	10	60,000	10,000
MNIST-RD	$28 \times 28 \times 1$	10	12,000	50,000
MNIST-RB	$28 \times 28 \times 1$	10	12,000	50,000
MNIST-BI	$28 \times 28 \times 1$	10	12,000	50,000
MNIST-RD+BI	$28 \times 28 \times 1$	10	12,000	50,000
Rectangles	$28 \times 28 \times 1$	2	1,200	50,000
Rectangle-I	$28 \times 28 \times 1$	2	12,000	50,000
Convex	$28 \times 28 \times 1$	2	8,000	50,000
MNIST-Fashion	$28 \times 28 \times 1$	10	60,000	10,000

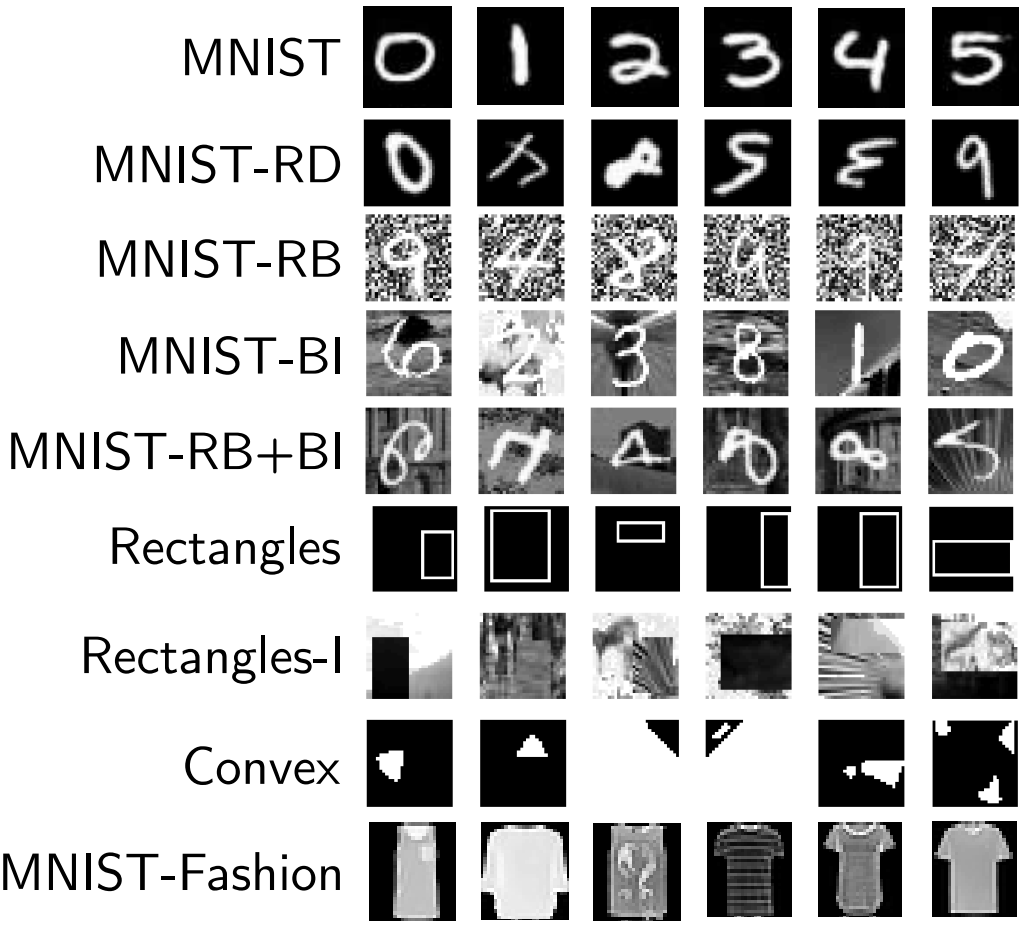


Figure 3.6: Image classification datasets used to evaluate the proposed *psoCNN* algorithm [91].

two classes: if the width of the rectangle is larger than the height, and vice-versa. The difference from the *Rectangles-I* to the *Rectangles* dataset is that the images in the *Rectangles-I* contain patches of random images. The *Rectangles* dataset contains 1,200 training images and 50,000 test images, while the *Rectangles-I* contains 12,000 training images and 50,000 test images. The *Convex* dataset contains black and white images of sizes equal to 28×28 with some polygonal shape. There are only two classes in this dataset: if the shape is convex or not. There are 8,000 training images and 50,000 test images in the *Convex* dataset.

The last dataset in the list is the *MNIST-Fashion* created by Xiao *et al.* [99]. It has the same number of images for training and testing and input sizes as the original *MNIST*, but, instead of numeric digits, it contains small images of fashion garments. It has a total of ten classes: *t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag*, and *ankle boot*.

An overview of all chosen datasets, with their input sizes, their number of classes, and the number of images used for training and testing, can be found in Table 3.1.

3.3.2 Peer Competitors' Algorithms

The two main peer competitors' algorithms to the proposed *psoCNN* are the *evoCNN* developed by Sun *et al.* [87] and the *IPPSO* developed by Wang *et al.* [96] because these two algorithms generate similar CNN architectures to the proposed *psoCNN* for image classification by using evolution or particle swarm optimization.

All other peer competitors' algorithms chosen to be compared with *psoCNN* are not models generated automatically. The following is the list of the peer competitors models used for comparison with the *psoCNN*:

- The original CNN models (LeNet-1, LeNet-4, and LeNet-5) created by LeCun

et al. [44].

- The recurrent CNN model (RCNN) created by Liang and Hu [100].
- A CNN architecture with DropConnect regularization developed by Wan *et al.* [101].
- The contractive autoencoders (CAE) developed by Rifai *et al.* [102].
- The models developed by Chan *et al.* based on principal component analysis (PCANet, RandNet, and LDANet) [103].
- For last, some recent state-of-the-art CNN architectures: AlexNet [57], VGG16 [23], GloogLeNet [24], and MobileNet [104].

The chosen peer competitors' algorithms give a reasonable coverage of currently available models using CNN and other machine learning algorithms to verify the performance of the proposed *psoCNN*.

3.3.3 Algorithm Parameters

The parameters used to test the proposed *psoCNN* are shown in Table 3.2. There are three categories of parameters: the ones related to the particle swarm optimization part of the algorithm, the ones related to the initialization of the swarm, and the ones related to the training phase (evaluation) of each particle.

The parameters related to the particle swarm optimization part of the proposed *psoCNN* are the number of iterations ($iter_{max}$) chosen to be equal to 10, the swarm size (N) equal to 20, and the C_g parameter equal to 0.5. The number of iterations is the stopping criteria of the proposed algorithm. A large number of iterations increases the probability of the algorithm finding good solutions, but it takes more time to run.

Table 3.2: *psoCNN* parameters used for evaluation [91].

Description	Value
Particle swarm optimization	
Number of iterations ($iter_{max}$)	10
Swarm size (N)	20
Cg	0.5
Swarm initialization	
Maximum number of convolutional filters ($filters_{max}$)	256
Maximum convolutional filter size (k_{max})	7×7
Maximum number of neurons in a FC layer (n_{max})	300
Maximum number of layers (l_{max})	20
Particle training	
Number of epochs for particle evaluation (e_{train})	1
Number of epochs for the global best (e_{test})	100

The swarm size defines how many particles are used by the algorithm. A large number of particles ensures better coverage of possible CNN architectures, but it also requires a lot more computational power to evaluate the entire swarm. The velocity operator uses the C_g parameter to control how fast a particle will approach the global best particle ($gBest$) in the swarm. A C_g close to 1.0 will make particles approach the $gBest$ very fast, leading to premature convergence.

The parameters related to the swarm initialization are the maximum number of convolutional filters ($filters_{max}$), convolutional filter size (k_{max}), number of neurons in a fully-connected (FC) layer (n_{max}), and number of layers (l_{max}). For each convolutional layer, its actual number of filters is chosen at random during the initialization to be between three and $filter_{max}$. The actual size of each convolutional filter (k_{max}) is chosen at random to be between 3×3 and 7×7. For the fully-connected layers, they are initialized with 1 to 300 neurons, also chosen at random. The maximum number of layers (l_{max}) of a particle is randomly chosen to be between 3 and 20. The values used during the initialization of the swarm controls how many CNN architectures are reachable by the algorithm. Thus, they need to be chosen according to the maximum

CNN architecture size allowed by the available computational equipment.

The last group of parameters is related to the evaluation of each particle during the searching process and the training of the final global best particle. During the process of particle evaluation, only a single epoch over the entire dataset is used to train each particle (e_{train}). After that, the particle's training accuracy is used to measure its quality. Better particles will have higher training accuracy. Each particle's training is also performed by using a 50% dropout of the neurons in the FC layers and batch-normalization to reduce the effects of overfitting. The final global best particle found by the algorithms is then trained (e_{test}) for a total of 100 epochs before reporting the results presented in the next section.

3.4 Experimental Results and Discussions

In this section, the results obtained by the proposed *psoCNN* is presented, compared with peer competitors' algorithms, and discussed in detail. Moreover, to ensure the statistical significance of the results presented here, they were obtained by running the proposed algorithm 30 times and using the test set of each presented datasets.

3.4.1 Results

The results for the *MNIST*, *MNIST-RD*, *MNIST-RB*, *MNIST-BI*, *MNIST-RD+BI*, *Rectangles*, *Rectangles-I*, and *Convex* datasets can be visualized in Table 3.3 and Figure 3.7, where the test error for each dataset is reported as a percentage. The *plus* symbol (+) next to each value represents the instances where the results of the proposed *psoCNN* are better than results from the peer competitors' algorithms in a statistically meaningful sense, while the *minus* symbol (−) represents the instances where the results of the proposed algorithm are worse than results from the other

algorithms. The (best) and (mean) indications shown in Table 3.3 represents the best and average results found by *psoCNN* in 30 independent runs. For the *MNIST-RD*, *MNIST-RB*, *MNIST-BI*, *MNIST-RD+BI*, *Rectangles-I*, and *Convex* datasets, the best CNN architectures found by *psoCNN* have test errors of 3.58%, 1.79%, 1.90%, 14.28%, 2.22%, and 1.7%, respectively, which are better than all peer competitors algorithms. Only for the *MNIST* and *Rectangles* datasets that the best CNN architectures found by *psoCNN* does not have the best test errors, 0.32% for *MNIST*, and 0.03% for *Rectangles*. However, on *MNIST*, *psoCNN* still has better results than 12 of the peer competitors, losing to only two of them (RCNN and DropConnect), while, on *Rectangles*, *psoCNN* only has worse results than those of the *evoCNN* algorithm.

The results for the *MNIST-Fashion* dataset are shown in Table 3.4 and Figure 3.8. Only for this dataset, the proposed *psoCNN* is tested *with* and *without* dropout and batch-normalization. The tests that do not use dropout and batch-normalization are indicated as ***psoCNN - dropout - BN*** in Table 3.4, while the tests that do use them are indicated as ***psoCNN + dropout + BN***. The best CNN architecture found by the proposed *psoCNN* has a test error of 5.53%, and the average test error on 30 runs of the *psoCNN* is 5.90%, which is only worse than two of the chosen peer competing algorithms: *evoCNN* and *MobileNet*.

3.4.2 Discussions

The results presented here were performed with a total of nine challenging image classification datasets, and comparisons were made to a total of 20 peer competitors' algorithms. The proposed *psoCNN* algorithm is capable of finding CNN architectures with the best results in six out of the nine datasets. The datasets that *psoCNN* did not produced the best results were the *MNIST*, *Rectangles*, and *MNIST-Fashion*.

Table 3.3: Test results on the *MNIST*, *MNIST-RD*, *MNIST-RB*, *MNIST-BI*, *MNIST-RD+BI*, *Rectangles*, *Rectangles-I*, and *Convex* datasets [91].

Model	MNIST	MNIST-RD	MNIST-RB	MNIST-BI	MNIST-RD+BI	Rectangles	Rectangles-I	Convex
LeNet-1 [44]	1.7% (+)	-	-	-	-	-	-	-
LeNet-4 [44]	1.1% (+)	-	-	-	-	-	-	-
LeNet-5 [44]	0.95% (+)	-	-	-	-	-	-	-
RCNN [100]	0.31% (-)	-	-	-	-	-	-	-
DropConnect [101]	0.21% (-)	-	-	-	-	-	-	-
CAE-1 [102]	2.83% (+)	11.59% (+)	13.57% (+)	16.7% (+)	48.10% (+)	1.48% (+)	21.86% (+)	-
CAE-2 [102]	2.48% (+)	9.66% (+)	10.90% (+)	15.5% (+)	45.23% (+)	1.21% (+)	21.54% (+)	-
PCANet-2 [103]	1.06%(+)	8.52% (+)	6.85% (+)	11.55% (+)	35.86% (+)	0.49% (+)	13.39% (+)	4.19% (+)
RandNet-2 [103]	1.27% (+)	8.47% (+)	13.47% (+)	11.65% (+)	43.69% (+)	0.09% (+)	17.00% (+)	5.45% (+)
LDANet-2 [103]	1.40% (+)	4.52% (+)	6.81% (+)	12.42% (+)	38.54% (+)	0.14% (+)	16.20% (+)	7.22% (+)
EvoCNN (best) [87]	1.18% (+)	5.22% (+)	2.8% (+)	4.53% (+)	35.03% (+)	0.01% (-)	5.03% (+)	4.82% (+)
EvoCNN (mean) [87]	1.28% (+)	5.46% (+)	3.59% (+)	4.62% (+)	37.38% (+)	0.01% (-)	5.97% (+)	5.39% (+)
IPPSO (best) [96]	1.13% (+)	-	-	-	34.50% (+)	-	-	8.48% (+)
IPPSO (mean) [96]	1.21% (+)	-	-	-	33% (+)	-	-	12.06% (+)
psoCNN (best)	0.32%	3.58%	1.79%	1.90%	14.28%	0.03%	2.22%	1.7%
psoCNN (mean)	0.44%	6.42%	2.53%	2.40%	20.98%	0.34%	3.94%	3.9%

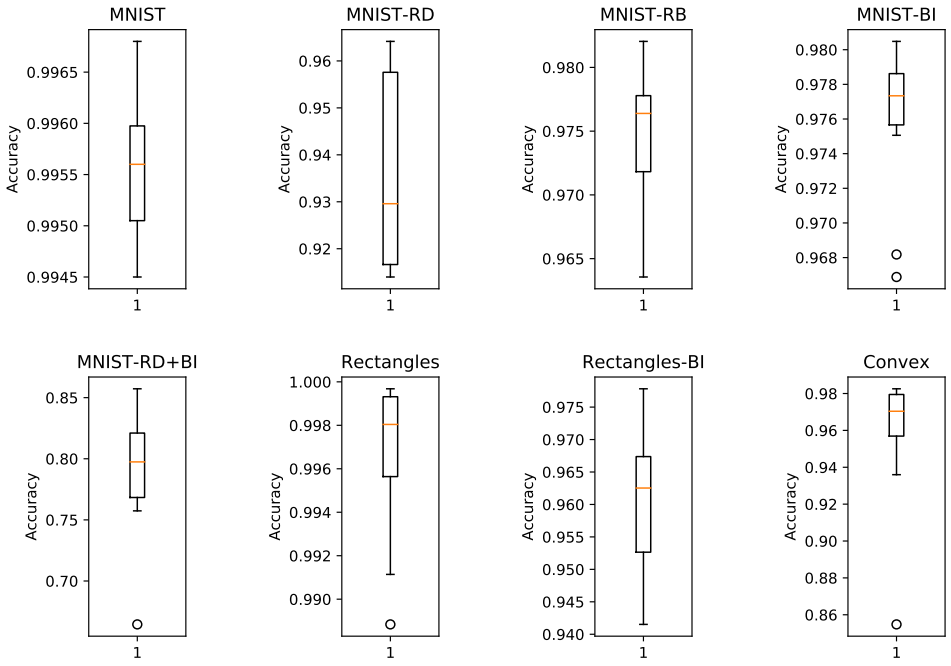


Figure 3.7: Boxplots of the results obtained by *psoCNN* for the *MNIST*, *MNIST-RD*, *MNIST-RB*, *MNIST-BI*, *MNIST-RD+BI*, *Rectangles*, *Rectangles-I*, and *Convex* datasets [91].

Table 3.4: Test results on the *MNIST-Fashion* dataset [91].

Model	**Test error**	**# parameters**
Human performance [1]	16.5% (+)	-
MLP 256-128-100 [1]	11.67% (+)	3M
GRU + SVM [1]	11.2% (+)	-
HOG + SVM [1]	7.4% (+)	-
AlexNet [1]	10.1% (+)	62.3M
3CONV + 3FC [1]	6.6% (+)	500k
VGG16 [1]	6.5% (+)	26M
GoogLeNet [1]	6.3% (+)	23M
evoCNN (best) [87]	5.47% (-)	6.68M
evoCNN (mean) [87]	7.28% (+)	6.52M
MobileNet [1]	5% (-)	4M
psoCNN − dropout − BN (best)	8.1% (+)	1.4M
psoCNN − dropout − BN (mean)	9.15% (+)	1.8M
psoCNN + dropout + BN (best)	5.53%	2.32M
psoCNN + dropout + BN (mean)	5.90%	2.5M

[1] https://github.com/zalandoresearch/fashion-mnist

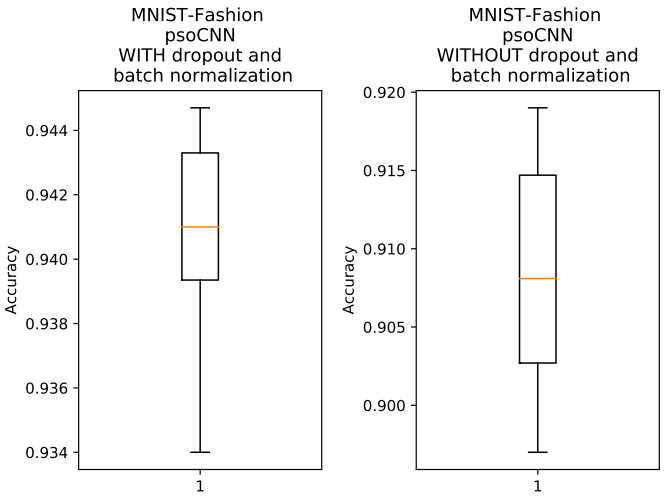


Figure 3.8: Boxplots of the results obtained by *psoCNN* for the *MNIST-Fashion* dataset [91].

However, *psoCNN* is always one of the top 3 models for these datasets.

Another important characteristic of the CNN architectures produced by *psoCNN* is that they have fewer parameters than most peer competitors' models. The number of parameters can be verified in Table 3.4 for the *MNIST-Fashion* dataset. The best CNN architecture found by *psoCNN* contains a total of 2.32 million parameters, which is almost half of the number of parameters of the most accurate model (MobileNet). It also found CNN architectures with less parameters than the ones from *evoCNN*, the most similar algorithms to the proposed *psoCNN* due to its use of population-based evolution.

The evolution of the global best particle ($gBest$) for ten independents runs of

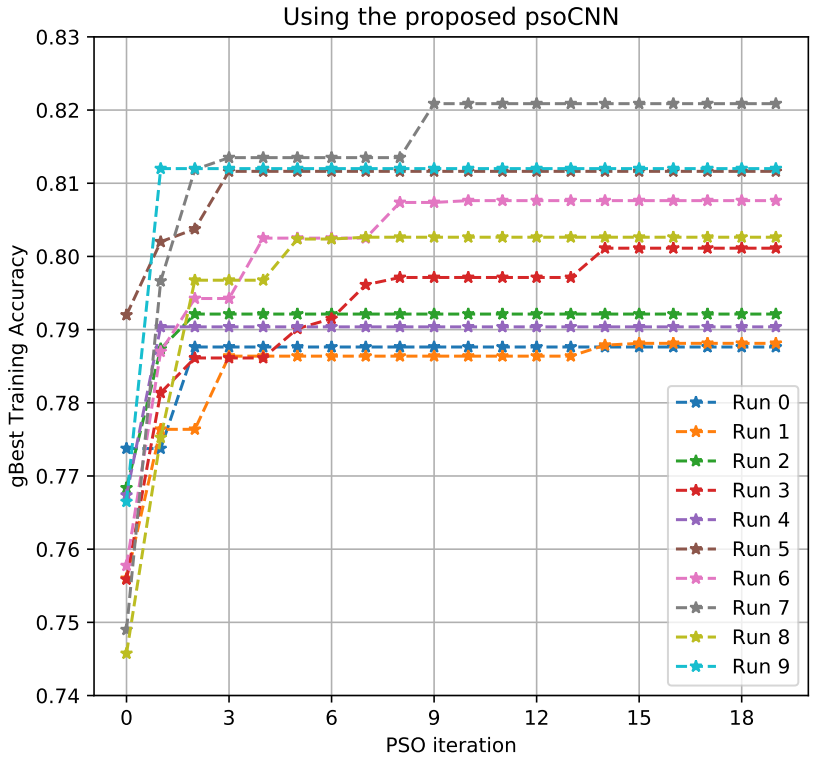


Figure 3.9: Evolution of the global best particle ($gBest$) in the *Convex* dataset for 10 independent runs of *psoCNN* [91].

the *psoCNN* in the *Convex* dataset is showed in Figure 3.9. It shows that the algorithm is capable of improving the *gBest* accuracy over time and that it is not merely performing random mutations. Furthermore, Figure 3.10 shows the evolution of the *gBest* particle using different numbers of epochs for particle evaluation in the *Convex* dataset. These results show that the proposed *psoCNN* algorithm is capable of improving the *gBest*, even using a large number of epochs for evaluation. Although using a single epoch for particle evaluation does not produce the highest training accuracy compared to five or ten epochs, the algorithm can easily find better *gBests* in this situation, and the final *gBest* particle is always retrained at the end of the process for a large number of epochs, which improves its accuracy eventually. Thus, it is unnecessary to evaluate particles for a large number of epochs.

The best CNN architectures found by *psoCNN* are shown in Table 3.5. One exciting aspect of all these networks is that they only have one single fully-connected (FC) layer at the end of their architectures. Springenberg *et al.* [105] showed that CNN architectures with only a single FC layer produce better results than the ones using multiple FC layers. These results have shown that *psoCNN* was able to find this type of CNN architectures by itself without using any prior knowledge.

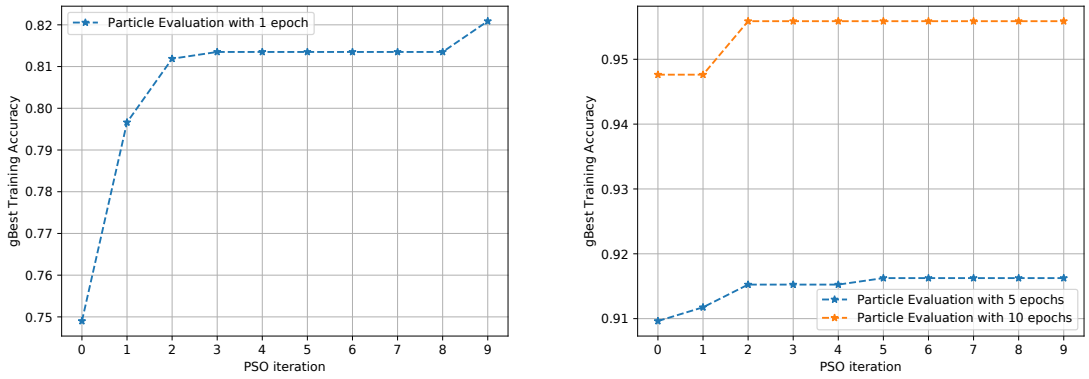


Figure 3.10: Effect of the number of epochs used during each particle evaluation on the Convex dataset [91].

Table 3.5: Best CNN architectures found by psoCNN on each dataset [91].

Dataset	Layer	Parameters
MNIST	Convolution	filter size: 4×4; number of filters: 77
	Convolution	filter size: 6×6; number of filters: 189
	Convolution	filter size: 5×5; number of filters: 91
	Convolution	filter size: 5×5; number of filters: 185
	Convolution	filter size: 6×6; number of filters: 244
	Average Pooling	filter size: 3×3; strides: 2×2
	Fully-Connected	output neurons: 10
MNIST-RD	Convolution	filter size: 6×6; number of filters: 189
	Convolution	filter size: 5×5; number of filters: 182
	Convolution	filter size: 6×6; number of filters: 236
	Convolution	filter size: 5×5; number of filters: 210
	Average Pooling	filter size: 3×3; strides: 2×2
	Fully-Connected	output neurons: 10
MNIST-RB	Convolution	filter size: 5×5; number of filters: 246
	Convolution	filter size: 3×3; number of filters: 216
	Convolution	filter size: 5×5; number of filters: 217
	Convolution	filter size: 5×5; number of filters: 156
	Convolution	filter size: 6×6; number of filters: 204
	Average Pooling	filter size: 3×3; strides: 2×2
	Fully-Connected	output neurons: 10
MNIST-BI	Convolution	filter size: 3×3; number of filters: 67
	Convolution	filter size: 4×4; number of filters: 126
	Convolution	filter size: 6×6; number of filters: 159
	Convolution	filter size: 3×3; number of filters: 252
	Convolution	filter size: 6×6; number of filters: 202
	Convolution	filter size: 6×6; number of filters: 202
	Fully-Connected	output neurons: 10
MNIST-RD+BI	Convolution	filter size: 4×4; number of filters: 247
	Convolution	filter size: 5×5; number of filters: 208
	Convolution	filter size: 4×4; number of filters: 122
	Convolution	filter size: 5×5; number of filters: 64
	Convolution	filter size: 4×4; number of filters: 144
	Convolution	filter size: 5×5; number of filters: 171
	Average Pooling	filter size: 3×3; strides: 2×2
	Fully-Connected	output neurons: 10
Rectangles	Convolution	filter size: 5×5; number of filters: 139
	Convolution	filter size: 6×6; number of filters: 113
	Convolution	filter size: 5×5; number of filters: 226
	Fully-Connected	output neurons: 2
Rectangles-I	Convolution	filter size: 5×5; number of filters: 127
	Convolution	filter size: 4×4; number of filters: 162
	Convolution	filter size: 6×6; number of filters: 212
	Convolution	filter size: 6×6; number of filters: 212
	Max Pooling	filter size: 3×3; strides: 2×2
	Fully-Connected	output neurons: 2
Convex	Convolution	filter size: 6×6; number of filters: 118
	Convolution	filter size: 4×4; number of filters: 224
	Convolution	filter size: 5×5; number of filters: 243
	Convolution	filter size: 3×3; number of filters: 71
	Convolution	filter size: 5×5; number of filters: 35
	Convolution	filter size: 5×5; number of filters: 160
	Fully-Connected	output neurons: 2

3.5 Final Remarks

In this chapter, a novel algorithm for CNN architecture searching is presented. The proposed algorithm is built on a particle swarm optimization framework with modified velocity and updating operators, allowing the use of a symbolic encoding scheme for CNN architectures, which is called here as *psoCNN*.

The results obtained show that *psoCNN* is capable of finding meaningful CNN architectures for a given image classification dataset. The results are also competitive with other similar algorithms. The results show that *psoCNN* can even find CNN architectures with fewer parameters than other algorithms and models can.

For last, the main argument for the use of Particle Swarm Optimization instead of purely evolutionary approaches is that the former converges faster than the latter. Indeed, using indirect measurements, *psoCNN* converged faster than *evoCNN*. Using a laptop Nvidia GTX 960M GPU, *psoCNN* took an average of 15.22 hours to find meaningful CNN architectures on the *MNIST* dataset, while *evoCNN* is reported to take up to four days using two powerful Nvidia GTX1080 GPUs.

CHAPTER IV

DEEP NEURAL NETWORK ARCHITECTURE PRUNING WITH EVOLUTION STRATEGIES

Another challenge faced by researchers and experts when using Deep Neural Networks (DNNs) is their high computational complexity. Most DNNs require lots of computational power for training and even deployment on consumer hardware. In this chapter, an algorithm for DNN architecture pruning based on evolution strategy (ES) is presented, which drastically reduces the amount of computational power needed to run the state-of-the-art models[1].

4.1 Introduction

Currently, deep neural networks are prevalent in the field of pattern recognition and computer vision. However, beyond the main problem of designing a meaningful DNN architecture, there is also another challenging problem in the use of such models by researchers and experts from other fields, which is their high computational cost. The development of DNN-based solutions requires two stages: training and inference. The training stage is the most computationally intensive of both, where the parameters of a DNN model is iteratively modified to learn the underlying structure of a specific database. The successful training of DNN models requires vast amounts of data, and multiple passes through the whole data, which can take months to finish,

[1]The present chapter is partially based on my submitted work, which is currently under review. A preprint of the submitted work can be found in [106].

even when using multiple powerful parallel processors such as general-purpose graphical processing units (GPUs). The inference phase or deployment phase is used once a well-suitable DNN model is found and fully trained. It consists of the deployment of the DNN model for use by consumers. Although, after deployment, a DNN model is only used when needed by the consumer, there are still many applications where real-time results are expected from the model, which is a challenging task given the massive number of parameters found in state-of-the-art DNN models.

Researchers developed many different approaches to deal with the high computational complexity required by DNNs. One can tackle this problem by developing specialized hardware solutions capable of running DNN models in real-time, such as FPGA-based solutions [107, 108] and low-power mobile GPUs [109]. However, software-wise solutions that can be used in the current hardware are more desirable by researchers than hardware-wise ones. The quantization of the number of bits needed to represent each parameter of a DNN model is one such solution [51] with the extreme case of using only one bit to represent a single parameter in the DNN model being another approach [52]. Some researchers developed approaches to reduce the computational complexity of mathematical operation performed within a DNN model [54, 104, 110]. However, such solutions do not address a common question: would it be possible to find DNNs with fewer parameters, but with similar accuracy to hand-crafted state-of-the-art ones? In other words, hand-crafted DNNs may contain too many parameters in excess, which is the central hypothesis used when performing DNN architecture pruning.

Convolutional Filter Pruning is the most used approach to reduce the computational complexity of DNNs and does not need the usage of any specialized software or hardware to achieve it. Since most of the processing of any given DNN comes from the convolutional layers instead of the fully-connected (FC) layers [55] and recent state-

of-the-art DNNs are not using as many FC layers as before [105], the elimination of neurons in FC layers has little to no effect in a DNN computational complexity. Thus, in this approach, only convolutional filters are selected and eliminated. In the literature, the filter selection is performed by using some statistical information about the current layer being pruned, such as the mean activation of each filter [53], the average percentage of zeros of each filter [50], and others. Filters can also be eliminated at random with no effect on the accuracy of the final DNN architecture [90], showing that randomized heuristics can produce good results during the pruning process. More details about Filter Pruning can be found in Subsection 2.3.2.

Most pruning algorithms use the percentage of parameters that one desires to eliminate from a DNN architecture. For example, if 25% of the parameters in a DNN need to be eliminated, the pruning algorithm needs to find an acceptable combination of convolutional filters that will produce the minimum degradation in the DNN accuracy capabilities. Thus, DNN architecture pruning can be seen as a combinatorial multi-objective optimization problem (MOP), where one would like to eliminate parameters in a DNN as much as possible while maintaining the DNN accuracy as far as possible. These two objectives are also conflicting. It is easy to see that the elimination of too many parameters will hurt the performance of a DNN model.

Therefore, in this chapter, a Convolutional Filter Pruning algorithm is presented, capable of using a random heuristic to choose which filters to eliminate, and of taking advantage of the multi-objective optimization nature of the problem to produce a set of candidate solutions. The proposed algorithm is called *DeepPruningES*, and it uses a modified multi-objective Evolution Strategy (ES) algorithm to prune DNN architectures. ES is an Evolutionary Algorithm (EA) that relies heavily on random changes of candidate solutions and is ideal for discrete combinatorial optimization

problems [70].

These are the main contributions of the proposed algorithm:

- A pruning algorithm that does not require prior knowledge about the number of parameters that needs to be eliminated.
- State-of-the-art DNN architectures can be pruned: convolutional neural networks (CNNs), residual neural networks (ResNets), and densely connected neural networks (DenseNets).
- The algorithm can efficiently find three pruned DNN models with different trade-offs between computational complexity and accuracy.

DeepPruningES is a population-based algorithm capable of pruning multiple types of DNN architectures and finding three solutions that can be used by a decision-maker accordingly to his or her needs. Specifically, the algorithm finds one solution with the best accuracy, but with the highest computational complexity, called *boundary heavy*, one solution with the smallest computational complexity, but with the worst accuracy, called *boundary light*, and, finally, one solution with the best trade-off between accuracy and computational complexity, called *knee*.

In the following sections, the proposed algorithm and the results obtained in six state-of-the-art, competing DNN models are presented in detail.

4.2 Proposed Algorithm

The general framework of the proposed *DeepPruningES* can be seen in Algorithm 5. The proposed algorithm performs DNN architecture pruning by eliminating convolutional filters from multiple layers. It is based on the *plus* version of the Evolution

Strategy (ES) algorithm, where the best individuals in the population are kept indefinitely. Specifically, *DeepPruningES* is a $(3+\lambda)-ES$ algorithm, where three solutions are always selected at every generation and are randomly mutated until λ offspring are generated. At the end of the pruning process, the algorithm returns three pruned DNN models: the *boundary heavy* solution, the *boundary light* solution, and the *knee* solution.

The following is a list of inputs used by the proposed *DeepPruningES*:

- The number of offspring that will be produced at every generation (λ_{size}).
- The maximum number of generation (gen), which is the only stopping criteria used by the algorithm.
- The mutation probability (p_m) of changing a single position in the genome of a candidate solution.
- The original DNN model that will be pruned (dnn).
- The number of epochs that a candidate solution will be retrained before its quality is evaluated (e_{eval}).
- The learning rate used during the evaluation of a candidate solution (α_{eval}).
- The number of epochs that the best solutions will be retrained or fine-tuned before saving them on disk (e_{fine}).
- For last, the learning rate used to retrain the best solutions found (α_{fine}).

Thus, there are five components in the proposed *DeepPruningES*, which are the initialization of the population, knee and boundary selection, offspring generation,

Algorithm 5: Proposed *DeepPruningES* [106]

Input : Offspring size (λ_{size}), maximum number of generations (gen), mutation probability (p_m), original DNN model (dnn), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}).

Output: Three DNN models: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

$\boldsymbol{\mu}, \boldsymbol{\lambda} \leftarrow$ *Initialize Population*(λ_{size}, dnn);
for g = *1 to gen* **do**
    $\mathbf{P} \leftarrow \boldsymbol{\mu} + \boldsymbol{\lambda}$
    $\boldsymbol{\mu} \leftarrow$ *Knee Boundary Selection*($\mathbf{P}$, dnn, e_{eval}, α_{eval});
    $\boldsymbol{\lambda} \leftarrow$ *Offspring Generation*($\boldsymbol{\mu}$, λ_{size}, p_m, dnn);
end
Fine-tuning($\boldsymbol{\mu}$, dnn, e_{fine}, α_{fine});
return $\mu.knee$, $\mu.heavy$, $\mu.light$;

fine-tuning of the best solutions, and a binary representation used to identify filters that will be eliminated. These components and the chosen representation are discussed in the following.

4.2.1 Filter Representation Scheme

DeepPruningES prunes DNN architectures by eliminating convolutional filters. Hence, the individuals' parameters used by the proposed algorithm only contains the representation of which filters should be discarded and which ones should be kept. This representation can be easily accomplished with the use of binary strings, where 1's represent the active filters, and 0's represent the inactive ones. Moreover, each convolutional filter in a given DNN architecture is represented by its physical location in this binary string.

As stated before, the proposed algorithm is capable of pruning CNNs, ResNets, and DenseNets. However, for each type of DNN architectures, a slightly different

representation scheme is used due to its connectivity pattern. The CNN architectures are represented by using only a single binary string, and it follows the logic explained before. The representation of CNN architectures is illustrated in Figures 4.1 and 4.2, where a three-layer CNN with a total of 32 convolutional filters is encoded using a 32-bit string.

The main challenge to represent ResNets is the fact that every shortcut connection is added with the output of a convolutional layer. These two entities are four-dimensional tensors, where the output tensor of a given convolutional layer is added with a tensor coming from a shortcut connection. These two tensors need to have the same dimensions due to the addition operation. Thus, if a single filter is eliminated from a shortcut connection, a filter needs to be also eliminated in the convolutional layer immediately before the shortcut connection. Moreover, ResNets uses blocks of shortcut connections with an equal number of filters. This connectivity pattern means that the elimination of a filter in one shortcut connection needs to be replicated in all subsequent shortcut connections.

To address this challenge, the proposed representation for ResNets uses two binary strings, as illustrated in Figure 4.3. One binary string, located on the left-hand side of Figure 4.3, is used to represent only the filters from convolutional layers located after the shortcut connection, which are represented in blue color and with a small vertical bar on the right side of the box. The other binary string, illustrated on the right-hand side of Figure 4.3, represents the filters from the shortcut connections. Thus, multiple convolutional filters are represented by one binary string. For example, each layer in green, the layers with small bars on the right side, has 16 filters with a total of 64 filters, but they are all represented with 16 bits because the elimination of one filter in one layer requires the elimination of one filter in all other layers.

DenseNets does not present the same challenge as ResNets because, although

they have multiple connections from multiple layers, they concatenate filters instead of adding the filters values in one single tensor. Concatenation increases the number of incoming filters in each layer, which is known as the *growth rate*. The proposed representation for DenseNets also uses two binary strings, and it is illustrated in Figure 4.4. However, one of the binary strings is only used in the DenseNet's bottleneck version, which uses a convolutional layer to reduce the number of input feature maps to the next layer. This version is also the one used in the experiments because it has better performance than the non-bottleneck one. The other binary string is used to represent the filters from the convolutional layers directly connected to the other layers. Thus, similar to the CNN representation, in the proposed DenseNet representation, each filter in the DNN architectures is represented by one bit in one of the

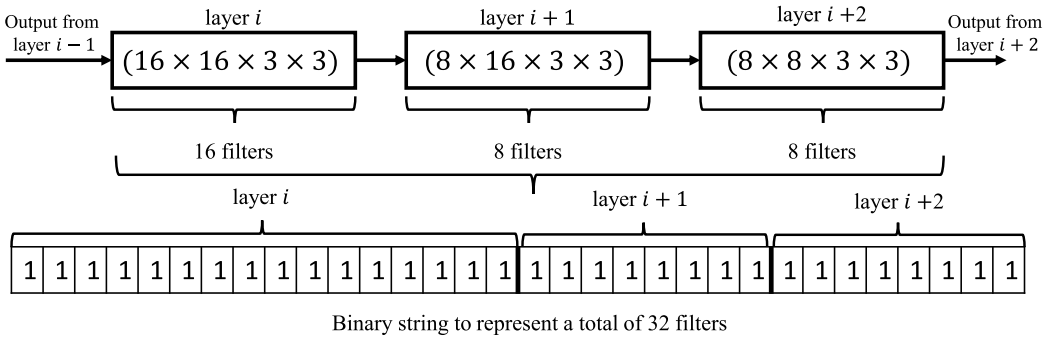


Figure 4.1: Representation of a three-layer CNN architecture on *DeepPruningES* [106].

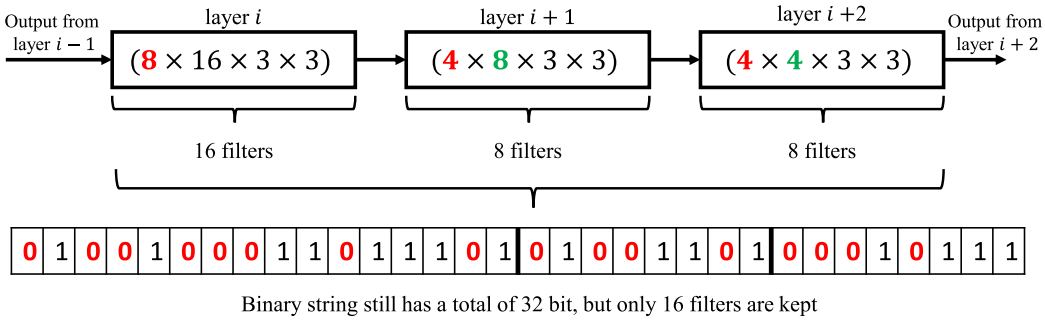


Figure 4.2: Representation of a three-layer CNN architecture on *DeepPruningES* where half of the filters are pruned [106].

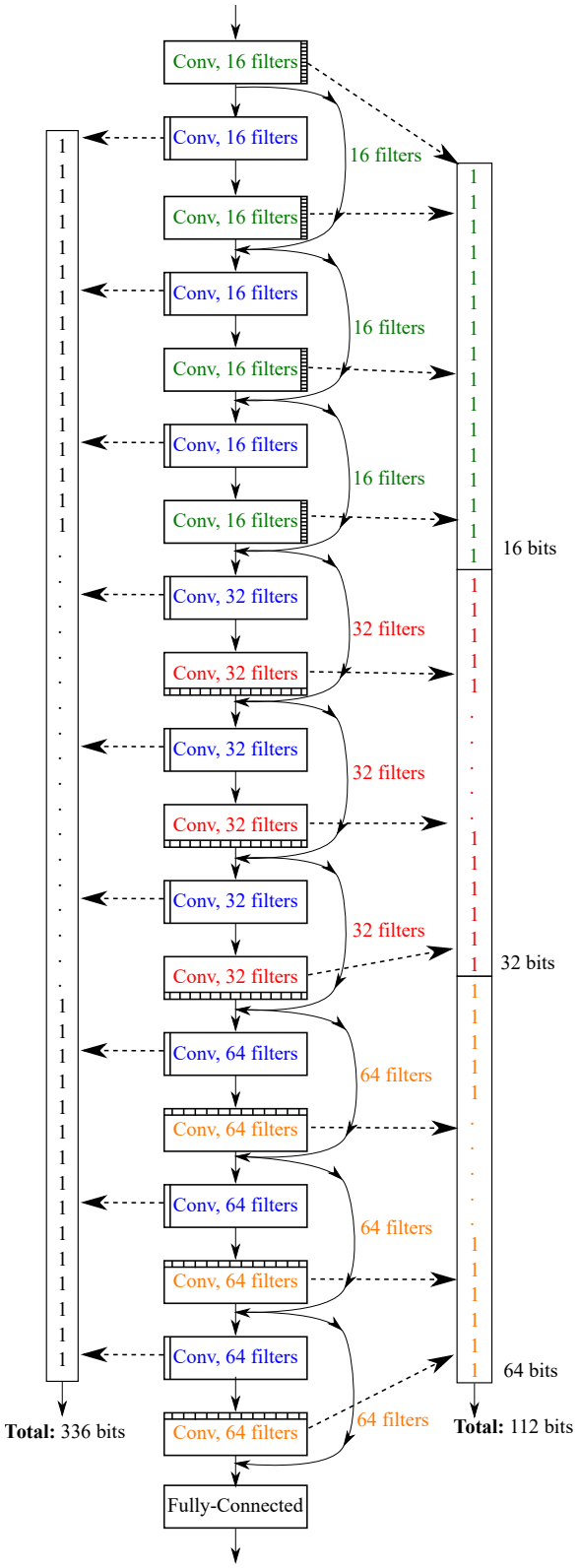


Figure 4.3: Representation of a 20 layer ResNet architecture on *DeepPruningES* [106].

binary strings.

4.2.2 Evaluation of Candidate Solutions

Each candidate solution in the population is evaluated in two objectives: its number of floating-point operations (FLOPs), and its training error. The number of floating-point operations can be directly counted by knowing the DNN architecture of the candidate solution. However, every time a filter is eliminated from a DNN architecture, its training error will be increased. Thus, every candidate solution needs to be retrained to decrease its training error.

The retraining phase of a candidate solution is done by randomly selecting 10% of the dataset to be used as input, which speeds up the retraining phase. It is not

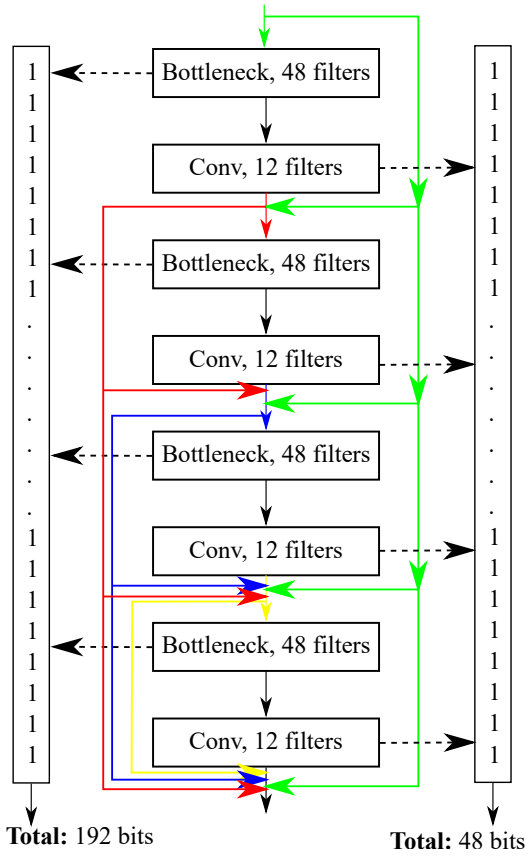


Figure 4.4: Representation of a Dense Block with four layers in *DeepPruningES* [106].

needed to use the whole dataset because all remaining parameters of a candidate solution are initialized to the parameters from the original pre-trained DNN model. Thus, the algorithm only needs to adjust the remaining parameters to accommodate the elimination of specific filters.

Two input parameters are used in Algorithm 5 to control the evaluation of candidate solutions: the number of epochs for individual evaluation (e_{eval}), and the learning rate for individual evaluation (α_{eval}). The e_{eval} parameter controls how many epochs a candidate solution will be retrained on the 10% of the dataset. While the α_{eval} is the learning rate used by the stochastic gradient descent (SGD) optimizer. Their chosen values are detailed in the next section.

4.2.3 Population Initialization

The proposed algorithm initialized all $3 + \lambda_{size}$ individuals in the population to be identical to the original DNN model being pruned, which is used as an input (dnn) in Algorithm 5. A DNN model well trained in the chosen dataset is used as input because all candidate solutions use this model parameters' values in their architectures to reduce the time required for particle evaluation. Thus, the individuals in the initial population are all identical to the original DNN model, and they are modified during the offspring generation.

4.2.4 Knee and Boundary Selection

In Multi-Criteria Decision Making (MCDM) problems, there is no single best solution to a problem. These types of problems have multiple objective functions and candidate solutions with many parameters. A small change in one of the candidate solution's parameters can produce a completely different result in one of the objective functions. Hence, decision-makers (DM) may need to use different solutions,

depending on his or her needs.

The population of candidate solutions is usually plotted in the objective space of the problem, making it easier to know which solution is better for use than the other based on the requirements at the moment. Then, the quality of candidate solutions can be evaluated by using geometric methods in the objective space. The so-called knee solution is a particular one with a strong geometric implication. The knee solution is usually the one that can be easily improved in any of the objectives with fewer changes in its parameters than the other solutions. Chiu *et al.* [111] developed the Minimum Manhattan Distance (MMD) method to find the knee solution by computing the Manhattan distances of all solutions located in the *Pareto optimal front*, where no solution is strictly better than the others, and selecting the one with the minimum distance. Because there are only three solutions of interest, the proposed algorithm does not impose a burden to find the whole *Pareto optimal front*. Thus, a modified MMD method is used to find the knee and boundaries solutions directly without using the whole *Pareto optimal front*.

The proposed knee and boundary solutions selection is presented in Algorithm 6 and illustrated in Figure 4.5. First, the algorithm evaluates the entire population to determine the number of FLOPs and training error of each candidate solution. Second, the algorithm determines the solutions with the smallest training error and with the smallest number of FLOPs to become the boundary heavy and the boundary light, respectively. Third, the Manhattan distance of each candidate non-dominated solution is computed accordingly to Algorithm 6, line 6. Finally, the knee solution is chosen to be the candidate solution with the minimum Manhattan distance in the population.

Algorithm 6: Knee and Boundary Selection [106]

Input : Individuals in the population ($\mathbf{P}$), original DNN model (dnn), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Three individuals: knee solution ($\mu.knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

$\mathbf{P} \leftarrow$ *Evaluate Population*($\mathbf{P}$, dnn, e_{eval}, α_{eval});
Find $min(f_1), min(f_2), max(f_1), max(f_2)$ in $\mathbf{P}$;
$\mu.heavy \leftarrow P_i$, where $f_1(P_i) = min(f_1)$;
$\mu.light \leftarrow P_j$, where $f_2(P_j) = min(f_2)$;
for $k = 1$ to len($\mathbf{P}$) **do**
    $dist(k) = \frac{f_1(P_k) - min(f_1)}{max(f_1) - min(f_1)} + \frac{f_2(P_k) - min(f_2)}{max(f_2) - min(f_2)}$;
end
$\mu.knee \leftarrow P_k$, where P_k has the minimum $dist(k)$;
return $\mu.knee$, $\mu.heavy$, $\mu.light$;

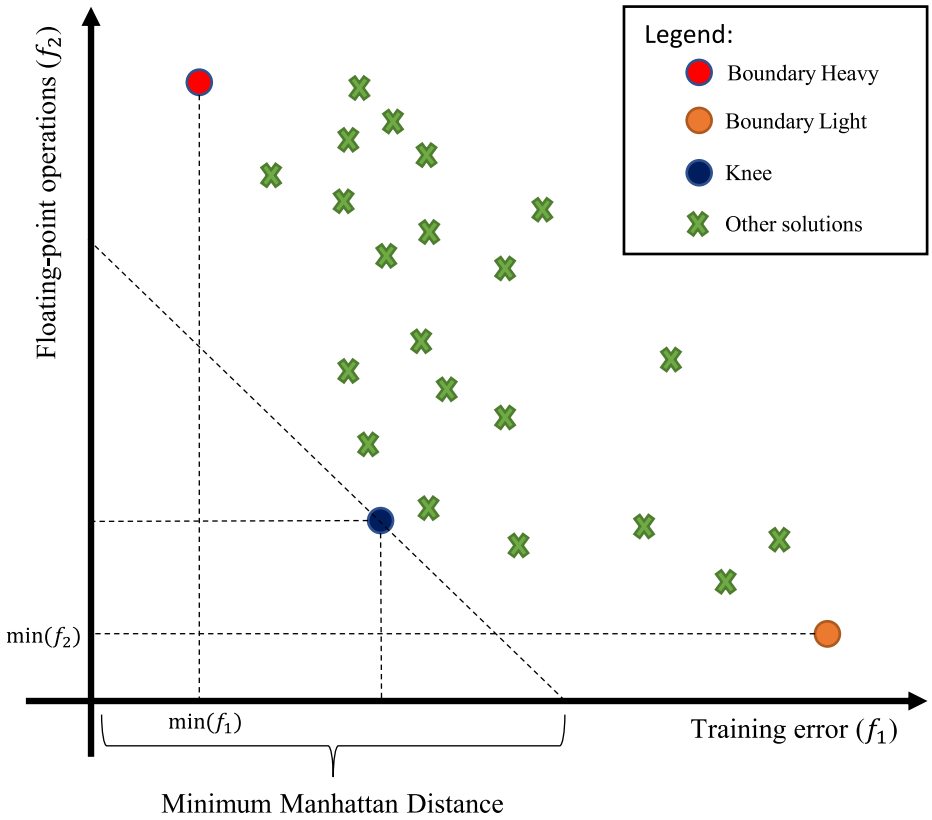


Figure 4.5: Example of *knee*, *boundary heavy*, and *boundary light* solutions [106].

4.2.5 Offspring Generation

After the knee and boundaries solutions are selected, they are used to generate λ_{size} offspring. *DeepPruningES* follows the standard ES offspring generation, where all offspring are generated by applying random mutations to the three solutions selected before. Each bit from a parent solution will be inverted with a probability equal to p_m. Once all offspring are created, the algorithm will go back to the selection of the knee and boundaries solutions, and the process will be repeated until the maximum number of generations (gen) has been reached.

4.2.6 Fine-Tuning of the Best Solutions

At the ending of the pruning process, the *knee*, *boundary heavy*, and *boundary light* solutions undergo a fine-tuning process to recover some of the classification performance they may have lost. The fine-tuning process is a retraining phase, where each solution is further trained in the whole dataset for many more epochs than during the pruning procedure. Thus, the two parameters used in the retraining are the number of epochs for fine-tuning (e_{fine}) and the learning rate for the fine-tuning (α_{fine}). Once the fine-tuning has finished, the pruned DNN models are saved in the disk, and their performance in both objectives is reported. All results reported in this chapter were obtained after the fine-tuning process.

4.3 Experimental Design

The chosen DNN architectures used to test the proposed *DeepPruningES*, and the algorithm parameters used in the experiments are discussed in this section.

4.3.1 Chosen DNN Models for Pruning

The proposed *DeepPruningES* is capable of pruning three types of DNN architectures: Convolutional Neural Networks (CNNs), Residual Neural Networks (ResNets), and Densely-connected Neural Networks (DenseNets). Thus, two state-of-the-art DNNs of each type are chosen to be pruned using the proposed algorithm. Moreover, only DNNs with excellent results in a challenging dataset were chosen for pruning.

Many researchers develop DNN models for use with the CIFAR10 dataset [43] because it is a challenging dataset, but small enough to be used in consumer-grade hardware. This dataset consists of 50,000 training images and 10,000 test images from ten different categories. Each image has a size equal to $3 \times 32 \times 32$ pixels, where the first dimension represents the red, green, and blue channels. Some samples of images in each class of the CIFAR10 dataset are shown in Figure 4.6. There also exists a dataset called CIFAR100 with similar characteristics to CIFAR10. Its main difference is that it used 100 classes instead of just ten with the same number of training and test images. However, there is no DNN architecture pruning works in the literature making use of this dataset.

The VGG16 and VGG19 are the two CNN models chosen to be pruned with the proposed algorithm. They have a total of 16 and 19 convolutional layers, respectively, and 3.15×10^8 and 4.01×10^8 floating-point operations (FLOPs), respectively. These networks also have good classification results on the CIFAR10 dataset with test errors equal to 6.06% and 6.18%, respectively. VGG16 and VGG19 are considered ones of the best CNN architectures for image classification tasks.

The ResNet56 and ResNet110 are the two state-of-the-art ResNet models chosen to be pruned. They have a total of 56 and 110 convolutional layers, and 1.27×10^8 and 2.57×10^8 FLOPs, respectively. Their test error on CIFAR10 is equal to 6.63%

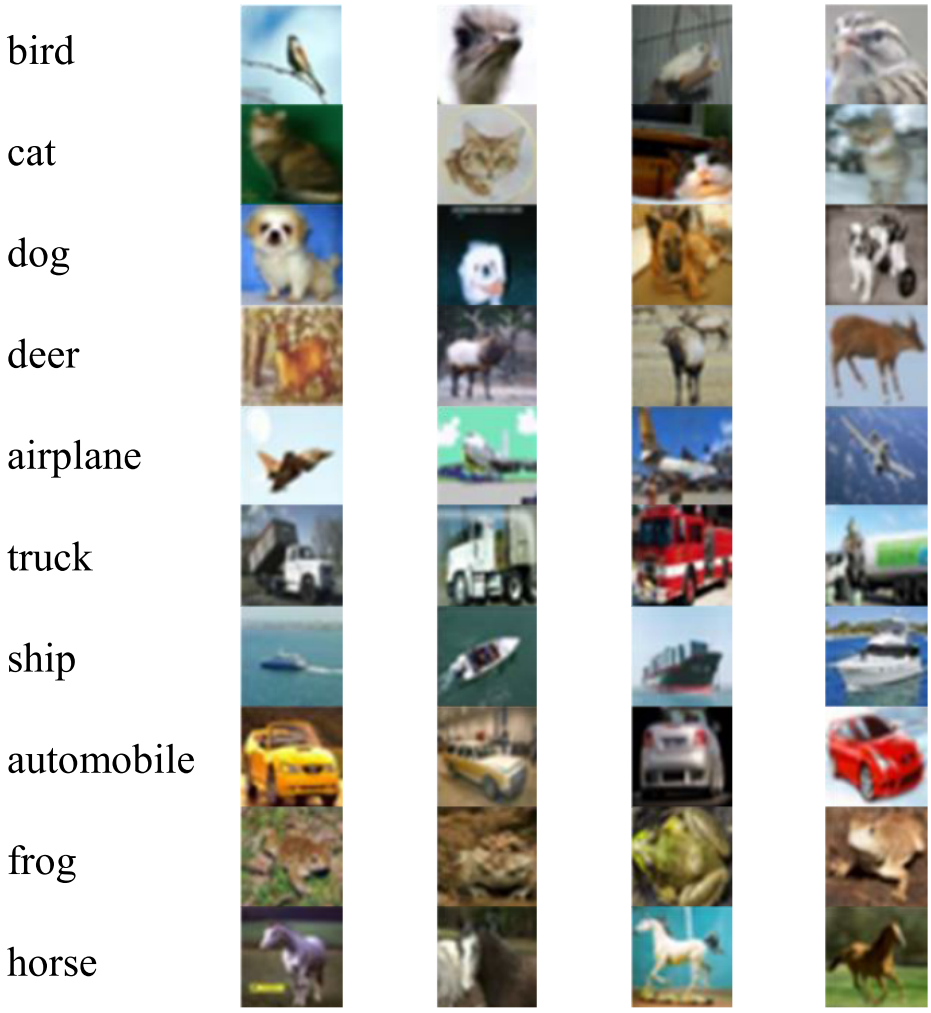


Figure 4.6: Example of images from each class in the CIFAR10 dataset [106].

and 6.2%, respectively. These two models are considered amongst the best ResNet architectures on the CIFAR10 dataset.

For last, the DenseNet50 and DenseNet100 are the two DenseNet models used to compare with the proposed algorithm. They have a total of 50 and 100 convolutional layers, and 0.93×10^8 and 3.05×10^8 FLOPs, respectively. Their test errors on CIFAR10 are 6.92% and 5.66%, respectively. These are also the best DenseNet models on the CIFAR10 dataset. The overview of all DNN models used to evaluate the proposed *DeepPruningES* can be seen in Table 4.1.

Table 4.1: Overview of the DNN architectures used to evaluate the proposed *DeepPruningES* [106].

DNN	# layers	# FLOPs	Test error on CIFAR10
VGG16	16	3.15×10^8	6.06%
VGG19	19	4.01×10^8	6.18%
ResNet56	56	1.27×10^8	6.63%
ResNet110	110	2.57×10^8	6.2%
DenseNet50	50	0.93×10^8	6.92%
DenseNet100	100	3.05×10^8	5.66%

4.3.2 Algorithm Parameters

The algorithm parameters used to prune all chosen DNN architectures are equal to the values indicated in Table 4.2. The number of offspring generated at every generation (λ_{size}) is equal to 20. All results are obtained with a maximum number of generation (gen) equal to 10. The probability of inverting a single bit in the binary strings representing convolutional filters (p_m) is equal to 0.1. The number of epochs and learning rate for individual evaluation (e_{eval} and α_{eval}) are equal to 5 and 0.1, respectively. For last, the number of epochs and learning rate used during the fine-tuning of the best solutions (e_{fine} and α_{fine}) are equal to 50 and 0.01, respectively.

Most of the parameters used by the proposed *DeepPruningES* are limited only by the amount of computational power available for use by the algorithm. A large number of offspring, generations, and epochs will require more computational power or more time to run. More offspring and generations would allow the algorithm to explore the objective space even more. However, a large mutation probability would reduce the algorithm's ability to find reasonable solutions because the candidate solutions' parameters would be guaranteed to change most of the time, avoiding the survival of a good combination of filters over the generations.

The number of epochs that each candidate solution is retrained during its evaluation follows the standard from other works in the literature in which the DNN is not

Table 4.2: Parameters used to evaluate the proposed *DeepPruningES* [106].

Parameter	Value
Offspring size (λ_{size})	20
Maximum number of generations (gen)	10
Mutation probability (p_m)	0.1
Number of epochs for individual evaluation (e_{eval})	5
Learning rate for individual evaluation (α_{eval})	0.1
Number of epochs for fine-tuning (e_{fine})	50
Learning rate for fine-tuning (α_{fine})	0.01

fully retrained [56, 49, 91]. The retraining will give a sense of the overall quality of the candidate solution. Thus, the model does not need to be retrained to perfection. The learning rate used during an individual evaluation is chosen to be higher than when fine-tuning because this parameter controls how strong the DNN weights are updated at every batch of images. A large learning rate will reduce the training error faster, but it can produce instability during long training sessions.

4.4 Experimental Results and Discussions

In this section, the results from the proposed *DeepPruningES* algorithm are presented and explained in detail.

4.4.1 Results

The results obtained with the six chosen DNNs are shown in Table 4.3. This table shows the knee, boundary heavy, and boundary light solutions found for each one of the chosen DNNs. It also contains the test errors, the numbers of FLOPs, and the percentage decreases in the number of FLOPs of the best solutions and the average of ten independent runs.

For the VGG16, the best knee, boundary heavy, and boundary light solutions have 65.49%, 32.01%, and 72.17%, respectively, fewer FLOPs when compared with

the original DNN model. Their final test error after fine-tuning are 9.04%, 8.21%, and 10.51%. On average, the algorithm pruned the knee, boundary heavy, and boundary light solutions by 61.58%, 20.88%, and 71.36%, respectively, and their average test errors are 9.58%, 8.6%, and 11.41%. These results are the expected ones because the knee solution is the best trade-off between the boundaries heavy and light solutions. For the VGG19 model, the best (and average) knee, boundary heavy, and boundary light solutions are pruned by 61.86% (57.15%), 32.56% (22.28%), and 71.74% (70.69%), respectively. Their best (and average) test errors are 9.04% (9.87%), 8.21% (8.77%), and 10.53% (12.03%), respectively.

For the ResNet56, the best (and average) knee, boundary heavy, and boundary light solutions are pruned by 66.23% (59.15%), 21.32% (15.23%), and 80.89% (77.67%), and their best (and average) test errors are 9.28% (9.98%), 8.11% (8.77%), and 11.42% (13.36%), respectively. In the case of the ResNet110 model, the best (and average) knee, boundary heavy, and boundary light solutions have 64.84% (59.89%), 16.72% (14.14%), and 83.29% (77.86%) fewer parameters than the original model, while their final test errors are 8.66% (9.42%), 7.43% (7.93%), and 10.27% (12.9%), respectively.

For the DenseNet50, the best (and average) knee, boundary heavy, and boundary light solutions are pruned by 56.05% (50.15%), 19.16% (16.59%), and 75.53% (73.91%), and their best (and average) test errors are 9.8% (10.43%), 8.91% (9.26%), and 13.04% (14.8%), respectively. For last, the best (and average) knee, boundary heavy, and boundary light solutions for the DenseNet100 are pruned by 63.64% (60.31%), 19.33% (18.24%), and 73.09% (71.16%) with test errors equal to 9.04% (9.37%), 8.34% (8.39%), and 10.47% (11.90%), respectively.

In general, the knee solutions are compressed around 63% with test error around

Table 4.3: Pruning results obtained with the proposed *DeepPruningES* [106].

DNN Model	DeepPruningES						
	Solution	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
VGG16	**Knee**	9.04%	9.58%	1.09×10^8	1.22×10^8	65.49%	61.58%
	Boundary Heavy	8.21%	8.6%	2.15×10^8	2.49×10^8	32.01%	20.88%
	Boundary Light	10.51%	11.41%	0.88×10^8	0.9×10^8	72.17%	71.36%
VGG19	**Solution**	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
	Knee	9.04%	9.87%	1.53×10^8	1.72×10^8	61.86%	57.15%
	Boundary Heavy	8.21%	8.77%	2.7×10^8	3.12×10^8	32.56%	22.28%
	Boundary Light	10.53%	12.03%	1.13×10^8	1.18×10^8	71.74%	70.69%
ResNet56	**Solution**	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
	Knee	9.28%	9.98%	0.432×10^8	0.523×10^8	66.23%	59.15%
	Boundary Heavy	8.11%	8.77%	1.01×10^8	1.08×10^8	21.31%	15.23%
	Boundary Light	11.42%	13.36%	0.244×10^8	0.286×10^8	80.89%	77.67%
ResNet110	**Solution**	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
	Knee	8.66%	9.42%	0.905×10^8	1.03×10^8	64.84%	59.89%
	Boundary Heavy	7.43%	7.93%	2.14×10^8	2.21×10^8	16.72%	14.14%
	Boundary Light	10.27%	12.9%	0.43×10^8	0.56×10^8	83.29%	77.86%
DenseNet50	**Solution**	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
	Knee	9.8%	10.43%	0.41×10^8	0.466×10^8	56.05%	50.15%
	Boundary Heavy	8.91%	9.26%	0.756×10^8	0.779×10^8	19.16%	16.59%
	Boundary Light	13.04%	14.8%	0.229×10^8	0.244×10^8	75.53%	73.91%
DenseNet100	**Solution**	**Test error (best)**	**Test error (mean)**	**# FLOPs (best)**	**# FLOPs (mean)**	**% Pruned (best)**	**% Pruned (mean)**
	Knee	9.04%	9.37%	1.11×10^8	1.21×10^8	63.64%	60.31%
	Boundary Heavy	8.34%	8.39%	2.46×10^8	2.49×10^8	19.33%	18.24%
	Boundary Light	10.47%	11.90%	0.82×10^8	0.879×10^8	73.09%	71.16%

9%; the boundary heavy solutions are compressed by 20% with test error of 8.2%; the boundary light solutions are compressed by 70% with test errors of around 11%. These results demonstrate that the proposed algorithm can be used to produce pruned DNN models with different trade-offs between computational complexity and classification accuracy.

4.4.2 Discussions

The proposed *DeepPruningES* algorithm is capable of pruning multiple types of DNN architectures while maintaining a good classification accuracy. The algorithm significantly pruned all six chosen DNN models with similar results across all of them. It was even capable of pruning ResNets and DenseNets, which were designed to have fewer parameters than standard CNNs.

Most pruning algorithms in the literature were tested with fewer DNN models than the ones presented here. Moreover, the results obtained by the *DeepPruningES* are comparable with other state-of-the-art pruning algorithms presented in Table 4.4, where the proposed algorithm can further prune DNN models than the algorithm developed by Li *et al.* [55] with slightly worse test errors on CIFAR10. *DeepPruningES*

Table 4.4: Pruning results from peer competitors using the CIFAR10 dataset [106].

Approach	DNN Model	% FLOPs Pruned	Test error
Li *et al.* [55]	VGG16	34.2%	6.60%
	ResNet56	27.6%	6.94%
	ResNet110	38.6%	6.70%
Ding *et al.* [112]	VGG16	81.39%	7.56%
	ResNet56	66.88%	9.43%

also obtains comparable pruning results to Ding *et al.* [112], and with comparable test errors on CIFAR10. These two works are the most similar ones in the literature to the proposed algorithm presented here, which were also tested on similar DNN models and dataset.

Although the test errors of the pruned solutions in *DeepPruningES* are slightly worse than the ones from Li *et al.* [55] and Ding *et al.* [112], they can be further improved by being trained from scratch. Figure 4.7 shows the training evolution of the knee solution of the VGG16 being trained from scratch for 200 epochs. Training from scratch means that all the parameters of the pruned DNN model are re-initialized using the Glorot method [97], and the network is trained using stochastic gradient descent (SGD) from there. Training from scratch was able to bring the test error down from 9.04% to 8.23%. Further improvements are still possible if other training optimizers and data augmentation are used.

As stated before, *DeepPruningES* uses the ES *plus* version, which is an elitist version of ES, which improves the final results by not eliminating the best individuals in the population. A version of DeepPruningES using the *comma* version was developed to test this statement. The population evolution of the proposed algorithm over nine generations using the *plus* version can be seen in Figure 4.8, while the *comma version* can be seen in Figure 4.9. The *comma* version of the algorithm is capable of pruning a DNN architecture, but it does not preserve the training error as well as the *plus*

version. Furthermore, in the *comma* version, the knee and boundaries solutions are all very close to each other and do not present good trade-off as the ones from the *plus* version.

4.5 Final Remarks

In this chapter, a DNN architecture pruning algorithm based on Evolution Strategy, called *DeepPruningES*, was presented and evaluated. The proposed algorithm works by eliminating convolutional filters from convolutional layers found in CNN, ResNet, and DenseNet architectures, and it does not require the use of any statistical information about the filters being pruned. It uses a modified Minimum Manhattan Distance approach to select three candidate solutions from the population with the best trade-off between computational complexity and classification performance. These three solutions can be used by decision-makers to fulfill their needs at the moment. *DeepPruningES* is capable of finding pruned solutions with comparable results to other algorithms in the literature by using a small population of only 20 individuals, and the evolution process only takes ten generations. A single run of the proposed

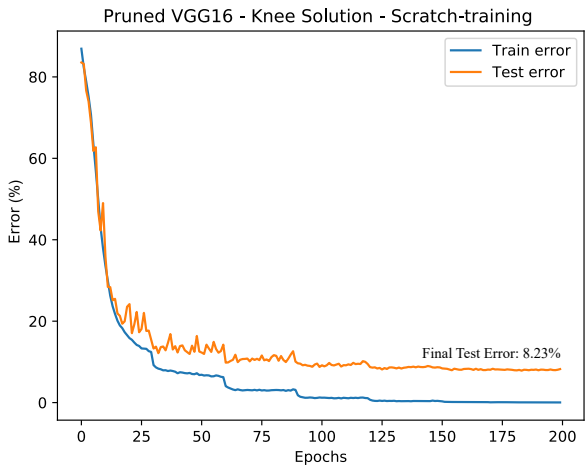


Figure 4.7: Knee solution from the pruned VGG16 network trained from scratch for 200 more epochs [106].

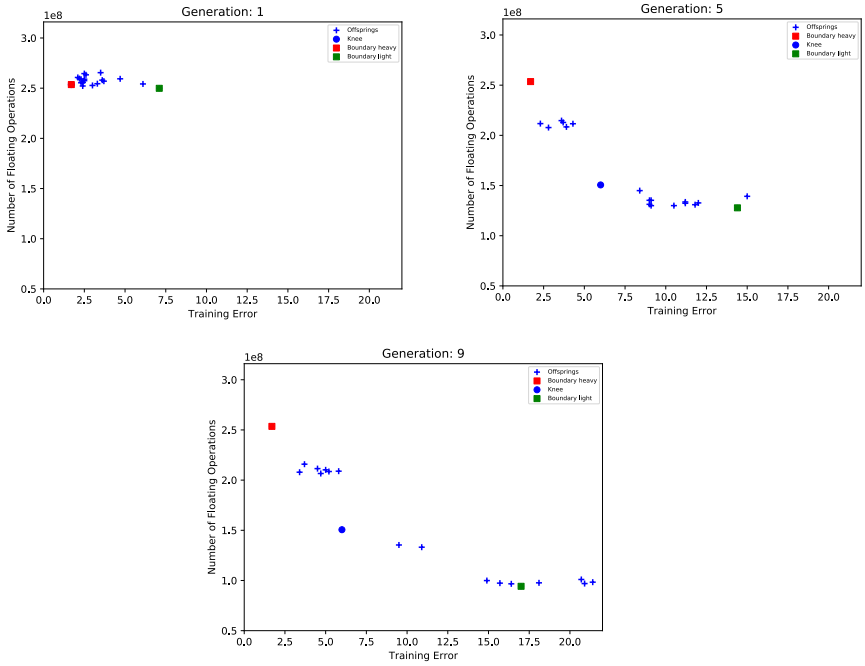


Figure 4.8: Evolution of the population when pruning VGG16 using the *plus* version of the proposed DeepPruningES [106].

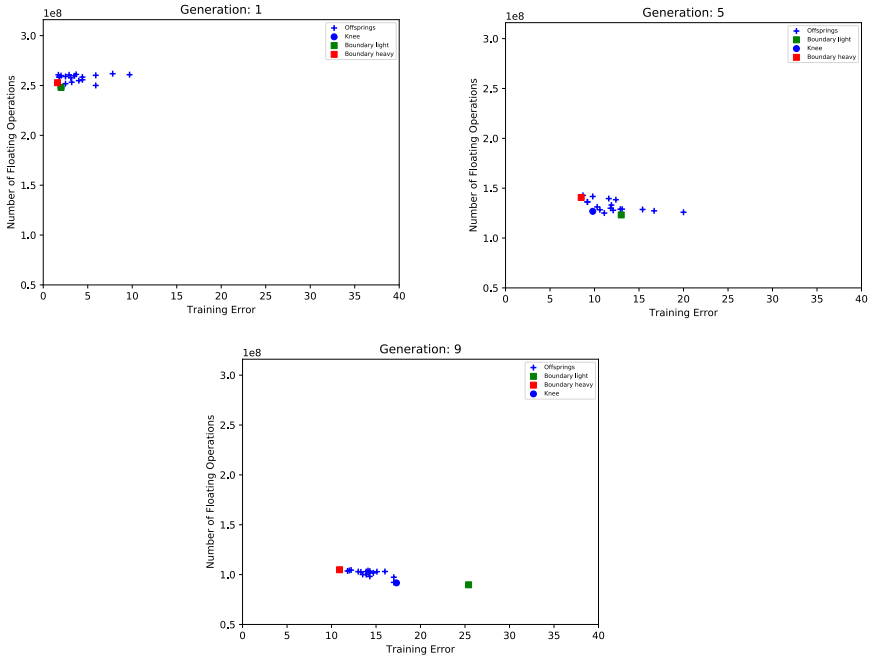


Figure 4.9: Evolution of the population when pruning VGG16 using the *comma* version of the proposed DeepPruningES [106].

algorithm can be performed in less than four hours using a mid-entry consumer-grade NVIDIA GTX 1060 GPU. Thus, the algorithm proposed in this chapter can be used in the development of DNN architectures to be used in mobile hardware.

CHAPTER V

AUTOMATIC SEARCHING AND PRUNING OF DEEP NEURAL NETWORKS FOR MEDICAL IMAGING DIAGNOSTICS

In the present chapter, DNN architecture searching and pruning algorithms are used together to solve the real-world problems of medical imaging diagnostics while also taking into account the user preference about the problem[1].

5.1 Introduction

As presented in Chapter II, multiple algorithms can be used to perform DNN architecture searching and pruning. For example, DNN architecture searching can be performed by Evolutionary Computation methods, such as Genetic Algorithms [40, 86, 89], Particle Swarm Optimization [96, 91], and others [84, 85]. On the other hand, DNN architecture pruning algorithms are mostly metaheuristic ones relying heavily on prior knowledge about the problem at hand and the topology of the DNN models [51, 50, 49, 55, 56]. To the best of the author's knowledge, DNN architecture searching and pruning are always treated as two separate and independent problems in the literature. Thus, in the present chapter, a unified framework is presented where these two problems are seen as two critical pieces for the automatic design of DNN models. Moreover, the proposed framework is applied to solve the real-world problems of medical imaging diagnostics.

Due to their extraordinary results, DNNs are being used in a variety of problems in

[1]The present chapter is partially based on my submitted work, which is currently under review.

the medical field. One crucial example of their use is in the field of medical diagnostics aided by images from a variety of sources, such as Computed Tomography (CT), x-rays, skin lesion photographs, and Magnetic Resonance Images (MRI). For example, Convolutional Neural Networks (CNNs) were used to segment tumor tissues on MRI brain scans [113, 114], to identify masses and non-masses in breast tomosynthesis images [115], and to characterize plaque compositions in carotid ultrasound [116]. Even a special type of DNN, called *U-Net*, was created to perform segmentation of regions of interest from medical images, where its input and output are both images [117]. The input is an image that needs to be segmented, and the output is a black and white image where the white pixels indicate the location of pixels belonging to the region of interest. The architecture of a *U-Net* resembles the ones found in autoencoders [117], where half of its structure uses convolutional layers to reduce the dimensionality of the input image, and the other half uses deconvolutional layers to create an output image.

Although DNNs have been used successfully in the field of medical imaging diagnostic, the main challenge remains: how to create a meaningful DNN architecture for a given problem? All of the previously cited works designed uniquely handcrafted DNN models by using expert knowledge of the problem. Thus, in this chapter, a framework for the creation of DNNs for use in medical imaging tasks, called *DNNDeepeningPruning*, is proposed. This framework has two phases: In the first phase, called *DNN Architecture Deepening*, a DNN model is created by iteratively deepening its architecture; in the second phase, called *DNN Architecture Pruning*, any parameter in excess is removed from the model's architecture. Furthermore, the second phase is guided by user preference between the model's computational complexity and the model's performance in the problem at hand.

Thus, the main contributions of the proposed framework are the following:

- The development of a simpler DNN architecture searching algorithm is presented where DNN models are found by only increasing their computational complexity, which, in turn, reduces the size of the algorithm's searching space.
- An improved Multi-Criteria Decision Making (MCDM) pruning algorithm is proposed where the user preference for one of the objectives is taken into consideration.
- A framework for DNN architecture designing is presented where accurate, and compact DNN models can be found by making the pruning stage an integral part of the algorithm.

5.2 Proposed Algorithm

The overview of the proposed *DNNDeepeningPruning* framework is presented in Algorithm 7. Its inputs are the chosen image classification dataset, and a set of parameters to control the DNN architecture deepening and pruning processes. Its outputs are three pruned models: the *boundary heavy*, the *boundary light*, and the user *preferable knee*. Each phase of the proposed framework is explained in detail in the following subsections.

5.2.1 DNN Architecture Deepening Algorithm

In the DNN architecture deepening phase, the algorithm only increases the computational complexity of a DNN model by iteratively adding randomly chosen blocks of residual layers (residual blocks) to the DNN architecture. At each iteration, the algorithm tests a total of G_{trial} blocks. The best one is added to the model, and a new iteration starts the process again. In order to assess the quality of each added block, the proposed algorithm adds a classification layer to the end of the current DNN

Algorithm 7: Proposed *DNNDeepeningPruning* Framework.

Input : **DNN Deepening:** Number of trials to test new blocks (G_{trial}). **DNN Pruning:** Offspring size (λ_{size}), number of iterations for pruning (G_{prune}), mutation probability (p_m), DNN model for pruning (dnn), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}). **Shared parameters:** image classification dataset ($\boldsymbol{X}$).

Output: Three DNN models: preferable knee solution ($prefer\_knee$), boundary heavy solution ($heavy$), and boundary light solution ($light$).

```
/* DNN Architecture Deepening                                    */
```
$dnn \leftarrow DNNDeepening(G_{trial}, \boldsymbol{X})$;
```
/* DNN Architectute Pruning                                      */
```
$prefer\_knee, heavy, light \leftarrow$
$DeepPruningES\_WIN(\lambda_{size}, G_{prune}, p_m, dnn, e_{eval}, \alpha_{eval}, e_{fine}, \alpha_{fine}, \boldsymbol{X})$;
```
/* Return best solutions                                         */
```
return $prefer\_knee$, $heavy$, $light$;

architecture, performs a small training phase using traditional backpropagation, and tests the model's performance in a validation set. The process of deepening and evaluation of a candidate solution is illustrated in Figure 5.1. Furthermore, the chosen evaluation metric used during deepening is the Area Under the Receiving Operating Characteristic Curve (ROC-AUC), which incorporates the concepts of True Positive Rate, which is equal to the $Sensitivity$, and the False Positive Rate, which is equal to $1 - Specificity$, into a single measure.

The proposed DNN Architecture Deepening algorithm is presented in Algorithm 8. Its inputs are the number of trials (G_{trial}) to test new blocks in a given iteration, and the chosen image classification dataset ($\mathbf{X}$). Its output is the best DNN model found during the deepening process. The algorithm works by adding blocks every iteration into the DNN model until the ROC-AUC of the validation set stops improving after two consecutive iterations in an early stopping fashion to avoid overfitting [118].

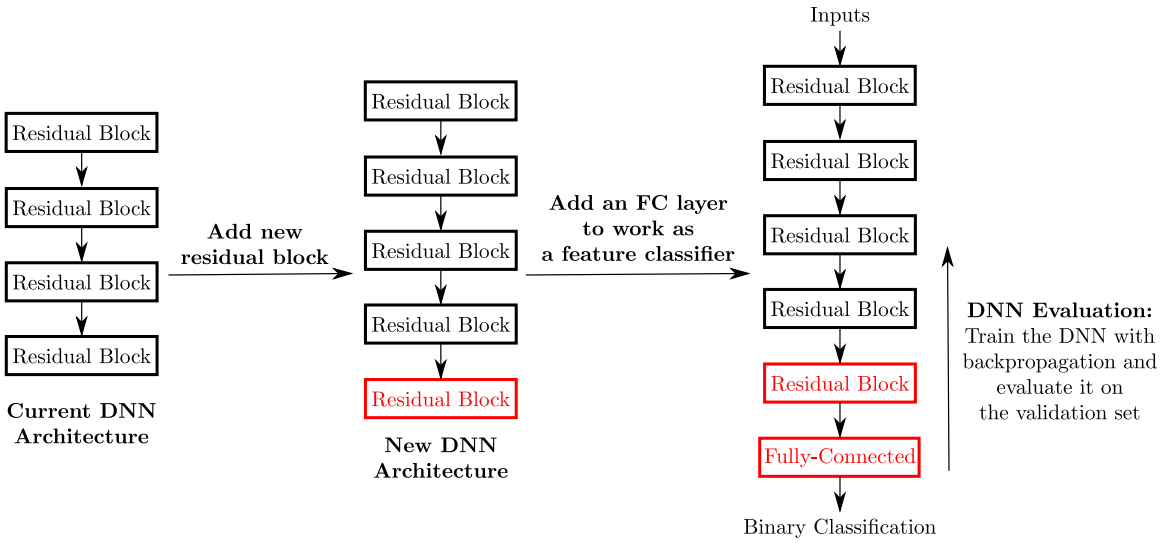


Figure 5.1: Proposed DNN deepening and evaluation.

Each block consists of three parameters: its number of residual layers, the number of convolutional filters for all residual layers, and if the block performs a downsampling operation.

Moreover, during the evaluation, each candidate solution is trained until the validation ROC-AUC stops improving for five epochs. In that sense, each added block is compared with others only after full training. The deepening algorithm also saves and loads the weights of the best blocks in each iteration, which allows for reduced training time. Another important key aspect of the proposed DNN architecture deepening algorithm is that it never deletes any block added in a previous iteration. Thus, the use of a pruning algorithm is crucial to allow the creation of compact DNN models.

5.2.2 DNN Architecture Pruning Algorithm

The proposed DNN architecture pruning algorithm presented in this chapter is an extension of the pruning algorithm previously presented in Chapter IV. It is still a two-objective Multi-Criteria Decision Making (MCDM) algorithm searching for three candidate solutions, but, instead of searching for the boundary heavy, the

Algorithm 8: Proposed DNN Architecture Deepening (*DNNDeepening*).

Input : Number of trials to test new blocks (G_{trial}), image classification dataset ($\boldsymbol{X}$).
Output: A DNN model.

```
/* Initializes a DNN with a single block                              */
dnn ← Creates an empty DNN object;
dnn ← AddRandomResidualBlock(dnn);
previous_fitness ← Evaluate(dnn, X);
BlocksWithoutImprovement = 0;
while BlocksWithoutImprovement < 2 do
    dnn_test ← dnn;
    for j = 1 to G_trial do
        dnn_temp ← dnn_test;
        /* Adds a new block at random to the DNN                      */
        dnn_temp ← AddRandomResidualBlock(dnn_temp);
        test_fitness ← Evaluate(dnn_temp, X);
        if j = 1 then
            dnn ← dnn_temp;
            best_fitness ← test_fitness;
        else if j > 1 and test_fitness is better than best_fitness then
            dnn ← dnn_temp;
            best_fitness ← test_fitness;
        end
    end
    if best_fitness is better than previous_fitness then
        BlocksWithoutImprovement = 0;
    else
        BlocksWithoutImprovement = BlocksWithoutImprovement + 1;
    end
end
return dnn;
```

boundary light, and the exact knee solutions, it searches for a preferable knee and the boundaries solutions. In this algorithm, the decision-maker can choose which of the two objectives is more important for him/her. Hence, the algorithm selects a non-dominated solution, excluding the boundary ones, which has its objective values closest to the preference given by the decision-maker. The preferable knee selection

Algorithm 9: Proposed DNN Architecture Pruning (*DeepPruningES-WIN*).

Input : Offspring size (λ_{size}), number of iterations for pruning (G_{prune}), mutation probability (p_m), original DNN model (dnn), user preference vector ($\mathbf{r}$), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}).

Output: Three DNN models: preferable knee solution ($\mu.prefer\_knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

$\boldsymbol{\mu}, \boldsymbol{\lambda} \leftarrow$ *Initialize Population*(λ_{size}, dnn);
for g = *1 to* G_{prune} **do**
    $\mathbf{P} \leftarrow \boldsymbol{\mu} + \boldsymbol{\lambda}$
    $\mathbf{F} \leftarrow$ *Non-dominated Selection*($\mathbf{P}$);
    $\boldsymbol{\mu} \leftarrow$ *Preferable Knee & Boundary Selection*($\mathbf{F}$, $\mathbf{r}$, dnn, e_{eval}, α_{eval});
    $\boldsymbol{\lambda} \leftarrow$ *Offspring Generation*($\boldsymbol{\mu}$, λ_{size}, p_m, dnn);
end
Fine-tuning($\boldsymbol{\mu}$, dnn, e_{fine}, α_{fine});
return $\mu.prefer\_knee$, $\mu.heavy$, $\mu.light$;

proposed here is a modified version of the Weight Induced Norm (WIN) approach developed by Chiu et al. [119], where the non-dominated solution with the smallest WIN value is chosen to be the preferable knee solution.

This new DNN architecture pruning algorithm is called here *DeepPruningES-WIN*, and it is shown in Algorithm 9. It is a $(3 + \lambda)$-ES algorithm, where a DNN model is given as input, and three pruned models are returned. Most of the input parameters are the same as the ones used in the *DeepPruningES* algorithm presented in Chapter IV, with the addition of a two-dimensional user preference vector ($\mathbf{r}$) used to rank the two-objective preference of a decision-maker. The user preference vector uses integer values to indicate a preference, and small values indicate the highest importance of the corresponding objective. For example, if $\mathbf{r} = [1, 2]$, it means that the first objective is more important than the second one for this particular decision-maker. Furthermore,

Algorithm 10: DNN Pruning: Non-dominated Selection

Input : Individuals in the population (**P**), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Set of non-dominated candidate solutions in the pareto front (**F**).

```
1  P ← Evaluate Population(P, dnn, e_eval, α_eval);
2  F = ∅;
3  P_i.isDominated = False, for i = 1 to len(P);
4  for i = 1 to len(P) do
5      if P_i.isDominated = False then
6          for j = 1 to len(P) do
7              if i ≠ j then
8                  if P_j ≻ P_i then
9                      P_i.isDominated = True;
10                     break;
11                 end
12             else
13                 P_j.isDominated = True;
14             end
15         end
16     end
17 end
18 for i = 1 to len(P) do
19     if P_i.isDominated = True then
20         F ← F ∪ P_i;
21     end
22 end
23 return F;
```

the algorithm has three key components: the Non-Dominated Selection (Algorithm 9, line 4), the Preferable Knee and Boundary Selection (Algorithm 9, line 5), and the Offspring Generation (Algorithm 9, line 6).

The *Non-Dominated Selection* is presented in Algorithm 10. It consists of pairwise comparisons between all candidate solutions to determine which ones are non-dominated with respect to the others. Once the non-dominated solutions are identified, the algorithm performs the *Preferable Knee and Boundary Selection*, according to Algorithm 11. Finally, during the *Offspring Generation*, the three selected candi-

Algorithm 11: DNN Pruning: Preferable Knee & Boundary Selection

Input : Individuals in the Pareto Front ($\mathbf{F}$), user preference vector ($\mathbf{r}$), original DNN model (dnn), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Three individuals: preferable knee solution ($\mu.prefer\_knee$), boundary heavy solution ($\mu.heavy$) and boundary light solution ($\mu.light$).

Find $min(f_1), min(f_2), max(f_1), max(f_2)$ in $\mathbf{F}$;
$\mu.heavy \leftarrow F_i$, where $f_1(F_i) = min(f_1)$;
$\mu.light \leftarrow F_j$, where $f_2(F_j) = min(f_2)$;
`// Compute Ranking Transforms`
$\phi_1 = 2 - r_1 + 1$ and $\phi_2 = 2 - r_2 + 1$;
`// Compute Weighting Vector`
$w_1 = \frac{\phi_1}{\phi_1+\phi_2}$ and $w_2 = \frac{\phi_2}{\phi_1+\phi_2}$;
for $k = 1$ to len($\mathbf{F}$) **do**
    `// Compute Weighted Induced Norms`
    $norm(k) = w_1 \cdot \frac{f_1(F_k)-min(f_1)}{max(f_1)-min(f_1)} + w_2 \cdot \frac{f_2(F_k)-min(f_2)}{max(f_2)-min(f_2)}$;
end
$\mu.prefer\_knee \leftarrow F_k$, where F_k has the minimum $norm(k)$;
return $\mu.prefer\_knee$, $\mu.heavy$, $\mu.light$;

date solutions are mutated until λ_{size} offspring have been generated. This process is repeated for G_{prune} iterations in which the best candidate solutions are returned.

During pruning, candidate solutions are evaluated in the following objectives: ROC-AUC of the validation set, and the number of Floating-Point Operations (FLOPs). Thus, the boundary heavy solution is the non-dominated solution with the smallest ROC-AUC value, while the boundary light solution is the one with the smallest number of FLOPs. The preferable knee solution will depend on the user preference. If the ROC-AUC is chosen to be more critical than the number of FLOPs, the preferable knee will be closer to the boundary heavy. If the number of FLOPs is chosen to be more critical than the ROC-AUC, the preferable knee will be closer to the boundary light solution. Figure 5.2 illustrates the behavior of the preferable knee in

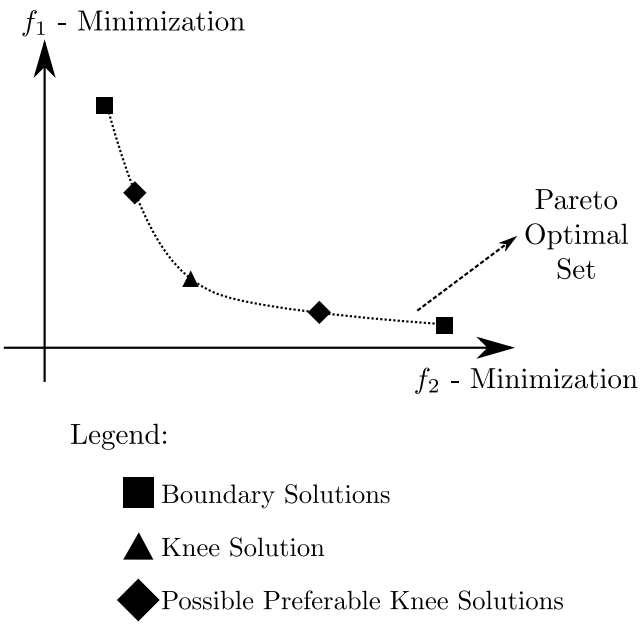


Figure 5.2: Example of preferable knee selection with the WIN approach.

a two-dimensional objective space, depending on the decision-maker preference.

5.3 Experimental Design

This section describes the experimental design used to evaluate the proposed algorithm. It includes the datasets selected for evaluation, the chosen evaluation criteria, and the algorithm parameters.

5.3.1 Medical Imaging Datasets

Because the proposed algorithm was designed to deal with binary classification problems, two binary classification datasets were chosen to evaluate it: one composed of images of skin lesions, and the other of chest x-ray images. Sample images of each dataset can be seen in Figure 5.3.

The first dataset was devised by the International Skin Imaging Collaboration in 2016 (ISIC2016), and it has pictures taken from benign and malignant (melanoma)

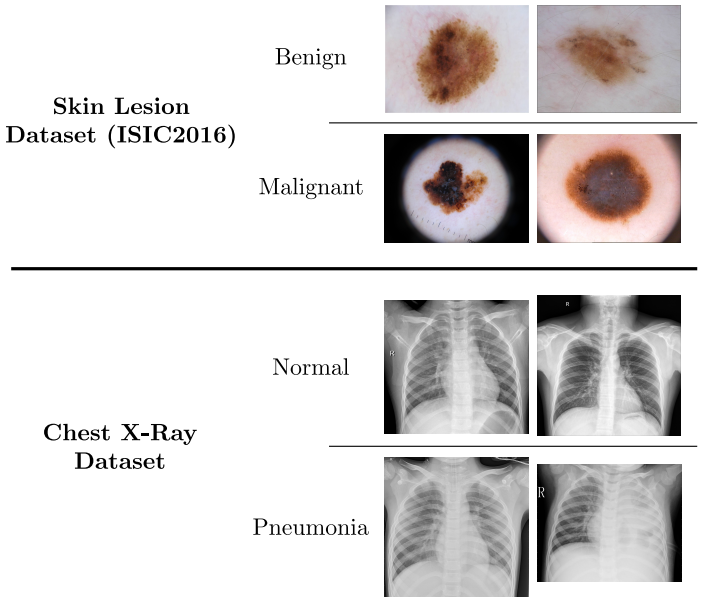


Figure 5.3: Chosen datasets to evaluate the proposed *DNNDeepeningPruning* algorithm.

skin lesions [120]. It has a total of 900 training images, where 727 samples are from benign lesions, and 173 are from malignant lesions, and 379 test images, where 304 are from benign lesions, and 75 from malignant lesions. The assessment of the proposed algorithm was done by randomly selecting 20% of the training images to be used as a validation set. Each image in this dataset has different resolution sizes. Thus, all images were resized to 224×224 pixels in resolution for use during training and evaluation.

The second dataset is composed of 5,232 training images and 624 test images of chest x-rays from patients of different ages and genders. The samples are classified as normal or pneumonia cases. In the training set, 1,349 images are from healthy patients, and 3,883 are from patients with pneumonia [121]. Similar to the previous dataset, 20% of the training images are selected at random to be used as a validation set. It also contains images with different resolution sizes, which are resized to 224×224 pixels.

5.3.2 Evaluation Criteria

The Area Under the ROC Curve (ROC-AUC) is used as the main selection criteria to select the best solutions produced by the DNN Deepening algorithm and presented in the next section. The ROC-AUC is used mostly in binary classification tasks, and it incorporates the number of true/false positives cases and true/false negatives cases into a single value ranging from 0.0 to 1.0, where a good classifier model is the one with the ROC-AUC value closest to 1.0. This metric is also important because it is computed using multiple decision thresholds to construct the ROC curve. Thus, models with acceptable performance only in a few decision thresholds still have low ROC-AUC values [122]. Furthermore, the results are presented using four more metrics: the model's accuracy, Sensitivity, Specificity, and number of FLOPs.

The model's accuracy is reported using a decision threshold equal to 0.5, which is similar to others' works, and it is computed by dividing the number of correctly classified data samples by the total number of data samples. In contrast, Sensitivity and Specificity are reported using a decision threshold (t) equal to:

$$t = \underset{x \in \mathbf{T}}{\operatorname{argmax}}(Se(x) \cdot (1 - Sp(x)))), \tag{5.1}$$

where $Se(x)$ is the model's Sensitivity using a decision threshold equal to x, and $Sp(x)$ is the model's Specificity using a decision threshold equal to x, and they are defined as follows:

$$Se(x) = \frac{TP(x)}{TP(x) + FN(x)} \text{ and } Sp(x) = \frac{TN(x)}{TN(x) + FP(x)}, \tag{5.2}$$

where, x represents a chosen decision threshold, $TP(x)$ the number of true positives, $FP(x)$ the number of false positives, $TN(x)$ the number of true negatives, and $FN(x)$

the number of false negatives given the threshold x. In other words, the decision threshold is the one that maximizes the $Sensitivity$ and $Specificity$ of the model at the same time, given the values from a ROC curve.

Additionally, a true positive (TP) classification is one that the model classified the data sample correctly as a positive case (presence of the condition). In contrast, a true negative (TN) classification is the one that the model classified the data sample correctly as a negative case (absence of the condition). Thus, false positives (FP) are cases where the model incorrectly classified the data as a positive case, and false negatives (FN) are cases where the model incorrectly classified the data is a negative case.

For last, the number of FLOPs in a given model is used to compare and contrast the resulted pruned models with handcrafted ones.

5.3.3 Algorithm Parameters

Unless stated otherwise, all results shown in this chapter were obtained using the algorithm parameters shown in Table 5.1, which shows the parameters by stages: DNN Architecture Deepening and DNN Architecture Pruning.

During the DNN Deepening stage, the algorithm was set to test a total of 10 residual blocks (G_{trial}) before moving to the next iteration. Furthermore, the minimum and the maximum number of layers in a residual block was randomly chosen to be between 4 and 12, respectively, while the number of convolutional filters in a given block was randomly chosen to be between 8 and 64. The evaluation of a given candidate solution was performed by training it with a learning rate of 0.001 until the validation AUC stopped improving for five epochs.

The best model of a total of five runs of the DNN Deepening algorithm was pruned using parameters with similar values to the ones presented in Chapter IV. The main

Table 5.1: Parameters used to evaluate the proposed algorithm.

DNN Architecture Deepening Parameters	
Number of trials to test new blocks (G_{trial})	10
Minimum number of layers in a residual block	4
Maximum number of layers in a residual block	12
Minimum number of convolutional filters in a residual block	8
Maximum number of convolutional filters in a residual block	64
Stops DNN evaluation after x epochs without improvement	$x = 5$
Learning rate to evaluate a candidate solution	0.001
DNN Architecture Pruning Parameters	
Offspring size (λ_{size})	30
Number of iterations for pruning (G_{prune})	10
Mutation probability (p_m)	0.2
User preference vector ($\mathbf{r}$)	[1, 2]
Number of epochs for individual evaluation (e_{eval})	3
Number of epochs for fine-tuning (e_{fine})	200
Learning rate for individual evaluation (α_{eval})	0.01
Learning rate for fine-tuning (α_{fine})	0.0001

addition is the user preference vector ($\mathbf{r}$) to control the selection of the preferable knee. In this case, while a model with reduced computational complexity is desired, maintaining a model's validation AUC as high as possible is considered more critical. Thus, $\mathbf{r}$ is chosen to be equal to $[1, 2]$, indicating that the first objective (validation ROC-AUC) is more important than the second (number of FLOPs). Furthermore, the algorithm minimizes the $(1 - \text{ROC-AUC})$, which is the same as maximizing the ROC-AUC.

5.4 Experimental Results

This section presents the experimental results obtained by the proposed *DNNDeepeningPruning* algorithm. The results of each stage are presented individually with their corresponding discussions.

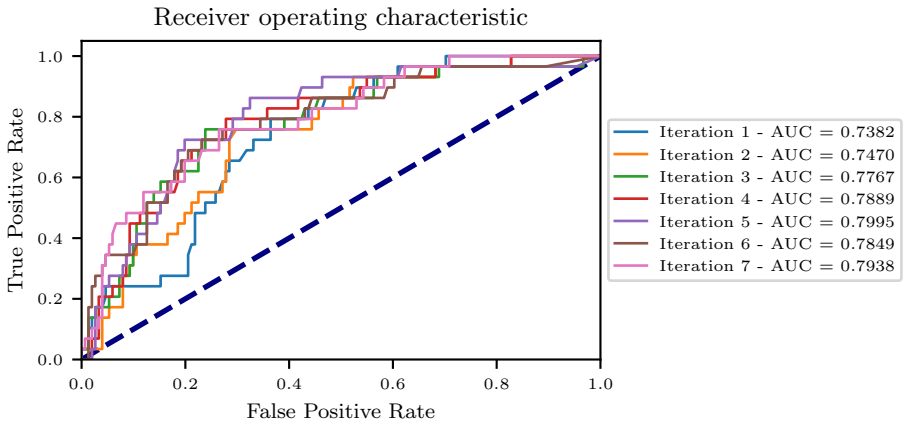
Table 5.2: DNN Deepening results on the selected datasets.

Skin Lesion Dataset (ISIC2016)					
Model	**Test AUC**	**Test Accuracy**	**Test Sensitivity**	**Test Specificity**	**# FLOPs**
ResNet18	0.6927 (+)	0.7942 (+)	0.5333 (+)	0.7961 (-)	1.826×10^9
ResNet34	0.6208 (+)	0.7731 (+)	0.720 (-)	0.4704 (+)	3.681×10^9
ResNet50	0.6275 (+)	0.7836 (+)	0.56 (+)	0.602 (+)	4.139×10^9
DNNDeepening (best)	0.7022	0.7995	0.64	0.6645	1.848×10^{10}
DNNDeepening (mean)	0.6843	0.8047	0.6333	0.6571	2.177×10^{10}
Chest X-Ray Dataset					
Model	**Test AUC**	**Test Accuracy**	**Test Sensitivity**	**Test Specificity**	**# FLOPs**
ResNet18	0.9223 (+)	0.7965 (+)	0.8872 (+)	0.8761 (-)	1.747×10^9
ResNet34	0.9197 (+)	0.8109 (+)	0.9333 (-)	0.812 (+)	3.602×10^9
ResNet50	0.9335 (-)	0.8237 (+)	0.8692 (+)	0.859 (+)	4.059×10^9
DNNDeepening (best)	0.9323	0.8558	0.9128	0.8462	3.833×10^{10}
DNNDeepening (mean)	0.9107	0.7782	0.9087	0.8128	2.227×10^{10}

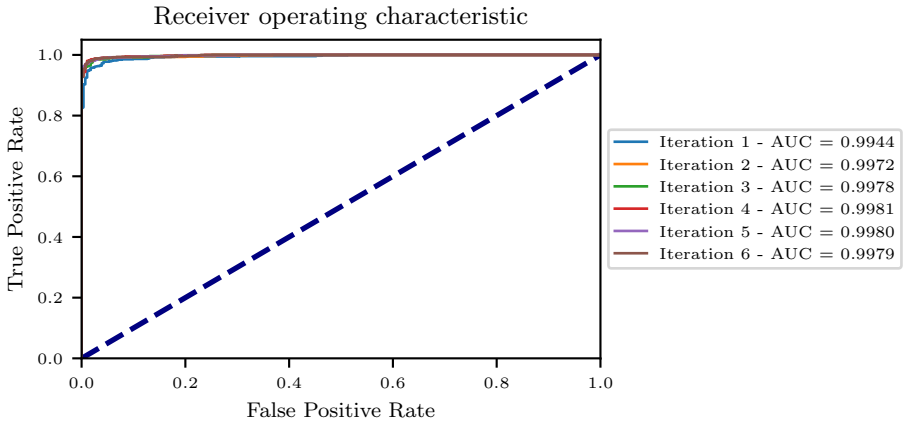
5.4.1 DNN Architecture Deepening Results and Discussion

The overall results of the proposed DNN architecture deepening algorithm is presented in Table 5.2. All results are obtained by evaluating the DNN models in the test set of each dataset. Furthermore, all models are trained and selected using the validation set, and the test set is used only to evaluate them. Notably, the results obtained by the DNN architecture deepening algorithm are compared with the *ResNet18*, *ResNet34*, and *ResNet50* because they have similar architectures. It is necessary to mention that the results of the proposed algorithm are obtained with five independent runs, and the one with the best validation ROC-AUC is reported on Table 5.2. Additionally, in this table, the best model found by the algorithm outperforms the handcrafted ones represented by a $(+)$ sign, while a $(-)$ sign represents situations where a handcrafted one outperformed the best model.

The results presented in Table 5.2 show that it is possible to obtain competitive results in image classification tasks by only adding layers to a DNN model. Moreover, Figures 5.4a and 5.4b shows that the validation ROC-AUC of the model being searched is continuously improving at every iteration, and, as expected from the way the algorithm was designed, it only stops improving in the last two iterations. The



(a) Validation ROC-AUC during *DNN Deepening* for the ISIC2016 dataset.



(b) Validation ROC-AUC during *DNN Deepening* for the Chest X-Ray dataset.

Figure 5.4: Evolution of the validation ROC-AUC during *DNN Deepening*.

reason that the proposed algorithm waits for two iterations before stopping the deepening process is to ensure that local optima architectures can be avoided.

Table 5.3 and Figures 5.5 and 5.6 present the best DNN model found by the proposed DNN Deepening algorithm for both datasets. For the ISIC2016 dataset, the best DNN found has a total of seven residual blocks and a total of 56 convolutional layers, while, for the Chest X-Ray dataset, the best DNN model has a total of six residual blocks and a total of 48 convolutional layers.

Table 5.3: Best DNNs found by the DNN Deepening in the selected datasets.

Skin Lesion Dataset (ISIC2016)				Chest X-Ray Dataset			
Block	**Convolutional Filters**	**Number of Layers**	**Downsampling**	**Block**	**Convolutional Filters**	**Number of Layers**	**Downsampling**
Residual Block 1	50	8	No	Residual Block 1	49	8	No
Residual Block 2	37	6	Yes	Residual Block 2	53	10	No
Residual Block 3	24	8	No	Residual Block 3	29	6	No
Residual Block 4	18	6	No	Residual Block 4	32	8	No
Residual Block 5	63	8	No	Residual Block 5	40	10	No
Residual Block 6	30	10	No	Residual Block 6	32	6	No
Residual Block 7	52	10	No				
Total Number of Layers:		56		**Total Number of Layers:**		48	

Table 5.4: DNN pruning results in the selected datasets.

Skin Lesion Dataset (ISIC2016)						
Model	**Test AUC**	**Test Accuracy**	**Test Sensitivity**	**Test Specificity**	**# FLOPs**	**% of FLOPs decreased**
Original DNN	0.7022	0.7995	0.64	0.6645	1.848×10^{10}	-
Boundary Heavy	0.6878	0.7731	0.6133	0.7138	1.240×10^{10}	32.87
Boundary Light	0.7023	0.8101	0.6800	0.6447	3.09×10^{9}	83.27
Preferable Knee	0.7076	0.8021	0.5867	0.7269	3.814×10^{9}	79.36
Chest X-Ray Dataset						
Model	**Test AUC**	**Test Accuracy**	**Test Sensitivity**	**Test Specificity**	**# FLOPs**	**% of FLOPs decreased**
Original DNN	0.9323	0.8558	0.9128	0.8462	3.833×10^{10}	-
Boundary Heavy	0.9334	0.8061	0.9154	0.8462	2.405×10^{10}	37.25
Boundary Light	0.9257	0.8295	0.9282	0.8162	7.485×10^{9}	80.47
Preferable Knee	0.9376	0.8510	0.8846	0.8718	1.429×10^{10}	62.70

5.4.2 DNN Architecture Pruning Results and Discussion

From Tables 5.2 and 5.3, it is easy to see that, by not allowing parameters to be eliminated, the DNN architecture deepening algorithm finds DNN models with higher computational complexity than the handcrafted ones, as expected. Thus, the pruning of DNN models is an essential part of the proposed DNN architecture searching algorithm proposed in this chapter.

The best models in each dataset found by the DNN Deepening algorithm were pruned using the proposed *DeepPruningES* algorithm with the WIN approach, and the results can be seen in Table 5.4. As stated before, the proposed pruning algorithm uses a user preference vector to select the so-called *preferable knee*. In this case, the algorithm has a preference to maintain the validation ROC-AUC of a candidate solution as high as possible instead of small computational complexity.

Table 5.4 also shows that, although the preferable knee solutions for the ISIC2016 and Chest X-Ray datasets were pruned by 79.36% and 62.70%, respectively. In both datasets, the preferable knee solution's Test AUC outperformed the original

model after pruning. The results in Table 5.4 shows that iteratively increase the computational complexity of a DNN model may improve its performance. However, it does add too many redundant parameters to the model. Thus, pruning algorithms can also be considered an essential part of the design of DNN architectures.

The evolution of the population of candidate solutions through the generations in each dataset can be seen in Figures 5.7a and 5.7b. Because the algorithm starts with an original model with all of its convolutional filters enabled, thus, most candidate solutions have a higher computational complexity than the ones in the last generation. As the convolutional filters are eliminated, the computational complexity and the validation ROC-AUC of the candidate solutions start to decrease. Consequently, at the end of the pruning process, the selected solutions are retrained for 200 epochs in order for them to regain some of their performance.

5.5 Final Remarks

This chapter presented a unified framework for DNN architecture designing consisting of a deepening and a pruning phases. The deepening phase allows a DNN model to grow more complex over time, while the pruning phase eliminates redundant parameters added during the deepening phase. The proposed framework was tested in two medical imaging datasets and showed competitive results with handcrafted networks and used less prior knowledge about the problem at hand. The work developed in this chapter also shows that DNN architecture searching and pruning can be seen as two essential parts in the automatic design of DNN architectures.

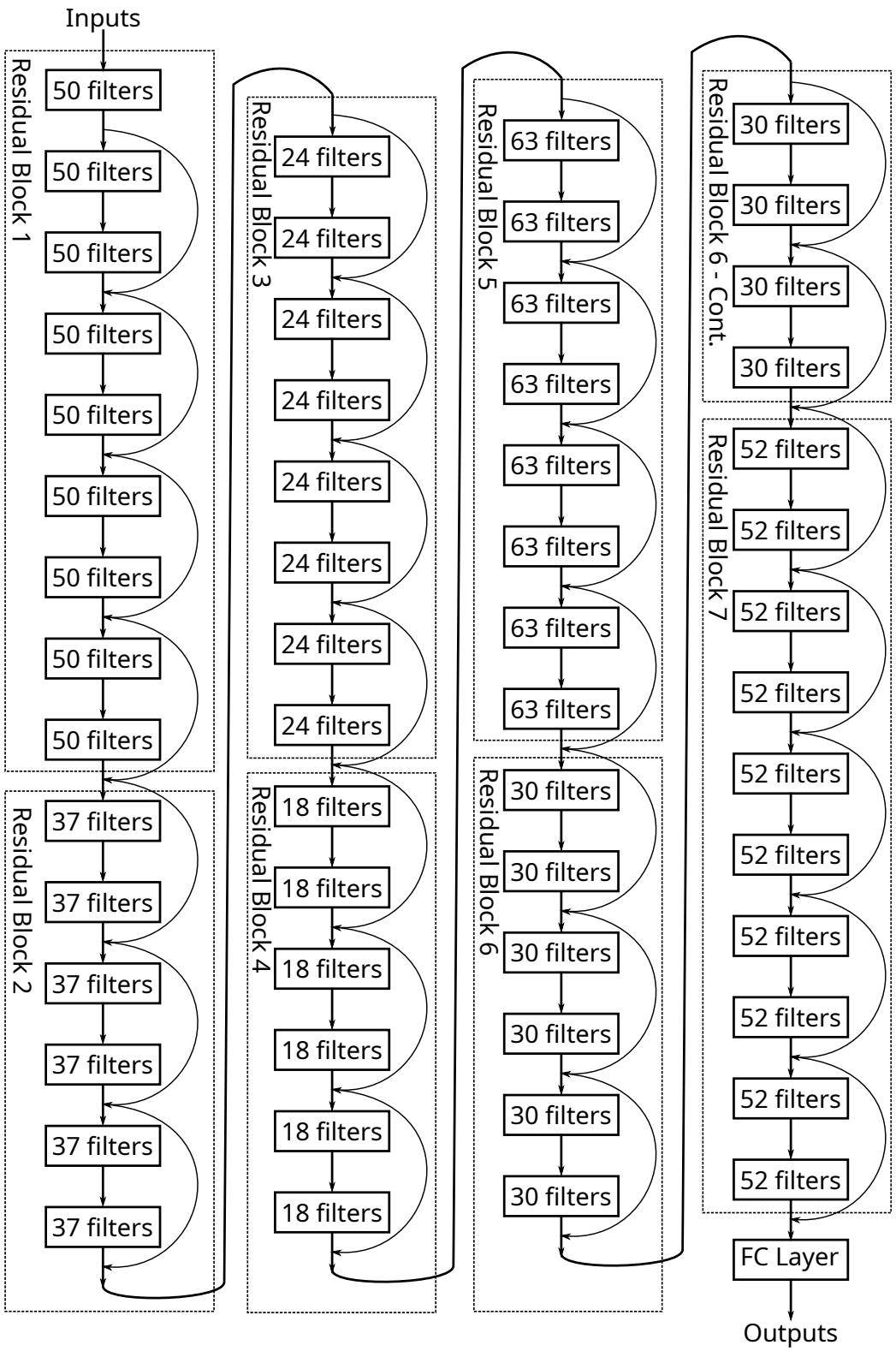


Figure 5.5: Best DNN found by the proposed *DNN Deepening* algorithm for the ISIC 2016 dataset.

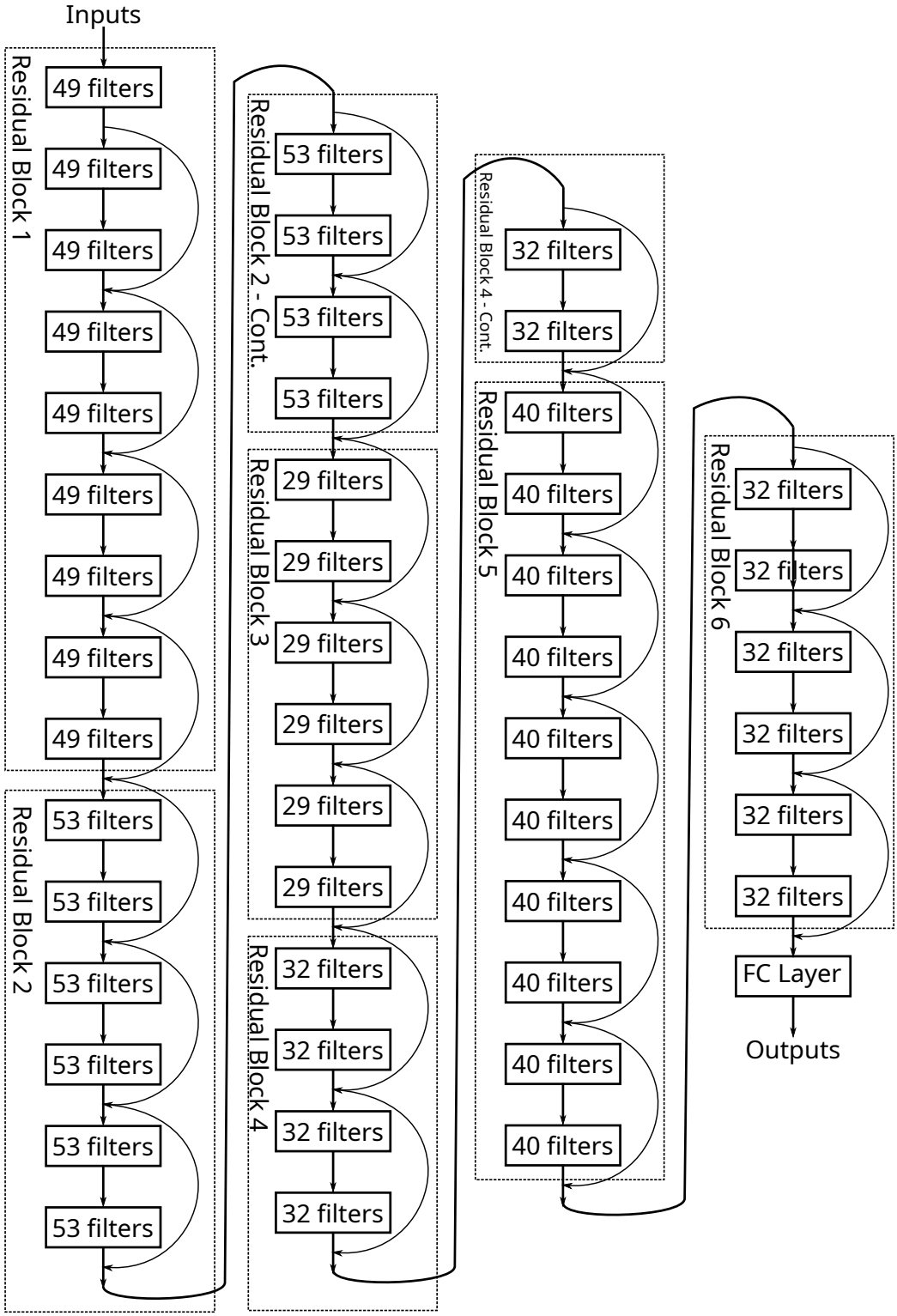
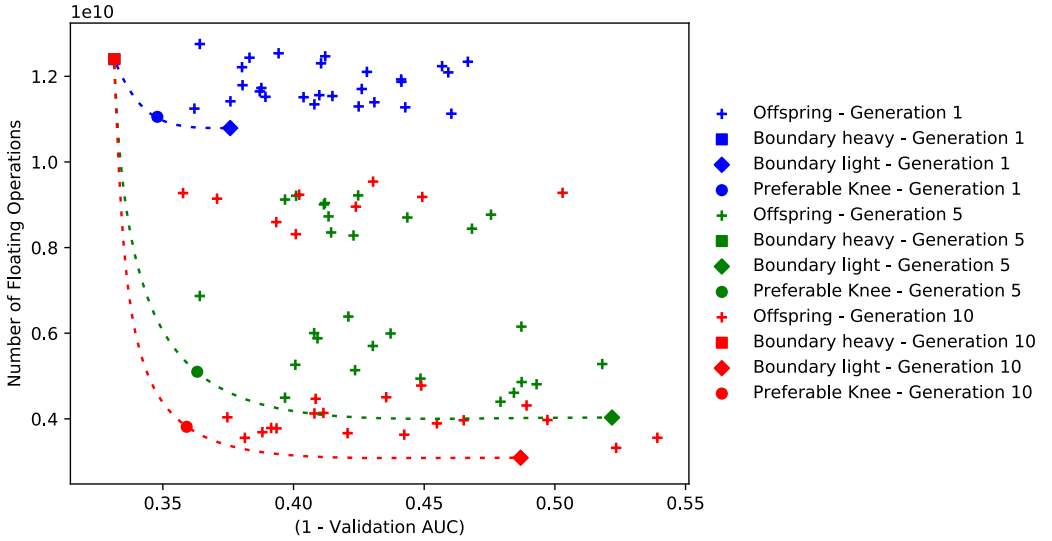
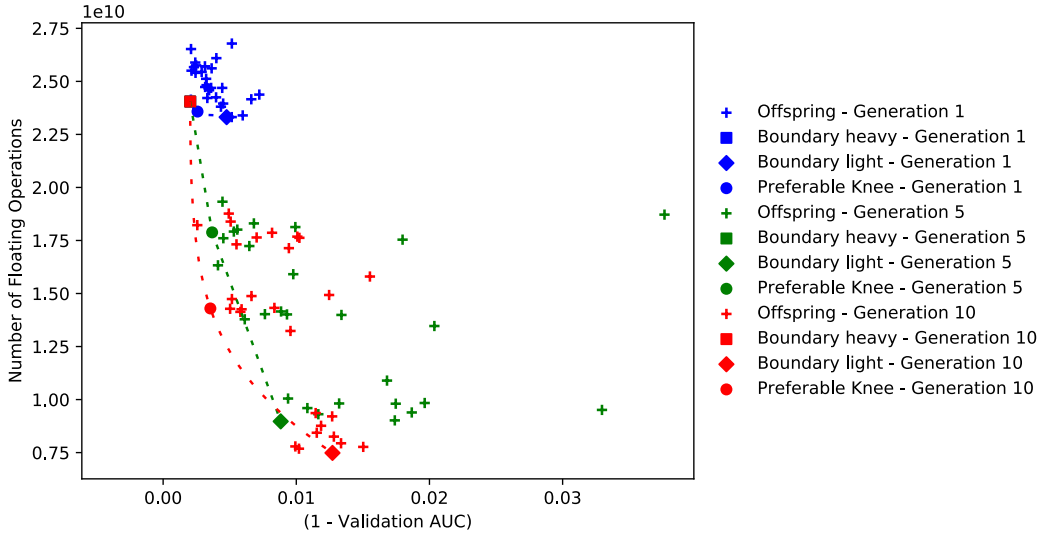


Figure 5.6: Best DNN found by the proposed *DNN Deepening* algorithm for the Chest X-Ray dataset.



(a) Evolution of the population on the ISIC2016 dataset.



(b) Evolution of the population on the Chest X-ray dataset.

Figure 5.7: Evolution of the population during the pruning stage in generations 1, 5, and 10.

CHAPTER VI

GENERATIVE ADVERSARIAL NETWORK ARCHITECTURE PRUNING WITH EVOLUTION STRATEGY

In this chapter, the DNN architecture pruning algorithm presented before is extended and used to prune a specific Generative Adversarial Network (GAN) used for chest x-ray image synthesis.

6.1 Introduction

Deep Learning (DL) models have the potential to revolutionize the field of medical diagnosis due to their astonishing classification capabilities. However, Deep Neural Networks (DNNs) require large amounts of data to be adequately trained, which limits their application in the medical field. Due to concerns about the privacy of patients and the need for experts to perform annotations, large amounts of medical data are expensive and challenging to obtain [123].

Deep Learning practitioners often rely on the use of data augmentation, such as image rotations and cropping, to develop DNN-based solutions for the medical field. However, data augmentation is often not enough to produce meaningful new data from a small dataset. Thus, Generative Adversarial Networks (GANs) are becoming an essential tool in the generation of new and useful data that can be later used to train large DNN models. GANs have been successfully used to synthesize Magnetic Resonance Images (MRIs) [124, 125], Computer Tomography (CT) images [126, 127],

chest x-ray images [128, 129, 130], and others. Therefore, due to their popularity in the field of medical imaging synthesis, GANs are promising candidates to further test the proposed DNN architecture pruning presented in Chapter IV.

Vanilla GANs usually are composed of two DNN models. One of them is called the *Generator*, and it is responsible for synthesizing new data. The other is called the *Discriminator*, and it is responsible for classifying images as real or fake. During training, the *Generator* tries to fool the *Discriminator* in thinking that synthesized images are real ones. GAN training is often compared to a money forger trying to trick a detective in thinking that fake money is real. However, the *Discriminator* is only needed when training the *Generator*. In general, the architecture of the *Discriminator* is a traditional Convolutional Neural Network with a single output, while the architecture of the *Generator* is composed of transposed convolutional layers, which upsamples its inputs to produce a higher resolution output. Once trained, the *Generator* can be used by itself to synthesize new images.

Thus, only the *Generator* is pruned with the proposed DNN architecture pruning algorithm because it is the end product of a GAN training session. Although the Generator has only transposed convolutional layers, its filters are arranged similarly to the ones found in standard CNN architectures. Thus, a similar filter representation used with CNNs in Chapter IV can be used to prune GANs. Moreover, the results presented in this chapter demonstrate that the use of DNN architecture pruning has many applications beyond image classification tasks. In the next sections, a description of the proposed algorithm, experimental design, and results are presented.

Algorithm 12: Proposed *GANPruningES*

Input : Offspring size (λ_{size}), maximum number of generations (gen), mutation probability (p_m), original *Generator* model ($generator$), original *Discriminator* model ($discriminator$), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}), number of epochs for fine-tuning (e_{fine}), learning rate for fine-tuning (α_{fine}).

Output: A pruned Generator model.

1 $\mathbf{1}, \boldsymbol{\lambda} \leftarrow$ *Initialize Population*(λ_{size}, *Generator*, *Discriminator*);
2 **for** *g = 1 to gen* **do**
3 $\quad \mathbf{P} \leftarrow \mathbf{1} + \boldsymbol{\lambda}$
4 $\quad \mathbf{1} \leftarrow$ *Knee Selection*($\mathbf{P}$, *Generator*, *Discriminator*, e_{eval}, α_{eval});
5 $\quad \boldsymbol{\lambda} \leftarrow$ *Offspring Generation*($\mathbf{1}$, λ_{size}, p_m, dnn);
6 **end**
7 *PrunedGenerator* $\leftarrow$ *Fine-tuning*($\mathbf{1}$, e_{fine}, α_{fine});
8 **return** *PrunedGenerator*;

6.2 Proposed Algorithm

The proposed GAN architecture pruning is called here as *GANPruningES*, and it is shown in Algorithm 12. It uses the concepts and ideas developed in the previously shown *DeepPruningES* to find a pruned *Generator* model. The main difference between *GANPruningES* and *DeepPruningES* is that *GANPruningES* only searches for a knee solution in a two-dimensional objective space because of the noisy loss function used in GANs. In practice, the proposed *GANPruningES* is a $(1 + \lambda)$-ES algorithm. Furthermore, it uses the same inputs from *DeepPruningES* with the addition of the original *Generator* and *Discriminator*. Thus, as with all pruning algorithm, a well-trained GAN is needed.

The *Knee Selection* and *Offspring Generation* (Algorithm 12, lines 4, and 5) are similar to the ones developed in Chapter IV. However, instead of selecting three solutions, the proposed *Knee Selection* only selects the individual with the minimum Manhattan distance in the population ($\mathbf{P}$), as shown in Algorithm 13. Another dif-

ference is the *Evaluate Population* function found in the first line of Algorithm 13. Due to the inherent instability produced when training GANs, *Evaluate Population* does not perform any form of training using backpropagation. An individual is evaluated by computing its GAN loss function over the entire dataset together with the fake images produced by the pruned *Generator*, as shown in Algorithm 14. At the end of the pruning process, the selected pruned *Generator* model is retrained from scratch for 10,000 iterations in the chosen medical imaging dataset in order to avoid any training instability.

6.3 Experimental Design

In this section, the chosen *Generator* and *Discriminator* architectures used during the pruning process, as well as the dataset chosen to train the original GAN is presented in detail.

6.3.1 Chosen GAN Architecture

Inspired by the work developed by Salehinejad *et al.* [128, 129], a Deep Convolutional Generative Adversarial Network (DCGAN) was the chosen candidate to synthesize medical images and to be pruned by the proposed *GANPruningES* algorithm. Both *Generator* and *Discriminator* have a total of eight layers. The *Generator* has eight transposed convolutional layers with *Batch Normalization* and $ReLU$ activation functions, illustrated in Figure 6.1a. The *Discriminator* has eight convolutional layers with $LeakyReLU$ activation functions [131], illustrated in Figure 6.1b. The chosen GAN architecture was trained for 2,000 iterations using the Wasserstein Distance as its loss function, which was defined in Chapter II.

Algorithm 13: Knee Selection

Input : Individuals in the population ($\mathbf{P}$), original *Generator* model (*Generator*), original *Discriminator* model (*Discriminator*), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: One individual: knee solution (*knee*).

$\mathbf{P} \leftarrow$ *Evaluate Population*($\mathbf{P}$, *Generator*, *Discriminator*, e_{eval}, α_{eval});
Find $min(f_1), min(f_2), max(f_1), max(f_2)$ in $\mathbf{P}$;
for $k = 1$ to len($\mathbf{P}$) **do**
    $dist(k) = \frac{f_1(P_k) - min(f_1)}{max(f_1) - min(f_1)} + \frac{f_2(P_k) - min(f_2)}{max(f_2) - min(f_2)}$;
end
$knee \leftarrow P_k$, where P_k has the minimum $dist(k)$;
return *knee*;

Algorithm 14: Evaluate Population

Input : Individuals in the population ($\mathbf{P}$), original *Generator* model (*Generator*), original *Discriminator* model (*Discriminator*), number of epochs for individual evaluation (e_{eval}), learning rate for individual evaluation (α_{eval}).

Output: Individual in the population with fitness scores ($\mathbf{P}$).

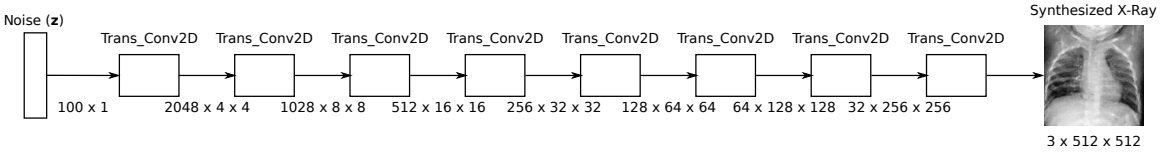
for $i = 1$ to len($\mathbf{P}$) **do**
    Decode the genetic code of the individual (P_i) in the population;
    Compute the average Wasserstein Distance over the entire dataset and fake images;
    Compute the number of FLOPs of the individual (P_i);
    The fitness score of the individual is equal to the average Wasserstein Distance and its number of FLOPs;
end
return $\mathbf{P}$;

6.3.2 Chosen Dataset for GAN Training and Pruning

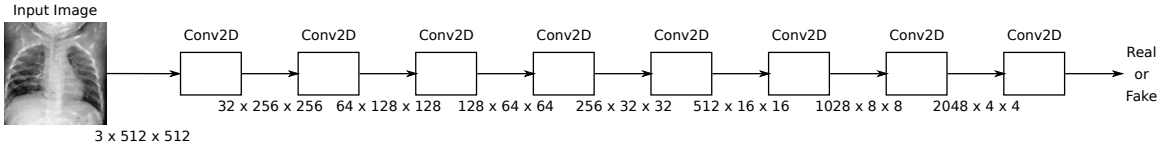
The chosen dataset to train and to prune the chosen GAN is called here as Chest X-Ray showed in Figure 6.2, and it was developed by Kermany *et al.* [121]. It contains a total of 5,232 training images in which 1,349 images are from healthy patients, while 3,883 are from patients diagnosed with pneumonia. There is also a test set with 624 images, where 234 images from healthy patients and 390 images from patients diagnosed with pneumonia. Because this is an unbalanced dataset, the GAN was trained using the 1,349 images of healthy patients to generate chest x-ray images of healthy people. The trained *Generator* can then be later used to generate a balanced dataset in which both classes have the same number of training samples.

6.4 Experimental Results

The results obtained from the original and pruned *Generator* models are presented and discussed in this section.



(a) Representation of the chosen *Generator* architecture.



(b) Representation of the chosen *Discriminator* architecture.

Figure 6.1: The chosen GAN architecture used for pruning.

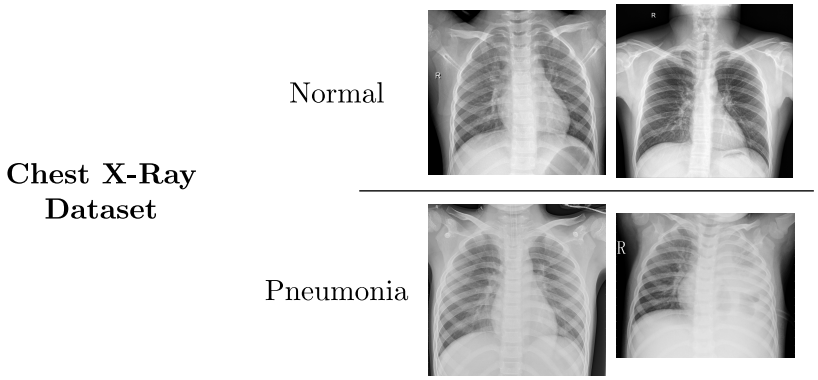


Figure 6.2: Chosen dataset to evaluate the proposed *GANPruningES* algorithm.

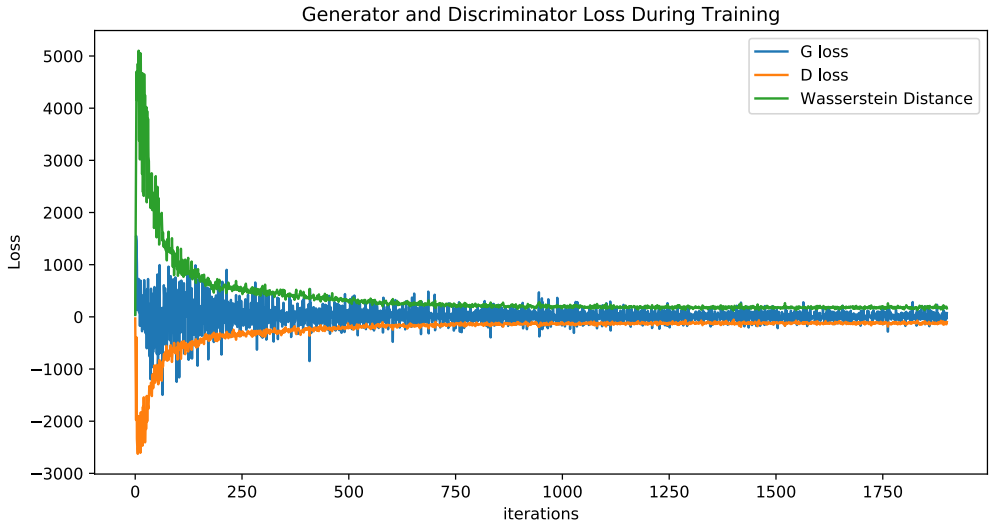


Figure 6.3: Losses of the chosen GAN architecture in the Chest X-Ray dataset for 2,000 iterations.

6.4.1 Original GAN Training Results

The original GAN was trained for 2,000 iterations using the Wasserstein Distance as the loss function. The evolution of the losses from the *Generator*, *Discriminator*, and the Wasserstein distance can be seen in Figure 6.3. The final Wasserstein distance was equal to 143.4226. Unfortunately, there is no other metric to measure the quality of training in the field of GANs. Thus, a fixed noise vector was used throughout the training, and its corresponding synthesized images from the *Generator* was saved. At

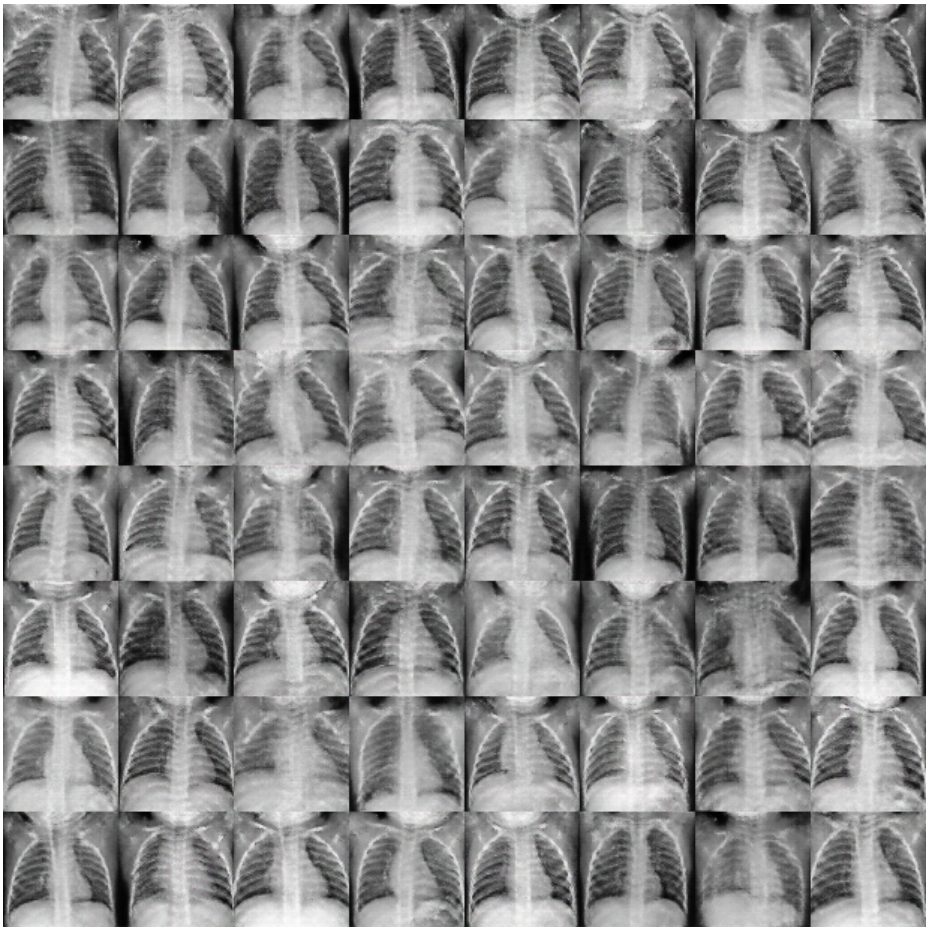


Figure 6.4: Images synthesized by the *original* Generator model.

the 2000^{th} iteration, the quality of the images generated is displayed in Figure 6.4.

6.4.2 Pruned GAN Results

As stated before, the final pruned *Generator* was trained from scratch for a total of 10,000 iterations using the original *Discriminator*, also trained from scratch. The losses of the pruned *Generator*, original *Discriminator*, and Wasserstein distance are shown in Figure 6.5. The final Wasserstein distance was equal to 682.0936. Figure 6.6 shows 64 synthesized images from the pruned *Generator* once the training was finished. Moreover, the proposed *GANPruningES* was capable of eliminating 70.51% of the number of Floating-Point Operations (FLOPs) from the original *Generator*.

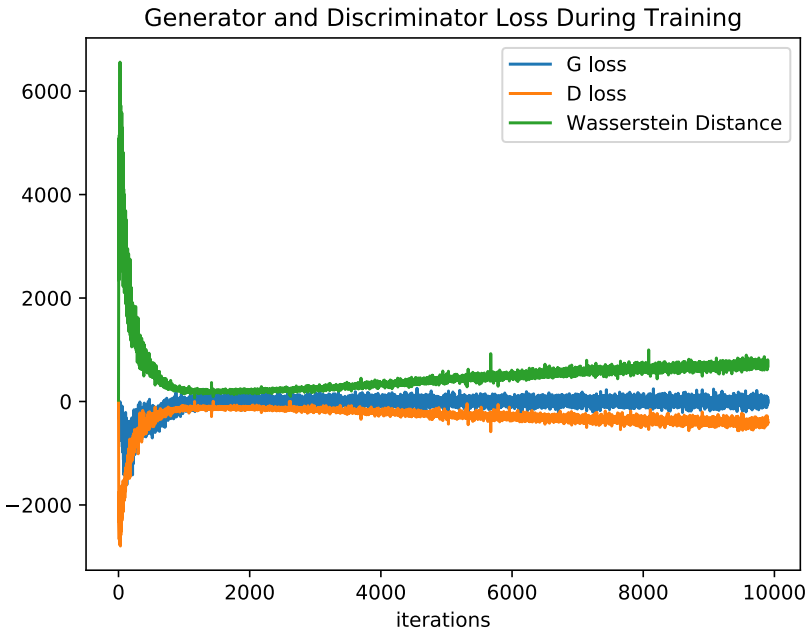


Figure 6.5: Losses of the *pruned* GAN architecture in the Chest X-Ray dataset for 10,000 iterations.

6.4.3 Discussion

The original *Generator* has a total of 2.08×10^7 FLOPs, while the pruned one has a total of 6.14×10^6 FLOPs, which is equal to 70.54% fewer in number of FLOPs than the original model. Moreover, comparing Figures 6.4 and 6.6, the pruned *Generator* was capable of generating chest x-ray images with a similar level of quality compared to the original one, even though the pruned *Generator* contains fewer parameters than the original one.

In order to further test the quality of the images synthesized by the pruned *Generator*, 5,068 images of the healthy patient were generated, where the original Generator generated 2,564 images, and the pruned *Generator* generated 2,564. These generated images were added to the original Chest X-Ray dataset and were used to fine-tuning an AlexNet model pre-trained on the ImageNet challenge. The overall results can be

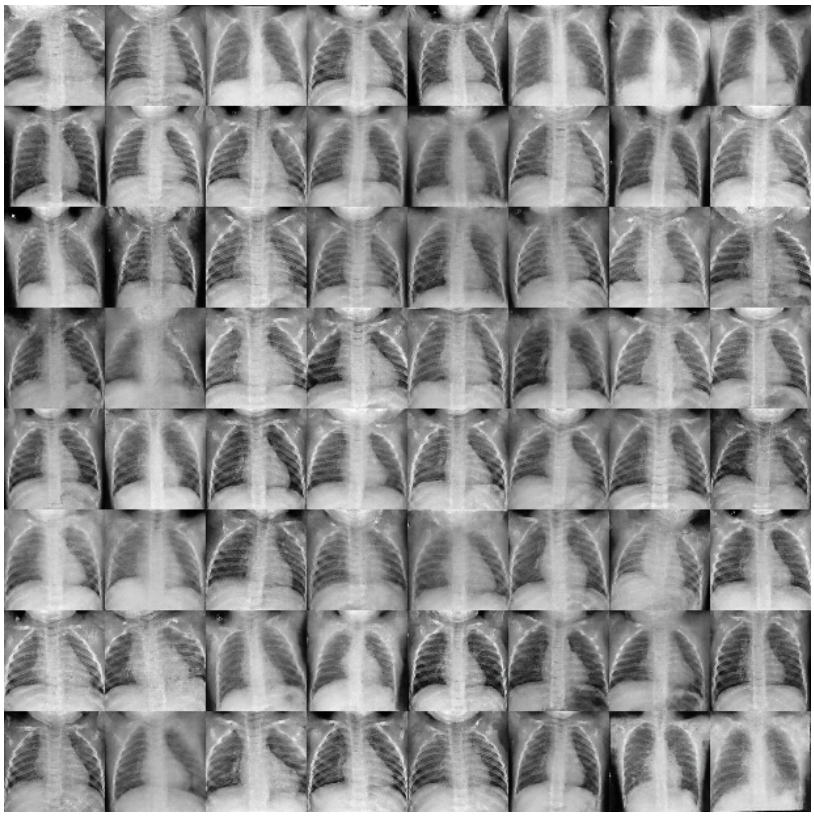


Figure 6.6: Images synthesized by the *pruned* Generator model.

Table 6.1: AlexNet finetuned with generated chest x-ray images.

	Only the original dataset	Original dataset + images from the **original** *Generator*	Original dataset + images from the **pruned** *Generator*
Test Acc	0.8125	0.8151	0.8237
Test AUC	0.9671	0.9656	0.9656
Test Sensitivity	0.9128	0.8897	0.9077
Test Specificity	0.9060	0.9145	0.8974

seen in Table 6.1. From these results, it is easy to see that the pruned *Generator* did not lose any of its synthesis capabilities as compared with the original one.

6.5 Final Remarks

In this chapter, the proposed DNN architecture pruning algorithm presented in Chapter IV was extended and used with Generative Adversarial Networks (GANs) architecture pruning. With minimal modifications, the proposed DNN architecture pruning was capable of reducing the number of Floating-Point Operations (FLOPs) of a *Generator* model without negative impacts on its synthesizing capabilities. This application demonstrates that it is possible to develop an evolutionary framework where compact GANs could be designed automatically. Such GANs would have a significant impact on training DNN models for use by the field of medical diagnostics of rare or uncommon diseases.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Deep Neural Networks (DNNs) are the most ubiquitous machine learning models currently available. They are used to solve a variety of real-world problems, such as object recognition, natural language understanding, and others. However, their development and deployment for consumer use is not a trivial task requiring vast amounts of time by trial and error multiple DNN architectures. The development of DNN-based solutions requires both expertise in the field of deep learning and massive amounts of computational power, which are not always available for interested researchers and experts from other fields. Thus, the development of algorithms and solutions that allows for the automatic generation of DNN architectures and the reduction of their computational complexity is critically needed to advance Deep Learning to other fields outside computer science and engineering. The main objective of this Ph.D. work was to address such problems in the development and deployment of DNN models.

The designing of meaningful DNN architectures can be seen as a discrete optimization problem, where one wants to find the best architecture for a given problem. However, there is no analytical approach available when searching for the best architectures. In that sense, Evolutionary Computation (EC) algorithms are good candidates for use in the designing of DNN architectures. ECs are nature-inspired

algorithms that have been used to solve all types of optimization problems where no analytical solution exists. In the present work, it is proposed the use of Particle Swarm Optimization (PSO), a type of EC algorithm, to search for DNN architectures automatically, called *psoCNN*. More specifically, the proposed algorithm is capable of finding meaningful CNN architectures, a particular type of DNN, used for image classification tasks. PSO is chosen because of its proven convergence speed compared with other EC algorithms, which in turn reduced the average running time for the algorithm to find CNN architectures. Indeed, the proposed *psoCNN* algorithm is capable of finding state-of-the-art CNN architectures faster than the peer competitors' algorithms. *psoCNN* is the best algorithm in six out of nine image classification datasets used to benchmark CNN models. Furthermore, in the other three datasets, *psoCNN* obtained results that are within the top three best algorithms. Therefore, PSO and other EC algorithms can be successfully used during the designing of DNN-based solutions with no prior knowledge required.

An important issue when DNN-based solutions are ready for deployment is that they require massive amounts of computational power to run in real-time. Many of these solutions are developed for deployment in consumer-grade hardware, which, in most cases, does not have enough computational power to run such models in real-time. Thus, the development of an algorithm to reduce the amount of computational power required by state-of-the-art DNN models is also presented in this work. The proposed algorithm is called *DeepPruningES*, and it eliminates convolutional filters from multiple convolutional layers by using Evolution Strategy (ES) and Multi-Criteria Decision making (MCDM). ES is another EC algorithm where candidate solutions are modified at random. In MCDM, a decision-maker (DM) uses information from multiple objective functions being optimized at the same time to choose which candidate solution better fits his or her needs at the moment. The proposed

DeepPruningES is capable of reducing the computational complexity of the state-of-the-art CNN, ResNet, and DenseNet architectures by up to 80% while maintaining reasonable classification performance. Furthermore, *DeepPruningES* uses MCDM to find three specific DNN models with different trade-offs between computational complexity and classification accuracy that can help decision-makers in choosing which one is better to be used in their available hardware.

To the best of the author's knowledge, no other work in the literature considered that DNN architecture searching and pruning can be seen as a unified framework to develop DNN-based solutions. The present work proposed the development of such a framework where DNNs are built as quickly as possible by adding residual blocks on the bottom of existing ones, which are later pruned, called *DNNDeepeningPruning*. Such methodology allowed for a reduced searching space of possible DNN architectures and sped up the searching procedure. Because the process of adding blocks does not allow for the elimination of any existing parameter, the newly created DNN model needs to be pruned in order to remove any parameter in excess. Users are allowed to control the pruning process by specifying their preference of models with reduced computational complexity or higher classification performance. The proposed framework was tested and evaluated with medical imaging diagnostics applications and achieved competitive results compared with hand-crafted DNN models.

For last, the proposed DNN architecture pruning algorithm was applied in the pruning of Generative Adversarial Networks (GANs) used to synthesize medical imaging data. The proposed pruning algorithm can successfully eliminate up to 70% of the parameters in a GAN with no impact on its image generation performance. These results show that DNN architecture pruning can be applied to all sorts of tasks, not only image classification ones.

Therefore, in the present work, highly accurate and compact DNN models used

for image classification and medical imaging diagnostics applications were developed automatically with the help of metaheuristic algorithms. The proposed algorithms and frameworks allow researchers and experts from other fields to take advantage of the learning powers of DNN models without being experts in the field of Deep Learning.

7.2 Future Work

Traditionally, DNN models are developed to solve problems with a well-defined number of classes or labels. For example, if an initial DNN model is created and trained to classify chest x-ray images as coming from healthy patients or patients with pneumonia, this model cannot be later re-purposed to identify which type of pneumonia, viral or bacterial, is present. One could think that adding an extra output to the DNN model and retraining it with new data would suffice. However, this method would result in *catastrophic forgetting* [132]. The main problem of training a DNN in a new problem is that the previously learned weights may not be that important in learning the new data, and they get erased. Thus, the model learns the new data but forgets the old one.

Similar to DNNs, this is a well known problem being in study since the 1990s [132, 133] that has regained attention recently with the popularity of Deep Learning. This problem is currently known as *Continual Learning*, and it is actively researched. Kirkpatrick *et al.* tried to overcome catastrophic forgetting in DNNs with the developed of Elastic Weight Consolidation, where a penalization term is added to the loss function helps maintain the weights related to the old data [134]. Zenke *et al.* used the idea of *Synaptic Intelligence*, where each DNN parameter contains its current value, its past value, and an estimate of importance, to overcome catastrophic forgetting

[135]. However, there are a paucity of evolutionary approaches that tries to overcome this problem. *Evolving Neural Turing Machines* are one of the few examples that use an evolutionary framework to deal with Continual Learning [136, 137]. Nevertheless, they are still very limited and cannot deal with image classification problems. Thus, there are many open opportunities to deal with such an essential problem with the help of evolutionary algorithms.

One suggestion of future work is to search for DNN architectures for Continual Learning using evolutionary algorithms. Specifically, instead of using a single DNN model to learn multiple tasks, a new DNN model could be searched for each new task. In this sense, multiple DNNs would be responsible for classifying different tasks. However, in this case, one would have to study how to fuse the information of multiple DNNs into a single classification output. One could also use evolutionary algorithms to train a single DNN architecture for Continual Learning. In this case, the architecture would be fixed, but the evolutionary algorithm would be used to preserve the weights necessary for old tasks while training the DNN in a new task.

The development of algorithms to deal with Continual Learning would also have ramifications to other fields, such as the field of dynamic optimization problems. In these types of problems, the environment is always changing over time, and new decisions need to be taken once a change is detected. In a sense, dynamic problems are similar to Continual Learning in which the DNN needs to learn the conditions of a new environment. However, these changes can happen at any time. Thus, a model needs to be able to identify such changes and needs to adapt to them as fast as possible. Dynamic problems are also a hot research area with many open opportunities. Many approaches are being used to tackle these problems, such as reinforcement learning [138, 139], and Multi-Criteria Decision Making [140]. Therefore, this is another field

that could benefit from the use of DNN architecture designing based on evolutionary algorithms. Specifically, DNN models and weights could be easily optimized with evolutionary algorithms once an environmental change is detected.

REFERENCES

[1] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 49, pp. 433–460, 1950. [Online]. Available: https://www.abelard.org/turpap/turpap.php

[2] S. J. Russell, P. Norvig, and E. Davis, *Artificial intelligence: a modern approach*, 3rd ed., ser. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010.

[3] C. M. Bishop, *Pattern recognition and machine learning*, ser. Information science and statistics. New York: Springer, 2006.

[4] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, Dec. 1989. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.4.541

[5] 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)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 770–778. [Online]. Available: http://ieeexplore.ieee.org/document/7780459/

[6] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 2261–2269. [Online]. Available: http://ieeexplore.ieee.org/document/8099726/

[7] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[8] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *arXiv:1406.1078 [cs, stat]*, Jun. 2014, arXiv: 1406.1078. [Online]. Available: http://arxiv.org/abs/1406.1078

[9] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, Apr. 1982. [Online]. Available: http://www.pnas.org/cgi/doi/10.1073/pnas.79.8.2554

[10] R. Girshick, "Fast R-CNN," in *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 1440–1448. [Online]. Available: http://ieeexplore.ieee.org/document/7410526/

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 779–788. [Online]. Available: http://ieeexplore.ieee.org/document/7780460/

[12] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean, "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, Dec. 2017. [Online]. Available: https://www.mitpressjournals.org/doi/abs/10.1162/tacl_a_00065

[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[14] Z.-H. Ling, S.-Y. Kang, H. Zen, A. Senior, M. Schuster, X.-J. Qian, H. M. Meng, and L. Deng, "Deep Learning for Acoustic Modeling in Parametric Speech Generation: A systematic review of existing techniques and future trends," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 35–52, May 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7078992

[15] T.-H. Wen, M. Gasic, D. Kim, N. Mrksic, P.-H. Su, D. Vandyke, and S. Young, "Stochastic Language Generation in Dialogue using Recurrent Neural Networks with Convolutional Sentence Reranking," *arXiv:1508.01755*, Aug. 2015, arXiv: 1508.01755. [Online]. Available: http://arxiv.org/abs/1508.01755

[16] T.-H. Wen, M. Gasic, N. Mrkšić, P.-H. Su, D. Vandyke, and S. Young, "Semantically Conditioned LSTM-based Natural Language Generation for Spoken Dialogue Systems," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1711–1721.

[17] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1989. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0893608089900208

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. [Online]. Available: http://www.nature.com/articles/323533a0

[19] M. Chen, S. Mao, and Y. Liu, "Big Data: A Survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, Apr. 2014. [Online]. Available: http://link.springer.com/10.1007/s11036-013-0489-0

[20] K.-S. Oh and K. Jung, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 6, pp. 1311–1314, Jun. 2004. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0031320304000524

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *2015 IEEE International Conference on Computer Vision (ICCV)*. Santiago, Chile: IEEE, Dec. 2015, pp. 1026–1034. [Online]. Available: http://ieeexplore.ieee.org/document/7410480/

[22] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 37. Lille, France: PMLR, Jul. 2015, pp. 448–456. [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.html

[23] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556*, Sep. 2014, arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556

[24] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA: IEEE, Jun. 2015, pp. 1–9. [Online]. Available: http://ieeexplore.ieee.org/document/7298594/

[25] M. Lin, Q. Chen, and S. Yan, "Network In Network," *arXiv:1312.4400 [cs]*, Dec. 2013, arXiv: 1312.4400. [Online]. Available: http://arxiv.org/abs/1312.4400

[26] J. S. Arora, O. A. Elwakeil, A. I. Chahande, and C. C. Hsieh, "Global optimization methods for engineering applications: A review," *Structural Optimization*, vol. 9, no. 3-4, pp. 137–159, Jul. 1995. [Online]. Available: http://link.springer.com/10.1007/BF01743964

[27] D. W. Marquardt, "An Algorithm for Least-Squares Estimation of Nonlinear Parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, Jun. 1963. [Online]. Available: http://epubs.siam.org/doi/10.1137/0111030

[28] M. Mitchell, *An introduction to genetic algorithms*, 5th ed., ser. Complex adaptive systems. Cambridge, Mass.: The MIT Press, 1999.

[29] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, Jan. 1965. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/7.4.308

[30] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4. Perth, WA, Australia: IEEE, 1995, pp. 1942–1948. [Online]. Available: http://ieeexplore.ieee.org/document/488968/

[31] J. Kennedy and R. C. Eberhart, *Swarm Intelligence*. Elsevier, 2001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9781558605954X50001

[32] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, Feb. 1996. [Online]. Available: http://ieeexplore.ieee.org/document/484436/

[33] X.-S. Yang, "Firefly Algorithms for Multimodal Optimization," in *Stochastic Algorithms: Foundations and Applications*, O. Watanabe and T. Zeugmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5792, pp. 169–178. [Online]. Available: http://link.springer.com/10.1007/978-3-642-04944-6_14

[34] C. J. A. Bastos Filho, F. B. de Lima Neto, A. J. C. C. Lins, A. I. S. Nascimento, and M. P. Lima, "A novel search algorithm based on fish school behavior," in *2008 IEEE International Conference on Systems, Man and Cybernetics*. Singapore, Singapore: IEEE, Oct. 2008, pp. 2646–2651. [Online]. Available: http://ieeexplore.ieee.org/document/4811695/

[35] X.-S. Yang and Suash Deb, "Cuckoo Search via Lévy flights," in *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*. Coimbatore, India: IEEE, 2009, pp. 210–214. [Online]. Available: http://ieeexplore.ieee.org/document/5393690/

[36] J. Schaffer, R. A. Caruana, and L. J. Eshelman, "Using genetic search to exploit the emergent behavior of neural networks," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 244–248, Jun. 1990. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0167278990900784

[37] M. Schoenauer and E. Ronald, "Neuro-genetic truck backer-upper controller," in *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. Orlando, FL, USA: IEEE, 1994, pp. 720–723. [Online]. Available: http://ieeexplore.ieee.org/document/349969/

[38] E. Ronald and M. Schoenauer, "Genetic lander: An experiment in accurate neuro-genetic control," in *Parallel Problem Solving from Nature — PPSN III*, G. Goos, J. Hartmanis, J. Leeuwen, Y. Davidor, H.-P. Schwefel, and R. Männer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 866, pp. 452–461. [Online]. Available: http://link.springer.com/10.1007/3-540-58484-6_288

[39] K. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 2. Honolulu, HI, USA: IEEE, 2002, pp. 1757–1762. [Online]. Available: http://ieeexplore.ieee.org/document/1004508/

[40] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, Jun. 2002. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/106365602320169811

[41] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*, ser. Natural computing series. Berlin ; New York: Springer, 2005.

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Miami, FL: IEEE, Jun. 2009, pp. 248–255. [Online]. Available: http://ieeexplore.ieee.org/document/5206848/

[43] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, Technical Report, Apr. 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[44] 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, Nov. 1998. [Online]. Available: http://ieeexplore.ieee.org/document/726791/

[45] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition," *arXiv:1310.1531 [cs]*, Oct. 2013, arXiv: 1310.1531. [Online]. Available: http://arxiv.org/abs/1310.1531

[46] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[47] Xue-Wen Chen and Xiaotong Lin, "Big Data Deep Learning: Challenges and Perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6817512/

[48] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 1, Dec. 2015. [Online]. Available: http://www.journalofbigdata.com/content/2/1/1

[49] J.-H. Luo and J. Wu, "An Entropy-based Pruning Method for CNN Compression," *arXiv:1706.05791*, Jun. 2017, arXiv: 1706.05791. [Online]. Available: http://arxiv.org/abs/1706.05791

[50] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures," *arXiv:1607.03250*, Jul. 2016, arXiv: 1607.03250. [Online]. Available: http://arxiv.org/abs/1607.03250

[51] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *International Conference on Learning Representations*, San Juan, Puerto Rico, USA, 2016, arXiv: 1510.00149. [Online]. Available: http://arxiv.org/abs/1510.00149

[52] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized Neural Networks," in *30th Conference on Neural Information Processing System*, Barcelona, Spain, 2016. [Online]. Available: https://papers.nips.cc/paper/6573-binarized-neural-networks.pdf

[53] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning Convolutional Neural Networks for Resource Efficient Inference," *arXiv:1611.06440*, Nov. 2016, arXiv: 1611.06440. [Online]. Available: http://arxiv.org/abs/1611.06440

[54] C. Ding, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, B. Yuan, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, and Y. Bai, "CIRCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-50 '17*. Cambridge, Massachusetts: ACM Press, 2017, pp. 395–408. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3123939.3124552

[55] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," in *International Conference on Learning Representations*, Toulon, France, 2017.

[56] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression," in *2017 IEEE International Conference on Computer Vision (ICCV)*. Venice: IEEE, Oct. 2017, pp. 5068–5076. [Online]. Available: http://ieeexplore.ieee.org/document/8237803/

[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[58] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesus, *Neural network design*, 2nd ed. Wrocław: Amazon Fulfillment Poland Sp. z o.o, 2014.

[59] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Dec. 2014, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980

[60] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2021068

[61] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 2672–2680. [Online]. Available: http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

[62] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," *arXiv:1511.06434 [cs]*, Jan. 2016, arXiv: 1511.06434. [Online]. Available: http://arxiv.org/abs/1511.06434

[63] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv:1701.07875 [cs, stat]*, Dec. 2017, arXiv: 1701.07875. [Online]. Available: http://arxiv.org/abs/1701.07875

[64] E. K. Burke and G. Kendall, Eds., *Search Methodologies*. Boston, MA: Springer US, 2014. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-6940-7

[65] K. Deb, "Multi-Objective Optimization Using Evolutionary Algorithms: An Introduction," Indian Institute of Technology Kanpur, Kanpur, India, Tech. Rep. KanGAL Report Number 2011003, Feb. 2011. [Online]. Available: https://www.egr.msu.edu/~kdeb/papers/k2011003.pdf

[66] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Computing Surveys*, vol. 45, no. 3, pp. 1–33, Jun. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2480741.2480752

[67] K. Miettinen, *Nonlinear multiobjective optimization*, ser. International series in operations research & management science. Boston: Kluwer Academic Publishers, 1999, no. 12.

[68] A. Sahu, S. K. Panigrahi, and S. Pattnaik, "Fast Convergence Particle Swarm Optimization for Functions Optimization," *Procedia Technology*, vol. 4, pp. 319–324, 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2212017312003271

[69] Wang Hu and G. G. Yen, "Adaptive Multiobjective Particle Swarm Optimization Based on Parallel Cell Coordinate System," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 1–18, Feb. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/6692894/

[70] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies – A comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3–52, 2002. [Online]. Available: http://link.springer.com/10.1023/A:1015059928466

[71] Xin Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, Sep. 1999. [Online]. Available: http://ieeexplore.ieee.org/document/784219/

[72] D. Whitley, T. Starkweather, and C. Bogart, "Genetic algorithms and neural networks: optimizing connections and connectivity," *Parallel Computing*, vol. 14, no. 3, pp. 347–361, Aug. 1990. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/016781919090086O

[73] S. W. Wilson, "Perception redux: Emergence of structure," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 249–256, Jun. 1990. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0167278990900795

[74] Z. Pan and L. Nie, "EVOLVING BOTH THE TOPOLOGY AND WEIGHTS OF NEURAL NETWORKS," *Parallel Algorithms and Applications*, vol. 9, no. 3-4, pp. 299–307, Jun. 1996. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/10637199608915582

[75] S. A. Harp, T. Samad, and A. Guha, "Designing application-specific neural networks using the genetic algorithm," in *Advances in neural information processing systems*, 1990, pp. 447–454. [Online]. Available: http://papers.nips.cc/paper/263-designing-application-specific-neural-networks-using-the-genetic-algorithm.pdf

[76] Y. Kassahun and G. Sommer, "Efficient reinforcement learning through Evolutionary Acquisition of Neural Topologies," in *Proceedings of the 13th European Symposium on Artificial Neural Networks*, Bruges, Belgium, Apr. 2005. [Online]. Available: https://www.informatik.uni-kiel.de/inf/Sommer/doc/Publications/yk/ESANN2005EANT.pdf

[77] N. T. Siebel and G. Sommer, "Evolutionary reinforcement learning of artificial neural networks," *International Journal of Hybrid Intelligent Systems*, vol. 4, no. 3, pp. 171–183, Oct. 2007. [Online]. Available: http://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/HIS-2007-4304

[78] V. Gudise and G. Venayagamoorthy, "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*. Indianapolis, IN, USA: IEEE, 2003, pp. 110–117. [Online]. Available: http://ieeexplore.ieee.org/document/1202255/

[79] M. Carvalho and T. Ludermir, "Particle Swarm Optimization of Feed-Forward Neural Networks with Weight Decay," in *2006 Sixth International Conference on Hybrid Intelligent Systems (HIS'06)*. Rio de Janeiro, Brazil: IEEE, Dec. 2006, pp. 5–5. [Online]. Available: http://ieeexplore.ieee.org/document/4041385/

[80] M. Carvalho and T. B. Ludermir, "Particle Swarm Optimization of Neural Network Architectures and Weights," in *7th International Conference on Hybrid Intelligent Systems (HIS 2007)*. Kaiserslautern, Germany: IEEE, Sep. 2007, pp. 336–339. [Online]. Available: http://ieeexplore.ieee.org/document/4344074/

[81] J.-R. Zhang, J. Zhang, T.-M. Lok, and M. R. Lyu, "A hybrid particle swarm optimization–back-propagation algorithm for feedforward neural network training," *Applied Mathematics and Computation*, vol. 185, no. 2, pp. 1026–1037, Feb. 2007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0096300306008277

[82] S. Kiranyaz, T. Ince, A. Yildirim, and M. Gabbouj, "Evolutionary artificial neural networks by multi-dimensional particle swarm optimization," *Neural Networks*, vol. 22, no. 10, pp. 1448–1462, Dec. 2009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0893608009001038

[83] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks," *Artificial Life*, vol. 15, no. 2, pp. 185–212, Apr. 2009. [Online]. Available: http://www.mitpressjournals.org/doi/10.1162/artl.2009.15.2.15202

[84] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin, "Large-scale Evolution of Image Classifiers," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 2902–2911, event-place: Sydney, NSW, Australia. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3305981

[85] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical Representations for Efficient Architecture Search," *arXiv:1711.00436*, Nov. 2017, arXiv: 1711.00436. [Online]. Available: http://arxiv.org/abs/1711.00436

[86] L. Xie and A. Yuille, "Genetic CNN," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[87] Y. Sun, G. G. Yen, and Z. Yi, "Evolving Unsupervised Deep Neural Networks for Learning Meaningful Representations," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 1, pp. 89–103, Feb. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8300639/

[88] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Completely Automated CNN Architecture Design Based on Blocks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–13, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8742788/

[89] ——, "Evolving Deep Convolutional Neural Networks for Image Classification," *IEEE Transactions on Evolutionary Computation*, pp. 1–1, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8712430/

[90] D. Mittal, S. Bhardwaj, M. M. Khapra, and B. Ravindran, "Studying the plasticity in deep convolutional neural networks using random pruning," *Machine Vision and Applications*, vol. 30, no. 2, pp. 203–216, Mar. 2019. [Online]. Available: http://link.springer.com/10.1007/s00138-018-01001-9

[91] F. E. Fernandes Junior and G. G. Yen, "Particle swarm optimization of deep neural networks architectures for image classification," *Swarm and Evolutionary Computation*, vol. 49, pp. 62–74, Sep. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2210650218309246

[92] D. Decoste and B. Schölkopf, "Training Invariant Support Vector Machines," *Machine Learning*, vol. 46, pp. 161–190, 2002. [Online]. Available: http://link.springer.com/10.1023/A:1012454411458

[93] D. Keysers, T. Deselaers, C. Gollan, and H. Ney, "Deformation Models for Image Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 8, pp. 1422–1435, Aug. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4250467/

[94] J. Sánchez, F. Perronnin, T. Mensink, and J. Verbeek, "Image Classification with the Fisher Vector: Theory and Practice," *International Journal of Computer Vision*, vol. 105, no. 3, pp. 222–245, Dec. 2013. [Online]. Available: http://link.springer.com/10.1007/s11263-013-0636-x

[95] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "A Particle Swarm Optimization-Based Flexible Convolutional Autoencoder for Image Classification," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8571181/

[96] B. Wang, Y. Sun, B. Xue, and M. Zhang, "Evolving Deep Convolutional Neural Networks by Variable-Length Particle Swarm Optimization for Image Classification," in *2018 IEEE Congress on Evolutionary Computation (CEC)*. Rio de Janeiro: IEEE, Jul. 2018, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8477735/

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

[98] H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, "An empirical evaluation of deep architectures on problems with many factors of variation," in *Proceedings of the 24th international conference on Machine learning - ICML '07*. Corvalis, Oregon: ACM Press, 2007, pp. 473–480. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1273496.1273556

[99] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," *arXiv:1708.07747*, Aug. 2017, arXiv: 1708.07747. [Online]. Available: http://arxiv.org/abs/1708.07747

[100] Ming Liang and Xiaolin Hu, "Recurrent convolutional neural network for object recognition," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA: IEEE, Jun. 2015, pp. 3367–3375. [Online]. Available: http://ieeexplore.ieee.org/document/7298958/

[101] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus, "Regularization of Neural Networks using DropConnect," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28. Atlanta, Georgia, USA: PMLR, Jun. 2013, pp. 1058–1066. [Online]. Available: http://proceedings.mlr.press/v28/wan13.html

[102] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive Auto-encoders: Explicit Invariance During Feature Extraction," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11. USA: Omnipress, 2011, pp. 833–840, event-place: Bellevue, Washington, USA. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104482.3104587

[103] T.-H. Chan, K. Jia, S. Gao, J. Lu, Z. Zeng, and Y. Ma, "PCANet: A Simple Deep Learning Baseline for Image Classification?" *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5017–5032, Dec. 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7234886/

[104] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, Apr. 2017, arXiv: 1704.04861. [Online]. Available: http://arxiv.org/abs/1704.04861

[105] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The All Convolutional Net," *arXiv:1412.6806*, Dec. 2014, arXiv: 1412.6806. [Online]. Available: http://arxiv.org/abs/1412.6806

[106] F. E. Fernandes Jr. and G. G. Yen, "Pruning Deep Neural Networks Architectures with Evolution Strategy," *arXiv:1912.11527 [cs]*, Dec. 2019, arXiv: 1912.11527. [Online]. Available: http://arxiv.org/abs/1912.11527

[107] J. Bottleson, S. Kim, J. Andrews, P. Bindu, D. N. Murthy, and J. Jin, "clCaffe: OpenCL Accelerated Caffe for Convolutional Neural Networks," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Chicago, IL, USA: IEEE, May 2016, pp. 50–57. [Online]. Available: http://ieeexplore.ieee.org/document/7529850/

[108] D. Danopoulos, C. Kachris, and D. Soudris, "Acceleration of image classification with Caffe framework using FPGA," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. Thessaloniki: IEEE, May 2018, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/8376580/

[109] M. Su, J. Tan, C.-Y. Lin, J. Ye, C.-H. Wang, and C.-L. Hung, "Constructing a Mobility and Acceleration Computing Platform with NVIDIA Jetson TK1," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. New York, NY: IEEE, Aug. 2015, pp. 1854–1858. [Online]. Available: https://ieeexplore.ieee.org/document/7336442/

[110] S. Lin, N. Liu, M. Nazemi, H. Li, C. Ding, Y. Wang, and M. Pedram, "FFT-based deep learning deployment in embedded systems," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden, Germany: IEEE, Mar. 2018, pp. 1045–1050. [Online]. Available: http://ieeexplore.ieee.org/document/8342166/

[111] W.-Y. Chiu, G. G. Yen, and T.-K. Juan, "Minimum Manhattan Distance Approach to Multiple Criteria Decision Making in Multiobjective Optimization Problems," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 6, pp. 972–985, Dec. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7465803/

[112] X. Ding, G. Ding, J. Han, and S. Tang, "Auto-Balanced Filter Pruning for Efficient Convolutional Neural Networks," in *AAAI Conference on Artificial Intelligence*, New Orleans, Louisiana, USA, 2018. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16450

[113] S. Pereira, A. Pinto, V. Alves, and C. A. Silva, "Brain Tumor Segmentation Using Convolutional Neural Networks in MRI Images," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1240–1251, May 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7426413/

[114] P. Moeskops, M. A. Viergever, A. M. Mendrik, L. S. de Vries, M. J. N. L. Benders, and I. Isgum, "Automatic Segmentation of MR Brain Images With a Convolutional Neural Network," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1252–1261, May 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7444155/

[115] R. K. Samala, H.-P. Chan, L. Hadjiiski, M. A. Helvie, J. Wei, and K. Cha, "Mass detection in digital breast tomosynthesis: Deep convolutional neural network with transfer learning from mammography: DBT mass detection using deep convolutional neural network," *Medical Physics*, vol. 43, no. 12, pp. 6654–6666, Nov. 2016. [Online]. Available: http://doi.wiley.com/10.1118/1.4967345

[116] K. Lekadir, A. Galimzianova, A. Betriu, M. del Mar Vila, L. Igual, D. L. Rubin, E. Fernandez, P. Radeva, and S. Napel, "A Convolutional Neural Network for Automatic Characterization of Plaque Composition in Carotid Ultrasound," *IEEE Journal of Biomedical and Health Informatics*, vol. 21, no. 1, pp. 48–55, Jan. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7752798/

[117] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, vol. 9351, pp. 234–241. [Online]. Available: http://link.springer.com/10.1007/978-3-319-24574-4_28

[118] L. Prechelt, "Early Stopping - But When?" in *Neural Networks: Tricks of the Trade*, G. Goos, J. Hartmanis, J. van Leeuwen, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, vol. 1524, pp. 55–69. [Online]. Available: http://link.springer.com/10.1007/3-540-49430-8_3

[119] W.-Y. Chiu, S. H. Manoharan, and T.-Y. Huang, "Weight Induced Norm Approach to Group Decision Making for Multiobjective Optimization Problems in Systems Engineering," *IEEE Systems Journal*, pp. 1–12, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8850326/

[120] D. Gutman, N. C. F. Codella, E. Celebi, B. Helba, M. Marchetti, N. Mishra, and A. Halpern, "Skin Lesion Analysis toward Melanoma Detection: A Challenge at the International Symposium on Biomedical Imaging (ISBI) 2016, hosted by the International Skin Imaging Collaboration (ISIC)," *arXiv:1605.01397 [cs]*, May 2016, arXiv: 1605.01397. [Online]. Available: http://arxiv.org/abs/1605.01397

[121] D. S. Kermany, M. Goldbaum, W. Cai, C. C. Valentim, H. Liang, S. L. Baxter, A. McKeown, G. Yang, X. Wu, F. Yan, J. Dong, M. K. Prasadha, J. Pei, M. Y. Ting, J. Zhu, C. Li, S. Hewett, J. Dong, I. Ziyar, A. Shi, R. Zhang, L. Zheng, R. Hou, W. Shi, X. Fu, Y. Duan, V. A. Huu, C. Wen, E. D. Zhang, C. L. Zhang, O. Li, X. Wang, M. A. Singer, X. Sun, J. Xu, A. Tafreshi, M. A. Lewis, H. Xia, and K. Zhang, "Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning," *Cell*, vol. 172, no. 5, pp. 1122–1131.e9, Feb. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0092867418301545

[122] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, Jun. 2006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016786550500303X

[123] H. Greenspan, B. van Ginneken, and R. M. Summers, "Guest Editorial Deep Learning in Medical Imaging: Overview and Future Promise of an Exciting New Technique," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1153–1159, May 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7463094/

[124] H.-C. Shin, N. A. Tenenholtz, J. K. Rogers, C. G. Schwarz, M. L. Senjem, J. L. Gunter, K. P. Andriole, and M. Michalski, "Medical Image Synthesis for Data Augmentation and Anonymization Using Generative Adversarial Networks," in *Simulation and Synthesis in Medical Imaging*, A. Gooya, O. Goksel, I. Oguz, and N. Burgos, Eds. Cham: Springer International Publishing, 2018, vol. 11037, pp. 1–11, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-00536-8_1

[125] C. Han, H. Hayashi, L. Rundo, R. Araki, W. Shimoda, S. Muramatsu, Y. Furukawa, G. Mauri, and H. Nakayama, "GAN-based synthetic brain MR image generation," in *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. Washington, DC: IEEE, Apr. 2018, pp. 734–738. [Online]. Available: https://ieeexplore.ieee.org/document/8363678/

[126] M. J. M. Chuquicusma, S. Hussein, J. Burt, and U. Bagci, "How to fool radiologists with generative adversarial networks? A visual turing test for lung cancer diagnosis," in *2018 IEEE 15th International Symposium on Biomedical Imaging (ISBI 2018)*. Washington, DC: IEEE, Apr. 2018, pp. 240–244. [Online]. Available: https://ieeexplore.ieee.org/document/8363564/

[127] M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "GAN-based Synthetic Medical Image Augmentation for increased CNN Performance in Liver Lesion Classification," *Neurocomputing*, vol. 321, pp. 321–331, Dec. 2018, arXiv: 1803.01229. [Online]. Available: http://arxiv.org/abs/1803.01229

[128] H. Salehinejad, S. Valaee, T. Dowdell, E. Colak, and J. Barfett, "Generalization of Deep Neural Networks for Chest Pathology Classification in X-Rays Using Generative Adversarial Networks," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Calgary, AB: IEEE, Apr. 2018, pp. 990–994. [Online]. Available: https://ieeexplore.ieee.org/document/8461430/

[129] H. Salehinejad, E. Colak, T. Dowdell, J. Barfett, and S. Valaee, "Synthesizing Chest X-Ray Pathology for Training Deep Convolutional Neural Networks," *IEEE Transactions on Medical Imaging*, vol. 38, no. 5, pp. 1197–1206, May 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8534421/

[130] M. Moradi, A. Madani, A. Karargyris, and T. F. Syeda-Mahmood, "Chest x-ray generation and data augmentation for cardiovascular abnormality classification," in *Medical Imaging 2018: Image Processing*, E. D. Angelini and B. A. Landman, Eds. Houston, United States: SPIE, Mar. 2018, p. 57. [Online]. Available: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10574/2293971/Chest-x-ray-generation-and-data-augmentation-for-cardiovascular-abnormality/10.1117/12.2293971.full

[131] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[132] M. McCloskey and N. J. Cohen, "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem," in *Psychology of Learning and Motivation*. Elsevier, 1989, vol. 24, pp. 109–165. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0079742108605368

[133] R. Ratcliff, "Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions." *Psychological Review*, vol. 97, no. 2, pp. 285–308, 1990. [Online]. Available: http://doi.apa.org/getdoi.cfm?doi=10.1037/0033-295X.97.2.285

[134] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the National Academy of Sciences*, vol. 114, no. 13, pp. 3521–3526, Mar. 2017. [Online]. Available: http://www.pnas.org/lookup/doi/10.1073/pnas.1611835114

[135] F. Zenke, B. Poole, and S. Ganguli, "Continual Learning Through Synaptic Intelligence," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 3987–3995, event-place: Sydney, NSW, Australia. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3306093

[136] R. B. Greve, E. J. Jacobsen, and S. Risi, "Evolving neural turing machines," in *Neural Information Processing Systems: Reasoning, Attention, Memory Workshop*, 2015.

[137] B. Lüders, M. Schläger, and S. Risi, "Continual learning through evolvable neural turing machines," in *NIPS 2016 Workshop on Continual Learning and Deep Networks (CLDL 2016)*, 2016.

[138] Z. Xu, Y. Wang, J. Tang, J. Wang, and M. C. Gursoy, "A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs," in *2017 IEEE International Conference on Communications (ICC)*. Paris, France: IEEE, May 2017, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7997286/

[139] Z. Li, T. Zhao, F. Chen, Y. Hu, C.-Y. Su, and T. Fukuda, "Reinforcement Learning of Manipulation and Grasping Using Dynamical Movement Primitives for a Humanoidlike Mobile Manipulator," *IEEE/ASME Transactions on Mechatronics*, vol. 23, no. 1, pp. 121–131, Feb. 2018. [Online]. Available: http://ieeexplore.ieee.org/document/7953692/

[140] F. Zou, G. G. Yen, and L. Tang, "A knee-guided prediction approach for dynamic multi-objective optimization," *Information Sciences*, vol. 509, pp. 193–209, Jan. 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0020025519308606

VITA

Francisco Erivaldo Fernandes Junior

Candidate for the Degree of

Doctor of Philosophy

Dissertation: AUTOMATIC DESIGN OF DEEP NEURAL NETWORK ARCHITECTURES WITH EVOLUTIONARY COMPUTATION

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Doctor of Philosophy in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma, USA in May, 2020.

Completed the requirements for the Master of Science in Electrical Engineering at Campinas State University, Campinas, São Paulo, Brazil, 2014.

Completed the requirements for the Technology degree in Industrial Mechatronics at Federal Institute of Education, Science and Technology of Ceará, Fortaleza, Ceará, Brazil, 2012.