

Yanan Sun
Gary G. Yen
Mengjie Zhang

Evolutionary Deep Neural Architecture Search: Fundamentals, Methods, and Recent Advances

Studies in Computational Intelligence

Volume 1070

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

The series “Studies in Computational Intelligence” (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

This series also publishes Open Access books. A recent example is the book Swan, Nivel, Kant, Hedges, Atkinson, Steunebrink: The Road to General Intelligence <https://link.springer.com/book/10.1007/978-3-031-08020-3>.

Indexed by SCOPUS, DBLP, WTI Frankfurt eG, zbMATH, SCImago.

All books published in the series are submitted for consideration in Web of Science.

Yanan Sun · Gary G. Yen · Mengjie Zhang

Evolutionary Deep Neural Architecture Search: Fundamentals, Methods, and Recent Advances



Springer

Yanan Sun
Department of Artificial Intelligence,
College of Computer Science
Sichuan University
Chengdu, Sichuan, China

Mengjie Zhang
School of Engineering and Computer
Science
Victoria University of Wellington
Wellington, New Zealand

Gary G. Yen
School of Electrical and Computer
Engineering, Intelligent Systems
and Control Laboratory
Oklahoma State University
Stillwater, OK, USA

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-031-16867-3 ISBN 978-3-031-16868-0 (eBook)
<https://doi.org/10.1007/978-3-031-16868-0>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

The use of Evolutionary Computation (EC) methods to create optimal or nearly optimal Deep Neural Network (DNN) architectures is referred to as evolutionary deep neural architecture design. The design process of architectures is often formalized as an optimization problem, where EC algorithms are correctly created to tackle the optimization problem.

DNNs have had remarkable success in many complicated practical applications in recent years. It is well known that the performance of a DNN is only promising when the architecture is appropriate. The architecture, on the other hand, is typically created by hand, needing a high level of skill that is in scarce supply in practice. A promising deep architecture is difficult to create in practice without this kind of skill, which is frequently DNN expertise and domain understanding of the problem to be solved. Reinforcement learning techniques, gradient-based optimization algorithms, and EC methods are the three common methods utilized to construct the architectures of DNNs in the literature. This book mainly focuses on the EC methods for deep neural architecture design.

In this book, we will first introduce the fundamentals of commonly used EC methods, including Genetic Algorithm (GA) [1], Particle Swarm Optimization (PSO) [2], Differential Evolution (DE) [3], and Genetic Programming (GP) [4]. Following that, we will go through two different forms of evolutionary deep neural architecture design algorithms. They are the design algorithms for unsupervised DNNs, and the design algorithms for supervised DNNs. In addition, we will also discuss some recent efforts to speed up the execution of such algorithms. These algorithms are primarily based on the authors recent work, which has been published in journals and international conferences devoted to EC and neural networks. We think that by presenting them all together in this book, readers will be able to absorb related information more quickly and conveniently.

This book is composed of five parts, and each part is fulfilled with the necessary content in different chapters. In the remainder of Part 1, we will provide an overview

of the fundamentals and backgrounds on EC and deep learning with a focus on different types of commonly used DNNs.

Chengdu, China
Stillwater, USA
Wellington, New Zealand

Yanan Sun
Gary G. Yen
Mengjie Zhang

References

1. John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.
2. James Kennedy and R Eberhart Particle Swarm Optimization. Ieee int. In *Conf. on Neural Networks*, volume 4, 1995.
3. Rainer Storn and Kenneth Price. Differential evolution âAS a simple and efficient heuristic for global optimization over continuous spaces, 1997. URL <https://doi.org/10.1023/A:1008202821328>.
4. Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

Contents

Part I Fundamentals and Backgrounds

1	Evolutionary Computation	3
1.1	Genetic Algorithms (GAs)	3
1.2	Particle Swarm Optimization (PSO)	4
1.3	Differential Evolution (DE)	5
1.4	Genetic Programming (GP)	6
1.5	Chapter Summary	7
	References	7
2	Deep Neural Networks	9
2.1	Deep Belief Networks	12
2.2	Stacked Auto-Encoders	13
2.2.1	Sparse Auto-Encoders	14
2.2.2	Weight Decay Auto-Encoders	15
2.2.3	Denoising Auto-Encoders (DAEs)	15
2.2.4	Contractive Auto-Encoders	16
2.2.5	Convolutional Auto-Encoders (CAEs)	17
2.2.6	Variational Auto-Encoders (VAEs)	18
2.3	Convolutional Neural Networks (CNNs)	19
2.3.1	CNN Skeleton	20
2.3.2	Convolution	20
2.3.3	Pooling	21
2.3.4	Reflect Padding	22
2.3.5	Batch Normalization (BN)	23
2.3.6	ResNet Blocks (RBs) and DenseNet Blocks (DBs)	23
2.4	Benchmarks for Deep Neural Networks	24
2.5	Chapter Summary	27
	References	28

**Part II Evolutionary Deep Neural Architecture Search
for Unsupervised DNNs**

3	Architecture Design for Stacked AEs and DBNs	39
3.1	Introduction	39
3.2	Related Work and Motivations	40
3.2.1	Unsupervised Deep Learning	40
3.2.2	Evolutionary Algorithms for Evolving Neural Networks	41
3.3	Algorithm Details	43
3.3.1	Framework of EUDNN	43
3.3.2	Evolving Connection Weights and Activation Functions	45
3.3.3	Fine-Tuning Connection Weights	47
3.3.4	Discussion	48
3.4	Experimental Design	49
3.4.1	Performance Metric	50
3.4.2	Peer Competitors	50
3.4.3	Parameter Settings	51
3.5	Experimental Results and Analysis	52
3.5.1	Performance of EUDNN	52
3.5.2	Analysis on Pre-training of EUDNN	54
3.5.3	Analysis on Fine-Tuning of EUDNN	55
3.5.4	Representation Visualizations	56
3.6	Chapter Summary	57
	References	58
4	Architecture Design for Convolutional Auto-Encoders	61
4.1	Introduction	61
4.2	Motivation of FCAE	62
4.3	Algorithm Details	64
4.3.1	Algorithm Overview	64
4.3.2	Encoding Strategy	64
4.3.3	Particle Initialization	65
4.3.4	Fitness Evaluation	66
4.3.5	Velocity Calculation and Position Update	67
4.3.6	Deep Training on Global Best Particle	70
4.4	Experimental Design	70
4.4.1	Peer Competitors	70
4.4.2	Parameter Settings	70
4.5	Experimental Results and Analysis	71
4.5.1	Overview Performance	71
4.5.2	Evolution Trajectory of PSOAO	72
4.5.3	Performance on Different Numbers of Training Examples	73
4.5.4	Investigation on x -Reference Velocity Calculation	74

4.6	Chapter Summary	75
	References	75
5	Architecture Design for Variational Auto-Encoders	79
5.1	Introduction	79
5.2	Algorithm Details	80
5.2.1	Algorithm Overview	80
5.2.2	Strategy of Gene Encoding	81
5.2.3	Initialization of Population	85
5.2.4	Evaluation	86
5.2.5	Crossover Operator and Mutation Operator	88
5.2.6	Environmental Selection	90
5.3	Experimental Design	90
5.3.1	Parameter Setting	91
5.3.2	Peer Competitors	91
5.3.3	Performance Evaluation	92
5.4	Experimental Results and Analysis	93
5.4.1	Overall Performance	93
5.4.2	Evolution Trajectory of EvoVAE	97
5.4.3	Running Time	99
5.4.4	The Obtained Architecture	99
5.4.5	Ablation Experiments	102
5.5	Chapter Summary	103
	References	104

Part III Evolutionary Deep Neural Architecture Search for Supervised DNNs

6	Architecture Design for Plain CNNs	109
6.1	Introduction	109
6.2	Algorithm Details	110
6.2.1	Algorithm Overview	110
6.2.2	Strategy of Gene Encoding	111
6.2.3	Initialization of Population	112
6.2.4	Evaluation of Fitness	113
6.2.5	Slack Binary Tournament Selection	115
6.2.6	Offspring Generation	116
6.2.7	Environmental Selection	117
6.2.8	Select and Decode Best Individual	118
6.3	Experimental Design	118
6.3.1	Peer Competitors	119
6.3.2	Parameter Settings	119
6.4	Experimental Results and Discussion	119
6.4.1	Overall Results	120
6.4.2	Performance of Weight Initialization	122
6.4.3	Discussion	123

6.5	Chapter Summary	124
	References	125
7	Architecture Design for RBs and DBs Based CNNs	127
7.1	Introduction	127
7.2	Algorithm Details	127
7.2.1	Algorithm Overview	128
7.2.2	Population Initialization	129
7.2.3	Fitness Evaluation	131
7.2.4	Offspring Generation	132
7.2.5	Environmental Selection	135
7.3	Experimental Design	136
7.3.1	Peer Competitors	136
7.3.2	Parameter Settings	136
7.4	Experimental Results and Analysis	137
7.4.1	Performance Overview	138
7.4.2	Evolution Trajectory	140
7.4.3	Designed CNN Architectures	142
7.5	Chapter Summary	143
	References	144
8	Architecture Design for Skip-Connection Based CNNs	147
8.1	Introduction	147
8.2	Algorithm Details	148
8.2.1	Algorithm Overview	148
8.2.2	Population Initialization	149
8.2.3	Fitness Evaluation	152
8.2.4	Offspring Generating	155
8.2.5	Environmental Selection	156
8.3	Experimental Design	157
8.3.1	Peer Competitors	157
8.3.2	Parameter Settings	158
8.4	Experimental Results and Analysis	159
8.4.1	Overall Results	159
8.4.2	Transferable Performance on ImageNet	163
8.4.3	Performance of Crossover Operator	164
8.4.4	Performance of Acceleration Components	166
8.4.5	Evolution Trajectory	167
8.5	Chapter Summary	168
	References	168
9	Hybrid GA and PSO for Architecture Design	171
9.1	Introduction	171
9.2	Algorithm Details	172
9.2.1	Overall Structure of the System	172
9.2.2	The Evolved CNN Architecture-DynamicNet	172

9.2.3	HGAPSO Encoding Strategy	173
9.2.4	HGAPSO Search	174
9.2.5	HGAPSO Fitness Evaluations	176
9.3	Experimental Studies	177
9.3.1	Parameter Settings	177
9.3.2	State-of-the-Art Methods Versus HGAPSO	177
9.3.3	Evolved CNN Architecture	178
9.3.4	One-Run Result on CIFAR-10 Dataset	179
9.4	Chapter Summary	179
	References	180
10	Internet Protocol Based Architecture Design	181
10.1	Introduction	181
10.2	Algorithm Details	182
10.2.1	Algorithm Overview	182
10.2.2	Encoding Strategy of Particle	183
10.2.3	Initialization of Population	185
10.2.4	Evaluation of Fitness	186
10.2.5	Update Particle with Velocity Clamping	186
10.2.6	Selection and Decoding of Best Individual	187
10.3	Experimental Design	188
10.3.1	Peer Competitors	188
10.3.2	Parameter Settings	188
10.4	Experimental Results and Analysis	189
10.4.1	Overall Performance	189
10.4.2	Evolved CNN Architectures	189
10.4.3	Trajectory Visualization	190
10.5	Chapter Summary	191
	References	192
11	Differential Evolution for Architecture Design	193
11.1	Introduction	193
11.2	Algorithm Details	193
11.2.1	DECNN Algorithm Overview	194
11.2.2	Adjusted IP-Based Encoding Strategy	194
11.2.3	Population Initialization	197
11.2.4	Fitness Evaluation	197
11.2.5	DECNN DE Mutation and Crossover	198
11.2.6	DECNN Second Crossover	198
11.3	Experimental Design	199
11.3.1	State-of-the-Art Competitors	199
11.3.2	Parameter Settings	199
11.4	Experimental Results and Analysis	199
11.4.1	DECNN Versus State-of-the-Art Methods	199
11.4.2	DECNN Versus IPPSO	201
11.4.3	Evolved CNN Architecture	201

11.5 Chapter Summary	202
References	202
12 Architecture Design for Analyzing Hyperspectral Images	203
12.1 Introduction	203
12.2 Algorithm Details	205
12.2.1 Algorithm Overview	205
12.2.2 Gene Encoding Strategy	205
12.2.3 Offspring Generation	207
12.2.4 Environmental Selection	209
12.3 Experimental Design	210
12.3.1 Benchmark Dataset	210
12.3.2 Peer Competitors	211
12.3.3 Parameter Settings	211
12.3.4 Training Details	212
12.4 Experimental Results and Analysis	212
12.4.1 Overall Results	213
12.4.2 Comparisons with Artificial-CNN	214
12.5 Chapter Summary	214
References	215

Part IV Recent Advances in Evolutionary Deep Neural Architecture Search

13 Encoding Space Based on Directed Acyclic Graphs	223
13.1 Introduction	223
13.2 Algorithm Details	224
13.2.1 Encoding Strategy Overview	224
13.2.2 Representation and Decoding Details	225
13.2.3 Initialization Algorithm Overview	226
13.2.4 Initialization Algorithm Details	228
13.3 Experimental Studies	232
13.3.1 Overview	232
13.3.2 Parameter Settings	232
13.3.3 Experimental Results	233
13.4 Chapter Summary	234
References	235
14 End-to-End Performance Predictors	237
14.1 Introduction	237
14.2 Related Work	238
14.3 Algorithm Details	239
14.3.1 Encoding	240
14.3.2 Training of the Random Forest	242
14.3.3 Performance Prediction	242
14.3.4 Strength and Weakness of E2EPP	243

14.4	Experimental Design	244
14.4.1	Peer Competitors	244
14.4.2	Parameter Settings	245
14.5	Experimental Results	246
14.5.1	Overall Results	246
14.5.2	Efficiency of E2EPP	248
14.5.3	Effectiveness of E2EPP	249
14.5.4	Comparison to Radial Basis Network	251
14.6	Chapter Summary	252
	References	253
15	Deep Neural Architecture Pruning	257
15.1	Introduction	257
15.2	Background	261
15.3	Algorithm Details	262
15.3.1	Genetic Representation of an Individual	263
15.3.2	Population Initialization	266
15.3.3	Individual Evaluation	266
15.3.4	Selection of Knee and Boundary	267
15.3.5	Offspring Generation	269
15.3.6	Fine Tuning	269
15.4	Experimental Design	269
15.4.1	Chosen CNN Architectures for Pruning	269
15.4.2	Algorithm Parameters	270
15.5	Experimental Results and Discussion	272
15.5.1	Experimental Results	272
15.5.2	Result Discussion	274
15.6	Chapter Summary	277
	References	277
16	Deep Neural Architecture Compression	281
16.1	Introduction	281
16.2	Related Work and Motivation	284
16.2.1	Convolutional Neural Network Compression	284
16.2.2	Evolutionary Algorithms and MMD	285
16.3	KGEA for Compressing DNNs	286
16.3.1	Convolutional Filter Pruning	286
16.3.2	Multi-objective Modeling for CNN Compression	287
16.3.3	KGEA	289
16.3.4	Encoding Scheme and Genetic Operators	294
16.3.5	Discussion	295
16.4	Experimental Studies	296
16.4.1	Experimental Settings	296
16.4.2	Experiments on Fully Convolutional LeNet	296
16.4.3	Experiments on VGG-19	302

16.5 Chapter Summary	307
References	308
17 Distribution Training Framework for Architecture Design	311
17.1 Introduction	311
17.2 Distributed Deep Learning	312
17.3 The Distributed Framework	315
17.3.1 Motivation	315
17.3.2 Framework Overview	316
17.3.3 Definition of the Data Packet	317
17.3.4 Server Node	318
17.3.5 Computing Node	319
17.4 Experimental Studies	320
17.4.1 Evolutionary Pelee	321
17.4.2 Speedup Analysis	323
17.4.3 Inconsistent Performance Node Analysis	324
17.4.4 Communication Analysis	325
17.4.5 Efficiency Analysis	326
17.5 Chapter Summary	327
References	327
Book Conclusions	331

Acronyms

AE	Auto-encoder
APoZ	Average percentage of zeros
BN	Batch normalization
CAE	Convolutional auto-encoder
CDAE	Convolutional denoising auto-encoder
CGP	Cartesian genetic programming
CNN	Convolutional neural network
CRBM	Convolutional restricted Boltzmann machine
DAE	Denoising auto-encoder
DAG	Directed acyclic graph
DB	DenseNet block
DBN	Deep belief network
DE	Differential evolution
DM	Decision-maker
DNN	Deep neural network
E2EPP	End-to-end performance predictor
EA	Evolutionary algorithm
EC	Evolutionary computation
ENAS	Evolutionary neural architecture search
FLOP	Floating-point operation
GA	Genetic algorithm
GAN	Generative adversarial network
GP	Genetic programming
GPU	Graphics processing unit
HIS	Hyperspectral image
IP	Internet protocol
KGEA	Knee guided evolutionary algorithm
LR	Low-rank
MCDM	Multi-criteria decision-making
MMD	Minimum Manhattan distance
MOP	Multi-objective optimization problem

MSE	Mean square error
NAS	Neural architecture search
NN	Neural network
PSO	Particle swarm optimization
RB	ResNet block
RBM	Restricted Boltzmann machine
ReLU	Rectifier linear unit function
RL	Reinforcement learning
SVHN	Street view house numbers
SVM	Support vector machine
UA	Unit alignment
VAE	Variational auto-encode

Part I

Fundamentals and Backgrounds

Chapter 1

Evolutionary Computation



EC is a class of nature-inspired algorithms that maintains a population of candidate solutions (individuals) and evolves toward the best answer(s). It has been frequently used to solve difficult real-world optimization problems since it evolves numerous solutions at the same time, which contribute to the notable characteristic of EC as being frequently insensitive to local minimal. EC approaches, in contrast to accurate optimization algorithms, do not require continuous problems or the extraction of gradient information, considerable domain expertise, or even the analytical description of issues (i.e., model). EC approaches have been frequently used in practice due to these problem characteristics.

The most well-known EC techniques are evolutionary algorithms (EAs) and swarm intelligence. In particular, swarm intelligence is driven by flocking and swarming behavior, such as PSO and ant colony optimization, while EAs are mostly inspired by human evolution, such as GAs, GP, DE, and so on. In the following, we will go through GAs, PSO, DE, and GP, which will be used as building blocks for the deep neural architecture search methods that follow.

1.1 Genetic Algorithms (GAs)

GAs are a type of heuristic population-based computational paradigm inspired by natural selection. They are also the most common sort of EA (in addition to GAs, EAs include GP, evolutionary strategy [1], and evolutionary programming). GAs are chosen in engineering domains where optimization problems are frequently non-convex and non-differentiable [2, 3] because of their gradient-free nature and insensitivity to the local minimum. GAs solve optimization issues by using a set of bio-inspired operators, including crossover, mutation, and selection, to mimic biological evolution [4, 5]. In general, a GA works like this:

- Step 1:* Initialization of a population of individuals, each of whom represents a potential solution to the problem using the encoding approach applied;
- Step 2:* Evaluation of each individual's fitness in the population based on encoded data and the fitness function;
- Step 3:* Selecting and mating prospective parent individuals from the present population and then using crossover and mutation operators to generate offspring;
- Step 4:* Evaluation of the fitness of the children produced;
- Step 5:* Environmental selection of a population of individuals with promise performance from the present population and their children, with the current population being replaced by the selected population;
- Step 6:* If the termination criterion is not met, proceed to Step 3; otherwise, return the person with the best fitness as the best solution to the problem.

Commonly, a maximal generation number is predefined as the termination criterion.

1.2 Particle Swarm Optimization (PSO)

PSO, like GAs, is also a population-based stochastic EC (swarm intelligence) technique that is often used for solving optimization problems without requiring domain knowledge. PSO is inspired by the social behavior of fish schooling or bird flocking [6, 7] and is distinguished from other heuristic algorithms by its basic principle including ease of implementation and computational efficiency. Individuals are termed particles in PSO, and each particle keeps the best solution from its own memory, while the population keeps track of the best answer from the history of all particles. Local best particles (commonly denoted by $pBest_i$ for the i th particle) and global best particles (commonly denoted by $gBest$) are frequently mentioned in both categories. In PSO, particles are supposed to collaborate and interact with the best local and global particles during the process, improving their search capacity and pursuing the goal. A typical PSO has the procedure as follows:

- Step 1:* Initialize the particles, as well as a counter $t = 0$ and a maximum iteration number max_t ;
- Step 2:* Evaluate the fitness values of particles;
- Step 3:* Choose the best $pBest_i$ particle from the memory of each particle;
- Step 4:* Choose the best particle $gBest$ from the entire particle history;
- Step 5:* Compute the velocity $\{v_1, \dots, v_i, \dots\}$ of each particle $\{x_1, \dots, x_i, \dots\}$ by Eq. (1.1);
- Step 6:* Update the position $\{p_1, \dots, p_i, \dots\}$ of each particle $\{x_1, \dots, x_i, \dots\}$ by Eq. (1.2);
- Step 7:* Increase t by 1, if $t < max_t$ to repeat Steps 2–6 otherwise go to Step 8;
- Step 8:* The position of $gBest$ is revealed.

$$v_i \leftarrow \overbrace{w \cdot v_i}^{\text{inertia}} + \overbrace{c_1 \cdot r_1 \cdot (p_g - p_i)}^{\text{global search}} + \overbrace{c_2 \cdot r_2 \cdot (p_p - p_i)}^{\text{local search}} \quad (1.1)$$

$$p_i \leftarrow p_i + v_i \quad (1.2)$$

In Eq. (1.1), c_1 and c_2 are acceleration constants, w signifies the inertia weight, r_1 and r_2 are random values between 0 and 1, and p_p and p_g denote the positions of $pBest_i$ and $gBest$, respectively. In addition, the velocity and position of the i -th particle x_i are denoted by v_i and p_i , respectively. In PSO, particles are supposed to find the ideal position by incorporating the terms “inertia”, “global search”, and “local search” into the velocity updating.

There are two subtraction operations in the “global search” and “local search”, as shown in Eq. (1.1). For the better of the understanding, we will give the details using the word $p_g - p_i$ from “global search” as an example to better grasp how such a subtraction operation works. Assume that the minimization problem to be solved is written as $f(z_1, z_2, \dots, z_n)$, with n choice variables $\{z_1, z_2, \dots, z_n\} \in \Psi$. When PSO is employed to solve this problem, the i -th particle x_i is randomly selected from Ψ and its position is defined by a specific value $p_i = \{z_1^i, z_2^i, \dots, z_n^i\} \in \Psi$. The location of x_i is adjusted toward the optimal solution through the interaction formulated by Eqs. (1.1) and (1.2). The optimization is performed after a number of iterations, and the final answer is the position of the global best particle, $gBest$. The length of a particle in this example is n , which is the number of choice variables as well as the dimension of the location. Clearly, $p_g - p_i = \{z_1^g - z_1^i, z_2^g - z_2^i, \dots, z_n^g - z_n^i\}$.

1.3 Differential Evolution (DE)

DE is an EC method that seeks for the best solutions to a problem using a population-based approach. It has been shown to be an effective and easy heuristic method for global optimization in continuous spaces [8]. In a DE algorithm, there are four major steps: initialization, mutation, crossover, and selection [9]. First, a population of potential vectors is generated at random. Second, mutation is applied according to Formula (1.3), where $\mathbf{v}_{i,g}$ denotes the i th temporary candidate vector of the g th generation; $\mathbf{x}_{r0,g}$, $\mathbf{x}_{r1,g}$ and $\mathbf{x}_{r2,g}$ denote three randomly selected candidates in the g th generation; and F is the differential rate is used to control the rate of evolution. Finally, crossover is done using Formula (1.4), where $u_{j,i,g}$ denotes the j th dimension of the i th candidate at the g th generation. For each applicant, at the start of the crossover procedure, a random number j_{rand} is generated for each candidate at the start of the crossover process, and then another random number $rand_j$ is generated for each dimension of each candidate vector, which is reported and compared with the crossover rate Cr and j_{rand} as shown in Formula (1.4) to determine whether the crossover applies on this dimension. Following the DE operators, a trial vector $\mathbf{u}_{i,g}$

is generated, which is then compared to the parent vector to determine which has the best fitness. The best candidate will be reported after iterating the stages of mutation, crossover, and selection until the stopping requirement is fulfilled.

$$\mathbf{v}_{i,g} = \mathbf{x}_{r0,g} + F \times (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \quad (1.3)$$

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } rand_j(0, 1) < Cr \text{ or } j = j_{rand} \\ x_{j,i,g} & \text{otherwise} \end{cases} \quad (1.4)$$

1.4 Genetic Programming (GP)

GP is a type of EA that uses a variable-length tree-based representation to solve problems. Population initialization, fitness evaluation, (parent) selection, and offspring generation are all critical processes in GP, as they are in other EAs. The population is frequently formed by evolution. Fitness functions are used to assess an individual's goodness, and genetic operators are mostly used to generate children.

When GP is used to solve real-world problems, some key parameters must be predefined, including the size of the population, the termination condition, the initialization method for constructing the population, the specific choices for genetic operators, and the fitness function. Also, the nodes in a tree-based GP are often classified into a terminal set and a function set, where the terminal set is often used as leaf nodes and the function set provides the available types of internal nodes in the tree. The terminal and function sets should also be predefined with the parameters aforementioned. The mean squared error (or other types of error/accuracy) measuring the difference between the program outputs, and the target/desired outputs of the training set is typically used as the fitness function. In addition, the termination condition of GP is typically a maximum number of generations or a specific value.

The full method, the growing method, and the ramping half-and-half approach [10] are the most often utilized population initialization methods. All internal nodes are added to each tree until a given depth is achieved in the complete population, and then leaf nodes are added to each leaf node. In the grow method, each node is randomly assigned to be a leaf or an internal node, and if a terminal is picked, the branch will stop growing. The aforementioned two procedures will construct half of the population for the ramping half-and-half method. Two parents must be chosen in order to produce kids. Typically, tournament selection selects each of the parent persons. A fixed number of people are randomly sampled from the population, and the one with the best fitness value is picked to be a parent. In reality, a fixed size, such as two, is commonly provided, which is known as the binary tournament selection.

The two genetic operators are mutation and crossover. The mutation process selects a person based on a likelihood, and then performs a random modification on the individual's position, which is likewise based on a probability. To accomplish the crossover operation, two parent solutions are chosen first, then two points from

the two parents are randomly chosen, and two new offspring are created by linking each component of the parent individuals. Environmental selection is frequently achieved through tournament selection. Elitism is also present in the environmental selection, with individuals with the highest fitness values being chosen.

1.5 Chapter Summary

This chapter provides a brief overview of the common EC algorithms, GAs, PSO, DE, and GP, which will also be used in later chapters. Other EC techniques can be easily checked from literature and will not be reviewed here since they will not be used directly in this book. In the next chapter, we will review deep learning and commonly used DNNs.

References

1. Janis, C. (1976). The evolutionary strategy of the equidae and the origins of rumen and cecal digestion. *Evolution*, 30(4), 757–774.
2. Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2), 182–197.
3. Sun, Y., Yen, G. G., & Yi, Z. (2018b). IGD indicator-based evolutionary algorithm for many-objective optimization problems. *IEEE Transactions on Evolutionary Computation*. <https://doi.org/10.1109/TEVC.2018.2791283>.
4. Mitchell, M. (1996). *An introduction to genetic algorithms*. MIT Press.
5. Schmitt, L. M. (2001). Theory of genetic algorithms. *Theoretical Computer Science*, 259(1-2), 1–61.
6. Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *Conference on neural networks* (Vol. 4). IEEE International.
7. Eberhart, R., & Kennedy, J. (1995). A new optimizer using particle swarm theory. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, 1995. MHS'95* (pp. 39–43). IEEE.
8. Storn, R., & Price, K. (1997) Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces. <https://doi.org/10.1023/A:1008202821328>.
9. Price, K. V., Storn, R. A., & Lampinen, J. A. et al. (2005). *Differential evolution: a practical approach to global optimization*, Chapter 2 (pp. 37–42). Springer.
10. Walker, Matthew. (2001). *Introduction to genetic programming*. Tech. Np: University of Montana.

Chapter 2

Deep Neural Networks



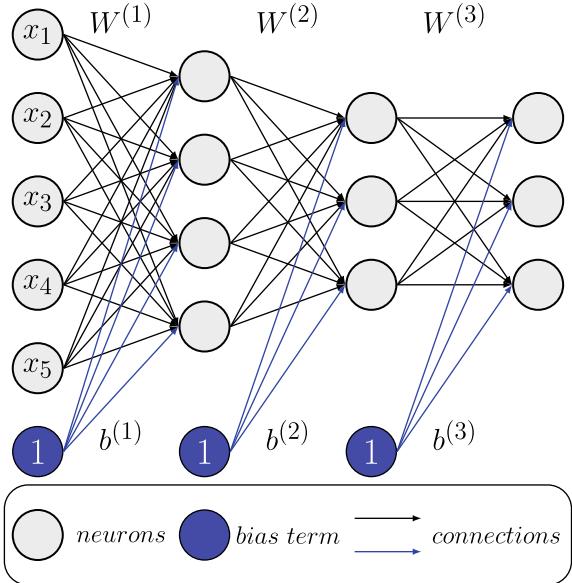
The neural networks (NNs) with deep architectures are referred to as DNNs. In general, there is no universal standard of how deep a CNN must be to be considered deep. In practice, a DNN is defined as a NN with at least four layers.

An artificial NN prompted by the computational functions of human brain neurons is typically referred to as a NN in the domains of computer science, engineering, and other fields related. Artificial NNs can be categorized into two types based on the direction in which information flows during computation: feed-forward NNs and recurrent NNs. For the sake of convenience, we will refer to “artificial NN” as “NN” in this book. Furthermore, unless otherwise stated, the NNs mentioned in this book will also pertain to feed-forward NNs.

An NN is frequently composed of multiple layers, each of which contains multiple neurons, each neuron is linked to the neurons in the next layer. The first layer is often referred to as the input layer, while the last layer is generally referred to as the output layer. In addition, the remaining layers are all referred to as hidden layers. Each neuron typically has numerous inputs and one output. The outputs of some or all of the neurons in the previous layer, as well as a “dummy” input called “bias”, make up the inputs of a neuron. In practice, the value of the bias term is frequently set to one. Figure 2.1 depicts a four-layer NN with five, four, three, and three neurons in the first, second, third, and fourth layers, respectively. A neuron in this example has full connections to all of the neurons in the following layer (excluding the neurons in the last layer), which is known as a “fully-connected” NN. Moreover, $x = [x_1, x_2, \dots, x_5]$ signifies the input data, W and b denote the connections from the neurons and the bias term, respectively, and the superscripts denote the index of the source layer from which the connections originate. $W^{(1)}$, for example, denotes the connections formed by the neurons in the first layer. Please notice that the connections in this figure are colored differently to distinguish between their sources, i.e., whether the connections are from neurons or from the bias terms.

The neuron will carry out two tasks: (1) summing the inputs through the connections (including the bias) while maintaining their individual weights, and (2) changing the weighted result using a activation function that is often non-linear (also

Fig. 2.1 An example of a four-layer feed-forward fully connected NN

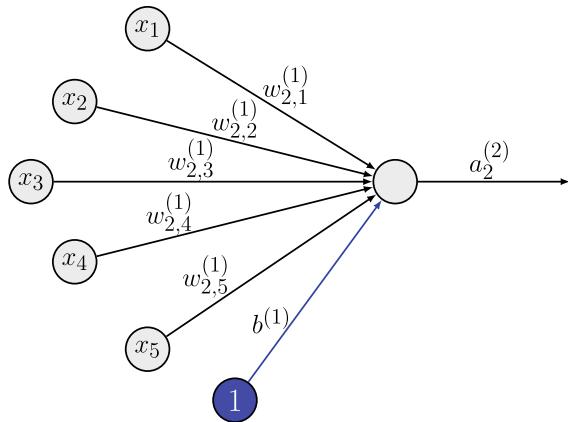


known as a transfer function in certain literature) to get the output. This technique is also known as the “computational model” of a neuron, and it is responsible for the computational function of NNs. For a better understanding, the computational model of the second neuron at the second layer is demonstrated in Fig. 2.2. In the following, we will continue to use W to symbolize the weights, and the notation $W_{i,j}^{(k)}$ to denote the weight of the connection from the j th neuron at the k th layer to the i th neuron at the $(k+1)$ th layer for the sake of clarity. We also use the symbols $z_i^{(k)}$ and $a_i^{(k)}$ to represent the weighted summation results and the output, respectively, and $f(\cdot)$ to represent the activation function. Equation (2.1) is often used to model the procedure of this neuron based on the knowledge introduced above.

$$\begin{cases} z_2^{(2)} = [w_{2,1}^{(1)}, w_{2,2}^{(1)}, w_{2,3}^{(1)}, w_{2,4}^{(1)}, w_{2,5}^{(1)}, b^{(1)}] \cdot [x_1, x_2, x_3, x_4, x_5, 1]^T \\ a_2^{(2)} = f(z_2^{(2)}) \end{cases} \quad (2.1)$$

Please note that all neurons have the same computational models; nevertheless, we chose this neuron as an example because Eq. (2.2) may be precisely specified with the relevant superscripts and subscripts, which may make the computational model easier to grasp. Furthermore, the activation function is element-specific, and the activation functions for each neuron might, in theory, differ. In practice, the hyperbolic tangent function and the sigmoid function, which are mathematically represented as $f(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and $f(x) = 1/(1 + e^{-x})$, respectively, are the common ones that are often employed for the traditional feed-forward fully-connected NNs in practice. The rectifier linear unit function (ReLU), i.e., $f(x) = \max(0, x)$, has been increasingly popular in recent years.

Fig. 2.2 An illustrated example showing the computational model of the neuron of NNs



DNNs have been extensively recognized for their promising performance in addressing difficult real-world issues. This is primarily owing to their non-linear transformation capacity, which can be enhanced even further as depth grows. In particular, most real-world items have non-linear relationships to varying degrees. As a result, dealing with uncertain non-linear situations naturally involves employing NNs with varied non-linear abilities.

However, their tremendous capabilities can only be demonstrated when the NNs reach their optimal state. One example is the optimal learnable parameters of NNs. The weights in terms of the connections, which can acquire the (near to, or at least local) ideal status via the learning algorithms, are the learnable parameters of NNs. Weight optimization is another name for this treatment. Please keep in mind that with some techniques, the weights are often arbitrarily set, and the most prevalent learning process is the gradient descent algorithm using back-propagation [1].

The gradient of the loss function in terms of the weights connecting to the neurons at the last layer is first determined during the back-propagation learning/training process, and then the gradient is back-propagated to the weight whose gradient has to be calculated through the connections. This is accomplished by multiplying the derivations of the activation functions and other terms numerically iteratively. Because sigmoid and hyperbolic tangent functions with derivations less than one are commonly used as activation functions, the gradients of the weights in the previous layers become very small as the depths of the NNs increase, and the weights updating for gradient descending cannot be truly updated (i.e., the value used for the updating is almost close to zeros). This is the source of the why the traditional NNs with deep architecture cannot demonstrate theoretical capacities. Gradient vanishing is another name for this issue.

DNNs have been highly popular in recent years, and they are now commonly utilized to solve difficult machine learning tasks. This is due to unique training techniques that may reasonably optimize the majority of the gradients of the DNN. Three different types of DNNs, which are the main methodologies of deep learning,

have been proposed to go along with the innovative training algorithm. Deep belief networks (DBNs), stacked auto-encoders (AEs), and convolutional NNs (CNNs) the three most common types of DNNs, which will be addressed separately in the next sections.

2.1 Deep Belief Networks

DBNs are made up of many restricted Boltzmann machines (RBMs) that are stacked together. An RBM is a network that consists of two layers: the input layer (also known as the visible layer) and the hidden layer [2]. The details of constructing the corresponding DBN for a set of RBMs are shown below. The RBMs are first trained one at a time, with the hidden layer of one RBM serving as the input layer for the next RBM to be trained. Second, the hidden layers of all RBMs are connected according to their training order, and the input layer of the first RBM is likewise used as the input layer of the DBN. The learnable parameters of the RBM, which were trained in the first stage, will be utilized to initialize the learnable parameters of the DBN that is generated. The DBN is subsequently trained to achieve optimal performance using the initialized learnable parameters. This is the first time a DNN has been efficiently trained in this manner [3].

The example of employing three RBMs to construct the matching DBN is shown in Fig. 2.3. There are three RBMs in this example (upper half), and the resulting DBN (lower half) includes four layers, including the input layer and hidden layer of the first RBM, as well as the hidden layers of the other three RBMs. All neurons

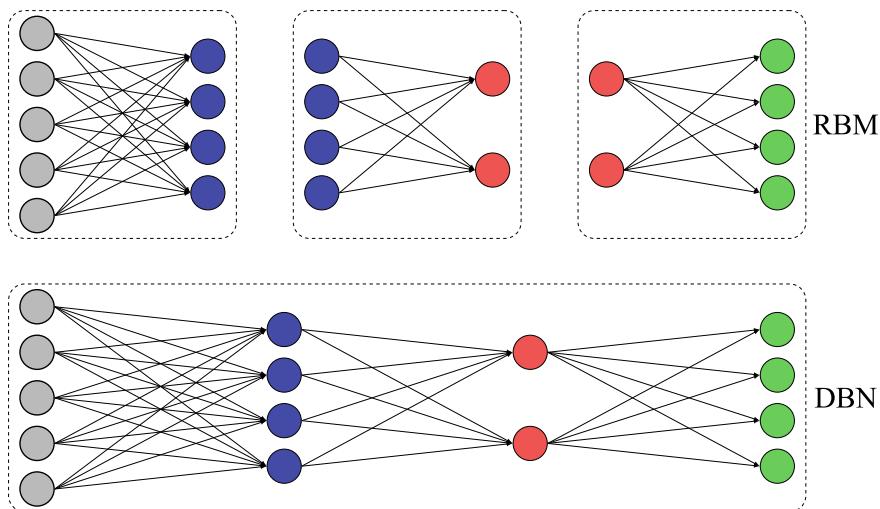


Fig. 2.3 The architectures of three RBMs and the resulted DBN

in the same layer are highlighted with the same color to make it easier to see the relationship between the RBM and the resulting DBN. Furthermore, neurons with the same color at different layers indicate that they are the same neurons (with the same values). Assume that the learning parameters of the three RBMs are collectively initialized as θ , and that their optimal status is indicated by θ^* after training. The learning parameters of the DBN will be set using θ^* . Pre-training is the process of training each RBM, and fine-tuning is the process of training the generated DBN using θ^* .

An RBM can be thought of as a two-layer NN, with connections occurring only between neurons in the input layer and neurons in the hidden layer, i.e. no connections between neurons in the same layers, which is also the main distinction between RBM and the Boltzmann machine. Furthermore, the neurons of an RBM only have two values: zero and one. The weights of the connections and the bias terms of all neurons in the input layer and the hidden layer are the learnable parameters of an RBM. In terms of mathematics, the training of an RBM is just an optimization process for the energy model; interested readers can learn more from [4].

2.2 Stacked Auto-Encoders

To construct the stacked AEs, multiple AEs are stacked. An AE [5] is a NN which is utilized to learn unsupervised efficient features. A standard AE consists of one input layer, one output layer, and one hidden layer. The output of the hidden layer can be used to represent the input data. The AE typically aims to map input data to the representation space and learn a collection of useful features. It is usual to locate or select finer, i.e. more efficient (discriminative), features to reduce dimension and improve discrimination in classification tasks. The output layer of a multi-layer perception with an AE is tuned to recover the input from the representation. Encoding is the process of mapping input to representation, and decoding is the process of reconstructing the representation to the AE output. The input data, the representation, and the reconstruction are designated as $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, and $\hat{x} \in \mathbb{R}^n$, respectively. The process of an AE is represented by Eq. (2.2):

$$\begin{cases} y = \mathcal{F}(x; \theta^e) \\ \hat{x} = \mathcal{G}(y; \theta^d) \end{cases} \quad (2.2)$$

where $\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $\mathcal{G} : \mathbb{R}^m \rightarrow \mathbb{R}^n$ refer to the encoder and the decoder, respectively, θ^e and θ^d means the parameters for the encoder and decoder, respectively, and both are often collectively represented by $\theta = \theta^e \cup \theta^d$ to denote the parameters of the AE. \hat{x} is a reconstruction of the input sample x . The reconstruction error can then be used to gauge the efficiency of the AE, based on which the cost function is shown in Eq. (2.3):

$$\mathcal{J}(\theta; X) = \frac{1}{N} \sum_{x \in X} L(\mathcal{G} \circ \mathcal{F}(x; \theta), x) \quad (2.3)$$

where N represents the number of data points in a collection of samples X , and L stands for the difference between samples where the square Euclidean distance $\|\hat{x} - x\|^2/2$ is commonly used. The optimal θ for the AE can be found by minimizing the cost function.

Equation (2.4) shows the settings for a symmetrical AE having the same numbers of encoders and decoders:

$$\left\{ \begin{array}{l} \mathcal{F} = \underbrace{F_1 \circ F_2 \circ \cdots \circ F_M}_M \\ \mathcal{G} = \underbrace{G_1 \circ G_2 \circ \cdots \circ G_M}_M \end{array} \right. \quad (2.4)$$

where M is the number of encoders/decoders, and $F_i(x, \theta_i^e)$ and $G_i(x, \theta_i^d)$ are neuron layers by which $f(w_i x + b_i), \theta_i = \{w_i, b_i\} \subset \theta$ are the parameters for the layer and f denotes the activation function. In most cases, especially when we pretrain a classifier network, the following AEs with only one layer shown in Eq. (2.5) are used.

$$\left\{ \begin{array}{l} y = f(w_1^e x + b_1^e) \\ \hat{x} = f(w_1^d x + b_1^d) \end{array} \right. \quad (2.5)$$

We can get a result that is comparable to principle component analysis when the activation function is linear and there are fewer hidden units than input units. In this case, the main components of the input space are the learned weights. In general, with a linear activation function and using more units in the hidden layer than in the input layer will lead to a trivial solution in which the weight is an identical matrix. To avoid this, a situation with a sigmoid activation function and more hidden units than input units are collectively considered.

Many beneficial AE versions have been created to prevent learning a trivial solution, over-fitting, and other unfavorable consequences. In the following, some representatives are introduced.

2.2.1 Sparse Auto-Encoders

If we utilize the cost function shown in Eq. (2.3) to train the network, we may be able to just learn an identity function. When the dimension of the encoding is higher than or equal to the number of input layers, a simple solution can be reached by simply copying the input.

With a nonlinear activation function, the representations will no longer be controlled solely by the input data, as there will be more units in the hidden layer than

in the input layer. By adding sparsity requirements to the cost function, one with the most zero components is particularly interesting. This is because the sparsity restrictions produce characteristics that are more invariant to geometric alterations of images (for some). This motivates the invention of the sparse AE which has the cost function expressed by Eq. (2.6):

$$\mathcal{J}_{AE+Sparse}(\theta; X) = \mathcal{J}(\theta; X) + \lambda S(y) \quad (2.6)$$

where $S(y)$ is the sparsity of y and λ is a penalty-balancing hyper-parameter.

2.2.2 Weight Decay Auto-Encoders

It is demonstrated that if we wish to improve the generalization ability of a NN, we need consider the balance between the information in the training samples and the complexity of the network. When the complexity of the network is valued higher than the information in the training set, the network will over-fit the data, whereas under-fitting occurs in the opposite circumstance. The number of parameters in a network often has anything to do with its complexity. As a result, limiting the number of factors as much as feasible can improve network formation. In practice, achieving this balance is usually tough.

Limiting the increase of the weights through some form of weight decay is another technique to confine a network and reduce its complexity. The cost function of AEs with weight decay is usually with the form represented by Eq. (2.7):

$$\mathcal{J}_{AE+WD}(\theta; X) = \mathcal{J}(\theta; X) + \lambda \sum_{w \in \theta} \frac{1}{2} \|w\|_F^2 \quad (2.7)$$

where $\|\cdot\|_F$ is the Frobenius norm, and λ is a hyper-parameter that governs how strongly the weights having large values are penalized. It is well recognized that this type of weight loss can help with generalization [6].

There is another way to look at weight loss. When we minimize the cost function in basic AEs with constrained sparsity, $S(y)$ becomes minimal. As a result, w^d grows very vast and w^e grows very little, rendering the dictionary of the representation insufficient. As a result of the poor representation, the solution is trivial. Weight loss is required from this standpoint.

2.2.3 Denoising Auto-Encoders (DAEs)

A DAE converts a corrupted example into an uncorrupted one [7]. It reduces the difference between the reconstruction of the original examples and the corrupted

examples. As a result, the network is able to learn a reliable representation. DAEs alter the reconstruction criterion, allowing for the avoidance of learning a trivial representation of an identity function. Equation (2.8) is the form of its cost function:

$$\mathcal{J}_{DAE}(\theta; X) = \frac{1}{N} \sum_{x \in X} L(\mathcal{G} \circ \mathcal{F}(x'; \theta), x) \quad (2.8)$$

where the corrupted x is denoted by x' . In this case, it is assumed that the high-dimensional input data is stored in a low-dimensional manifold. DAEs learn a stochastic operator $p(x|X)$ that maps the corrupted X back to its uncorrupted version during training. Corrupted examples are more likely than uncorrupted examples to be outside and distant from the manifold. The mapping learns from examples that include both corrupted and uncorrupted data to the manifold created by corrupted instances, so expanding the learning territory and improving learning ability.

DAEs attempt to predict corrupted data from uncorrupted values in terms of likelihood. To predict any subset of variables from the remaining variables, it is sufficient that randomly selected subsets of missing patterns capture the joint distribution of a set of variables.

2.2.4 Contractive Auto-Encoders

A contractive AE [8] employs a training criterion that promotes the learned representation y to be as invariant as feasible to the input data x while also reconstructing the input data. The criterion is used to reduce the summation of a reconstruction error and the Frobenius norm of the corresponding Jacobian matrix, which is the derivation of each hidden unit output with respect to each input. Contractive AEs have cost function expressed by Eq. (2.9):

$$\mathcal{J}_{CAE}(\theta; X) = \mathcal{J}(\theta; X) + \lambda \|\partial a_e / \partial X\|_F^2 \quad (2.9)$$

where the second term will be very small when we minimize \mathcal{J}_{CAE} , therefore w^d will tend to be infinity and w^e will tend to be zero to meet the first term. To avoid a trivial solution, we usually use a tied weight, in which w^d is forced to be the transpose of w^e . b^d and b^e , on the other hand, are both initialized and trained.

By penalizing large-value components in the matrix, the representation will retain consistency for tiny permutations of input data. As a result, the learned representation will be more resilient and generalizable.

2.2.5 Convolutional Auto-Encoders (CAEs)

When image data is given to the above AEs, it must first be converted into vector form, which alters their basic structures and reduces subsequent performance. For example, consider an image with the format $I \in \mathbb{R}^{n \times n}$, in which the pixel $I_{j,k}$ ($1 < j < n, 0 < k < n$) is close to the pixel $I_{j-1,k}$. The connection between $I_{j,k}$ and $I_{j-1,k}$ will be modified when I is vectorized to $V \in \mathbb{R}^{n^2}$, and they may no longer be neighbors in V . The importance of neighboring information in addressing image-related issues has been demonstrated in numerous studies [9–13]. Masci et al. [14] proposed the CAEs to overcome this problem, in which image data is directly provided in 2-dimensional form. The encoder in CAEs consists of one convolutional layer followed by one pooling layer, with only one deconvolutional layer in the decoder. For learning the hierarchical representations that improve the final classification performance, many trained CAEs are stacked to a CNN. CAE variants have been proposed in response to the advantages of CAEs in addressing data in the original 2-dimensional form. For instance, Norouzi et al. [15] proposed the convolutional RBMs [3, 16] (CRBM). By stacking a group of trained CRBMs, Lee [17] proposed the convolutional DBN. In addition, Zeiler et al. [18, 19] introduced inverse convolutional ones based on the sparse coding schema [20], which prompted Kavukcuoglu et al. [21] to develop convolutional stacked sparse coding for handling object identification problems. Du et al. [22] recently presented the convolutional DAE (CDAE), which learns convolutional filters using DAE [7].

In practice, CAEs have been more frequently applied to 3-dimensional data. In a general case, assume that CAEs are used for image classification tasks and that each image $X \in \mathbb{R}^{w \times h \times c}$, where w, h, c refer to the image width, height, and number of channels, respectively. The architecture of one CAE [14] is shown in Fig. 2.4.

The deconvolutional operation is the same as the convolutional operation with the reversed parameter settings. The deconvolutional operation, in particular, conducts the convolutional operation on the feature map that is the outcome of the corresponding convolutional operation using the filter and stride that were used in the

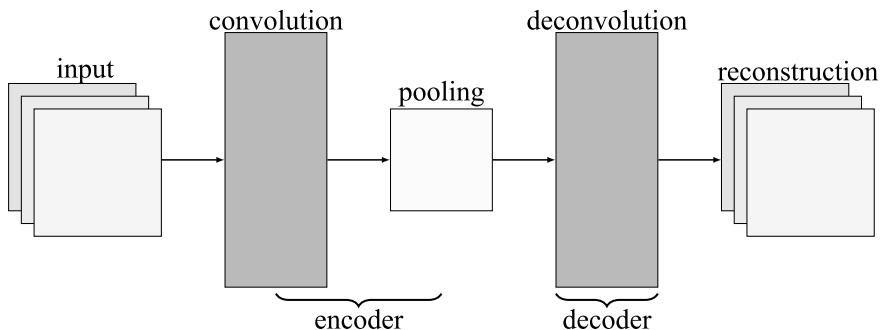


Fig. 2.4 The illustrative architecture of a CAE

corresponding convolutional operation. Extra zeros may be padded into the input of deconvolutional operation to ensure that the output of the deconvolutional operation has the same size as the input of the corresponding convolutional operation.

Equation (2.10) represents the mathematical form of the CAE, where $\text{conv}(\cdot)$, $\text{pool}(\cdot)$, and $\text{de_conv}(\cdot)$ denote the convolutional, pooling, and deconvolutional operations, respectively, $F(\cdot)$ and $G(\cdot)$ denote the element-wise nonlinear activation functions, b_1 and b_2 denote the corresponding bias terms, r and \hat{X} denote the learned features and reconstruction of X , $l(\cdot)$ measures the differences between X and \hat{X} , and Ω denotes a regularization term. The CAE is trained by minimizing L , and then parameters for convolution, bias terms, and deconvolutional operations are found. Encoders of multiple trained CAEs are combined to construct a CNN for learning hierarchical features that improve final classification performance [23].

$$\begin{cases} r = \text{pool}(F(\text{conv}(X) + b_1)) \\ \hat{X} = G(\text{de_conv}(r) + b_2) \\ \text{minimize } L = l(X, \hat{X}) + \Omega \end{cases} \quad (2.10)$$

2.2.6 Variational Auto-Encoders (VAEs)

In contrast to traditional AEs, the VAE developed by Kingma et al. [24] is a powerful generative AE that can provide high-quality representations for a large number of unlabeled samples and generate virtual instances in the controllable smooth latent space [25].

The VAEs have a high capability of approximating the intractable posterior density in a more generic manner and can effectively do inference. These advantages combine to make the VAEs popular and effective for a variety of machine learning tasks, including natural image processing [26], text caption [27], and audio data recognition [28]. Furthermore, the latent representations can be controlled more accurately thanks to the variational lower bound in the optimization of VAEs, and precise modelling may yield better representations than previous AE variations. Certain downstream tasks, like classifying images [29], could benefit from this. Furthermore, because VAEs learn the characteristics of a given probabilistic distribution, the production of new instances may be readily controlled by changing the distribution parameters. For example, Kulkarni et al. [30] used a convolutional VAE to train an interpretable representation of images, and the model can generate new instances of the same item with varied poses and illumination, implying that we may use a VAE to render a static image with different posture orientations. Furthermore, VAEs are more stable throughout the training phases than other generative paradigms, such as generative adversarial networks (GANs) [31], and have the capacity to create crisp examples comparable to GANs [32]. Another significant advantage is that, when compared to variational inference, VAEs are typically more efficient for large datasets [25], because VAEs are still relatively fast when using a single feed-forward

NN to be a stochastic function of the input variables, whereas variational inference typically encounters a large computational complexity as the number of samples increases.

VAE is a form of generative model, as previously stated. The generative model is designed to estimate the distributions $P(X)$ for a given dataset $X = \{x_i\}_{i=1}^N$ consisting of N i.i.d. samples of continuous variable x , where the real samples should have a high probability and the random noise should get a low probability [33]. We suppose that the production of real data follows a random two-step procedure in a more generic case: (1) Some prior distributions $p(z; \theta)$ are used to produce a continuous latent variable z . (2) The conditional distribution $p(x|z; \theta)$ is responsible for the observed data x . In the general situation, the true parameters θ and the latent variable z are invisible, and the posterior $p(z|x)$ is intractable. The VAEs can conduct efficient posterior inference on data with intractable posteriors by maximizing a variational lower limit $\mathcal{L}(\theta, \phi; x_i)$, which can be calculated using Eq. (2.11):

$$\begin{aligned}\mathcal{L}(\theta, \phi; x_i) = & -\text{KL}(q(z|x_i; \phi) \| p(z; \theta)) \\ & + \mathbb{E}_{q(z|x_i; \phi)}[\log p(x_i|z; \theta)]\end{aligned}\quad (2.11)$$

where the learned weights of the VAE are denoted by ϕ and θ , respectively.

We assume that the ground-truth posterior $p(z|x_i; \theta)$ is an approximation Gaussian with approximately diagonal covariance, and that the prior distribution $p(z; \theta)$ is a multivariate Gaussian distribution [24]. In this situation, $\mathcal{L}(\theta, \phi; x_i)$ can be expressed in the form provided by Eq. (2.12):

$$\begin{aligned}\mathcal{L}(\theta, \phi; x_i) \simeq & \frac{1}{2} \sum_{j=1}^J (1 + \log \sigma_{i,j}^2 - \mu_{i,j}^2 - \sigma_{i,j}^2) \\ & + \frac{1}{L} \sum_{l=1}^L \log p(x_i|z_l; \theta)\end{aligned}\quad (2.12)$$

where μ_i and σ_i^2 refer to the mean value and variance value of the distribution, respectively, which are the outputs of the encoder of the VAEs. Furthermore, the symbol J signifies the dimension of the underlying manifold, while the symbol L denotes the sample size of Monte Carlo technique sampling from estimated posterior distributions.

2.3 Convolutional Neural Networks (CNNs)

These days, CNNs are one of the most used types of DNNs, especially for classification tasks. CNNs work by extracting features from raw data without requiring considerable domain knowledge, and then using those features to perform the tasks. The number of independent parameters (weights and biases) of CNN is signifi-

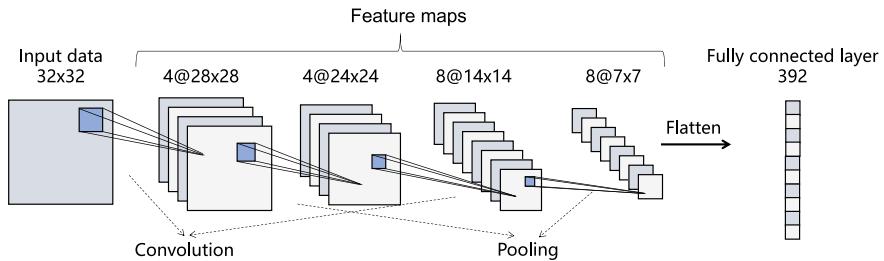


Fig. 2.5 An overview architecture of CNN

cantly smaller than the number of connections due to weight sharing—many different groups of connections share the same set of weights, which is a critical attribute of CNNs for classification tasks. Image classification, speech recognition, and natural language processing have all benefited from the usage of CNNs.

2.3.1 CNN Skeleton

Figure 2.5 presents a schematic CNN architecture, which includes two operations for convolution, two operations for pooling, four groups of feature maps resulted, and a fully connected layer at the tail. By flattening all elements of the fourth group of feature maps, the final layer, which is a completely connected layer, receives the flattened data as input. At the top of the architecture, the convolutional and pooling layers can be combined, while the fully connected layers are constantly placed on top of each other at the bottom. The sizes of the respective layers are also shown in Fig. 2.5. In particular, the input data and the output data are with the sizes of 32×32 and 392×1 , respectively, while the other numbers denote the feature map configurations. For example, there are 8 feature maps with the same size of 14×14 as implied with $8@14 \times 14$.

The convolution and pooling procedures, which associated with the convolutional layer and the pooling layer, respectively, are documented in the following sections, although the fully connected layer is not covered because it is considered common information.

2.3.2 Convolution

For a convolutional feature map to be constructed from an input image of size $n \times n$, a filter must be pre-defined for that image. When it comes to a filter (also known as a matrix), the size of the filter (i.e., the filter width and the filter height) is set at random before the filtering process begins. With a step size equal to a stride width,

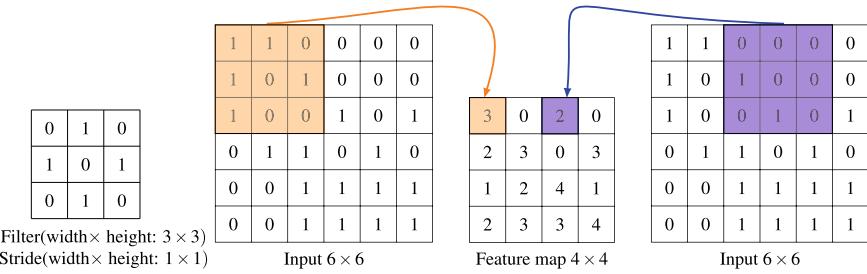


Fig. 2.6 An illustration example of convolutional operation

this filter travels from left to right of the input data, then travels down with a step size equal to a stride height to reach the rightmost bottom of the input image until it reaches the leftmost bottom. There are two types of convolutional operations: VALID (without padding) and SAME (with padding), based on whether or not padding zeros are used to maintain the same size between the feature map and the input data. Each component in the feature map is equal to the sum of the products of each filter and the associated components this filter overlaps. It will need three unique filters if the data has more than three channels, each of which concatenates on its own channel, and the outputs are calculated by element-wise summation.

An illustration of the VALID convolutional operation is presented in Fig. 2.6, where the filter is 3 in both width and height, the input data is 6×6 in size, and the stride is 1 in height and width. The generated feature map is 4×4 in size, the colored shadow areas in the input data indicate where the filter overlaps with the input data at different points in the data, the connection weight values are represented by the numbers in the filter, with each shade representing a convolutional output, as displayed in Fig. 2.6. Nonlinearity, such as ReLU [34], is used to update the convolutional output for each filter before they are saved into the feature map. The width and height of filter, the number of feature maps, the width and height of stride, the type of convolutional operation, and the connection weights in the filter are obviously involved parameters in one convolutional operation.

2.3.3 Pooling

Although there are some differences in terms of element-wised output and related feature map values, it would appear that pooling and convolutional processing are extremely similar in many other respects. The pooling operation, much like the convolutional operation, makes use of a fixed window with the name kernel to calculate the mean or maximum value of the elements over which it slides. The size of the slide is referred to as the “stride”. Figure 2.7 illustrates an example of the pooling operation. In this illustration, the kernel value is 3×3 , the stride width and height are both 3, the input data is 6×6 , and the shadows with varying hues relate to the

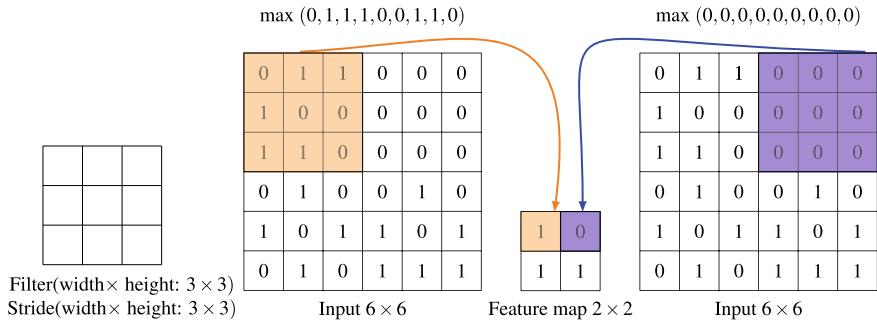


Fig. 2.7 An illustration example of pooling operation

Fig. 2.8 An illustration example of reflect padding

0.7	0.8	0.7	0.6	0.7
0.2	0.1	0.2	0.3	0.2
0.7	0.8	0.7	0.6	0.7
0.2	0.3	0.2	0.3	0.2
0.7	0.8	0.7	0.6	0.7

two slide positions and the subsequent pooling values. In this particular instance, the MAX pooling operation is carried out. The pooling technique takes into account a number of elements that are plainly relevant, including the width and the height of kernel, the width and the height of stride, and the type of pooling. In addition, the AVERAGE pooling is another popular one which uses the average value in analogy to the max value in the MAX pooling operation.

2.3.4 Reflect Padding

The reflect padding is a typical method for image padding that returns the input image to the state it was in when it was first created. It does this by reflecting the data that is being fed into the system; for instance, if we want to add a new row to the top of the input image using reflect padding, the row is obtained via vertical and diagonal reflection from the second row in the input image. A schematic representation of how reflect padding operates is shown in Fig. 2.8. The first and last components in the first green row, for example, both have a value of 0.7 as a result of diagonal reflection. Through the process of vertical reflection, we can see that the middle parts of the blue area have the same value as those in the second row.

2.3.5 Batch Normalization (BN)

A CNN has the potential to function effectively [35] if the BN layer can be used concurrently. In particular, the BN layer makes it possible for a higher learning rate, significantly increases the training speed, and eliminates the risk of gradient vanishing or divergence. The BN layer of the CNN immediately goes to work whenever a new batch of data is added to it. It does this by calculating the mean and the standard deviation of the new data [36].

2.3.6 ResNet Blocks (RBs) and DenseNet Blocks (DBs)

ResNet [37] and DenseNet [38] are the names of two potential state-of-the-art CNNs that have been proposed in recent years. In general, the popularity of ResNet and DenseNet may be attributed, in large part, to their respective building blocks, which are denoted by the acronyms RBs and DBs.

Figure 2.9 illustrates an example of an RB, which consists of three convolutional layers and one skip connection. It is important to note that this book only discusses the building blocks that are utilized in the construction of deeper networks. There is in fact an additional kind of block in ResNet, and it is the kind that is typically utilized for networks with fewer than 34 layers. In this particular illustration, the convolutional layers are denoted by the letters *conv#1*, *conv#2*, and *conv#3*. A smaller number of 1×1 filters is applied to *conv#1* in order to minimize the spatial dimension of the input. This is done so that the computational cost of *conv#2* can be brought down. When attempting to learn features on *conv#2* that have the same spatial dimension, larger filters such as 3×3 are applied. On *conv#3*, the application of filters with a size of 1×1 is repeated, but this time the spatial dimension is increased to produce more features. As the final outcome of the RB, which is denoted by \oplus , the input is applied to the output of *conv#3*, which is also the case. Noting that in the circumstance that the spatial sizes of the input and the output of *conv#3* are not the identical, a series of convolutional operations using filters with a size of 1×1 are applied to the input in order to accomplish the addition and achieve the same spatial size that *conv#3* has.

Figure 2.10 depicts a DB in its entirety, where only four convolutional layers are kept for the sake of simplicity and convenience of discussion. In point of fact, a DB

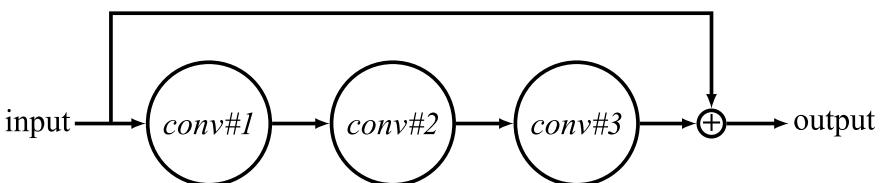


Fig. 2.9 An illustration example of RB including three convolutional layers

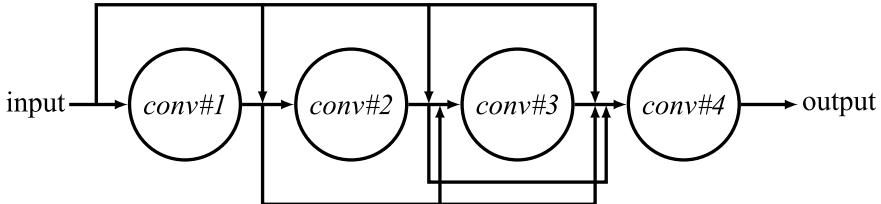


Fig. 2.10 An illustration example of DB including four convolutional layers

may consist of arbitrary number of convolutional layers, the parameters of which may be modified by users. Every convolutional layer in a DB accepts inputs not only from the layer below it but also from all of the convolutional layers that came before it. A k option is also available for altering the spatial size of the input and output of the particular convolutional layer. This option may be found in the same place as the previous one. In the case that the a is the spatial size of the input, then $a + k$ will be the spatial size of the output. This can be accomplished by performing a convolutional operation using the suitable number of filters.

The mechanism that lies behind the efficacy of RBs and DBs has been explored [39, 40], and it has been observed that RBs and DBs can lessen the detrimental effects that are brought on by the gradient vanishing problem [41]. A deep architecture is able to effectively learn the hierarchical representations of the input data with the help of RBs or DBs, which in turn helps to improve the classification accuracy of the final output. In addition, it has been hypothesized that dense connections in DBs can recycle low-level features in order to increase the discrimination of features that are learned in the highest layers of CNNs [38]. Because of the many excellent qualities that RBs and DBs possess, it was decided to employ them as the fundamental building blocks in the corresponding algorithms presented in this book.

2.4 Benchmarks for Deep Neural Networks

There have been multiple popular benchmarks for the research of DNNs, which will be briefly introduced.

MNIST and Fashion During the early stages of deep learning research, the handwritten digits benchmark test dataset was quite popular. In general, there are two types of this benchmark: conventional MNIST [11] and the variants based on MNIST [42]. Specifically, the variants include MNIST-basic (a subset of the conventional MNIST), MNIST-back-rand (MNIST with random noise background), MNIST-rot (a rotated version of MNIST), MNIST-back-image (MNIST with random image background), and MNIST-rot-back-image (MNIST-rot with random image background). For reference, Fig. 2.11 shows examples from the MNIST benchmark. The MNIST and their variants all have the same dimensions of 28×28 .

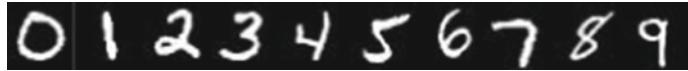


Fig. 2.11 A collection of digit samples ranging from 0 to 9 taken from MNIST benchmark dataset



Fig. 2.12 Examples from the STL-10 dataset

The Fashion benchmark dataset is similar to MNIST regarding the size of each image and the sizes of training data and test data, in addition that the Fashion dataset is to recognize 10 fashion objects (e.g., trousers, coats, and so on), in contrast to the numbers in the MNIST dataset.

Geometries and Rectangles Geometries and rectangles are included in this category of benchmark datasets, including the tall and wide rectangles dataset (Rectangles) [42], rectangles dataset with random image background (Rectangles-image), and convex sets recognition dataset (convex). The images in this category are also 28×28 in size.

STL-10 The STL-10 dataset [43], which contains 100,000 in unlabeled images, 5,000 in training images, and 8,000 in test images from 10-category natural image object identification, is a commonly used dataset for unsupervised learning. The objects in STL-10 are bird, airplane, cat, car, dog, deer, horse, ship, monkey, truck. Each image is a 96×96 3-channel RGB image. Furthermore, the unlabeled images include those that fall outside of the 10 categories. Examples from this benchmark dataset are shown in Fig. 2.12.

Caltech-101 Caltech-101 [44] is a 101 image classification dataset with images ranging in weight and height from 80 to 708 pixels. The majority of the images are 3-channel RGB with some grey, with distinct numbers of images in each category ranging from 31 to 800. Furthermore, most images only show a tiny portion of the image, while other regions are filled with noise, making classification more difficult. The feature learning methods are additionally challenged two facts of this dataset: short number of images and non-identical number of images in each category. Examples from this benchmark dataset are shown in Fig. 2.13.

Street View House Numbers (SVHN) The SVHN dataset [45] is a real picture collection for home numbers that was obtained from Google Street View images. On the official website, the datasets can be accessed in one of two different forms.



Fig. 2.13 Examples from the Caltech-101 dataset



(a) CIFAR-10



(b) CIFAR-100

Fig. 2.14 a, b illustrate instances from each of the three CIFAR-10 and CIFAR-100 categories, each with 10 examples

For the sake of convenience, throughout this book, we will be utilizing the format-2 dataset, which has 73,257 images for training and 26,032 images for testing. Every image in the collection has had its resolution increased to 32 by 32 pixels regardless of its original size. The format-2 dataset has categories that range from 0 to 9, just like MNIST does. The SVHN dataset, on the other hand, is significantly more challenging due to the fact that it requires the classifier to differentiate digits from natural scene pictures.

CIFAR-10 and CIFAR-100 CIFAR-10 and CIFAR-100 are two extensively used image classification benchmark datasets [46] for natural objects, such as birds, boats, and planes. Each set contains 50,000 images for training and 10,000 images for testing. CIFAR-10 differs from CIFAR-100 in the number of objects to be classified, where CIFAR-10 contains 10 different objects while CIFAR-100 contains 100 different objects. CIFAR-10 and CIFAR-100 both include nearly same number of training images in their benchmarks, but the training images for each class are roughly the same in the respective benchmarks. In addition, CIFAR-10 is also known as CIFAR-10-BW, which refers to the grey variant and was popular for unsupervised deep learning models in the early time.

As can be seen in Fig. 2.14, the images in each row represent benchmarks from the same category, and the text in the left line refers to the class name. In Fig. 2.14 the object to be classified in each image has a different resolution to each other, blends with the backdrop, and occupies a distinct position, which generally increases the challenge in accurately recognizing the objects.

ImageNet ImageNet [47] is an image dataset structured based on the WordNet hierarchy [48]. A “synonym set” or “synset” is the term used to represent any relevant concept in WordNet that may be described by several words or word phrases. WordNet has over 100,000 synsets, with nouns accounting for the majority over 80,000. The goal of ImageNet is to provide on average 1,000 images to illustrate each synset. Images of each concept are quality-controlled and human-annotated. In its completion, ImageNet will offer tens of millions of cleanly labeled and sorted images for most of the concepts in the WordNet hierarchy.

The ImageNet dataset is the backbone of ILSVRC (ImageNet Large Scale Visual Recognition Challenge). ILSVRC uses a subset of ImageNet images for training the algorithms and some of ImageNet’s image collection protocols for annotating additional images for testing the algorithms. In the literature, the ImageNet dataset commonly used also refers to the “ImageNet 1K” or “ISLVRC2012”. Specifically, there are 1,000 different categories in this dataset, with approximately a total of 1,200,000 images in the training set and 50,000 in the validation set. The test set is not available publicly, and the results reported in the literature are often based on the validation set. In addition, the sizes of images are different in ImageNet, and on an average size of about 400×350 . ImageNet is now considered the most challenging image dataset.

2.5 Chapter Summary

In this chapter, we provided a brief overview of the deep learning particularly DNNs. In particular, we briefly described commonly used DNNs, including both unsupervised and supervised. At the end, we reviewed commonly used benchmark image datasets that were used to verify the performance of the DNNs, algorithms and systems. Together with Chaps. 1 and 2, the fundamentals and backgrounds of the book, Part I, have been provided.

The next part of the book, Part II, will provide a number of the state-of-the-art methods for designing and developing different types of unsupervised DNNs. Readers will be able to learn different aspects with special considerations of evolutionary design of such methods for automatically developing unsupervised DNNs for image classification.

References

- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 541–551.
- Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the trade* (pp. 599–619). Springer.
- Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786), 504–507.
- Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), 1527–1554.
- Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1), 1–127.
- Hinton, G. E. (1987). *Learning translation invariant recognition in a massively parallel networks*. Springer.
- Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11, 3371–3408.
- Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning* (pp. 833–840).
- Pluim, J. P. W., Antoine Maintz, J. B., & Viergever, M. A. (2003). Mutual-information-based registration of medical images: a survey. *IEEE Transactions on Medical Imaging*, 22(8), 986–1004.
- Pass, G., Zabih, R., & Miller, J. (1997). Comparing images using color coherence vectors. In *Proceedings of the fourth ACM international conference on Multimedia* (pp. 65–73). ACM.
- LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324. ISSN 0018-9219. 10.1109/5.726791.
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), 84–90. <https://doi.org/10.1145/3065386>
- Peng, X., Zhao, B., Yan, R., Tang, H., & Yi, Z. (2017). Bag of events: An efficient probability-based feature extraction method for aer image sensors. *IEEE Transactions on Neural Networks and Learning Systems*, 28(4), 791–803.
- Masci, J., Meier, U., Cireşan, D., & Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning-ICANN, 2011*, 52–59.
- Norouzi, M., Ranjbar, M., & Mori, G. (2009). Stacks of convolutional restricted boltzmann machines for shift-invariant feature learning. In *IEEE conference on computer vision and pattern recognition, 2009. CVPR 2009.* (pp. 2735–2742). IEEE.
- Smolensky, P. (1986). *Information processing in dynamical systems: Foundations of harmony theory*. DTIC Document: Technical report.
- Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 609–616). ACM.
- Zeiler, M. D., Krishnan, D., Taylor, G. W., & Fergus, R. (2010). Deconvolutional networks. In *2010 IEEE conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 2528–2535). IEEE.
- Zeiler, M. D., Taylor, G. W., & Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. In *2011 IEEE International Conference on Computer Vision (ICCV)* (pp. 2018–2025). IEEE.
- Olshausen, B. A., & Field, D. J. (1997). Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision Research*, 37(23), 3311–3325.

21. Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., & Cun, Y. L. (2010). Learning convolutional feature hierarchies for visual recognition. In *Advances in neural information processing systems* (pp. 1090–1098).
22. Bo, D., Xiong, W., Jia, W., Zhang, L., Zhang, L., & Tao, D. (2017). Stacked convolutional denoising auto-encoders for feature representation. *IEEE Transactions on Cybernetics*, 47(4), 1017–1027.
23. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.
24. Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. In *International conference on learning representations*.
25. Kingma, D. P., Welling, M., et al. (2019). An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4), 307–392.
26. Chen, X., Song, J., Hilliges, O. (2019a). Unpaired pose guided human image generation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) workshops*.
27. Pu, Y., Gan, Z., Henao, R., Yuan, X., Li, C., Stevens, A., Carin, L. (2016). Variational autoencoder for deep learning of images, labels and captions. In *Advances in neural information processing systems* (pp. 2352–2360).
28. Blaauw, M., & Bonada, J. (2016). Modeling and transforming speech using variational autoencoders. In *Interspeech* (pp. 1770–1774).
29. Kingma, D. P., Mohamed, S., Rezende, D. J., & Welling, M. (2014). Semi-supervised learning with deep generative models. In *Advances in neural information processing systems* (pp. 3581–3589).
30. Kulkarni, T. D., Whitney, W. F., Kohli, P., & Tenenbaum, J. (2015). Deep convolutional inverse graphics network. In *Advances in neural information processing systems* (pp. 2539–2547).
31. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).
32. Dai, B., & Wipf, D. (2018). Diagnosing and enhancing vae models. In *International conference on learning representations*.
33. Doersch, C. (2016). *Tutorial on variational autoencoders*. [ArXiv:abs/1606.05908](https://arxiv.org/abs/1606.05908).
34. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 315–323).
35. Chang, Y., Yan, L., Fang, H., Zhong, S., & Liao, W. (2018). Hsi-denet: Hyperspectral image restoration via convolutional neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 57(2), 667–682.
36. Ioffe, S., Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research* (pp. 448–456). PMLR.
37. He, K., Zhang, X., Ren, S., Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).
38. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).
39. Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015a). Training very deep networks. In *Advances in neural information processing systems 28: 29th annual conference on neural information processing systems 2015* (pp. 2377–2385).
40. Orhan, E., & Pitkow, X. (2018). Skip connections eliminate singularities. In *International conference on learning representations*. <https://openreview.net/forum?id=HkwBEMWCZ>.
41. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>
42. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning* (pp. 473–480). ACM. <https://doi.org/10.1145/1273496.1273556>.

43. Coates, A., Ng, A., & Lee, H. (2011). An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 215–223).
44. Fei-Fei, L., Fergus, R., & Perona, P. (2007). Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding*, 106(1), 59–70.
45. Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., Ng, A. Y. (2011). *Reading digits in natural images with unsupervised feature learning*
46. Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical Report, University of Toronto. <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
47. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 248–255). IEEE.
48. Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11), 39–41.

Part II

Evolutionary Deep Neural Architecture Search for Unsupervised DNNs

In recent years, tremendous progress has been gained in computer vision, partly because of the excellent potential of DNNs [1]. The DNNs are intended to acquire meaningful representations with high-level features through the nonlinear combination of lower-level features. Such deep architectures have been the models for image processing tasks that are considered to be state of the art because of their ability to greatly outperform competitors that utilize shallow architectures and other traditional methods. The DNNs are frequently utilized as supervised learning models and place a heavy emphasis on the utilization of a substantial number of labeled samples throughout the training process. In contrast to the unlabeled examples, which may be found more frequently in practice and also are easy to acquire, it should come as no surprise that providing a rich supervision indication for DNNs can be challenging to find in a number of different contexts. This encourages the use of unsupervised learning with DNNs, which can take advantage of a huge number of unlabeled data to produce better representations for the network after it has been fully trained.

Researchers have found that unsupervised pre-training does help supervised training of the DNNs [2]. DBNs and stacked AEs are the basic unsupervised DNN techniques for learning meaningful representation, which is the result of the raw input data being processed by several nonlinear transformations in the DNNs. In practice, meaningful representations have the potential to significantly improve the performance of the machine learning tasks that come after it.

In general, arbitrary DNNs have the ability to construct or learn representations. Unfortunately, representations do not always have any useful content; more specifically, it is not accurate to say that all representations resulted in excellent potential when those representations replace the raw data that is supplied to machine learning algorithms. This is because the representations learned are not always meaningful. In point of fact, the representations are the results that have undergone multiple iterations of nonlinear adjustments from the input data [3]. These transformations are motivated mostly by the mammalian hierarchical visual system [4–6]. Equation (1) is the mathematical expression that summarizes the learned representations of the input data $X \in \mathbb{R}^m$:

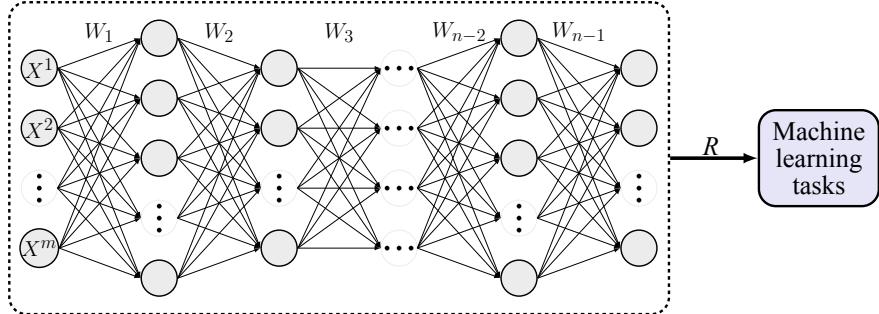


Fig. II.1 An illustration of a typical flowchart of representation learning and how it relates to machine learning tasks

$$\left\{ \begin{array}{l} R_1 = f_1(W_1 X) \\ R_2 = f_2(W_2 R_1) \\ \dots \\ R_n = f_n(W_n R_{n-1}) \\ R = R_n \end{array} \right. \quad (1)$$

where f_1, \dots, f_n represents collection of element-wise nonlinear activation functions, W_1, \dots, W_n stand for a set of connection weights, and R_1, \dots, R_n stand for the learned representations (output) at the depth or layer $1, 2, \dots, n$ of the network, among which $R = \{R_i | 2 \leq i \leq n\}$ means the representations. In addition, the flowchart of learning representations and its purpose in machine learning tasks is depicted in Fig. II.1.

In addition, the greedy layer-wise training approach, which is comprised of two phases referred to respectively as pre-training and fine-tuning [7], is utilized in order to optimize the weights in DBNs and stacked AEs. The pre-training phase is depicted in Fig. II.1 for convenience. During this phase, a series of input-hidden-output NNs, which are all three-layer NNs having a different number of units in each layer, are trained independently by reducing reconstruction error. During the fine-tuning phase as depicted in Fig. II.2b, such hidden layers with the weights trained in the pre-training phase are sequentially stacked together. After that, a classifier functioned by a classification layer is attached to the trail, which targets at performing fine-tuning to optimize the related loss function indicated by the specific task that is being performed.

Unsupervised DNN methods are greatly preferred mostly because of their demands requiring less labeled data particularly in the modern Big Data era where most data received are unlabeled. However, a significant challenge that arises when training these models is determining how to ensure that the representations that are learned are meaningful. To be more specific, during the pre-training phase for training one NN unit, which is illustrated in Fig. II.3, let $X \in R^n$ signify the input data, $W \in R^{k \times n}$ signify the matrix of connection weights between the input layer

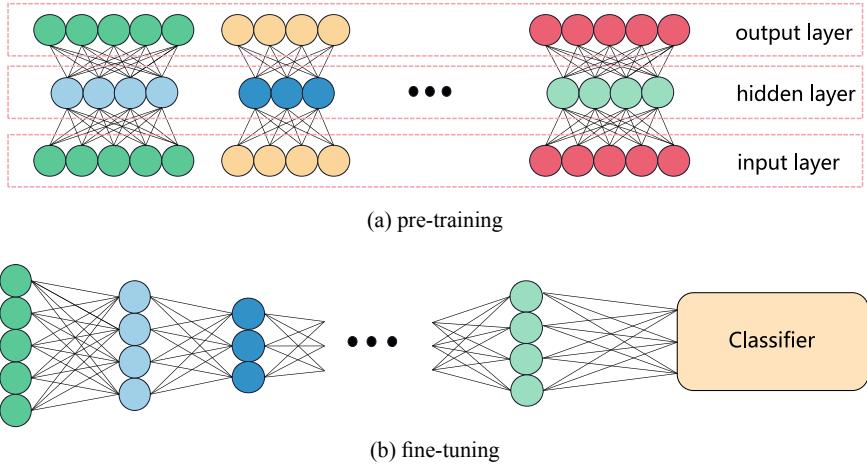
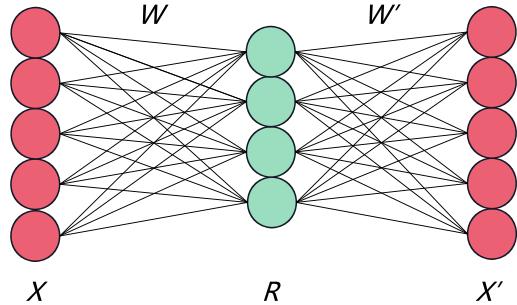


Fig. II.2 The traditional training process of unsupervised DNNs

Fig. II.3 An illustration example of unsupervised DNN model



to the hidden layer, and $W' \in R^{n \times k}$ signify the connection weight matrix from the hidden layer to the output layer. According to Eq. (2), the NN unit is working to reduce reconstruction error L that occurs between the input data X and the output data X' . It is important to notice that the bias terms, which represent an additional type of connection weight that is commonly seen in NNs, have been incorporated into W and W' in this case for the sake of simplicity.

$$\begin{cases} R = f(WX) \\ X' = f(W'R) \quad (2) \\ L = l(X, X') \end{cases}$$

The output resulting from the hidden layer is denoted by R in Eq. (2), which stands for the representations that have been learned, f stands for the activation function, and l stands for the function that measures the discrepancies between X and X' .

There is a strong correlation between the settings of the hyper-parameters and the architecture of DNNs and the performance of the DNN algorithms. To be more specific, hyper-parameters are the factors that must be assigned before training the models. These parameters include the number of weights, the values of the coefficients, and the types of activation functions and the prior terms. The potential of DNN algorithms cannot be showcased to its full potential since the existing optimization strategies for looking for ideal hyper-parameter settings and design have some weaknesses.

The neural architecture search (NAS) tries to automatically determine the best NN architecture for a given task by exploring the architecture search space [8], so that the automatically tuned architecture can achieve better prediction performance on unknown data. As a result, NAS automation makes deep learning approaches much more available to end-users, particularly for practitioners with limited domain-specific knowledge or DNN training experience. NAS research dates back to the 1990s [9], and early methods frequently evolved network architectural topologies, weights, and hyper-parameters all at the same time [10].

Within the recent few years, there has been a rise in research efforts by the NAS community [8] to automate the search process for network architecture, thanks to the resurrection of artificial intelligence and deep learning. Based on their base optimizers, existing NAS algorithms can be grouped into three categories: gradient-based algorithms [11], reinforcement learning (RL)-based algorithms [12], and EA-based algorithms, i.e., evolutionary neural architecture search (ENAS) algorithms [13].

The gradient-based approaches allow for fully differentiable architecture optimization. Differentiable architecture search [11] is widely regarded as a classic work that uses a continuous relaxation of the architecture representation to evolve network design using a gradient-based technique. Unfortunately, because gradient-based methods take a lot of graphics processing unit (GPU) memory to update all of the parameters, they can induce competition among processes when they are optimized together [14].

The RL-based algorithms perceive the architecture design process to be an activity. Baker et al. [15] developed the MetaQNN technique based on RL algorithms, in which states represented premature architectures, and architectures were modified using the ϵ -greedy method [16]. The work in [12] offered an alternate strategy based on the policy gradient method. They used a RNN as a controller to forecast the actions that are used to build a network according to prior actions in this approach. Cai et al. [17] used an RL agent as the meta-controller, which uses function-preserving changes to grow the network depth or layer breadth. The RL-based NAS, on the other hand, relies on hyper-parameters to provide stability, which forces controllers to perform tens of actions in order to receive a positive reward and is often ineffective [18].

In contrast to RL-based algorithms, EA-based algorithms are designed to discover specialized architecture through the creation of new individuals [13, 19]. Experiment evidence suggests that RL-based algorithms necessitate greater processing resources than EA-based algorithms [20] do. GA and PSO are two more common algorithms in addition to traditional EA. The network designs are encoded by the particles in PSO-based algorithms like SFCAE [21], SOBA [22], and BQPSO [23], and the velocity

and position of each particle are updated toward the ideal network architecture in each iteration. Many GA-based NAS algorithms have recently emerged, owing to the fact that GAs can tackle problems that are difficult to address directly.

Sun et al. [24] introduced a new technique for solving image classification issues, which uses the GA to update the architectures and connection weight initialization values of DNNs. Assuncao et al. [25] used the GA to construct the architecture of the AEs, with the goal of obtaining compressed representations on a specific dataset. Lu et al. proposed the NSGANet [26] and its variant NSGANetV1 [27] algorithms to approximate the entire Pareto frontier between the computational budget and predictive performance, with NSGANetV1 extending the earlier proof-of-principle NSGANet algorithm to consider the limitations of existing algorithms in terms of multi-objective optimization and vast computational resources.

Existing works on automatic architecture design, on the other hand, have apparent flaws. For example, Google [13] proposed a work on directly evolving CNNs for image classifications over 250 high-performance servers for more than 400 h. Although computational resources are not always available to all interested researchers, evolutionary techniques would undoubtedly be useful for creating deep NNs. Most researchers lack the computing resources that Google possesses. Furthermore, existing works for automatically designing AEs are primarily focused on ordinary AEs [25, 28], with strategies that cannot be directly applied to more complicated AEs such as CAEs [29] and VAEs [30]. The solutions to these problems will be detailed in the following chapters.

In this part, three representative approaches of ENAS for unsupervised NNs will be introduced: the architecture design for stacked AE and DBN method (EUDNN) [31], the architecture design for CAE method (EvoCAE) [21], and the architecture design for variational AE method (EvoVAE) [32], which are all for the architecture of unsupervised DNNs.

References

1. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 5210(7553), 436–444.
2. Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P. & Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 110(Feb), 625–660.
3. Delalleau, O. & Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *Advances in Neural Information Processing Systems*, pp. 666–674.
4. Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 1600(1), 106–154.
5. Lee, H., Ekanadham, C. & Ng, A. Y. (2008). Sparse deep belief net model for visual area v2. In *Advances in Neural Information Processing Systems*, pp. 873–880.
6. Utgoff, P. E., & Stracuzzi, D. J. (2002). Many-layered learning. *Neural Computation*, 140(10), 2497–2529.
7. Hinton, G. H., Osindero, S. & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 180(7), 1527–1554.

8. Wistuba, M., Rawat, A., & Pedapati, T. (2019). A survey on neural architecture search. <https://arxiv.org/abs/1905.01392>
9. Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9), 1423–1447.
10. Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2) 99–127.
11. Liu, H., Simonyan, K., & Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations*.
12. Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv e-prints*, art. arXiv:1611.01578, November 2016.
13. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). Large-scale evolution of image classifiers, pp. 2902–2911.
14. Dong, X., & Yang, Y. (2019). Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pp. 1761–1770.
15. Baker, B., Gupta, O., Naik, N., & Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.
16. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge.
17. Cai, H., Chen, T., Zhang, W., Yu, Y., & Wang, J. (2018). Efficient architecture search by network transformation. In *Thirty-Second AAAI conference on artificial intelligence*.
18. Chen, Y., Meng, G., Zhang, Q., Xiang, S., Huang, C., Mu, L., & Wang, X. (2019). Renas: Reinforced evolutionary neural architecture search. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pp. 4787–4796.
19. Elsken, T., Metzen, J. H., & Hutter, F. (2018). Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv e-prints*, art. arXiv:1804.09081.
20. Sun, Y., Xue, B., Zhang, M., Yen, G. G., & Lv, J. (2020). Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 500(9), 3840–3854.
21. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2019). A particle swarm optimization-based flexible convolutional autoencoder for image classification. *IEEE transactions on neural networks and learning systems*, 300(8), 2295–2309.
22. Fielding, B., & Zhang, Li. (2018). Evolving image classification architectures with enhanced particle swarm optimisation. *IEEE Access*, 6, 68560–68575.
23. Li, Y., Xiao, J., Chen, Y., & Jiao, L. (2019). Evolving deep convolutional neural networks by quantum behaved particle swarm optimization with binary encoding for image classification. *Neurocomputing*, 362, 156–165.
24. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2020). Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 240(2), 394–407..
25. Assuncao, F., Sereno, D., Lourenco, N., Machado, P., & Ribeiro, B. (). Automatic evolution of autoencoders for compressed representations. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8.
26. Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., & Banzhaf, W. (2019). Nsga-net: neural architecture search using multi-objective genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 419–427.
27. Lu, Z., Whalen, I., Dhebar, Y., Deb, K., Goodman, E., Banzhaf, W., & Boddeti, V. N. (2020). Multi-objective evolutionary design of deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*.
28. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2018). An experimental study on hyper-parameter optimization for stacked auto-encoders. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8..
29. Masci, J., Meier, U., Ciresan, D., & Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning—ICANN 2011*, pp. 52–59.

30. Kingma, D. P., & Welling, M. (2014). Auto-encoding variational bayes. In *International Conference on Learning Representations*.
31. Sun, Y., Yen, G. G., & Yi, Z. (2019). Evolving unsupervised deep neural networks for learning meaningful representations. *IEEE Transactions on Evolutionary Computation*, 23(1), 89–103.
32. Chen, X., Sun, Y., Zhang, M., & Peng, D. (2020). Evolving deep convolutional variational autoencoders for image classification. *IEEE transactions on neural networks and learning systems, early access*.

Chapter 3

Architecture Design for Stacked AEs and DBNs



3.1 Introduction

As introduced in Part II, altering n in Eq. (1) could learn numerous different representations, but only those that perform exceptionally well on the machine learning tasks linked with them are given attention. These representations are also referred to as meaningful representations based on literature evaluations [1–5]. Assuming that R_j are meaningful representations, it is obvious that hyper-parameter settings, including the number of layers, j , and the chosen activation function types of f_1, \dots, f_j , and the parameter values comprising of the values of each element in $\{W_1, \dots, W_j\}$ would strongly reflect whether or not the learned R_j are meaningful. As for this aim, the BP algorithm [6], which uses gradient information to train parameter values, is the most extensively used algorithm. However, because of its local search features, which can easily be stuck in local minima [7], its performance is heavily influenced by the initialization configuration. Although multiple BP implementations have been presented, such as SGD, AdaGrad [8], RMSProp [9], and AdaDelta [10], extra hyper-parameters, such as momentum initialization values and balance factors, are included and must be fine-tuned beforehand. Furthermore, several techniques [11, 12] for optimizing the hyper-parameters have been presented, however they frequently need domain expertise and are depend on the problem. As for this goal, the grid search approach was created, which maintains its dominance in the selection of suitable hyper-parameters [13]. When the hyper-parameters are continuous numbers, however, the grid search method typically misses the optimal hyper-parameter combinations.

According to the foregoing research on the possibilities of unsupervised deep NNs for acquiring meaning representations, as well as ENAS revealed in Part II, an effective and efficient approach for learning meaningful representations named EUDNN has been designed. The following is a list of the contributions made in EUDNN:

1. It has been designed that evolutionary approaches evolve DNNs with a huge amount of parameters for tackling high-dimensional input with limited processing resources. EUDNN can be implemented in academic settings with restricted computer resources thanks to its architecture.
2. Unsupervised models have been directed to know their usefulness in advance using a fitness evaluation technique designed in EUDNN, this can enable the acquired representations to be meaningful with little correctly constructed prior expertise.
3. DNNs that have a huge amount of parameters are an example of a large-scale global optimization problem. Because of this, a single evolutionary technique cannot deliver the best possible results for these networks. To achieve this purpose, a local search strategy is designed into EUDNN to ensure the desired performance.

The rest of this chapter is structured as follows. To begin, Sect. 3.2 illustrates related work and rationale for EUDNN. In Sect. 3.3, the specifics and debates of EUDNN are described. To assess the performance, a series of experiments are conducted on EUDNN against selected peer rivals, with the results quantified by the selected performance metrics examined in Sect. 3.4.

3.2 Related Work and Motivations

In Sect. 3.2.1, the work that has been done on unsupervised deep learning methods will be discussed, point out their limitations when it comes to learning meaningful representations, and justify the goals of designing EUDNN. In addition, the existing works will be discussed in Sect. 3.2.2.

3.2.1 *Unsupervised Deep Learning*

In Eq. (3.14), lowering L to R does not imply that the acquired representations R are meaningful because there is no information about the associated classification problem in this stage, and whereas R is only meaningful when it can increase the performance of the related classification problem, any R will result in a minimal L . To achieve this goal, literatures have given unsupervised deep learning algorithms with various priori information [2, 3, 14, 15], denoted as Θ , and the reconstruction error is translated into $L = l(X, X') + \lambda\Theta$, where λ specifies a balance factor to define the weight of the related priori term. While such prior information might aid in the meaning of the learned representations, major concerns remain. First of all, this prior knowledge is built on several assumptions that may or may not apply to the current circumstance. Second, the prior information is offered explicitly for general tasks, with the intention that performance on specific tasks may increase. Third, choosing

the most appropriate priori term for the current objective is tricky. Last but not least, the balance factor λ is a difficult-to-determine hyper-parameter in practice [3].

In order to address these issues, EUDNN employs a method that was previously devised [5]. To be more exact, while evaluating the effectiveness of EAs, only a tiny portion of the data that has been labeled is utilized, and the learned representations are immediately quantified by employing the classification problem that was utilized in the step of fine-tuning. Individuals which have a good effect on the classification problem are predicted to have offspring with increased effectiveness for the next generation, which will result in meaningful learned representations. There are no additional expenditures associated with EUDNN because the labeled data utilized can be added from fine-tuning stage and classification goals are identical as in the fine-tuning stage.

3.2.2 *Evolutionary Algorithms for Evolving Neural Networks*

Because EUDNN seeks to evolve deep NNs, only the studies of NEAT and HyperNEAT [16, 17] will be discussed here, despite the fact that several relevant literatures for evolving NNs were cited in Part II. Please keep in mind that the works in [18–24] were suggested two decades ago and cannot be used to deep NNs; the work in [25] exclusively dealt with weight pruning, while the work in [26] used a direct method of evolution and had no general meaning. The shortcomings of NEAT and HyperNEAT in evolving deep NNs are initially documented in the following sections. The motivations for EUDNN are described in conjunction with the obstacle that EAs face in evolving deep NNs, namely the upper bound encoding problem. Furthermore, EAs cannot fully solve optimization tasks with a huge amount of variables, and the arguments for this are presented.

To increase the adaptive complexity of developing NNs in an indirect approach, NEAT has been designed. The NEAT has two categories of genes: node genes and connection genes. The type of unit indicating the unit is input, hidden or output and an identification number are encoded in node genes, which are then used to represent all the units of the evolved NN. One node gene includes five components, and the connection genes are used to represent the relationship information among the node genes. In particular, the information includes the amounts of the input and output units, the value of the connection, one bit indicating whether the connection is active or not, and one innovation number recording the index of the connection gene in an enhanced manner. Individuals are initially just given the input and output units of the network, as well as the randomized connections among them, during the evolutionary process. Individuals are remerged and modified after that. To be more explicit, connection mutations and node mutations are the two forms of mutations. When a couple of node genes is connected, connection genes have been increased by one gene. When it comes to node mutations, one hidden node is developed first, followed by the appropriate connection gene, which splits an existing connection into two parts. Although NEAT is flexible in terms of NN evolution, it requires a

predictable number of outputs, which is unrealistic in the field of deep learning. Furthermore, because NEAT directly encodes each connection and unit, it is not ideal for generating deep NNs with a huge amount of connections and units. The connective compositional pattern producing networks (CPPN) [17, 27] have been presented to address this deficiency regarding the inability to evolve deep NNs.

By merging NEAT and the CPPN encoding technique, HyperNEAT has been designed. The CPPN, in particular, uses a single low-dimensional grid to construct NEAT connections with a set of predefined nonlinear functions. To be more explicit, any point in the coordinate system is selected and put into a list of compositional functions to accomplish the transformation from genotype to phenotype. Because any points could be selected from the low-dimensional coordinate system, this makes it possible to represent many connections at a low computational cost. In this aspect, HyperNEAT would have the most possibility for evolving a deep NN, however the amount of the output must be determined previously. Furthermore, because of the nature of large-scale global problems, all of the values of the connections in HyperNEAT are produced by genetic operators in evolution, which does not ensure the greatest performance in creating a deep NN. As a result, this approach uses recurrent connections or connections between the same layers, which are not suited for acquiring compact meaningful representations.

The hyper-parameter configurations and parameter values have a significant influence on the effectiveness of deep learning algorithms. One of the major hyper-parameters in the pre-training phases is the size of hidden layers. When using EA techniques to find the sizes, one issue that naturally arises is how to define the upper bound of the sizes of the hidden layers given the strategy of a fix-length encoding gene. Indirect encoding helps to alleviate this problem, but it limits the generalization of evolved NNs and the architecture space that may be utilized [18]. Alternatively, because a large number would necessitate more computing cost and, as a result, reduce model performance, there is no way to predict how large the upper bound should be. Nevertheless, Yang et al. [28] analytically demonstrated that the meaningful representations of the input data are found in the original space. If the input data has n dimensions, the size of the related hidden layer should not exceed n . n orthogonal n -dimensional basis vectors are enough to cover a n -dimensional space. As a result, only one basis r_1 of n -dimensional space is needed to compute, and the other $(n - 1)$ n -dimensional basis vectors may be obtained explicitly using Eq. (3.1) to locate the null space. Please note that multiple solutions might theoretically be obtained by computing the null space of one vector. In practice, only the orthogonal basis generated from the singular value decomposition for the associated null space needs to be considered. For this reason, the task with n^2 parameters by explicitly encoding around n parameters using a GA, which is a computationally efficient gene encoding strategy, is easily be represented.

$$\text{null space}(r_1) = \{x \in R^n | r_1 x = 0\} \quad (3.1)$$

EUDNN models unsupervised deep NNs with a huge amount of parameters using the computationally efficient gene encoding approach outlined above. Regardless of

the approach employed, the number of parameters in the initial issues remains identical, despite the fact that the encoded parameters have been significantly reduced. In reality, the impacts of one gene in the used encoding approach are equal to various variables in the initial issues. The computationally efficient gene encoding technique suggested herein, for example, uses just 1,000 genes for a NN with 100,000 parameters. Consequently, one gene contains on average 100 parameters, and changing one gene using crossover and mutation operations can change 100 parameters. Furthermore, it is widely known that the performance of EAs performance is ensured by their exploration search provided by mutation operators and exploitation search provided by crossover operators, which introduce global and local search capabilities, respectively. Because a small change in one gene in EUDNN causes numerous parameters to change that affects global behavior, it may be said that EAs lack search strategy from the problem to be solved. Furthermore, the data processed by deep learning algorithms is frequently of high dimension, resulting in thousands of choice variables in the encoded chromosomes of EAs, despite the fact that the encoding scheme has conserved a lot of space in comparison to other methods. Widespread investigations have shown that EAs struggle to achieve the optimal results on issues with large input dimensions. A local search technique is implemented into EUDNN to resolve this concern and provide the desired performance.

3.3 Algorithm Details

The details of EUDNN are presented in this section. To be more detailed, the framework is shown first in Sect. 3.3.1, which is made up of two independent stages. Next, both stages are described in Sects. 3.3.2 and 3.3.3, respectively. Last, the over-fitting problem-prevention in EUDNN and notable distinctions from its peer rival are highlighted in Sect. 3.3.4.

3.3.1 Framework of EUDNN

It is assumed that the learned representations are for a classification problem wherein the meaningful representations can increase performance in terms of a higher correct classification rate (CCR) for the sake of development. Furthermore, in this classification task, given a set of data D , a portion of D denoted by $D_{train} = \{(x_1, y_1), \dots, (x_k, y_k)\}$ is regarded as the training data, where x_i denotes the input data and y_i is the relevant label, the remaining data is considered the test data D_{test} for assessing whether or not meaningful representations can be learned. In addition, Fig. 3.1 shows the flow chart of EUDNN, which simply demonstrates the two steps of the design: (1) finding the best architectures in deep NNs, the desired initialization of connection weights, and the activation functions; (2) fine-tuning all

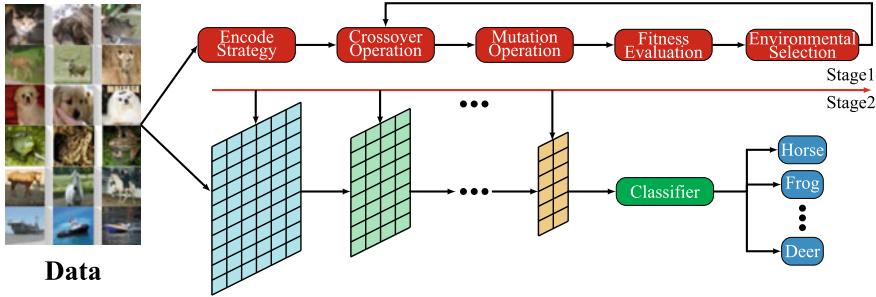


Fig. 3.1 The flow chart of EUDNN is divided into two steps. The first is dedicated to identifying ideal architectures and desirably initialized connection weight parameter values. The second involves fine-tuning them for improved performance

the parameter values in connection weights from the appealing initialization. Both steps are also termed as pre-training and fine-tuning, respectively.

Algorithm 1: Framework of EUDNN

Input: Maximum number of layers p , classifier $C(\cdot)$, training data D_{train} , test data D_{test} .

Output: Predicted labels of D_{test} .

```

1  $i \leftarrow 0$ ;
2 while  $i < p$  do
3    $i \leftarrow i + 1$ ;
4    $| W_j, f_j(\cdot) \leftarrow$  Evolving the appropriate connection weights and activation functions;
5 end
6 Fine-tuning connection weights  $W_1, \dots, W_p$ ;
7  $Y_{test} = C(f_p(W_p \times \dots f_2(W_2 \times f_1(W_1 \times D_{test}))))$ ;
8 Return  $Y_{test}$ .

```

To achieve this, one efficient genetic strategy presented in Sect. 3.2.2 is used to encode the potential architectures and affiliated huge amounts of parameters in connection weights by a group of individuals, and then EA is used to evolve and select the individual with the optimal performance according to the fitness measures. The method introduced in Sect. 3.2.1 is then used to ensure that the learned representations are meaningful, i.e., a small portion of data D_f from D_{train} is randomly selected, and the representations of D_f are learned based on the models encoded by the individuals, then fed with the affiliated classification task to select the ones that give the highest CCR for evolution. In the second step, a fine-tuning technique, which offers the exploitation local search, is used to save the optimal performance ever obtained, which supports the exploration global search during first step, based on the studies in Sect. 3.2.2. In essence, these two processes work together to ensure that unsupervised deep NNs learn meaningful representations.

Algorithm 1 presents the framework of EUDNN. The first stage is described in lines 2–5, while the second stage is defined in line 6. Finally, in lines 7 and 8, the projected labels of the test data are derived and returned. Following that, the specifics of these two steps are described.

3.3.2 Evolving Connection Weights and Activation Functions

A variety of repetitive subprocesses are involved in getting all of the optimal connection weights and their accompanying activation functions. Firstly, how to acquire the appropriate connection weight and also its activation function in Algorithm 2 will be discussed in this section. The full procedure is then described.

Algorithm 2: Evolving Connection Weights and Activation Functions

Input: Input data, population size m , probability of crossover ρ , probability of mutation μ .
Output: Evolved connection weight W and activation function $f(\cdot)$.

```

1 Initialize a population  $P$  having the size of  $m$ ;
2 while stopping termination are not met do
3   Evaluate the fitness value of each individual in  $P$ ;
4    $Q \leftarrow$  Generate new offspring with a probability of  $\rho$  from two parents chosen using a
      binary tournament selection;;
5    $Q \leftarrow$  Perform the mutation with a probability of  $\mu$  for each individual in  $Q$ ;
6    $S \leftarrow$  Choose the best-fitness individual from  $P \cup Q$ ;
7    $P \leftarrow S \cup$  Choose  $(m - 1)$  individuals from  $(P \cup Q) \setminus S$  using a binary tournament
      selection;
8 end
9 Evaluate the fitness value of each individual in  $P$ ;
10  $ind_{best} \leftarrow$  Choose the best-fitness individual from  $P$ ;
11 Return  $W$  and  $f(\cdot)$  encoded by  $ind_{best}$ .

```

In particular, m individuals encoding the information of potential ideal connection weights and their accompanying activation functions are initiated first in Algorithm 2 (line 1). The evolution then begins (lines 2–8) and continues until the termination conditions are met, such as reaching the maximum generations. The fitness of all individuals is evaluated initially during each generation (line 3). Then, using the probability ρ , new offspring are generated, and their parents are chosen from P using the binary tournament selection (line 4). After that, with the probability μ , all the children in Q are altered (line 5). Moreover, lines 6 and 7 describe environmental selection wherein the best individual is first to be saved for elitism, and then $(m - 1)$ individuals are picked from the left-over solutions in $P \cup Q$ using binary tournament selection. Whenever the evolution is accomplished, the optimal solution for converting the best connection weight and activation function is chosen from the current population (lines 9 and 10).

Following that, the specifics of the applied gene encoding technique will be explored, with the understanding that the fundamental concepts have already been laid forth in Sect. 3.2.2. The potential connection weight for producing meaningful representations in [28] has been pointed out to be in a subspace of the original feature space. As a result, the searching for the best connection weight might be limited in the input data space. The assumption is that the input data is n -dimensional in nature. First, the certain linear independent n -dimensional vectors are supplied as a set of basis $S = [s_1, \dots, s_n]$ that can span a n -dimensional space. The vector a_1 is then linearly merged with the components $b = [b_1, \dots, b_n]$ by the basis in S . Equation (3.1) can then be used to construct the orthogonal complements $\{a_2, \dots, a_n\}$ in terms of a_1 . Please note that the ability of $\{a_1, a_2, \dots, a_n\}$ to span the range of input data is self-evident. Finally, a binary encoded string showing whether the appropriate basis is available or not is used to choose a subset of these bases that cover a subspace of the original space for creating the best connection weight. Furthermore, the chromosome contains the activation function for the cheval. A list of selected activation functions with various nonlinear capabilities is provided, and their orders in the list are picked to indicate which one is chosen. Furthermore, Fig. 3.2 is included to show how the connection weight and activation function are intended to be efficiently encoded. When the optimal connection weight W_i and its relating activation function f_i for the i -th layer are found using Algorithm 2, the $(i+1)$ -th layer can be optimized that use the same algorithm by defining the input data to $f_i(W_i \times R_i)$, where R_i signifies the representations at the i -th layer.

Each component in b is expressed using nine bits in the used gene encoding method, with the leftmost bit indicating whether the coefficient is positive or negative. Then one bit is utilized to specify if the connection weight is chosen using the basis a_j ($j \in [2, \dots, n]$). Finally, the activation function is represented by two bits. In

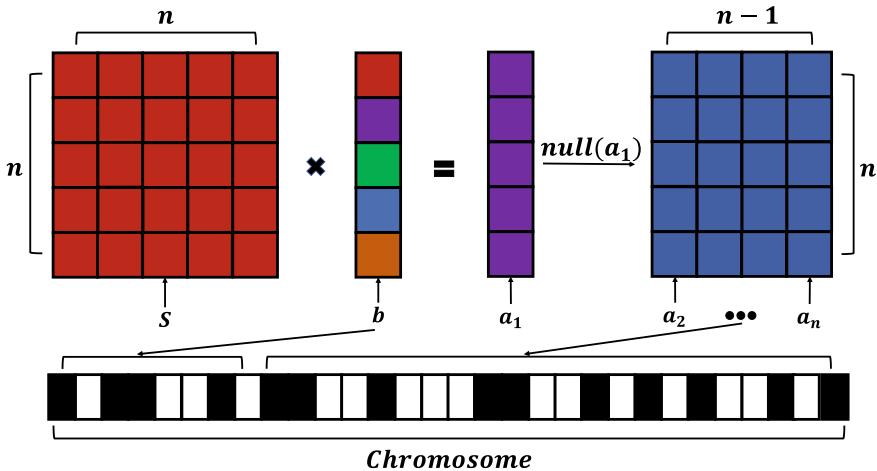


Fig. 3.2 The flow chart of encoding the potential connection weight and activation function

addition to the widely used sigmoid and hyperbolic tangent functions, ReLU, which has lately been shown to perform better in some situations, is now being examined. As a result, the n -dimensional input data requires $(10n + 1)$ bits per chromosome. If using the real number encoding approach would need a memory space that is a multiple of eight, and is one of the main reasons why EUDNN uses the binary encoding method as part of the so-called computationally efficient gene encoding strategy.

Furthermore, due to the favorable computational efficiency and linear nature for greater discriminating ability if the learned representations are meaningful, the linear support vector machine (SVM) [29] is used for assessing the efficacy of individuals.

3.3.3 Fine-Tuning Connection Weights

To boost performance even more, the second stage includes an exploitation mechanism that uses a local search method to fine-tune parameter values of connection weights. Having fine-tuned all connection weights and activation functions in the first step, they are integrated with the classifier to finish the process of fine-tune. In this case, fine-tuning is accomplished through the BP method. In the second stage, any local search strategy can be employed. Two factors are mostly responsible for using BP: (1) the gradient message in the loss function is often analytic, and the BP algorithm is naturally used in the majority of design concepts; (2) numerous BP libraries for accelerating computing with GPUs have been implemented, and the cost of computing may be significantly decreased, particularly when processing high-dimensional data. Furthermore, the value of zero is given according to community convention when the ReLU function is not differentiable at the location of zero [30].

Algorithm 3 also includes details on fine-tuning. The classifier $Y_{pred} = C(f_p(W_p \times \dots \times f_2(W_2 \times f_1(W_1 \times X))))$ in line 2 computes the predicted labels Y_{pred} of the training examples X . Line 3 then measures the difference L between both the predicted labels and the ground truth, which varies depending on the classifier employed. Then, until the terminating conditions are satisfied, the deviation of L with regard to W_i is calculated, and W_i is changed with the step size α (lines 4–6). The fine-tuned W_1, \dots, W_p is then kept for future tasks. Additionally, Fig. 3.3. depicts the flow chart for this phase.

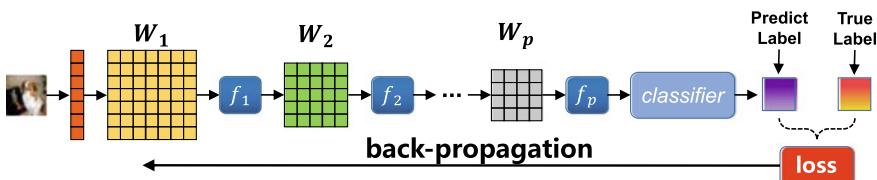


Fig. 3.3 The flow chart for the second stage of EUDNN

Algorithm 3: Fine-tune Connection Weights

Input: Training data X and labels Y ; connection weights W_1, \dots, W_p ; activation functions $f_1(\cdot), \dots, f_p(\cdot)$; classifier $C(\cdot)$; step size α .

Output: Fine-tuned W_i, \dots, W_p .

```

1 while stopping termination are not met do
2    $Y_{pred} \leftarrow$  Calculate the predicted labels of  $X$ ;
3    $L = l(Y, Y_{pred}) \leftarrow$  Calculate the value of loss function;
4   for  $i = 1, \dots, p$  do
5      $| \quad W_i = W_i - \alpha \frac{\partial L}{\partial W_i}$ 
6   end
7 end
8 Return  $W_i, \dots, W_p$ .

```

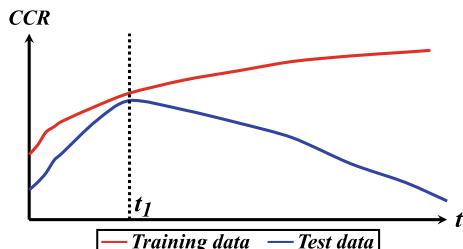
3.3.4 Discussion

This section focuses on the over-fitting problem prevention mechanism designed in EUDNN, as well as the important differences between it and the direct evolutionary feature extraction algorithm (DEFE) [31], which uses a similar gene encoding technique as EUDNN does.

The over-fitting issue refers to a poor generalization capacity of the model, in which the trained model achieves a higher CCR on training data but a lower CCR on test data. Due to the purpose of training a classification model is to get a better CCR on test data, some mechanisms should be in place to prevent over-fitting. Given a set of Vapnik Chervonenkis (VC) [32] dimension-based models, the lower the VC dimension, the simpler the associated model, and the higher the generalization of a simpler model. The VC dimension could be considered as a measure of the complicity of several models capable of performing a single job. [33]. Since the amount of parameters is proportional to the value of the VC dimension [1], and deep NN designs typically have a large number of parameters, over-fitting is a common problem in these models.

Figure 3.4 depicts a typical instance of CCR on training data (red curve) and checking on test data (green curve) as the training process progresses. CCR on both data is continuously increasing until t_1 , and then CCR on the training data proceeds to rise while CCR on the test data starts to fall after the training period exceeds t_1 ,

Fig. 3.4 The change of CCR values on training data and test data as training process continues



which suggests the existence of an over-fitting issue. As mentioned above, the optimal performance of EUDNN cannot be ensured during the first stage of training, therefore the second step is introduced to assist EUDNN in achieving the best performance. As a result, the issue of over-fitting would not occur in the first step of EUDNN as claimed. This is because the first stage ends before t_1 . However, the over-fitting issue may appear after t_1 , which corresponds to the second stage of EUDNN. As a result, some rules must be used just in the second stage to prevent this problem. The “early stop” approach is used here, thus, $D_{validate}$ is uniformly picked from D_{train} as the validation data to replace the testing on test data in Fig. 3.4, since initially noticed that the CCR of validation data has begun to drop while the CCR of training data continues to rise, i.e., the precise time t_1 is discovered, the fine-tuning operation is completed, and the model with the top performance is obtained. Next, the difference between EUDNN and DEFE will be discussed.

In the literature, DEFE appears to learn only linear representations of input data and shallow representations of input data. DEFE is unable to learn meaningful representations [34] as a result of these two findings. To be more precise, the representations R learned by DEFE can be written as $R = WX$ [31], where W is the transformation matrix representing the connection weight in deep NN models and X is the input data. As can be seen, because deep linear transformations are equivalent to a one-layer linear representation, and DEFE will only learn linear representations, there is no nonlinear transformation that can be performed on WX , for nonlinear representation learning, on the other hand, EUDNN will provide sets of activation functions with diverse nonlinear transformation capabilities.

In conclusion, because of its linear nature, DEFE cannot be used to acquire meaningful representations, whereas the achievement of deep NNs is mostly owing to the meaningful representations learned via deep nonlinear transformations, that have been specifically achieved by EUDNN.

3.4 Experimental Design

The performance of EUDNN is evaluated using simulations according to a set of image classification benchmarks versus a set of peer rivals. Specifically, the CCR on the test data is considered by the selected performance metric during the comparisons. This is followed by the examination of the selected peer competitors, along with the justification for their selection. Next, the details of the chosen performance metric and the specifications of parameter values used by these compared methods are shown. In the end, both quantitative and qualitative experimental results are displayed and thoroughly examined.

3.4.1 Performance Metric

Because the learned representations are intermediate outcomes, it is technically impossible to determine whether they are meaningful or not. Commonly, these learned representations are applied to a particular classification issue, and then a classifier is used to explore the CCR. Typically, a better CCR indicates that the learned representations are more meaningful. Due to the fact that the benchmarks used in this researches are multi-class classification problems, the Softmax regression classifier is often employed to compute the corresponding CCR in compliance with community standards.

A set of training data and their accompanying labels with k different integer values are supposed to be denoted as $\{x_1, \dots, x_m\}$, and $\{y_1, \dots, y_m\}$, respectively, where $x_i \in \mathcal{R}^n$ and $y_i \in \{1, \dots, k\}$. To be specific, the label of the sample x_i ($i \in \{1, \dots, m\}$) is predicted by Eq. (3.2) with the Softmax regression,

$$\arg \max_j p_j(x_i) = \frac{\exp(\theta_j^T x_i)}{\sum_{l=1}^k \exp(\theta_l^T x_i)} \quad (3.2)$$

where $\Theta = [\theta_1, \dots, \theta_k]^T$ are obtained by minimizing

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{j=1}^k f(y_i, j) \log \frac{\exp(\theta_j^T x_i)}{\sum_{l=1}^k \exp(\theta_l^T x_i)} \right] \quad (3.3)$$

in which $f(y_i, j) = 1$ if $y_i = j$, otherwise $f(y_i, j) = 0$.

3.4.2 Peer Competitors

Because the goal of EUDNN is to evolve an unsupervised DNN to learn meaningful representations, approaches relevant that may be used to the evolution of deep NNs, such as NEAT, HyperNEAT, DBNs, and variants of stacked AEs, should all be used as peer competitors. However, due to the aforementioned issues, it is impossible to acquire meaningful representations by the use of the NEAT or the HyperNEAT. Therefore, they are excluded from the comparison algorithms. Experiments involving comparisons are conducted with DBNs in addition to stacked AE variants. RBMs and AEs are employed in studies to compare the performance of learned representations to that of EUDNN since they are the building blocks for training DBNs and stacked AEs, respectively. To be more specific, RBMs and AEs as unsupervised deep NN models will be evolved, dubbed EUDNN/RBM and EUDNN/AE, respectively, to undertake comparisons against rivals. In recent years, many AE variations, such as the SAEs, DAEs, and CAEs, are presented with various regularization terms to learn meaningful representations and have also shown similar performance in several

tasks. Consequently, in addition to the DBNs, they are included as peer rivals in the comparisons.

3.4.3 Parameter Settings

The parameters in the second step of EUDNN and the competing ones are the same for a fair comparison. As a result, the description of such generic parameter values will be firstly discussed in this section. Because the optimal performance of the examined algorithms is frequently highly reliant on the specific benchmark dataset as well as parameter choices, to provide a fair comparison, evaluating these parameters on corresponding training data using a commonly accepted range and then compare the best performance of each compared method on test data.

According to community convention, SAE, DAE, CAE, and Softmax regression are all trained using the SGD approach, which has learning rates of 0.0001, 0.001, 0.01, 0.1, and batch sizes of 10, 100, 200, respectively. A total of 30 trials are carried out to compare the various methods. Furthermore, during the training of the Softmax regression, a performance monitor is inserted into each epoch to save the optimal CCR across the test dataset as the optimal performance of the algorithm which supplies the learned representations to the Softmax regression. The SAE, DAE, CAE, and RBM use a *log* function with a 0.5 interval and a maximum depth of 5 and the number of units in each layer being between 200 and 3000 recommended by [35]. The outcomes assessed by the chosen performance metric must be compared in statistical due to the heuristic nature of the first stage defined in EUDNN. Base on community convention, there was a 5% important threshold used in these investigations employing a Mann-Whitney-Wilcoxon rank-sum test [36].

Furthermore, the binary corrupted degree of DAE, the sparsity of SAE and the contractive component of CAE are each set to 10%, 30%, 50% and 70%, respectively. The CD- k method is chosen to train RBM because of its nature, and k is set to 1 according to the recommendation [35]. To accelerate the first stage of EUDNN, 20% of the training dataset is chosen at random for fitness evaluation in each iteration. Furthermore, the weights and biases of the connections are set to $[-4 \times 6/\sqrt{n_{number}}, 4 \times 6/\sqrt{n_{number}}]$ with uniform sampling, where n_{number} represents the total amount of units in two neighboring layers according to the experiences presented at [37].

Due to the fact that the second-stage parameter values in EUDNN are similar to those of peer competitors, the first-stage parameter settings for evolution-related parameters are declared. Specifically, at this step, a single chromosome has the potential to split into three parts: coefficients that are related to the main basis (Part 1), which are utilized to denote the vector a_1 in Fig. 3.2, and coefficients that are related to projected space (Part 2), which are utilized to show which bases are selected for the connection weight, and coefficients that are related to activation functions (Part 3) are also included. It is envisaged that crossover operations will be pushed in Parts 1 and 2 in order to improve exploiting local search, which delivers much higher

performance than exploratory global search. This is because Parts 1 and 2 have such a large influence on the performance of the connection weight. As a result, in Parts 1 and 2, the one-point crossover operator is used. In addition, EUDNN considers three frequently used nonlinear activation functions, with one to be chosen for the associated connection weight. As a result, it is expected that the information about how the activation function does not change frequently, as it is difficult to identify which is the best. As a result, Parts 2 and 3 are regarded to be one part for the crossover operation. When the result in Part 3 becomes invalid, a random value is chosen to replace it. In addition, the identical values are used for the probabilities of crossover and mutation as in the community convention, i.e., 0.1 in mutation and 0.9 in crossover.

3.5 Experimental Results and Analysis

Based on the design motivation, EUDNN (1) utilizes an evolutionary method and a local search strategy to guarantee that deep NNs learn meaningful representations, (2) utilizes an evolutionary method in the first stage to assist deep NNs in discovering ideal architectures and initialized weights, hence providing a better starting position for the second stage, and (3) uses a local search strategy in the second stage to significantly enhance the expected performance. As a result, to assess the performance of EUDNN, a series of tests has been carefully designed.

3.5.1 Performance of EUDNN

A number of well-designed experiments and comparisons are done to quantify if the representations learned by EUDNN are meaningful. EUDNN/RBM and EUDNN/AE are two implementations of EUDNN over unsupervised NN models, respectively (i.e., RBMs and AEs). After that, using the setups described above, they are utilized to acquire the representations alongside selected peer competitors. Following that, the Softmax regression measure is used to determine if the learned representations have improved the related classification problems using CCR, indicating whether the learned representations are meaningful or not.

Table 3.1 lists the mean values and standard deviations of CCR produced by these compared algorithms over 30 individual runs, with the top results over the same benchmark marked in bold. Furthermore, using the rank-sum test, the symbols “+”, “-”, and “=” indicate whether the CCR of EUDNN on the following benchmarks is statistically superior to, inferior to, or equal to those from the corresponding peer competitors, accordingly. To perform this statistical test, the higher CCR produced by EUDNN/AE and EUDNN/RBM with the same benchmark will be chosen firstly, and then perform the rank-sum test on the selected results. In addition, the last row

Table 3.1 The CCR of EUDNN/RBM and EUDNN/AE upon chosen benchmark and peer competitors. Specifically, the best mean values are highlighted in bold face, the symbols “+”, “-”, and “ \approx ” denote whether EUDNN statistically are better than, worse than, and equal to that of the corresponding peer competitors, respectively, using the employed rank-sum test

Benchmark	EUDNN		CAE	DAE	DBN	SAE
	RBM	AE				
MNIST-basic	0.9633(0.00473)	0.9674(0.00616)	0.9635(0.00831)(+)	0.9580(0.00352)(+)	0.9658(0.00550)(+)	0.9776(0.00585)(-)
MNIST	0.9885(0.00255)	0.9878(0.00751)	0.9843(0.00699)(+)	0.9820(0.00506)(+)	0.9771(0.00959)(+)	0.9832(0.00891)(+)
MNIST-back-rand	0.8386(0.00054)	0.88443(0.00076)	0.5741((0.00779)(+)	0.7725(0.00531)(+)	0.8221(0.00130)(+)	0.8851(0.00934)(=)
MNIST-rot	0.7549(0.00286)	0.7952(0.00917)	0.7706(0.00754)(+)	0.7274(0.00757)(+)	0.7639(0.00568)(+)	0.7852(0.00380)(+)
MNIST-rot-back-image	0.8879(0.00815)	0.8925(0.00906)	0.6574(0.00913)(+)	0.8691(0.00127)(+)	0.8830(0.00098)(=)	0.8733(0.00632)(+)
MNIST-back-image	0.4830(0.00469)	0.4325(0.00569)	0.4010(0.00337)(+)	0.4022(0.00012)(+)	0.4587(0.00794)(+)	0.4638(0.00162)(+)
Rectangles-image	0.7716(0.00048)	0.7521(0.00689)	0.7810(0.00784)(=)	0.7598(0.00451)(+)	0.7628(0.00913)(+)	0.7725(0.0002)(-)
Rectangles	0.9681(0.00829)	0.9627(0.00311)	0.6275(0.00602)(+)	0.9232(0.00166)(+)	0.9622(0.00154)(=)	0.9408(0.00263)(+)
CIFAR-10-BW	0.4331(0.00962)	0.4798(0.00107)	0.4860(0.00775)(+)	0.4309(0.00005)(+)	0.4598(0.00869)(+)	0.4423(0.00817)(+)
Convex	0.8085(0.00826)	0.8113(0.00052)	0.8016(0.00996)(+)	0.7930(0.00538)(+)	0.7895(0.00443)(+)	0.8053(0.00878)(+)
	+/-=		9/0/1	10/0/0	8/0/2	7/2/1

of Table 3.1 summarizes how many times EUDNN is superior to, inferior to, or equal to the equivalent peer competition over the studied benchmarks.

The statistical results bias the results obtained by the statistical significance tools, i.e., the Mann-Whitney-Wilcoxon rank-sum test. As shown in Table 3.1, EUDNN/AE achieves the best mean values on the MNIST-rot-back-image, MNIST-rot, CIFAR-10-BW, and Convex benchmarks, as well as the top rank-sum results on the CIFAR-10-BW, Convex, and MNIST-rot benchmarks. Furthermore, in the MNIST and MNIST-back-image benchmarks, both the best mean values and the greatest rank-sum results go to EUDNN/RBM. Despite the fact that the best result of EUDNN on the EUDNN/AE over the MNIST-basic benchmark is somewhat worse than those of SAE, which wins the best mean value and rank-sum results, EUDNN/AE beats all other peer rivals. In addition, SAE achieves the best mean values on the MNIST-basic and MNIST-back-rand benchmarks, while the best result of EUDNN produced by EUDNN/AE on MNIST-back-rand is statistically equivalent to that of SAE, as well as surpassing competitor algorithms. On Rectangles-image, optimal result of EUDNN obtained by the EUDNN/RBM is inferior to that of CAE and SAE, despite EUDNN/RBM and CAE having statistically identical results. Furthermore, the optimal results of EUDNN/AE on MNIST-rot-back-image and EUDNN/RBM on Rectangles are statistically equal to those of DBN, despite EUDNN/RBM and EUDNN/AE having the best mean values on these two benchmarks, respectively. As a conclusion, EUDNN outperforms the selected peer rivals by 34 over the 40 comparisons, demonstrating its superior ability to learn meaningful representations using unsupervised NN models.

3.5.2 Analysis on Pre-training of EUDNN

It has been hypothesized that the first stage in EUDNN makes it easier for unsupervised NN based models to learn optimal designs and more appropriately initialized parameter values. In this regard, experiments are performed on a component-by-component basis on optimized architectures and initialized parameter settings, which in turn to evaluate the effects of each component on the overall design for the design justification. The settings of the initialization parameters, on the other hand, are dependent on the architecture. This results in an experiment that is difficult to build, in which only the architectural settings are modified to examine how the learned architectures impact performance. As a consequence of this, the primary focus of this investigation is on the quality of the initial parameter values.

To that purpose, the architecture settings with which EUDNN performs well in terms of best mean EUDNN/AE and EUDNN/RBM values for each benchmark from Table 3.1 will be collected. After that, peer competitors repeat the experiments using the stored architecture settings and randomly initialized parameter values. Finally, the learned representations are input into the performance metric in question to see if they are meaningful. The experimental results are shown in Fig. 3.5 with the CCR on the horizontal axis and the benchmarks on the vertical axis.

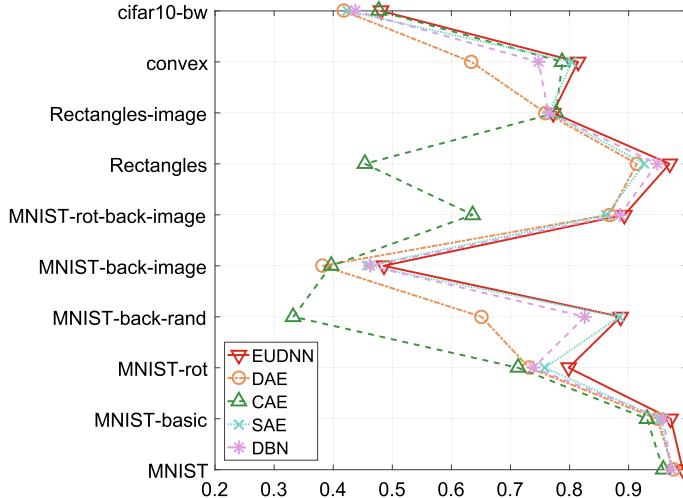


Fig. 3.5 Comparison of EUDNN against selected peer competitors using the configurations resulting EUDNN having the best CCR

As can be shown in Fig. 3.5, the majority of the peer competitors get worse CCRs than EUDNN does. Specifically, EUDNN demonstrates the best CCR upon MNIST-rot, MNIST, MNIST-rot-back-image, MNIST-back-image, CIFAR-10-BW, and Convex benchmarks, which is in line with the facts presented in Table 3.1. In addition, MNIST basic and MNIST back-rand benchmarks both point to EUDNN as having the best CCR as well. In addition to EUDNN with the initialized parameter values set by the evolutionary approach, all the results shown in Fig. 3.5 were produced by the compared algorithms using identical architectural configurations and generally used parameter initializing techniques for the second stage. The first stage of EUDNN used an evolutionary scheme that greatly aided the learning of meaningful representations since that in a local search the success of the strategy is heavily dependent on its starting location.

3.5.3 Analysis on Fine-Tuning of EUDNN

Whether the local search approach used in the second stage improves the overall performance of EUDNN when compared to solely using evolutionary approaches in the first stage needs to be examined. To do this, the potential CCR acquired by EUDNN from Table 3.1 will be selected firstly, where the outcomes of EUDNN are achieved collectively by the evolutionary approach used in the first stage and the local search strategy used in the second stage. Then choosing the results acquired without using the local search approach, i.e., the results obtained by EUDNN during the

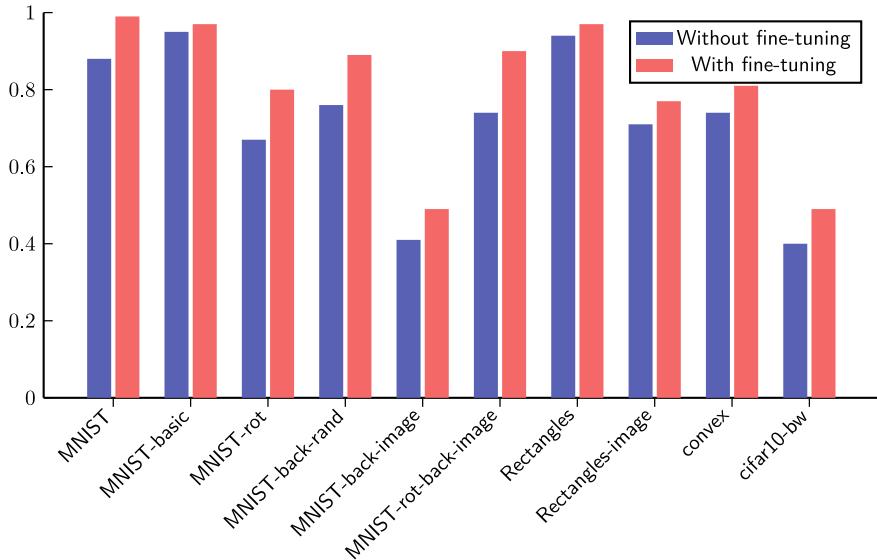


Fig. 3.6 CCR comparisons when the fine-tuning is used (ref) or not (blue) in EUDNN

first stage. Finally, for quantitative comparisons, these results are shown in Fig. 3.6, where the vertical axis represents the CCR and the horizontal axis represents the benchmarks.

The performance of the second stage of EUDNN has been improved over all of the benchmarks studied, as shown in Fig. 3.6, when comparing to the algorithm which only uses the first stage. The CCR has improved by 12.83% on MNIST, and around 20% on MNIST-back-rand, MNIST-rot, MNIST-rot-back-image, MNIST-back-image, and CIFAR-10-BW. In conclusion, the results of these experiments show that the second-stage local search approach improves the performance of EUDNN significantly, allowing the learned representations to be more meaningful and satisfying the design purpose.

3.5.4 Representation Visualizations

Several quantitative studies have been presented above to demonstrate how well EUDNN performs when it comes to generating meaningful representations based on unsupervised deep NN algorithms. Here, an experiment of a qualitative nature is provided for the purpose of properly understanding what the representations obtained via EUDNN using visualization techniques, which is a typical approach used by related research to intuitively analyze the learned representations [1–5].

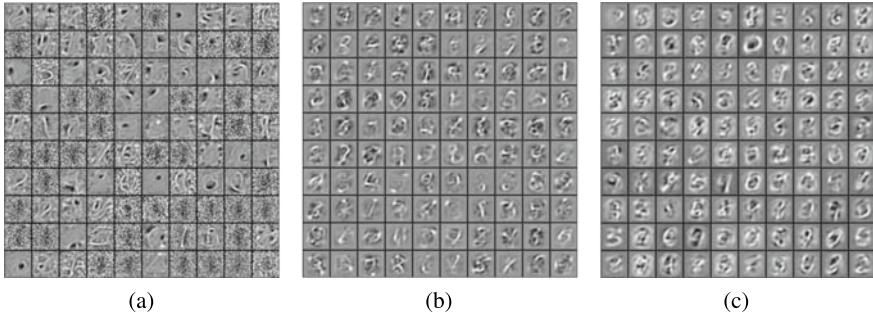


Fig. 3.7 Visualizations resulted from EUDNN using the activation maximization approach over the MNIST dataset at depths one to three (from left to right)

The activation maximization method [38] is used to depict the learned representations of EUDNN across the MNIST dataset for this purpose, and a number of 100 randomly chosen patch visualizations are shown in Fig. 3.7. Only representations with depths of one, two, and three are represented here because understanding the visualizations of representations gained at depths larger than one is challenging, and visualizations acquired at depths larger than three do not offer a frame of reference for making comparisons. Furthermore, with 10,000 trials and a constant learning rate of 0.1, the SGD is used to optimize the activation maximization. Figure 3.7a depicts the learned representations on depth one that are typically used to describe the visualization [38].

In Fig. 3.7a, It is obvious that certain strokes are learned across most patches, and a percentage of the representations is similar to the RBM, which can be regarded as evidence of the efficiency of EUDNN. This is due to the fact that representations of the MNIST dataset that are quite similar to one another have been reported in a variety of different literatures. Figure 3.7b, c show visualizations of representations at depths of two and three, respectively. Owing to the increase hierarchical nature of these representations, they are challenging to identify intuitively and interpret. However, by comparing the learned representations to the studies simulated in the literature [38], it may be inferred that EUDNN has learned the relevant representations. In conclusion, these visualizations provide a qualitative insight that EUDNN has learned the relevant representations efficiently.

3.6 Chapter Summary

In this chapter, we introduced and discussed how to develop a GA-based algorithm to automatically design the architectures of stacked AEs and DBNs. In addition, we have also justified the effectiveness of the developed algorithm through experiments.

In the next chapter, we will introduce how to use PSO for automated design and development of another type of unsupervised DNN, CAEs for image classification.

References

1. Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1), 1–127.
2. Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on machine learning* (pp. 1096–1103). ACM.
3. Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th international conference on machine learning* (pp. 833–840).
4. Sun, Y., Mao, H., Guo, Q., & Yi, Z. (2015a). Learning a good representation with unsymmetrical auto-encoder. *Neural computing and applications* (pp. 1–7).
5. Sun, Y., Mao, H., Sang, Y., & Yi, Z. (2015b). Explicit guiding auto-encoders for learning meaningful representation. *Neural computing and applications* (pp. 1–8).
6. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3), 1.
7. Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of 8th Annual Conference Cognitive Science Society* (pp. 823–831).
8. Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12, 2121–2159.
9. Tieleman, T., & Hinton, G. (2012). *Rmsprop*. COURSERA: Lecture.
10. Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. [arXiv:1212.5701](https://arxiv.org/abs/1212.5701).
11. Bergstra, J. S., Bardenet, R., Bengio, Y., Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems* (pp. 2546–2554).
12. Bergstra, J., Yamins, D., Cox, D. D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *ICML* (1), 28, 115–123.
13. Lerman, P. M. (1980). Fitting segmented regression models by grid search. *Applied statistics* (pp. 77–84).
14. Olshausen, B. A., & Field, D. J. (1997). Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision Research*, 37(23), 3311–3325.
15. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, 19, 153.
16. Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 99–127.
17. Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2), 131–162.
18. Whitley, D., Starkweather, T., & Bogart, C. (1990). Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel Computing*, 14(3), 347–361.
19. Whitley, L. D. (1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *ICGA*, 89, 116–123.
20. Montana, D. J., & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. In *International joint conference on artificial intelligence* (Vol. 89, pp. 762–767).
21. Christian, S. F., & Lebiere, C. (1990). The cascade-correlation learning architecture. In *Advances in neural information processing systems* 2. Citeseer.
22. Frean, M. (1990). The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2(2), 198–209.

23. Sietsma, J., & Dow, R. J. F. (1991). Creating artificial neural networks that generalize. *Neural Networks*, 4(1), 67–79.
24. Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9), 1423–1447.
25. Liu, J., Gong, M., Miao, Q., Wang, X., & Li, H. (2017a). Structure learning for deep neural networks based on multiobjective optimization. *IEEE Transactions on Neural Networks and Learning Systems*.
26. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).
27. Stanley, K. O. (2006). Exploiting regularity without development. In *Proceedings of the AAAI fall symposium on developmental systems* (p. 37). AAAI Press Menlo Park.
28. Yang, J., Frangi, A. F., Yang, J.-Y., Zhang, D., & Jin, Z. (2005). Kpca plus lda: a complete kernel fisher discriminant framework for feature extraction and recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(2), 230–244.
29. Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3), 273–297.
30. Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of ICML* (Vol. 30).
31. Zhao, Q., Zhang, D., & Lu, H. (2006). A direct evolutionary feature extraction algorithm for classifying high dimensional data. In *Proceedings of the National Conference on Artificial Intelligence*, (Vol. 21, pp. 561). AAAI Press, MIT Press, 1999.
32. Vapnik, V. (2010). *Statistical learning theory*. DBLP.
33. Vapnik, V. N. (1997). The nature of statistical learning theory. *IEEE Transactions on Neural Networks*, 8(6), 1564–1564.
34. Hinton, G. E., & Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786), 504–507.
35. Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade* (pp. 599–619). Springer.
36. Steel, R. G. D., Dickey, D. A. J. H. et al. (1997). *Principles and procedures of statistics a biometrical approach*. Number 519.5 S813 1997. WCB/McGraw-Hill.
37. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
38. Erhan, D., Bengio, Y., Courville, A., Vincent, P. (2009). Visualizing higher-layer features of a deep network. *University of Montreal*, 1341.

Chapter 4

Architecture Design for Convolutional Auto-Encoders



4.1 Introduction

Although the CAE and its variations have proven benefits in a variety of applications, one key restriction is that their stacked architectures are incompatible with those of state-of-the-art CNNs. The amount of convolutional and pooling layers in the stacked CNN is the same because each CAE contains a pooling layer and a convolutional layer in the encoder. State-of-the-art CNNs, on the other hand, have different amounts of convolutional and pooling layers. The constraint on the number of convolutional and pooling layers contained in CAEs ought to be lifted because the architecture of CNN is one of the important ingredients contributing to the final performance. However, because of the non-differentiable and non-convex properties in practice, it is intractable to determine optimal numbers for convolutional layers and pooling layers, which is related to NAS.

Since the features of placing no condition, for example, convex or differentiable, on the problems that need to optimize, PSO is widely used to numerous applications in real world [1–3], containing the architecture design for shallow and median-scale NNs, such as [4–9]. In order to optimize NN architectures, these methods use an implicit technique to encode each NN connection and then use PSO or its variations to find the best solution. They cannot be used for CAEs and CNNs, or even DBNs and Stacked AEs, which belong to the deep learning algorithms with a large number of connection weights, making implementation and successful optimization in current existing architectural optimization techniques based on PSO [10] prohibitively expensive.

In particular, CAEs without limitations on the number of convolutional layers and pooling layers would be much desirable for constructing state-of-the-art CNNs, as previously suggested. However, until the architecture is determined, the absolute numbers of these levels are unclear. In addition, there are many different types of architectures involved, each with a distinct number of decision variables. In this regard, particles of varied lengths will be required when PSO is used for this problem.

Unfortunately, the canonical PSO does not include a mechanism for updating the velocity of particles of different lengths. The following four aims are specified to collectively address the aforementioned issues:

1. A flexible CAE (FCAE) that allows for numerous convolutional and pooling layers is designed. The FCAE has no requirements for the number of convolutional and pooling layers, and can stack to multiple types of CNNs.
2. A PSO-based architecture optimization algorithm, i.e., PSOAO, is designed for the FCAE. In PSOAO, an efficient encoding technique for encoding the FCAE architecture is offered, which include hundreds of thousands of parameters per particle, as well as a reliable mechanism for updating the velocity of particles with varied lengths.
3. The performance of FCAE is verified on MNIST, CIFAR-10, Caltech-101 and STL-10 dataset when its architecture is optimized using the PSOAO, and is compared with peer competitors in terms of classification accuracy.
4. The efficiency of the velocity update approach through quantitative comparisons with its competitors is investigated. In addition, the evolution efficiency of PSOAO is also conducted.

The rest of this chapter is laid out as follows. Section 4.2 shows the motivation of FCAE. In Sect. 4.3, the details of PSOAO are presented. After that, Sects. 4.4 and 4.5 document the design of experiments and the analysis of experimental results, respectively.

4.2 Motivation of FCAE

Figure 2.4 of Chap. 2 shows a CAE which is made up a convolutional layer, a pooling layer, and a deconvolutional layer. The encoder is the transformation that goes via the convolutional layer and subsequently the pooling layer. The decoder is the transition that occurs through the deconvolutional layer. During the creation of a CNN employing CAEs, encoders from several CAEs are stacked according to their training orders, and the input data for the current encoder is the output data from the preceding one. However, this approach cannot be used to form either the most advanced CNNs or the typical deep CNNs.

Figure 4.1a, b demonstrate a CNN architectures stacked by CAEs and a state-of-the-art CNN called VGGNet [11], respectively. CAEs are clearly incapable of stacking into VGGNet based on these two examples. The reason is that CAEs stack CNNs using the same construction components, namely a two-layer network with a convolutional layer and a pooling layer. As a result, a collection of these building blocks is used to construct the stacked CNN, which has the number of convolutional layers equaling to that of pooling layers. In CNNs that are stacked by CAEs, a pooling

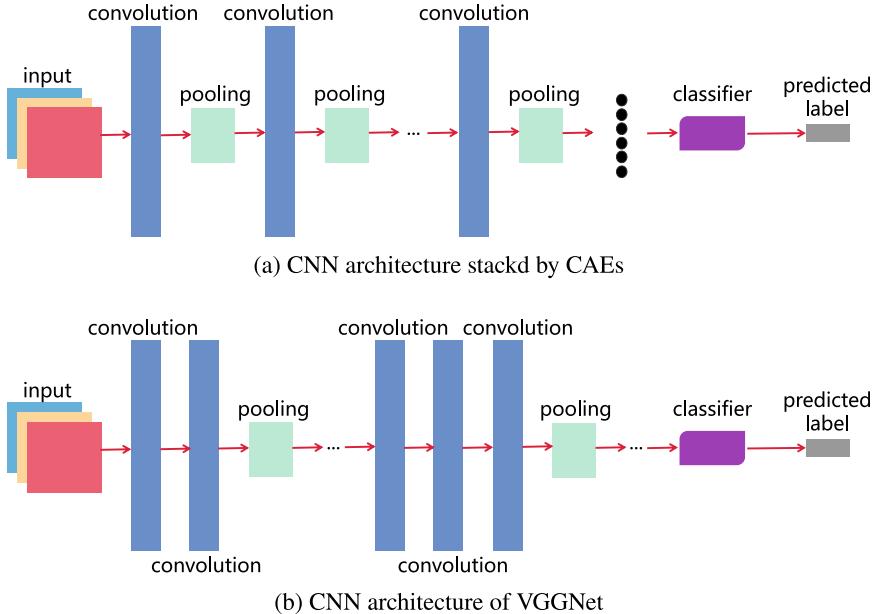


Fig. 4.1 Illustrations of the architectures stacked by CAEs and VGGNet

layer is required to come after a convolutional layer in the sequence of layers. As shown in Fig. 4.1b, there are identical numbers of layers for convolutional operation and pooling operation, in addition to this, a convolutional layer does not always follow by the pooling layer.

According to the preceding explanation, the limitation is sourced from the architecture of CAE, which consists of a convolutional layer and a pooling layer following it. As a result, there is a concern that there needs to be an update to the architecture of CAE. This serves as an inspiration for the design of the FCAE, in which layers of pooling and convolution with non-identical numbers are allowed in the encoder component of an FCAE, and a pooling layer can be applied after a series of convolutional layers. In addition, it is also not straightforward employing PSO to design architecture in FCAE. Specifically, the particle x_i , $g\ Best$, and $p\ Best_i$ must all have the same or fixed length for velocity updating to work. Particles illustrate the probable ideal architecture of FCAE when PSO is employed for architecture optimization. Because the ideal FCAE architecture for completing the task at hand is uncertain, particles of varying lengths will arise. As a result, a new velocity update method must be devised in this regard.

4.3 Algorithm Details

The PSOAO algorithm for FCAE will be described in detail in this section, which include the encoding technique mainly considering the FCAE representation, particle initialization, fitness evaluation, and velocity calculation and position update mechanisms.

4.3.1 Algorithm Overview

Algorithm 1: The Framework of PSOAO

```

1  $\mathbf{x} \leftarrow$  Initialize particles with encoding strategy;
2 for each step in the predefined maximal generation do
3   Evaluate fitness value of each particle in  $\mathbf{x}$ ;
4   Update global best particle  $g\ Best$  and local best particle  $p\ Best_i$ ;
5   Calculating velocity of each particle;
6   Updating position of each particle;
7 end
8 Return global best particle  $g\ Best$  to perform deep training.

```

The framework of PSOAO is outlined in Algorithm 8. To begin, particles are initialized at random using the encoding approach (line 1). The particles then begin to evolve until the number of generations surpasses the predetermined number (lines 2–7). At last, the $g\ Best$ particle is chosen to conduct deep training for final performance to solve the tasks at hand (line 8). Please note that it is assumed the best performance of a NN model is reached by training T^* epochs, and the deep training means the model has gone through T epochs of training, where T is not less than T^* .

During evolution, the first is to evaluate the fitness for each particle (line 3), and then the $p\ Best_i$ and $g\ Best$ are modified according to the fitness (line 4). Following that, the velocity and position of each particle (line 5–6) are computed and updated for the following generation of evolution. The most import components of PSOAO are discussed in the following.

4.3.2 Encoding Strategy

In Definition 1, the concept of FCAE is described by generalizing the building blocks in all CNNs for the sake of development. When both the number of convolutional layers and the number of pooling layers are set to one, the CAE is obviously a particular type of FCAE.

Definition 1 One encoder and one decoder make up a flexible FCAE, where the encoder is made up of different number of convolutional and pooling layers, and the decoder is the inverse of the encoder. \square

PSOAO suggests an approach for encoding one potential FCAE architecture into a single particle using variable-length particles. Due to the fact that the decoder component of an FCAE is the inverse form of the encoder, for reducing the computing complexity, the variable-length encoding approach only encodes the encoder part in the particle. The number of convolutional layers and pooling layers in each particle varies. All the PSOAO encoded information for FCAE can be summarized as follows: the encoded information of convolutional layer, which contains the width and height of filter, width and height of stride, type of convolution, number of feature maps, and the coefficient of l_2 ; the encoded information of pooling layer, which contains the width and height of filter, width and height of stride, and pooling type. This encoded information is based on the convolution and pooling operation introduced in Sect. 4.2, where the weight decay regularization term l_2 is used to avoid the overfitting problem [12]. In principal, only the convolutional layers are affected by this term of regularization since only the convolutional layers have weight parameters. Furthermore, rather than encoding the convolutional layer type, the encoding method will default to using the SAME type, since the output size does not differ from the input size when using the SAME convolutional layers, making automatic architecture design effective and efficiency. This parameter does not need to be encoded because, as explained in Sect. 4.2, CAE favors the MAX one.

A encoding approach with variable length is created to represent FCAEs with diverse architectures by particles in the PSOAO algorithm. The main reason for this is that before optimization, the ideal architecture is unknown, and the encoding approach with fixed length that is frequently used to impose limitations on architectures is inapplicable in this case. The maximum length should be defined in advance if the typical encoding approach with fixed length is used. The maximum length, on the other hand, is difficult to determine and must be fine-tuned for optimal performance. For the optimal architecture of FCAE to tackle complicated issues, it is inefficient using a small number to denote the maximum length. In contrast, a large number consumes a lot of needless computation and degrades efficiency within the same predetermined number of evolution generation. In addition, each particle has two sorts of layers, making the fixed-length encoding technique more complex to implement. With the suggested variable-length encoding approach, all of the information of an ideal FCAE architecture may be dynamically represented during the search process for exploitation and exploration.

4.3.3 Particle Initialization

The process for particle initialization with the specified population size and the maximum numbers of convolutional and pooling layers is shown in Algorithm 2.

Algorithm 2: Particle Initialization

Input: The maximum number of convolutional and pooling layers N_c and N_p , the size of population N

Output: The population \mathbf{x}_0 initialized by this algorithm

```

1  $\mathbf{x} \leftarrow \emptyset;$ 
2 while  $|\mathbf{x}| \leq N$  do
3    $conv\_list \leftarrow \emptyset;$ 
4    $n_c \leftarrow$  An integer which is uniformly generated between  $[1, N_c]$ ;
5   while  $conv\_list \leq n_c$  do
6      $| conv\_unit \leftarrow$  Use random setting to initialize a convolutional layer;
7      $conv\_list \leftarrow conv\_list \cup conv\_unit;$ 
8   end
9    $pool\_list \leftarrow \emptyset;$ 
10   $n_p \leftarrow$  An integer which is uniformly generated between  $[1, N_p]$ ;
11  while  $|pool\_list| \leq n_p$  do
12     $| pool\_unit \leftarrow$  Use random setting to initialize a pooling layer;
13     $pool\_list \leftarrow pool\_list \cup pool\_unit;$ 
14  end
15  generate a particle  $x$  using  $conv\_list$  and  $pool\_list$ ;
16   $\mathbf{x} \leftarrow \mathbf{x} \cup x;$ 
17 end
18 Return  $\mathbf{x}.$ 

```

Specifically, lines 3–8 indicate the convolutional layer initialization, whereas lines 9–14 show the pooling layer initialization, where the random settings correspond to the configurations of the information that is encoded in these two types of layers. Due to the fact that the decoder component of FCAE can be directly derived from the encoder component, in order to reduce computing complexity, the PSOAO algorithm just comprises the encoder component in each particle as highlighted above.

4.3.4 Fitness Evaluation

The particle fitness evaluation in PSOAO is shown in Algorithm 6. As mentioned above, the target function for training CAE is the reconstruction error plus the loss of the regularization component. However, the numbers and values of weights have a significant impact on the loss of the regularization term employed in FCAE (i.e., the l_2 loss), since different designs have varied weight numbers and weight values. The reconstruction error is used as the fitness instead of the l_2 loss to see if the sole thing being reflected by the quality of particle. Assume the batch training data is $\{d_1, d_2, \dots, d_n\}$, where $d_i^{jk} \in R^{w \times h}$ denotes the pixel value at (j, k) of the i -th image in the batch training data, and the dimension of every image is $w \times h$, the FCAE weights are $\{w_1, w_2, \dots, w_m\}$, and the reconstructed data is $\{\hat{d}_1, \hat{d}_2, \dots, \hat{d}_n\}$, the term of $\sum_{i=1}^m w_i^2$ calculates the l_2 , while $\frac{1}{n} \sum_{k=1}^n \sum_{l=1}^w \sum_{m=1}^h (\hat{d}_k^{lm} - d_k^{lm})^2$ calculates the reconstruction error.

Algorithm 3: Fitness Evaluation

Input: The training set D_{train} , the training epoch number N_{train} , the population \mathbf{x} .

Output: The population \mathbf{x} with their evaluated fitness

- 1 Calculating the l_2 loss as well as the error of reconstruction for the FCAE that is encoded by each particle in \mathbf{x} , and train the weights with the given amount of epochs denoted by N_{train} ;
 - 2 Calculating the error of reconstruction for each batch D_{train} data, and use the mean error of reconstruction to indicate the fitness of the particle corresponded;
 - 3 **Return** \mathbf{x} .
-

In order to employ gradient-based techniques to train the weight parameters, the number of training epoch required by a deep learning system is usually in the size of $10^2\text{--}10^3$. In population-based algorithms, this significant computational issue is exacerbated. This amount is given at a relatively modest figure (e.g., 5 or 10) in the PSOAO algorithm to speed up the training. For example, One epoch of training on the CIFAR-10 dataset including 50,000 training images will take 2 minutes on a single GTX1080 GPU card. With a population size of 50 for 50 generations, it will take roughly a year to train each particle using 10^2 epochs, which is not suitable for normal academic study. The most common way for overcoming this obstacle is to use heavy computation resources, e.g., the LEIC algorithm introduced by Google in 2017, in which 250 processors are used for roughly 11 days utilizing GA on the CIFAR-10 dataset for architectural discovery. During the architecture search, there is no need for the final performance evaluation on every particle over a significant number of training epochs. Instead, it could be a feasible option that training particles with fewer epochs and selecting a particle with promising performance, then deep training it with adequate training epochs. This motivates that the fitness evaluation of particles is conducted using a modest number of training epochs in the PSOAO method. The $gBest$ and $pBest_i$ are chosen based on the evaluated fitness to steer the search to the best result. While the evolution is finished, the $gBest$ is chosen and a one-time deep training process is undertaken to achieve the best results. Using this strategy, PSOAO can be significantly speed up while using fewer computational resources while maintaining its promising performance.

4.3.5 Velocity Calculation and Position Update

Because the particles in PSOAO have varying lengths, Eq. (1.1) cannot be applied directly. A approach called “ x -reference” to update the velocity was devised in order to solve this problem. The lengths of $gBest$ and $pBest_i$ in x -reference correspond to the current particle length x , i.e., $gBest$ and $pBest_i$ maintain the same length as x . The current particle x has the same length as the $pBest_i$, since the $pBest_i$ is picked from the memory of each particle, while the $gBest$ is taken from all particles, and the x -reference is solely used to the global search component in Eq. (1.1). The details of the x -reference approach are shown in Algorithm 4.

Algorithm 4: The x -reference Approach for Velocity Calculation

Input: Current particle x , the global best particle $gBest$, the constant of acceleration c_1
Output: The velocity updating for the global search component.

```

1  $r_1 \leftarrow$  sampling a number at random from  $[0, 1]$ ;
2  $cg \leftarrow$  The convolutional layers extracted from  $gBest$ ;
3  $cx \leftarrow$  The convolutional layers extracted from  $x$ ;
4  $pos\_c \leftarrow \emptyset$ ;
5 if  $|cg| < |cx|$  then
6    $c \leftarrow$  Using encoded information of 0 to Initialize  $|cx| - |cg|$  convolutional layers;
7    $cg \leftarrow$  Padding  $c$  to the end of  $cg$ ;
8 else
9    $c \leftarrow$  Truncating the final  $|cg| - |cx|$  convolutional layers from  $cg$ ;
10 end
11 for  $i = 1$  to  $|cx|$  do
12    $p_{x_i}, p_{cg_i} \leftarrow$  From  $cx$  and  $cg$ , extracting the  $i$ -th convolutional layer position
13    $pos\_c \leftarrow pos\_c \cup c_1 \cdot r_1 \cdot (p_{cg_i} - p_{x_i})$ ;
14 end
15  $pos\_p \leftarrow$  Analogizing the operations in lines 2-14 on the  $x$  and  $gBest$  pooling layer;
16 Return  $pos\_c \cup pos\_p$ .
```

During the global search phase of the velocity updating in Eq. (1.1), the x -reference approach is used twice in the same way. The first is on the convolutional layer (lines 2–14) of $gBest$ and x , while the second is on the pooling layer (line 15). New convolutional will be padded to the end of cg and initialized with zero if the number of convolutional layers cg from $gBest$ is less than the number of x . Otherwise, additional convolutional layers in the end of cg are truncated. After Algorithm 4 calculates the global search, the inertia and local search parts of Eq. (1.1) are calculated as usual, and Eq. (1.2) calculates the complete velocity and updates the particle position.

Figure 4.2 shows an example of x -reference velocity updating approach for a better understanding. In particular, Fig. 4.2a shows the $gBest$ and x variables that are utilized to perform the velocity updating for the global search phase. The pooling and convolutional layers from $gBest$ and x are collected in Fig. 4.2b. Because the lengths of convolutional layers and pooling layers from x are 2 and 4, respectively, and those lengths from $gBest$ are 3 and 2, the tail of the pooling layer component from $gBest$ is padded with the other two pooling layers, while the final convolutional layer in the convolutional layer part of $gBest$ is truncated. The padded pooling layers, in particular, are constructed with encoded information values of zeros. The update between the convolutional layer and the pooling layer from $gBest$ and x is shown in Fig. 4.2c. The outcomes of this update are shown in Fig. 4.2d.

The purpose of the intended x -reference velocity updating approach, regardless of whether padding or truncating is used, is to maintain the length of convolutional and pooling layers in $gBest$ same as in x . The following is a description of the mechanism that underpins this design. In PSOAO, there is a population of particles of various lengths who has the same purpose of finding the best FCAE architectures

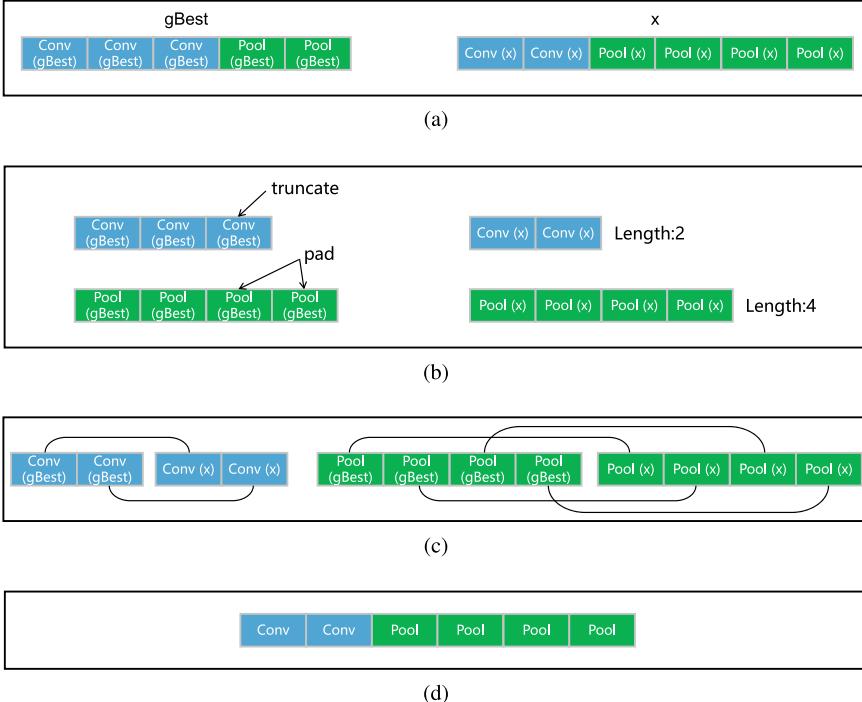


Fig. 4.2 An illustration of how the x -reference velocity updating approach works

for performing image classification tasks. When every particle follows the length of $gBest$, i.e., the $gBest$ -reference velocity updating approach, all these particles in the second generation will have the same length as $gBest$. The current particle x , $pBest_i$, and $gBest$ could all be the same length from the third generation since the $pBest_i$ is picked from memory of each particle. As a result, all particles take part in the optimization with one FCAE depth and merely change the information that is encoded. In fact, the fluctuation in length for $gBest$ can be interpreted as an search behavior of exploration, whereas the exploitation search behavior for encoded information can be noticed. The length of $gBest$ will be constant until it finishes when all particles are with the same length. When using the $gBest$ -reference velocity updating approach, the search ability of exploration is lost. Furthermore, maintaining the length of x at $gBest$ might be interpreted as a loss of diversity, in population-based algorithms, this might easily lead to premature convergence. A poor performance will come from both the premature phenomenon and the loss of exploration search.

4.3.6 Deep Training on Global Best Particle

When PSOAO has completed its evolution, $gBest$, the global best particle, is chosen for deep training. Particles are only trained for a few epochs, as described in Sect. 4.3.4, which cannot achieve optimal performance to meet the needs of real-world applications. Deep training is required to address this issue. The deep training method is typically identical to the fitness evaluation in Sect. 4.3.4, with the exception of a larger epoch number, such as 100 or 200.

4.4 Experimental Design

This section details the parameter settings and peer competitors for the trials assessing the performance of the FCAE architectures optimized by PSOAO.

4.4.1 Peer Competitors

For the comparisons on the investigated benchmark datasets for image classification, those peer competitors mentioned in Sect. 4.1 for FCAE are used. In particular, the state-of-the-art CDAE [13], the CAE [14], and the CRBM [15] are the three peer competitors. In addition, two frequently used AE variants are also included in order to provide a full comparison, they are the Sparse AE [16] and the DAE [17].

4.4.2 Parameter Settings

On the chosen benchmark datasets, the peer competitors SCAE, SCRBM, stacked sparse AE, and SDAE were recently explored, and their architectures are manually tuned with extensive domain knowledge [13]. Because their classification results are taken straight from the source papers, no parameter settings are required. Furthermore, on the chosen benchmark datasets, the state-of-the-art SCDAE provides classification results using only one or two building blocks. In addition, experiments on SFCAE using less than three building blocks are conducted to ensure a fair comparison. To maintain consistency with its peer competitors, the SFCAE is evaluated using the selected benchmark datasets in their raw form, without any preprocessing or data augmentation being performed beforehand.

PSO related parameters are given in PSOAO based to their conventions [18], i.e., the initial velocity is set to 0, both the acceleration constants c_1 and c_2 are set to 1.496172, and the inertia weight w is set to 0.72984. MNIST and CIFAR-10 training sets, as well as unlabeled STL-10 data, are naturally employed in their fitness

evaluations. Since every category of the Caltech-101 dataset has a non-identical and small number of images, a selection of 30 images at random is made from each category for use in fitness evaluations and also as the training set in accordance with the recommendations included in [13]. Unaffordable computational cost will be resulted by the incorrect settings of convolutional and pooling operations, so setting the kernel size between two and five with the same width and height, the maximum number of pooling layers to one, the number of feature maps between 20 and 100, and the number of convolutional layers to five in the exploration of each particle. Note that the studies only look at square pooling kernels and convolutional filters, according to the state-of-the-art CNN conventions [19, 20]. Furthermore, the coefficient of l_2 term is set between 0.0001 and 0.01, in practice, this is a common range for NNs training.

FCAE is used for deep training, with the PSOAO specified architecture and the initialized weights utilizing the widely known Xavier approach [21], with a 50% Dropout [22] and a fully connected layer consisting of 512 units following the conventions of the deep learning community. For investigating the results of classification, the corresponding test set to the trained model are fed, using the Adam [23] optimizer with the default settings, the commonly employed ReLU [24] activation function, and the BN [25] technique for speeding up the training. Tests on each benchmark dataset with the trained model are also separately repeated five times to ensure consistency in the findings to be compared. Analyzing the Caltech-101 dataset according on its convention [15], i.e., analyzing the accuracy of classification for each category of images and providing the mean and standard deviations for the entire dataset, due to the extreme imbalance of training data.

TensorFlow [26] is used to implement PSOAO, and every single clone of the source code is executed on a single GTX1080 GPU card. Based on the statistics, PSOAO consumed 81.5, 118, 230, and 22.5 h on CIFAR-10, MNIST, STL-10, and Caltech-101 dataset, respectively.

4.5 Experimental Results and Analysis

4.5.1 Overview Performance

On the benchmark datasets, Table 4.1 displays the classification accuracy values of FCAE in terms of mean and standard deviations, their architecture is optimized by PSOAO and its peer competitors. Because the standard deviations of SCRBM on CIFAR-10, SDAE on MNIST, and SCAE on both CIFAR-10 and MNIST are not available in the literature, only their mean classification accuracy values are presented. The sources of the related mean classification accuracy values are referenced in Table 4.1, and the results of the best mean classification are with bold. The SFCAE with one building block is referred to as SFCAE-1, while the SFCAE with two build-

Table 4.1 The classification accuracy values achieved by FCAE in comparison to that of its peer competitors on the investigated benchmark datasets

Algorithm	CIFAR-10	MNIST	STL-10	Caltech-101
Stacked Sparse AE	74.0 (0.9)	96.29 (0.12)	55.5 (1.2)	66.2 (1.2)
SDAE	70.1 (1.0)	99.06 [17]	53.5 (1.5)	59.5 (0.3)
SCAE	78.2 [14]	99.29 [14]	40.0 (3.1)	58.0 (2.0)
SCRBM	78.9 [27]	99.18 [15]	43.5 (2.3)	65.4 (0.5) [15]
SCDAE-1	75.0 (1.2)	99.17 (0.10)	56.6 (0.8)	71.5 (1.6)
SCDAE-2	80.4 (1.1)	99.38 (0.05)	60.5 (0.9)	78.6 (1.2)
SFCAE-1	78.9 (0.3)	99.30 (0.03)	61.2 (1.2)	79.8 (0.0)
SFCAE-2	83.5 (0.5)	99.51 (0.09)	56.8 (0.2)	79.6 (0.0)

ing blocks is referred to as SFCAE-2, and the terms SCDAE-1 and SCDAE-2 are with the same meaning.

On all benchmark datasets, FCAE beats both traditional CAEs (i.e., SCRBM and SCAE) and traditional AEs (i.e., SDAE and Stacked sparse AE), as demonstrated in Table 4.1. Furthermore, on these benchmark datasets, FCAE outperforms state-of-the-art CAEs (i.e., the SCDAE-1 and SCDAE-2). In addition, SFCAE-2 achieves the highest results on MNIST and CIFAR-10, while SFCAE-1 achieves the best results on STL-10 and Caltech-101. Note that SFCAE-2 outperforms SFCAE-1 on Caltech-101 and STL-10, this is owing to the fact that these two benchmark datasets have a smaller number of training samples, and deeper architectures are more susceptible to overfitting. Because MNIST and CIFAR-10 have a far larger number of training samples (60,000 in MNIST and 50,000 in CIFAR-10), a more complex design naturally leads to higher classification accuracy. In conclusion, when the architecture of FCAE is improved using PSOAO, On each of the four image classification test datasets, FCAE achieves results that are superior to those of its peer competitors.

4.5.2 Evolution Trajectory of PSOAO

Figure 4.3 presents the evolution trajectories of PSOAO on the investigated benchmark datasets during the training phases, where the vertical axis represents the fitness values of the global best particle $gBest$, and the horizontal axis represents the number of generations that have passed since the beginning of the evolution.

PSOAO has successfully converged under the required maximum generation number, as shown in Fig. 4.3a, b, c, d. It has converged on all benchmark datasets for both SFCAE-1 and SFCAE-2 since about the 15th generation, and on the Caltech-101 and CIFAR-10 datasets for SFCAE-2 since about the 5th generation. On the STL-10

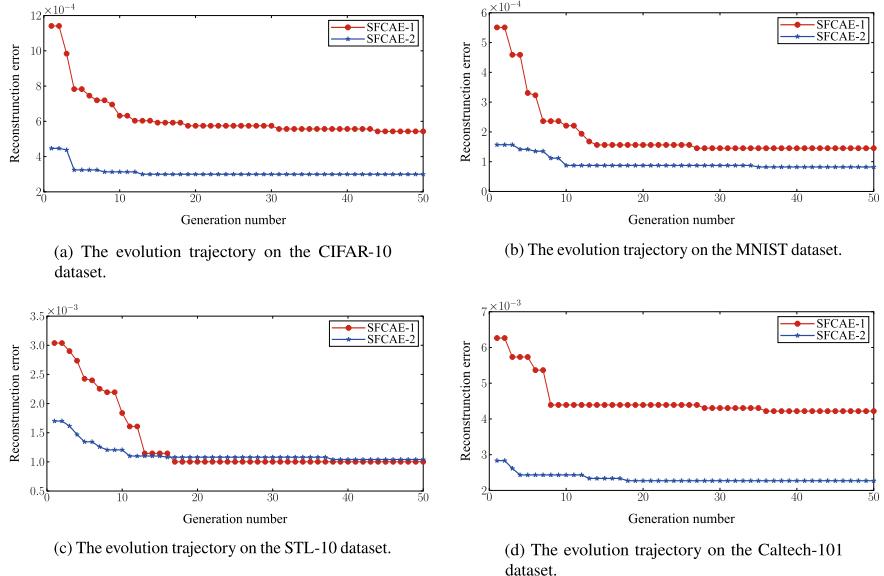


Fig. 4.3 Evolution trajectories of PSOAO in automatically discovering the architectures of FCAE on the chosen benchmark datasets

dataset (shown in Fig. 4.3c), the reconstruction error of SFCAE-2 is less than that of SFCAE-1, which is due to their varied input data.

4.5.3 Performance on Different Numbers of Training Examples

The classification performance of FACE, whose architecture is optimized by PSOAO, is also investigated on varying the number of training samples. SCRBM [15], ULIFH [28], SSE [29], and SCAE [14] are peer rivals on the MNIST dataset, while SCAE [14], K-means (4k feat) [30], and Mean-cov-RBM [27] are peer competitors on the CIFAR-10 dataset. The literature has published their matching experimental results, which are frequently used by comparisons between various types of CAEs, which is why these benchmark datasets and peer rivals were chosen.

Tables 4.2 and 4.3 exhibit the experimental results of FCAE-2 on varying amounts of training samples from the MNIST and CIFAR-10 benchmark datasets, with the architecture confirmed in Sect. 4.5.1. The symbol “–” indicates that no result has been recorded in the relevant literature. The most accurate classification is marked in bold.

Tables 4.2 and 4.3 show that SFCAE-2 outperforms all peer competitors on these two datasets. SFCAE-2 outperforms the foundational work of CAE (SCAE) with a

Table 4.2 The accuracy of classification achieved by FCAE-2 in comparison to that of its peer rivals on varying amounts of training samples taken from MNIST

# Samples	1K	2K	3K	5K	10K	60K
SCRBM	97.38	97.87	98.09	98.41	–	99.18
ULIFH	96.79	97.47	–	98.49	–	99.36
Stacked sparse AE	97.27	–	98.17	–	–	98.50
SCAE	92.77	–	–	–	98.12	99.29
SFCAE	97.49	98.45	98.80	99.09	99.16	99.51

Table 4.3 The accuracy of classification achieved by FCAE-2 in comparison to that of its peer rivals on varying amounts of training samples taken from CIFAR-10

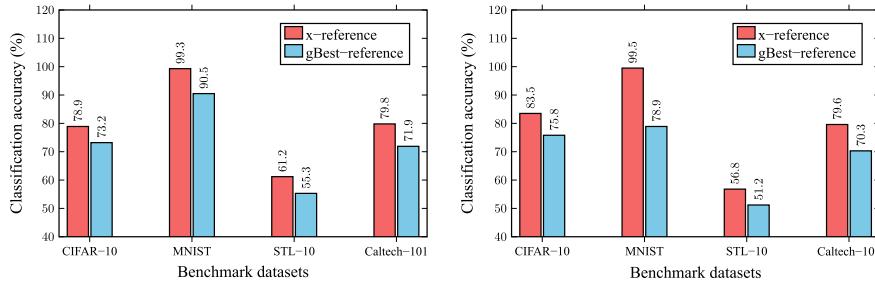
# Samples	1K	10K	50K
SCAE	47.70	65.65	78.20
Mean-cov-RBM	–	–	71.00
K-means (4k feat)	–	–	79.60
SFCAE	53.79	73.96	83.47

improvement of 4.72% classification accuracy on MNIST and an improvement of 6.09% classification accuracy on CIFAR-10, although with a significantly smaller number of training samples (1K). These findings demonstrate the scalability of SFCAE-2 in dealing with various quantities of training samples.

4.5.4 Investigation on x -Reference Velocity Calculation

To further evaluate the superiority of x -reference velocity updating approach, substituting it with the $g\ Best$ -reference velocity updating method in the PSOAO algorithm and compare the results on the specified benchmark datasets. To do so, setting the length of $p\ Best_i$ equal to the length of $g\ Best$. Zeros are padded if the length of the convolutional layers in $p\ Best_i$ is less than that of $g\ Best$. Otherwise, the appropriate segments of $p\ Best_i$ will be truncated. This approach is used in pooling layers of $p\ Best_i$ as well as these two types of layers in x . The position of each particle is then updated using Eqs. (1.1) and (1.2). The experimental results are shown in Fig. 4.4, with FSCAE-1 results shown in Fig. 4.4a and FSCAE-2 results in Fig. 4.4b.

As shown in Fig. 4.4a, SFCAE-1 improves classification accuracy by 5.7, 8.8, 5.9, and 7.9% using the x -reference velocity updating approach on the CIFAR-10, MNIST, STL-10, and Caltech-101 benchmark datasets, respectively. The classification accuracy gains of 7.7, 10.6, 5.6, and 9.6% on these investigated benchmark datasets, as shown in Fig. 4.4b, demonstrate the promising performance of SFCAE-2.



(a) The classification accuracy of SFCAE-1 by truncating *gBest* and *x*.
 (b) The classification accuracy of SFCAE-2 by truncating *gBest* and *x*.

Fig. 4.4 Classification accuracy comparisons between the *x*-reference and *gBest*-reference velocity updating strategy in PSOAO

4.6 Chapter Summary

In this chapter, we discussed and introduced a method based on PSO to automatically design the optimal architectures of CAEs. In addition to the experiment conducted to reveal the overall performance of the designed algorithm, the evolution trajectory, performance with different numbers of training examples, as well as the investigation on the *x*-reference velocity calculation have also been conducted for the same goal. These experiments have collectively demonstrated the effectiveness of the algorithm. In the next chapter, we will introduce the automatic design for another type of AEs, i.e., the VAEs, which are often used for generative tasks.

References

1. Xue, B., Zhang, M., Browne, W.N. (2013). Particle swarm optimization for feature selection in classification: A multi-objective approach. *IEEE Transactions on Cybernetics*, 43(6), 1656–1671.
2. Mohammed, A. W., Zhang, M., & Johnston, M. (2009). Particle swarm optimization based adaboost for face detection. In *IEEE congress on evolutionary computation, CEC'09*. (pp. 2494–2501). IEEE.
3. Setayesh, M., Zhang, M., & Johnston, M. (2013). A novel particle swarm optimisation approach to detecting continuous, thin and smooth edges in noisy images. *Information Sciences*, 246, 28–51.
4. Jianbo, Y., Wang, S., & Xi, L. (2008). Evolving artificial neural networks using an improved pso and dpsos. *Neurocomputing*, 71(4), 1054–1060.
5. Settles, M., Rodebaugh, B., Soule, T. (2003). Comparison of genetic algorithm and particle swarm optimizer when evolving a recurrent neural network. In *Genetic and evolutionary computation-GECCO 2003* (pp. 200–200). Springer.
6. Da, Y., & Xiurun, G. (2005). An improved pso-based ann with simulated annealing technique. *Neurocomputing*, 63, 527–533.

7. Juang, C.-F. (2004). A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEEE Transactions on Systems, Man, and Cybernetics. Part B, Cybernetics : A Publication of the IEEE Systems, Man, and Cybernetics Society*, 34(2), 997–1006. <https://doi.org/10.1109/TSMCB.2003.818557>.
8. Lu, W. Z., Fan, H. Y., & Lo, S. M. (2003). Application of evolutionary neural network method in predicting pollutant levels in downtown area of hong kong. *Neurocomputing*, 51, 387–400.
9. Salerno, J. (1997). Using the particle swarm optimization technique to train a recurrent neural model. In *International Conference on Proceedings of the 1997 on Tools with Artificial Intelligence* (pp. 45–49). IEEE.
10. Omidvar, M. N., Li, X., Mei, Y., & Yao, X. (2014). Cooperative co-evolution with differential grouping for large scale optimization. *IEEE Transactions on Evolutionary Computation*, 18(3), 378–393.
11. Simonyan, K., Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
12. Krogh, A., & Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems* (pp. 950–957).
13. Bo, D., Xiong, W., Jia, W., Zhang, L., Zhang, L., & Tao, D. (2017). Stacked convolutional denoising auto-encoders for feature representation. *IEEE Transactions on Cybernetics*, 47(4), 1017–1027.
14. Masci, J., Meier, U., Cireşan, D., & Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning - ICANN*, 2011, 52–59.
15. Lee, H., Grosse, R., Ranganath, R., & Ng, A. Y. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning* (pp. 609–616). ACM.
16. Shin, H.-C., Orton, M. R., Collins, D. J., Doran, S. J., & Leach, M. O. (2013). Stacked auto-encoders for unsupervised feature learning and multiple organ detection in a pilot study using 4d patient data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1930–1943.
17. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11, 3371–3408.
18. Bratton, D., & Kennedy, J. (2007). Defining a standard for particle swarm optimization. In *Proceeding of 2007 IEEE Swarm Intelligence Symposium* (pp. 120–127). IEEE.
19. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
20. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
21. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
22. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958. ISSN 1532-4435.
23. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
24. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323).
25. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, volume 37 of Proceedings of Machine Learning Research* (pp. 448–456). PMLR.

26. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. [arXiv:1603.04467](https://arxiv.org/abs/1603.04467).
27. Krizhevsky, A., Hinton, G. (2010). Convolutional deep belief networks on cifar-10. Unpublished Manuscript (p. 40).
28. Huang, F. J., Boureau, Y.-L., LeCun, Y., et al. (2007). Unsupervised learning of invariant feature hierarchies with applications to object recognition. In *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07.* (pp. 1–8). IEEE.
29. Weston, J., Ratle, F., Mobahi, H., Collobert, R. (2012). Deep learning via semi-supervised embedding. In *Neural networks: Tricks of the trade* (pp. 639–655). Springer.
30. Coates, A., Ng, A., & Lee, H. (2011). An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 215–223).

Chapter 5

Architecture Design for Variational Auto-Encoders



5.1 Introduction

Most VAEs were developed with symmetrical architecture in mind, which means that the encoder and decoder must have the same number of layers. However, when completing the step of unsupervised pre-training step, the decoder portion is deprecated and will never be employed when fine-tuning image classification problems. As a result, maintaining a symmetrical architecture is not nearly necessary [1]. However, new complications develop when the asymmetrical architecture is used, despite the fact that it is a quite appealing decision. This shows that if the asymmetrical architectures of CVAEs are planned to improve, one needs to evaluate the architectures of each partition on its own before making any changes to the architectures as a whole. In addition, for the correct forwarding of the asymmetrical CVAEs, it is essential to make certain that the decoder appropriately fits the resolution of the raw input data in the appropriate manner. This also adds another layer of complication to the manual design process for VAE designs.

In this chapter, the EvoVAE algorithm based on GA is introduced in order to successfully solve the architecture design of VAE that heavily depends on human expertise, and also disentangle the symmetrical requirement of VAE, i.e., the number of layers can be chosen by the each block of the VAEs. The following contributions are highlighted:

1. Develop a flexible architecture for VAEs that is capable of resolving the issues that are caused by the standard architectures for VAEs, which have a design that is symmetrical. The architectures of the VAEs are built out of these distinct components, which come in a total of four different varieties. The h -block, μ -block, σ -block, and t -block are the names that are given to these blocks in the order that they are listed. Any of the designs can have their settings changed as long as each block is flexible enough to accommodate the change. Because of this, the architectures are not always symmetrical, and the best architectures are often asymmetrical in character.

2. Create a brand new method of variable-length gene encoding that is able to accurately depict VAE designs of any level of depth. The method of fixed-length encoding is often employed in previous studies to discover the optima architectures. However, the depth of these architectures cannot be altered while they are being optimized. The depth is an important hyper-parameter that determines VAE performance. The ideal depth will be unknown until the optimal architectures are found. Incorrect values for the allowed length in the fixed-length encoding prevent the discovery of the best possible architecture. The encoding approach is able to perform an automatic examination to identify the optimal depth of the network.
3. In order to handle the variable-length encoding method, it is important to construct corresponding crossover and mutation processes. Although the method of variable-length encoding can automatically uncover the most effective VAE designs, the initial genetic operators are solely applicable for fixed-length one. In order to achieve this objective, efficient operators for crossover and mutation have been developed. This has enabled to considerably boost the capability of determining the appropriate depth of the VAEs, which is a necessary step toward achieving the design of EvoCAE.
4. Carry out the appropriate experiments in order to demonstrate that EvoCAE is superior than its contemporaries in the industry. By using SVHN, MNIST, CIFAR-10, and SVHN datasets, the performance of EvoVAE is compared to that of its nine peer competitors, including such CAE, AE, SAE, and DAE. The results of the experiments demonstrate that EvoVAE has the ability of automatically discovering VAE architectures and delivering improved performances.

5.2 Algorithm Details

EvoVAE algorithm is described in detail in this section. In particular, Sect. 5.2.1 provides an overview of EvoVAE, and Sects. 5.2.2–5.2.6 introduce the primary components of EvoVAE .

5.2.1 Algorithm Overview

The framework of EvoVAE is shown in Algorithm 1. Firstly, the gene encoding strategy is used to initialize the population P_0 (line 1). Fitness evaluation method is used to evaluate the individuals in the population P_0 in order to determine their fitness values (line 2). Secondly, the program will initiate the process of evolution from one generation to the next, generating individuals with enhanced performance in accordance with the design, and will continue doing so until the number of generations reaches a maximum value T (lines 3–8) that was previously specified. Finally, the individual who has demonstrated the highest performance is chosen for the deep

training in the final round (line 9). This is because of the limited number of epochs available for acceleration [2] and individual training during the evolutionary process is frequently insufficient, which has also been employed for the design of other algorithms in this book.

There are five steps in the evolutionary process (lines 3–8). To create the mating pool from P_{t-1} (line 4), the binary tournament selection [3] is used. The next step is for genetic operators (line 5) to generate a population of offspring, which is then evaluated for its fitness value (line 6). Last but not least, for the purpose of environmental selection, the existing population comes together with the generated offspring, and the individuals that perform best survive into the next generation serving as parent solutions (line 7).

In the following, the major components of EvoVAE will be discussed in detail.

Algorithm 1: Framework of EvoVAE

Input: The maximal generation number T

Output: The best individual p_{best}

```

1  $P_0 \leftarrow$  Initialize population using the gene encoding technique;
2 Perform fitness evaluations on the individuals residing in  $P_0$  using the approach designed for
   fitness evaluations. ;
3 for  $t = 1$  to  $T$  do
4    $C_t \leftarrow$  Utilize the binary tournament selection to select parent solutions;
5    $O_t \leftarrow$  Utilize genetic operators to generate the offspring from  $C_t$ ;
6   Evaluate the fitness values of individuals in  $O_t$ ;
7    $P_t \leftarrow$  Utilize environmental selection strategy to perform environmental selection from
       $P_{t-1} \cup O_t$ ;
8 end
9  $p_{best} \leftarrow$  Determine the most qualified member of  $P_T$  and decode it to the appropriate VAE
   for deep training.
10 return  $p_{best}$ 

```

5.2.2 Strategy of Gene Encoding

The initial step in employing GA is to describe the various answers to the issue that chromosomes will solve using the appropriate encoding approach. Each chromosome in the algorithm represents a possible VAE architecture.

In order to explain the details of representing VAEs having asymmetrical architectures, the whole VAEs need to break down into four separate components firstly. To begin, letting σ -block denote the variance branch and μ -block denote the mean branch, which are responsible for calculating the parameter variance and mean of distribution $p(z|x)$, respectively. The h -block is positioned in front of the μ -block and the σ -block so that the common feature extraction can be carried out as described in the reasoning. In addition, the t -block is a representation of the decoder, which

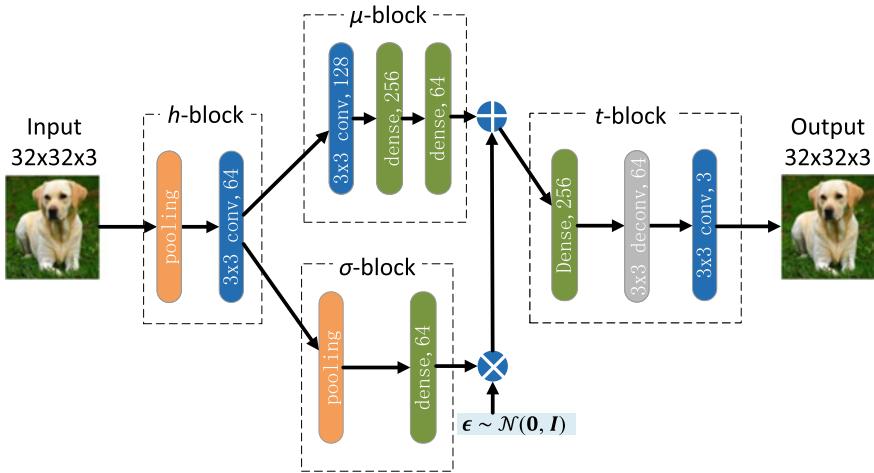


Fig. 5.1 The skeleton of asymmetrical CVAE architecture

is responsible for outputting the rebuilt data. The μ -block and the σ -block are both sequential networks made up of dense layers (i.e., the fully connected layers in other types of DNNs), convolutional layers, and pooling layers, all of which have adjustable positions and settings. The h -block is a sequential network made up of pooling layers and convolutional layers. The decoder, also known as the t -block, is another sequential network, but it uses the layers of deconvolutional operation rather than the layers of pooling operation. This is done so that the features can restore their original resolution.

A VAE with an asymmetrical convolution is illustrated as its skeleton in Fig. 5.1. The gray boxes with dots serve as an outline for each block. The rebuilt image is output by the VAE after it has been fed the RGB image with dimensions of 32×32 .

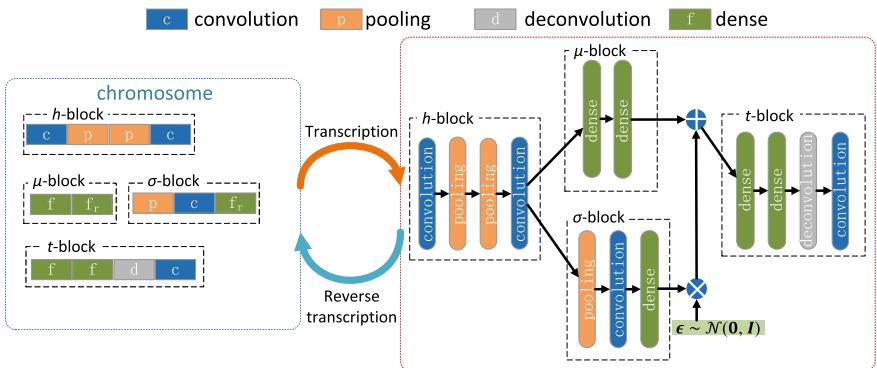


Fig. 5.2 An illustration example of the chromosomes designed in EvoCAE

Whenever the image is delivered via the h -block, it is changed into a tensor that is $64 \times 16 \times 16$. After that, the μ - and σ -blocks are forwarded, which transforms it into a 64-dimensional vector. It is recommended that the tensors having the dimension of 3 are flattened to a vector before being input into a dense layer. In the final step of the deconvolutional operations, the vector having the dimension of 64 is converted into a vector having the dimension of 256 in the t -block before being unflattened back into a $1 \times 16 \times 16$ tensor. This brings the total number of dimensions to 256. The last convolutional layer produces a tensor with dimensions of $3 \times 32 \times 32$. There is a clear difference in the total number of layers present in each block. When taking such an asymmetrical approach, one of the possibilities open to us is the development of optimal sub-architectures for the various blocks that make up the VAEs.

Algorithm 2: Strategy of Gene Encoding

Input: The VAE model m .
Output: The chromosome p .

- 1 Initialize h -block, μ -block, σ -block, and t -block all with \emptyset ;
- 2 Extract h -block, μ -block, σ -block and t -block from m as c_h , c_μ , c_σ and c_t , respectively;
- 3 **for** $l \in c_h$ **do**
- 4 $u \leftarrow$ Generate a unit u ;
- 5 **if** l is convolutional layer **then**
- 6 | Encode the convolutional layer l in unit u ;
- 7 **else if** l is pooling layer **then**
- 8 | Encode the pooling layer l in unit u ;
- 9 **else if** l is deconvolutional layer **then**
- 10 | Encode the deconvolutional layer l in unit u ;
- 11 **else if** l is dense layer **then**
- 12 | Encode the dense layer l in unit u ;
- 13 **end**
- 14 $h\text{-block} \leftarrow h\text{-block} \cup u$;
- 15 **end**
- 16 Use steps in Lines 3-15 to update μ -block, σ -block and t -block;
- 17 $p \leftarrow h\text{-block} \cup \mu\text{-block} \cup \sigma\text{-block} \cup t\text{-block}$;
- 18 **return** p ;

When coping with a specific challenge, the appropriate network depth is not known in advance, evolution requires a gene strategy of variable length. In order to accommodate the ideal depth of VAEs, the chromosome length can be adjusted throughout the process of optimization to fit a variable-length structure with various numbers and types of layers. Furthermore, each block of the VAEs is not be confined in order to maintain a symmetrical architecture, such as maintaining the widths of the output features and the number of layers identical for all blocks. To further comprehend the variable-length gene encoding technique and the asymmetrical manner, Fig. 5.3 illustrates three chromosomes of varied lengths. The figure clearly displays how the gene encoding strategy reflects changing length and asymmetry. The method known as fixed-length encoding, in contrast to the method known as variable-length encod-

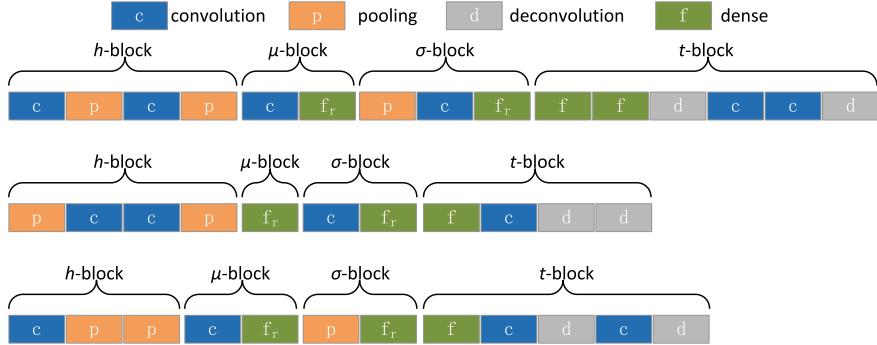


Fig. 5.3 An illustration to three varied-length chromosomes in EvoVAE

ing, is unable to solve this issue since it operates on the assumption that the depth of the entire network is always the same. In network architecture, depth is generally considered to be an important hyper-parameter. This is due to the fact that the depth has a considerable impact on the performance of DNN, as indicated by [4, 5]. The performance of architectural evolution would decrease if a fixed-length technique was used since this happens when the global optimum is a long way from the network depth that has been selected.

The information required to construct a VAE is contained within a chromosome, and the genetic required information for the network layer is kept within the individual units of chromosome. A genetic unit can be encoded as a particular network layer, and a particular network layer can also be decoded as a particular genetic unit in the chromosome. Both of these operations can take place simultaneously. Within the EvoVAE, the following different kinds of units are observed: dense, convolutional, deconvolutional, and pooling units. In specifically, the dense units are associated with the dense layers, and their primary purpose is to encode the number of features. The attribute values in the units are all whole numbers. The MAX unit and the AVERAGE unit in the pooling units are represented by using zero and one, respectively. It is important to note that the ground truth dimension of the manifold that is embedded in such a high-dimensional measurement space is not something that familiar with. Because of this, building a latent gene on the chromosome that is responsible for estimating the inherent dimension of the underlying generative manifold of the data that was input is possible. In summary, the encoded information for a dense unit and a deconvolution unit are the number of features and the number of feature maps, respectively, while that of a convolution unit includes the number of feature maps and the width and height of the kernel. In addition, the information encoded for a pooling unit are the type of pooling: AVERAGE or MAX.

The architecture of a VAE is encoded on a chromosome. The details of encoding a known VAE onto a chromosome is shown in Algorithm 2. To begin, separating each block of a particular VAE model m independently (line 2). A genetic unit u is then created for each layer l in a block, with the information encoded in the genetic unit

u (lines 3–16). A new chromosome p (line 17) is created by combining the updated h -block, μ -block, σ -block, and t -block. Only the encoder and decoder tails and heads of a valid model have dense layers, whereas only the decoder tails and heads contain deconvolutional layers. Figure 5.2 depicts a chromosome and a VAE model that it corresponds with; the chromosome is on the left and the decoded network model is on the right. All four blocks of the chromosome contain two convolutional units, two pooling units, two dense units, and one convolutional unit in the h -block of chromosome. The decoded network model has blocks for every chromosomal region. Note that the latent gene is represented by f_r in the μ -block and the σ -block because the latent gene functions as a dense layer.

Algorithm 3: Initialization of Population

Input: The size of population N , the maximal number of dense layers N_f , the maximal number of convolutional N_c , the maximal number of pooling layers N_p , the maximal number of deconvolutional layers N_d and the maximal dimensionality of the latent representation R .

Output: The initialized population P_0 .

- 1 $P_0 \leftarrow \emptyset;$
- 2 **for** $i = 1$ **to** N **do**
- 3 | Sample integers $n_{hc} \in [1..N_c]$ and $n_{hp} \in [1..N_p]$;
- 4 | h -block \leftarrow Merge the randomly generated n_{hc} convolutional units and n_{hp} pooling units;
- 5 | Sample integers $n_{\mu c} \in [0..N_c - n_{hc}]$, $n_{\mu p} \in [0..N_p - n_{hp}]$ and $n_{\mu f} \in [1..N_f]$;
- 6 | μ -block \leftarrow Merge the randomly generated $n_{\mu c}$ convolutional units, $n_{\mu p}$ pooling units and $n_{\mu f}$ dense units;
- 7 | Generate σ -block in the same manner as the μ -block generation process;
- 8 | Sample integers $n_{tf} \in [1..N_f]$, $n_{tc} \in [1..N_c]$ and $n_{td} \in [1..N_d]$;
- 9 | t -block \leftarrow Merge the randomly generated n_{tf} dense units, n_{tc} convolutional units, n_{td} deconvolutional units;
- 10 | Shuffle the convolutional units, the pooling units and the deconvolutional units in each block;
- 11 | Sample the dimensionality of the underlying manifold $r \in [2..R]$;
- 12 | Generate two dense units f_r^1 and f_r^2 with r neurons;
- 13 | μ -block $\leftarrow \mu$ -block $\cup f_r^1$, σ -block $\leftarrow \sigma$ -block $\cup f_r^2$;
- 14 | $p_i \leftarrow h$ -block $\cup \mu$ -block $\cup \sigma$ -block $\cup t$ -block;
- 15 | $P_0 \leftarrow P_0 \cup p_i$;
- 16 **end**
- 17 **return** P_0 ;

5.2.3 Initialization of Population

A population of individuals is created at random using the designed gene encoding strategy. Each individual is made up of four sections, as previously stated. The number of each sort of unit is produced first for each section, then the same number of units is randomly initialized based on the selected number.

Algorithm 3 explains the process of initializing population in detail. In order to generate the h -block, taking a random sample of an integer value of $n_{hc} \in [1..N_c]$ and another integer value of $n_{hp} \in [1..N_p]$ (line 3), where n_{hc} represents the number of convolutional units and n_{hp} represents the number pooling units generated by the h -block. The n_{hc} convolutional layers and the n_{hp} pooling layers, both of which are represented by the genetic units, are then created at random, and the genetic units are merged into one list to produce the h -block (line 4). The network models for any other blocks, including the t -block (line 9), the μ -block (line 6), and the σ -block (line 7), can all be easily obtained by using a procedure that is analogous to the one described above.

The convolutional units, pooling units, and deconvolutional units of each block are then randomly permuted so that various types of layers in the network model are mixed together. The explanation for this is due to the fact that the order of layers in DNN models could potentially have an effect on the overall performance. The encoder is comprised of three blocks: the h -block, the μ -block, and the σ -block. As a direct consequence of this, the maximum number of units of a particular type that can be stored in the encoder section is constrained by the maximum number that was specified beforehand. Using the maximum number of convolutional units as an illustration, the numbers of convolutional units in the h -block, μ -block, and σ -block all fulfill the condition $\max(n_{hc} + n_{\mu c}, n_{hc} + n_{\sigma c}) \leq N_c$. To do this by inserting an additional latent genetic fragment into the chromosome of each individual (line 13), along with an integer selected at random from a specified range R (line 11). At last, each of the building blocks, including the one of latent size r (line 14), is combined into a single individual. This process is going to be repeated until there are N individuals in the population.

5.2.4 Evaluation

During the course of training, if the reconstruction error is rather large, the network will decrease the output of the variance component to improve its ability to recover the original image. As the reconstruction error is minimized, the loss of the KL divergence component grows. At the same time, the optimization shifts its focus to ensure that the output of that part is as standard normal as possible. As a result, it is doable to incorporate the antagonistic interaction between them into the process of training phase and to approach it as a multi-objective optimization problem (MOP). Finding better network architectures that have the lowest average losses requires using the training VAE loss function as the fitness function. While trying to solve this MOP, one of goals is to single out the individual who is the most capable of correctly classifying images. As a result of this goal, the promising model that has the lowest loss typically has a greater ability to model images.

Individuals in P_t can be evaluated and given a final fitness score using Algorithm 4. First, the decoding of the chromosome to the network model m_i is shown in line 3, as well as the initialization of the VAE parameters for the network model. Line 4

Algorithm 4: Evaluation of Performance

Input: The individual list P_t , unsupervised training epoch T_u , the unsupervised training dataset D_{ut} , the learning rate γ , and batch size b

Output: The population with fitness P_t

```

1 for  $i = 1$  to  $|P_t|$  do
2    $f \leftarrow -\infty;$ 
3   Decode the VAE represented by the individual  $P_t^i$  as  $m_i$  and initialize its weight;
4    $K = |D_{ut}|/b;$ 
5   for  $t = 1$  to  $T_u$  do
6     Shuffle the training dataset  $D_{ut}$ ; for  $k = 1$  to  $K$  do
7        $X \leftarrow$  Get the  $k$ -th mini-batch data in  $D_{ut}$ ;
8       Compute the loss  $l_k$  by Equation (5.1) after feeding  $X$  to  $m_i$ ;
9        $W_u \leftarrow W_u - \gamma * \frac{\partial l_k}{\partial W_u};$ 
10    end
11     $f \leftarrow \max(-\frac{1}{K} \sum_{k=1}^K l_k, f);$ 
12  end
13  Assign  $f$  to the  $i$ -th individual  $P_t^i$ ;
14 end
15 return  $P_t$ ;

```

uses the size of the unsupervised training dataset D_{ut} and the batch size b to compute the number of iterations in each epoch. Prior to each epoch, D_{ut} is shuffled because randomization is critical for training. The evaluation algorithm returns the fitness value of individual as the lowest average loss on D_{ut} (line 13) for unsupervised training because it does not require a supervised signal. Line 6–10 explains how to finish the unsupervised training of an epoch: D_{ut} is used to sample a mini-batch of data from $\{\mathbf{x}_i\}_{i=1}^b$, with $\mathbf{x}_i \in \mathbb{R}^{h \times w \times c}$ denoting i -th image in the batch data. The reconstructed data is represented by $\{\hat{\mathbf{x}}_i\}_{i=1}^b$. The loss on the training mini-batch can be calculated using Eq. (5.1), which is obtained from Eq. (2.12).

$$\frac{1}{2b} \sum_{i=1}^b \left(\sum_{m=1}^h \sum_{l=1}^w \sum_{k=1}^c (x_i^{m,l,k} - \hat{x}_i^{m,l,k})^2 + \sum_{j=1}^r (\mu_{i,j}^2 + \sigma_{i,j}^2 - \log \sigma_{i,j}^2 - 1) \right) \quad (5.1)$$

where $x_i^{m,l,k}$ is the pixel in the k th channel of \mathbf{x}_i at position (m, l) . mean square error (MSE) is used as the reconstruction error and divided by the number of samples b because the pixel values of each image are normalized to $[-1, 1]$. The loss function is calculated when the mini-batch is fed into one VAE model. As a second step, determining the gradient of the unsupervised loss with respect to the encoded and decoded parameters W_u . The parameters in each of the four blocks that make up the VAE are included in the set of W_u parameters that are associated with the unsupervised pre-training. The SGD can be used for parameter updating. Other popular optimizers, such as Adam [6], are also suggested and might be used.

5.2.5 Crossover Operator and Mutation Operator

Crossover operator and mutation operator are the two major genetic operators in GAs. In general, the crossover operator acts as a local search to ensure that the dual chromosomes are mixed in the encoding space. The mutation alters a portion of a chromosome at random to maintain the diversity of population and improves the ability to escape local minima as part of the global search.

Due to the fact that both parents should have the same genetic origin having the identical number of genes, traditional GAs allow crossover operations for fixed-length individuals. So as to determine the ideal depth of VAEs, using the variable-length gene encoding technique is recommended, which requires the redesign of a correct crossover operation. Using a variable-length gene encoding technique just makes the nonidentical problem worse because the parents to be matched may have varying lengths. The unit alignment (UA) approach for recombination of dual individuals with variable-length chromosomes was introduced [7] to address this problem. This approach was inspired by human chromosome crossover and forces each chromosome to align at the head and exchange gene information in the shared portions. In crossover, two chromosomes are swapped in homologous segments but the unique regions stay intact, according to studies. Depending on the layer type, the units in the same position in a block can be mismatched in EvoVAE, allowing for variable block lengths. For example, in Fig. 5.4a, the type of the first unit in the *mu*-block of chromosome-1 differs from that of chromosome-2. Information cannot be hybridized between different types of units, hence the method cannot be used on the algorithm itself. In EvoVAE, a method for achieving chromosomal crossover that involved a multi-block alignment is designed.

In Fig. 5.4, each block is delineated by a dotted box of a gray color, and the units that represent the various layers that make up the block, such as the convolutional layers and the pooling layers, are shown in a variety of colors. In the example of the chromosomes of parents illustrated in Fig. 5.4a, the length of each chromosome and the type of the related genes are different from one another. This is a very regular occurrence during the process of evolution, and the multi-block alignment technique is a method that can successfully solve it. The different blocks separately in this technique are extracted; all crossover operations are based on the same block. Because of this, there is no information that is shared across the various blocks of parents. After that, creating a series of lists based on the different types of genetic units that are contained within the blocks and order them according to their location within the blocks. More specifically, the following steps are outlined in Fig. 5.4b to show how the crossover can be achieved on the *t*-blocks of the two chromosomes: (1) First, removing the *t*-blocks from both sets of chromosomes of parents, then creating separate lists for the various types of units that are contained within each *t*-block, and finally, arranging the lists so that they correspond to the head of each pair of chromosomes. (2) Using the SBX [8], the crossover between the two genetic units is represented by the blue arrow with two heads that is located on the left side of Fig. 5.4b. In the event that the list does not contain a comparable unit, there will be

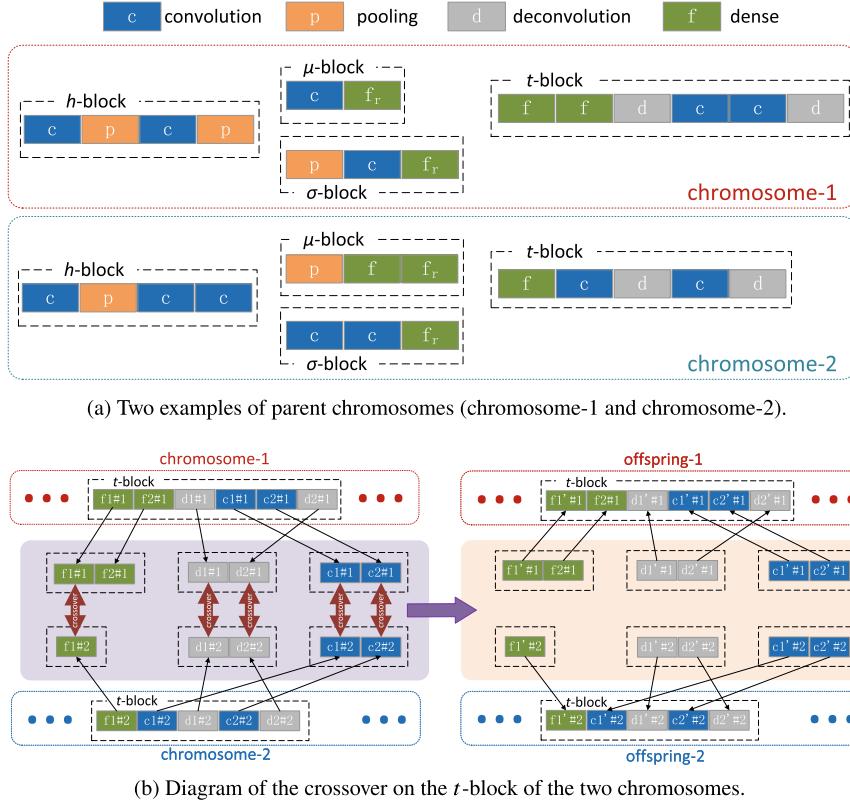


Fig. 5.4 An illustration to the multi-block alignment strategy for the crossover

no information exchanged. (3) Once the process of crossing in the unit list has been finished, each unit will be returned in the list to its original position on the chromosomes that correspond to it. Because of this, as seen in the bottom right corner of Fig. 5.4b, two children that are a combination of chromosome-1 and chromosome-2 are obtained. The multi-block alignment has, as far as can be seen, successfully performed the swapping of two chromosomes without changing the lengths of either of them. When compared, the other three components can be identified and determined.

In EvoVAE, three mutation operations are designed: *addition*, *deletion*, and *modification*, which take effect on the offspring generated by the crossover. A uniform distribution is used to choose which mutation method to employ when the mutation process starts. The chromosome lengths are alternated through addition and deletion operations. The three operations are described in the following sections.

Addition—A new layer can be added to a certain block by utilizing the addition operation. To begin, a block in the given chromosome denoted by x is randomly selected and then produce the unit denoted by u . This unit represents a particular layer, the type of which and its parameters being chosen at random from a uniform

distribution with the settings already determined. It is important to keep in mind that a valid VAE architecture is subject to the following constraints: deconvolutional units can only be located in the t -block; pooling units can only be located in the blocks representing the encoder of the VAE; and dense units can only be located in the tail of the μ -block and σ -block. As a consequence of this, the type of the u must be addressed in a manner that differs according to the block x . In the end, the value u is introduced arbitrarily into the block x at a point that is chosen at random but still adheres to the requirements.

Deletion—In order to remove a layer from a particular block, the deletion operation must be performed. Randomly pick a certain block, denoted by x , from inside the designated chromosome. After that, one of the units in the block x will be removed.

Modification—The values of the following characteristics in the genetic unit of a given chromosome are altered when the modification operation is performed: the number of features in dense units, the kernel sizes and the number of features in deconvolutional units, the kernel sizes and number of features in convolutional units, and the type of pooling units. The Polynomial Mutation [9] is utilized to the values of the characteristics for each genetic unit u in the chromosome. This is done because the Polynomial Mutation has a promising performance when it comes to processing mutations on real numbers.

5.2.6 Environmental Selection

Environmental selection aims to generate the parent population surviving into the next generation using the already existing population as well as their offspring. The method of environmental selection is carried out through mixing elitist selection with civilian selection [7]. In order to achieve this goal, the most optimal partition first selected from among them to serve as the elitism. This will guarantee that the optimization results will converge. In the meantime, in order to select the remaining civilians from among those who have not been chosen as elite, the competition known as the binary tournament is used. This is done in order to make certain that the next generation of individuals would have a diverse make up. At last, the population of the next generation will be comprised of a mix of elitists and regular individuals.

5.3 Experimental Design

A series of well-designed experiments are carried to illustrate the efficacy of EvoVAE. The implemented specifics and peer competitors employed for the experiments are explained in the following sections.

5.3.1 Parameter Setting

The evolution parameters of EvoVAE are set in accordance with GA community convention. In particular, the probability of crossover operator is specified as 0.8, while that of mutation operator is specified as 0.1. Both SBX [8] and polynomial mutation [10] distribution indexes are set to 20. As advised by [11], the population size and generation number are set with the same number of 20. Furthermore, according to the Pareto principle [7], the proportion of elitism is fixed at 20%.

The number of convolutional layers is specified from one to six. The channels of the convolutional layers are specified from 20 to 128, which is also applied to those of the deconvolutional layers. The number of dense layer is specified as one or two, and the corresponding number of neurons is specified from 64 to 256. In addition, the number of pooling layers is specified from one to two, the kernel size is specified as two to five, and the size of the latent representation r is specified from two to 128. Furthermore, convolution kernel and stride are made by ensuring that both their length and width are the same. This decision was made based on our prior experience with the parameter settings used in well-known network architectures. The “same” padding operation is used for the standard convolutional layers to guarantee that the resolution of input is identical to that of output. As a result, the pooling operation, with the kernel and stride sizes both set to two, is exclusively responsible for the resolution reduction. This setup not only follows the principles of deep learning community, but it also makes it easier to change the parameters of network and the resolution of output during the evolution process, lessening the complexity of gene decoding.

5.3.2 Peer Competitors

The AE, CAE [12], DAE [13], sparse AE [14], and their respective stacked versions, i.e., the stacked AE, the stacked sparse AE, the stacked DAE, the stacked CAE, and the stacked CDAE, are also selected for the comparisons.

Because the architectures of the selected peer competitors are manually created, for a fair comparison, an equal number of candidates are generated as populations in EvoVAE for each AE. Their architectures are created by doing a random search applying the same architecture parameter ranges and candidate numbers as were used in the EvoVAE. For the comparison, the one having the best results is selected. This is as a result of the fact that the reconstruction error is a standard criterion for AE training that is extensively applied. There is a cap of two on the number of pooling layers that can be used in the method. In the architecture, once a pooling layer is introduced, the encoder will be given two convolutional layers, while the decoder will be given a deconvolutional layer and a convolutional layer. Both of these layers will be added after the pooling layer has been added. As a direct consequence of this, the maximum number of layers that an algorithm can support is 10. To ensure

that the results of the experiments are comparable, the number of hidden layers of all staked AEs has been equalized to 10.

Furthermore, the peer competitor is the semi-supervised learning (SSL) model approach [15], which is based on VAEs and has lately achieved the best performance. In the subsequent tests, SSL is used in the model of M1, and renames “SSL-M1”. In order to acquire the specific experimental results, the network design is reimplemented such that it is consistent with the original paper. The SSL-M1 is a hand-crafted architecture that consists of two hidden dense layers, each with 600 neurons, and uses the softplus activation function to develop both the encoder and the decoder. They decided to use 50 as the value for the dimension of the underlying manifold.

Other existing solutions in the field of machine learning, such as k -nearest neighbor (KNN) where the k is one, SVM [16], and learning with local and global consistency (LLGC) [17], are also utilized as baselines to address the image classification tasks in the following experiments. This is due to the fact that these solutions have a wide range of applications and are easy to reproduce. Take into consideration the fact that LLGC is a SSL algorithm. The conclusions of these three algorithms may assist the reader establish an intuitive grasp regarding the challenging of a specific task.

5.3.3 Performance Evaluation

Directly classifying images using generic AEs is tricky. The method as suggested in [2, 18] is used to build a classifier on top of the encoder part of an AE. In the process of image classification that makes use of the AE, it is normal practice to incorporate both the pre-training phase and the fine-tuning phase. In particular, during the unsupervised pre-training phase of the AE, the vast unlabeled samples are used in order to train both the encoder and the decoder of the AE. In the second phase of the process, the labeled samples are utilized to train the classifier that will be added to the AE. It is important to take note that following the completion of the first phase, the decoder is often deprecated and is never used in the second phase. In the following, the specifics of the training method that the EvoVAE and other AE-like competitors use will be discussed.

In the initial phase of training for VAE-like models, follow the protocol that has been established by the community and set the sampling frequency L of Eq. (2.12) to one. Simultaneously, the Adam optimizer [6] is used in our experiments to speed up the training convergence. The initial learning rate is 0.001, the batch size is 128, and the weight decay is 0.00001. On the image data that has been provided, do not undertake any extra specific preparations, such as data augmentation or any other methods.

It is generally required to train the network to a magnitude of 10^2 or 10^3 on the whole training set in actual experiments during the initial step of unsupervised pre-training when the network is being trained without supervision. The computational

complexity of the population-based EvoVAE, on the other hand, makes it impossible to train with such a large number of epochs. In point of fact, do not require quite as many iterations of the process as [2] suggests. Because of this, in the unsupervised pre-training phase of the EvoVAE algorithm, the epoch number is specified as 20, which resulted in a significant increase in the processing performance of algorithm.

Once EvoVAE has determined the most effective architecture for the VAE, it will proceed to deep train the obtained network model for a longer period of time so that it can classify images. A classifier is added to μ -block, which is a subnetwork with two dense layers and a dropout [19] layer. In the beginning of the deep training process, setting the number of epochs to 400, and later on, when the algorithm is fine-tuning, set it to 100. Both of these parameters are consistent across all of the AE-like models that tested in the experiments.

PyTorch, a popular open source deep learning framework, is used in all of the experiments. The instructions are duplicated and run on a system that has two NVIDIA RTX 2080TI GPUs. A ReLU [20] is utilized as the activation function of the hidden layers in all of the models that have been utilized. In all of the models that have been utilized for the purpose of training acceleration and stability, the BN layer is inserted after each convolutional layer. All DNN models begin with five different random seeds each, and during the supervised fine-tuning phase, each model receives separate training under supervision. After that, the means and standard deviations of the five outcomes are displayed in order to reduce the amount of disruption generated by the random values. In passing, the network architecture optimization on the aforementioned datasets takes approximately three days to complete with the two 2080TI GPUs.

5.4 Experimental Results and Analysis

5.4.1 Overall Performance

In this section, taking on image classification problems utilizing VAEs that have been optimized by EvoVAE. The accuracy of classification is then compared to that of competitors. The generic AEs are unsupervised learning techniques that cannot be strictly adhered to image classification tasks, as stated in Sect. 5.3. In this chapter, the image classification is done in the second phase of fine-tuning, and the first phase of unsupervised pre-training can be considered an auxiliary process to improve the initial weight of the model for the second phase, so that the fine-tuning phase can benefit from the first phase. The image classification is done in the second phase of fine-tuning in this chapter. Altering the total number of labeled samples is one of the common criteria used in the second phase of the fine-tuning process, such as in the work in [2]. This is one of the ways to evaluate the effectiveness of the training mechanism. In theory, a promising unsupervised learning model may obtain improved performance on supervised tasks even with a small number of labeled sam-

ples if enough of those data were labeled. Experiments are built with varying amounts of labeled data so that the capability of such unsupervised learning methods can be evaluated. Because the SVM and KNN algorithms are unable to acquire knowledge by first analyzing a large number of unlabeled data, it is recommended that they be applied immediately to the samples that have been labeled. As a consequence of this, it can also be able to show the details of unsupervised learning algorithms in improving the effectiveness of supervised learning methods under this particular scenario.

Tables 5.1, 5.2 and 5.3 illustrate the experimental results of the EvoVAE on the three benchmark datasets, with architectures optimized by themselves, with the best results indicated in bold. Adjust the amount of labeled samples on the MNIST dataset between 100, 600, 1,000, 2,000, 3,000, 5000, 10,000, and 60,000. Adjust the amount of labeled samples on the SVHN dataset from 100 to 600, 1,000 to 2,000, 5,000 to 10,000, and 40,000. This changes to 100, 600, 1,000, 2,000, 3,000, 5,000, 10,000, and 50,000K on the CIFAR-10 dataset, correspondingly.

Table 5.1 Experimental results of the compared algorithms on the MNIST dataset

Methods	100	600	1 K	2 K	3 K	5 K	10 K	60 K
KNN	73.09	86.03	88.58	90.65	92.20	93.67	94.89	96.88
SVM	79.34	91.32	93.09	94.60	95.19	95.99	96.72	98.33
LLGC	87.86	94.00	94.35	95.28	95.37	95.83	96.24	96.93
AE	75.36 (2.43)	87.81 (0.13)	90.41 (0.45)	92.54 (0.17)	94.25 (0.06)	95.45 (0.15)	96.94 (0.18)	98.40 (0.07)
CAE	78.95 (0.91)	92.73 (0.43)	94.81 (0.35)	96.64 (0.16)	97.07 (0.11)	97.05 (0.28)	98.01 (0.09)	98.95 (0.02)
DAE	76.78 (3.19)	88.27 (0.23)	91.16 (0.42)	93.89 (0.20)	94.97 (0.27)	96.22 (0.18)	97.24 (0.14)	98.63 (0.04)
sparse AE	78.87 (0.86)	88.71 (0.33)	90.44 (0.17)	92.92 (0.21)	94.32 (0.27)	95.60 (0.16)	96.97 (0.02)	98.35 (0.06)
stacked AE	73.64 (1.30)	86.93 (0.40)	89.05 (0.37)	92.14 (0.26)	93.58 (0.40)	95.06 (0.32)	96.58 (0.11)	98.34 (0.06)
stacked CAE	77.29 (2.04)	93.04 (0.34)	95.03 (0.28)	96.51 (0.21)	97.31 (0.15)	97.89 (0.07)	98.51 (0.10)	99.37 (0.04)
stacked DAE	73.85 (1.37)	88.87 (0.28)	90.95 (0.26)	93.40 (0.22)	94.43 (0.18)	95.26 (0.26)	96.56 (0.18)	98.15 (0.06)
stacked sparse AE	22.91 (1.24)	80.72 (1.69)	84.93 (1.29)	90.74 (0.67)	92.02 (0.35)	94.52 (0.20)	96.03 (0.11)	98.10 (0.12)
stacked CDAE	81.33 (0.94)	94.53 (0.11)	96.05 (0.17)	97.26 (0.16)	97.78 (0.08)	98.29 (0.09)	98.74 (0.03)	99.41 (0.02)
SSL-M1+FC ^a	72.78 (1.27)	88.21 (0.21)	90.64 (0.40)	93.42 (0.23)	94.51 (0.22)	95.51 (0.13)	96.95 (0.14)	98.43 (0.09)
EvoVAE(Ours)	87.65 (1.49)	95.92 (0.44)	96.76 (0.31)	97.89 (0.27)	98.38 (0.20)	98.79 (0.16)	99.14 (0.13)	99.48 (0.06)

^aUsing the VAE network design specified in the literature [15] and assess it in the designed paradigm for fair comparison

Table 5.2 Experimental results of the compared algorithms on the SVHN dataset

Methods	100	600	1 K	2 K	5 K	10 K	40 K	FULL ^a
KNN	13.80	18.36	19.92	23.12	27.69	31.44	37.54	42.37
SVM	12.54	12.97	17.22	20.31	29.52	36.74	51.68	54.34
LLGC	12.92	18.83	21.24	25.39	28.01	31.34	45.29	48.39
AE	16.39 (1.60)	40.69 (2.00)	50.15 (0.89)	59.45 (0.43)	68.07 (0.72)	72.92 (0.61)	79.13 (0.10)	82.70 (0.15)
CAE	17.26 (1.86)	18.86 (1.45)	18.15 (1.76)	19.59 (0.00)	17.88 (3.41)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)
DAE	16.42 (0.92)	38.93 (1.05)	48.97 (1.62)	59.39 (0.79)	63.67 (0.68)	73.10 (0.70)	79.83 (0.25)	82.78 (0.25)
sparse AE	18.49 (1.43)	43.55 (1.15)	52.11 (1.34)	60.02 (0.94)	68.19 (0.46)	72.67 (0.50)	80.17 (0.20)	83.17 (0.18)
stacked AE	15.72 (1.01)	38.97 (1.04)	47.87 (1.44)	57.16 (0.38)	65.84 (0.66)	70.71 (0.49)	78.52 (0.30)	81.98 (0.17)
stacked CAE	18.82 (1.54)	19.59 (0.01)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)	89.15 (0.10)
stacked DAE	15.08 (0.68)	42.66 (1.55)	51.85 (1.03)	61.11 (0.63)	68.42 (0.93)	73.71 (0.51)	81.01 (0.13)	83.73 (0.16)
stacked sparse AE	19.59 (0.00)	19.08 (1.53)	21.96 (4.74)	24.48 (2.56)	41.50 (2.38)	53.08 (4.33)	76.29 (0.43)	80.98 (0.18)
stacked CDAE	17.58 (2.45)	20.02 (1.29)	17.54 (3.26)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)	19.59 (0.00)	89.56 (0.19)
SSL-M1+FC	16.00 (1.10)	42.58 (1.82)	49.71 (0.98)	57.99 (1.33)	66.41 (0.68)	71.53 (0.29)	78.92 (0.34)	82.38 (0.12)
EvoVAE(Ours)	16.74 (1.37)	55.69 (2.09)	66.13 (2.55)	75.32 (2.73)	79.31 (1.97)	82.51 (1.56)	90.37 (0.97)	92.01 (0.70)

^aFull labels of the training dataset are employed

According to the results, the VAE that was optimized by EvoVAE produced competitive results in majority of the experiments, despite having variable quantities of labeled samples on the datasets that were selected. EvoVAE achieves the highest performance out of all the algorithms that are tested, with the exception of situations including samples labeled with 100 where the LLGC, stacked sparse AE, and CAE achieve the best results, respectively. On the CIFAR-10 dataset, the algorithm demonstrates that EvoVAE is superior than its peers by a significant margin, outperforming them by a large margin overall. In addition, it has been found that the promotion of unsupervised learning wanes as the number of training samples increases. The EvoVAE algorithm is just 2.45% more accurate than the inferior stacked CDAE strategy when applied to full samples from the SVHN dataset. To preserve consistency, the training parameters, such as the number of training epochs and the starting learning rate, are adjusted to be the same for all models. Nevertheless, this strategy may prevent some of the models in Tables 5.2 and 5.3 from converging on a solution.

Table 5.3 Experimental results of the compared algorithms on the CIFAR-10 dataset

Methods	100	600	1 K	2 K	3 K	5 K	10 K	50 K
KNN	18.45	23.13	23.99	24.49	25.72	27.17	29.09	35.39
SVM	24.15	34.09	36.75	39.15	40.88	43.11	45.61	51.40
LLGC	17.18	22.15	22.62	26.02	26.86	28.08	29.81	36.06
AE	26.45 (1.02)	33.95 (0.44)	36.18 (0.32)	39.77 (0.43)	41.53 (0.27)	44.49 (0.40)	47.55 (0.19)	55.65 (0.28)
CAE	27.35 (0.89)	33.89 (3.89)	32.85 (6.40)	32.05 (7.94)	29.41 (4.64)	29.65 (4.30)	37.02 (3.54)	42.73 (16.37)
DAE	25.48 (1.73)	33.81 (0.40)	36.35 (0.67)	40.28 (0.65)	42.03 (0.25)	44.50 (0.21)	48.22 (0.26)	55.87 (0.13)
sparse AE	24.64 (0.69)	32.97 (0.81)	35.26 (0.38)	38.71 (0.33)	40.51 (0.46)	43.52 (0.53)	46.84 (0.40)	54.40 (0.36)
stacked AE	24.93 (1.01)	32.34 (0.93)	34.67 (0.98)	37.45 (0.33)	39.70 (0.22)	42.32 (0.35)	46.13 (0.15)	54.66 (0.33)
stacked CAE	25.76 (1.13)	27.83 (6.88)	31.73 (4.82)	26.27 (3.46)	26.08 (4.40)	35.95 (5.50)	41.84 (1.72)	63.42 (1.37)
stacked DAE	25.51 (1.15)	32.61 (0.44)	35.86 (0.22)	39.28 (0.45)	41.07 (0.25)	43.97 (0.45)	47.74 (0.16)	55.68 (0.21)
stacked sparse AE	15.84 (0.77)	21.46 (0.46)	23.83 (1.55)	28.27 (1.15)	29.82 (1.39)	33.87 (0.20)	38.74 (0.36)	49.34 (0.32)
stacked CDAE	26.20 (1.26)	37.42 (1.50)	38.51 (3.23)	45.35 (1.12)	44.04 (3.18)	46.01 (1.51)	54.79 (1.24)	66.80 (2.11)
SSL-M1+FC ^a	24.05 (1.13)	33.13 (0.92)	35.14 (0.55)	38.29 (0.15)	40.26 (0.29)	43.16 (0.32)	46.54 (0.36)	54.55 (0.41)
EvoVAE(Ours)	26.81 (1.96)	40.96 (2.03)	45.12 (1.33)	51.59 (1.09)	54.79 (0.95)	59.04 (0.88)	64.58 (1.35)	76.12 (1.94)

It can be discovered that the two-phase training mechanism performs significantly better than the baseline algorithms in the vast majority of instances. On the MNIST dataset, the best result obtained by the EvoVAE algorithm is 8.31% better than the best result obtained by the SVM algorithm while using samples that have been labeled with 100. On the SVHN dataset, the accuracy gap is very glaring and obvious. When considering samples of 2,000, this improvement rises to a level of 55.01%. In addition, the two-phase training mechanism seems to be less noticeable in some instances when there are only a few samples available for analysis. On the CIFAR-10 dataset with 100 labeled samples, the best AE is just 3.2% more accurate than the best baseline algorithm. Because the two-phase training mechanisms may be the source of this constraint, an end-to-end semi-supervised algorithm should be researched to replace the naive two-phase training mechanism. Furthermore, the experimental results show that the LLGC is superior on datasets with a small number of labeled samples that are less complicated. With 100 labelled samples on the MNIST dataset, the LLGC technique outperforms SVM by 8.52% and EvoVAE by

0.21%. The performance of the LLGC on the CIFAR-10 dataset is comparable to that of the KNN.

In conclusion, the results that have been shown in this section demonstrate the promising performance of EvoVAE on a variety of different numbers of training samples. The results of the competition demonstrate that EvoVAE is scalable when it comes to dealing with image classification tasks on the datasets that were chosen.

5.4.2 Evolution Trajectory of EvoVAE

The EvoVAE optimization trajectories on the three benchmark datasets is displayed in Fig. 5.5 to confirm that EvoVAE effectively optimizes the architectures of the VAEs and obtains promising architecture, where the x -axis denotes the loss of the best VAE architecture and the y -axis denotes the corresponding generation.

The CIFAR-10 dataset, which contains real objects, is significantly more challenging than the other two datasets shown in Fig. 5.5. The loss curve on CIFAR-10 demonstrates, in addition, that the corresponding VAE on the dataset has a loss that is more than 100. On the other hand, the losses that occur on the MNIST and SVHN datasets are lower than those that occur on the CIFAR-10 dataset. The loss changes around 1, particularly when compared to the MNIST dataset; it has been converging ever since the 6th generation. Since around the 11th generation, EvoVAE has converged on the SVHN dataset, and its loss was approximately 54.4 at that point. On the CIFAR-10 dataset, it reached a point of convergence in the 16th generation, and it looks to be steadily decreasing ever since.

Producing virtual images for end-users who need synthetic data is another role that the VAEs are responsible for performing. In order to generate a synthetic fake sample, one can first sample from a multivariate normal distribution to obtain a random latent variable z' , and then one can input z' into the decoder of the VAE which will provide the desired result. The produced images is plotted in Fig. 5.6 throughout the evaluation of each individual to better understand the performance

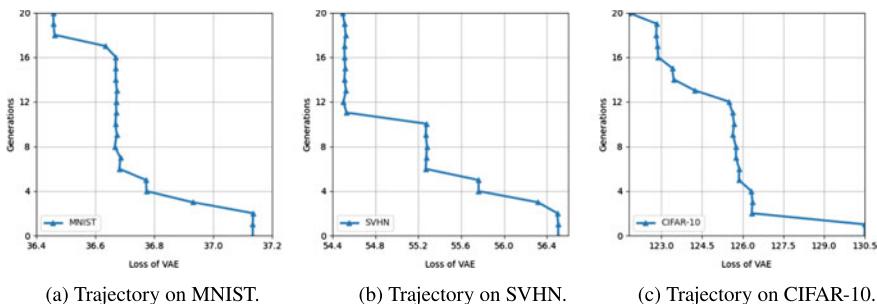


Fig. 5.5 The trajectory of evolved VAEs in discovering architectures on the three benchmarks

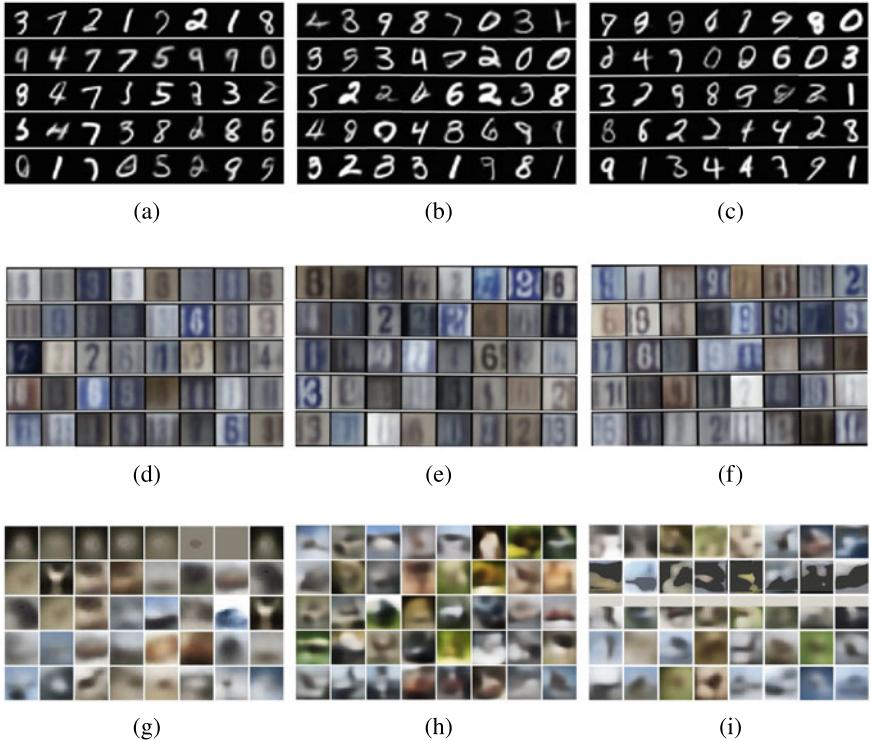


Fig. 5.6 Images generated at random, each row in the subfigure is produced by individuals chosen at random during evolution

of the architecture search. Figure 5.6a–c, d–f, g–i show the produced images on the benchmark datasets of MNIST, SVHN, and CIFAR-10, respectively. The images produced by the population at the generation of the first, the middle, and the last are depicted in the figures from left to right.

Within the identical limited epoch number of training, there is a significant qualitative enhancement in sharpness comparing to the first generation. This is something that was observed in datasets such as the SVHN and the CIFAR-10. If taking a look at Fig. 5.6f, for instance, there is a greater variety of digits than there are in Fig. 5.6d. On the other hand, there is not a discernible change in image quality from generation to generation on the MNIST dataset. In the meanwhile, when compared to GANs, VAEs generate images that are significantly more blurry, particularly for complex datasets such as the CIFAR-10. There have also been attempts made to use the VAEs to generate high-quality images, such as human faces [21], but this is not the main focus of this research.

5.4.3 Running Time

The unsupervised training time (min) and testing time (ms) of each comparison method are shown in Tables 5.4 and 5.5. The training speed of the EvoVAE algorithm is the slowest on the dataset in the unsupervised pre-training phase, while stacked CDAE is the second slowest method. The LLGC method is the most time-consuming during the testing phase. That is due to the fact that it takes a long time to calculate the whole affinity matrix. Furthermore, SVM and AEs with only dense layers take the least amount of time.

5.4.4 The Obtained Architecture

Table 5.6 shows the optimized architectures of the VAEs on various datasets, with the string “ $c@k \times k$ ” indicating “ $k \times k$ ” convolution with c channels in the parameter setting of a convolutional or deconvolutional layer. The types of network layers, the specific parameters, and the dimensionalities r of the underlying manifolds are provided for each block. Convolutional, pooling, deconvolutional, and dense layers are represented by the letters “C”, “P”, “D” and “F” in the layer type, respectively. The “Avg” symbol denotes the average pooling layer, whereas the “Max” symbol denotes MAX layer.

Table 5.6 shows that, despite the fact that the optimized architectures for the three datasets varied, the μ -block is always more complex than the σ -block. The μ -blocks, on the other hand, have more layers than the σ -block. The symmetrical convention, however, requires that the σ -block have the same architecture as the μ -block in most applications. Nevertheless, based on the results, this constraint may not be suitable, emphasizing the need for EvoVAE algorithm to be implemented in an asymmetrical manner.

Furthermore, because the MNIST dataset is simpler than natural scene images, the generated t -block of the VAE on the MNIST dataset is shallower than those on the other datasets. This point could also be supported by the dimensionalities of the underlying manifolds. On the MNIST, SVHN, and CIFAR-10 datasets, the dimensionalities r are 16, 66, and 70, respectively. When the intrinsic structures of the images become more intricate, decoders must include additional parameters to reconstruct the image from compact representations z .

Another unusual phenomenon has caught our attention. Except for the h -block and a few critical parameters, the optimized architectures on the SVHN and CIFAR-10 datasets are extremely comparable. However, these two datasets are obviously different from a human standpoint, thus two completely different architectures should be obtained. This finding shows that a comparable architecture could handle two datasets that are somewhat distinct from a human perspective. It is hoped that this phenomenon can be investigated in future work.

Table 5.4 Unsupervised training time (min) of the compared algorithms on each dataset

Datasets	KNN	SVM	LLGC	AE	CAE	DAE	SAE	Stacked AE	Stacked CAE	Stacked DAE	Stacked sparse AE	Stacked CDAE	SSL	EvoVAE
MNIST	0	–	–	13	30	14	15	15	31	20	208	58	18	119
SVHN	0	–	–	21	49	20	14	26	36	26	33	173	38	198
CIFAR-10	0	–	–	14	32	14	15	18	27	18	22	97	34	68

Table 5.5 Testing time per sample (ms) of the compared algorithms on each dataset

Datasets	KNN	SVM	LLGC	AE	CAE	DAE	SAE	Stacked AE	Stacked CAE	Stacked DAE	Stacked sparse AE	Stacked CDAE	SSL	EvoVAE
MNIST	–	0.19	240	0.21	0.38	0.21	0.20	0.21	0.63	0.26	0.25	0.62	0.22	0.87
SVHN	–	0.23	1162	0.22	0.39	0.22	0.22	0.27	0.62	0.27	0.27	0.63	0.23	0.70
CIFAR-10	–	0.22	645	0.21	0.38	0.21	0.21	0.27	0.61	0.26	0.27	0.62	0.23	0.83

Table 5.6 The architectures of the VAEs optimized by EvoVAE on the benchmark datasets

Dataset	<i>h</i> -block		μ -block		σ -block		<i>r</i>	<i>t</i> -block	
	Type	Configuration	Type	Configuration	Type	Configuration	Configuration	Type	Configuration
MNIST	C	31@4 × 4	C	109@3 × 3	C	99@4 × 4	16	F	181
	P	Max	C	88@2 × 2	C	120@5 × 5		D	73@2 × 2
	P	Avg	C	127@5 × 5	C	48@4 × 4		C	103@4 × 4
			C	94@3 × 3	F	118		C	30@4 × 4
			F	224					
			F	164					
SVHN	P	Avg	C	57@3x3	C	68@3 × 3	70	F	218
	C	42@3 × 3	C	63@3 × 3	F	134		C	69@2 × 2
	C	111@2 × 2	F	142				C	33@3 × 3
	P	Max						C	126@4 × 4
								C	87@3 × 3
								D	61@2 × 2
								C	43@2 × 2
CIFAR-10	C	41@3 × 3	C	57@3 × 3	C	79@2 × 2	66	F	246
	P	Avg	C	62@3 × 3	F	134		C	55@2 × 2
	C	55@3 × 3	F	168				C	30@3 × 3
	C	115@2 × 2						C	123@4 × 4
	P	Max						C	87@2 × 2
								D	65@2 × 2
								C	69@4 × 4

5.4.5 Ablation Experiments

The performances of the genetic operator are investigated through ablation experiments. To investigate the effect that the Random Sampling can be obtained from EvoVAE, Random Sampling is firstly perform with a sample size of 400 on the chosen datasets. Then, for the comparisons, their loss values on each dataset are given, and the results are shown in Table 5.7. Table 5.7 illustrates EvoVAE algorithm has lower loss values on the three benchmark datasets, demonstrating that it outperforms the Random Sampling method.

The effectiveness of crossover and mutation operators has been evaluated using ablation studies of both operators. In addition, instead of employing random sampling within EvoVAE framework, experiments was ran without both crossover operator and mutation operator. Table 5.8 shows the experimental results. Please note that the experimental results of the framework of EvoVAE random sampling are identical to those of the “Random Sampling” shown in Table 5.7. That is because they are essentially the same. Table 5.8 shows that on two of the three benchmark datasets, genetic operators may optimize the final architecture with lower loss values. EvoVAE enabled with mutation and crossover provides the greatest performance on the MNIST and CIFAR-10 datasets. However, while the EvoVAE solely with the crossover outperforms the Random Sampling on the SVHN dataset, the algorithm that only uses the mutation achieves the best result. That might be because the SVHN

Table 5.7 The final losses of the optimized architectures obtained by the Random Sampling and EvoVAE on the chosen datasets

Methods	MNIST	SVHN	CIFAR-10
Random sampling	36.72	55.49	124.26
EvoVAE	36.52	54.44	121.75

Table 5.8 The final losses of the optimized architectures on the chosen datasets with different search strategies

Strategies	MNIST	SVHN	CIFAR-10
EvoVAE (Random sampling)	36.72	55.49	124.26
EvoVAE (without crossover)	36.59	53.28	122.46
EvoVAE (without mutation)	36.81	54.71	126.36
EvoVAE	36.52	54.44	121.75

dataset has more local minima, and the crossover degrades the ability to execute a global search.

5.5 Chapter Summary

In this chapter, we introduced the EvoVAE algorithm, which is a GA-based algorithm for automatically designing the optimal architectures of VAEs. The effectiveness of EvoVAE is firstly confirmed by experiments on the designed architectures against peer competitors. In addition, the evolution trajectory, running time, and the obtained architectures of EvoCAE are also investigated with the same goal.

This chapter and the previous two chapters focused on the automated design of unsupervised DNNs for image classification. While these are very useful when lacking labels in the image datasets, the majority of the methods developed for image classification are still supervised DNNs, particularly CNNs and their variations. Many different EC methods including GAs, PSO, DE, and GP have been used, and integration of machine learning methods such as surrogate and transfer learning have also been introduced. These will be described in the next part of the book, Part III.

References

1. Lee, K. H., Kang, S. J., Kang, W. H., & Kim, N. S. (2016). Two-stage noise aware training using asymmetric deep denoising autoencoder. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5765–5769). IEEE.
2. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2019c). A particle swarm optimization-based flexible convolutional autoencoder for image classification. *IEEE Transactions on Neural Networks and Learning Systems*, 30(8), 2295–2309.
3. Miller, B. L., Goldberg, D. E., et al. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3), 193–212.
4. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
5. Bengio, Y., & Delalleau, O. (2011). On the expressive power of deep architectures. In *International conference on algorithmic learning theory* (pp. 18–36). Springer.
6. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
7. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2020a). Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24(2), 394–407.
8. Deb, K., & Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. *Complex Systems*, 9(2), 115–148.
9. Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms* (Vol. 16). Wiley.
10. Deb, K., & Agrawal, S. (1999). A niched-penalty approach for constraint handling in genetic algorithms. In *Artificial neural nets and genetic algorithms* (pp. 235–243). Springer.
11. Sun, Y., Xue, B., Zhang, M., Yen, G. (2019b). Completely automated cnn architecture design based on blocks. *IEEE Transactions on Neural Networks and Learning Systems*, 1–13. <https://doi.org/10.1109/TNNLS.2019.2919608>.
12. Masci, J., Meier, U., Cireşan, D., & Schmidhuber, J. (2011). Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning-ICANN*, 52–59.
13. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11, 3371–3408.
14. Lee, H., Battle, A., Raina, R., Ng, A. Y. (2007). Efficient sparse coding algorithms. In *Advances in neural information processing systems* (pp. 801–808).
15. Kingma, D. P., Mohamed, S., Rezende, D. J., & Welling, M. (2014). Semi-supervised learning with deep generative models. In *Advances in neural information processing systems* (pp. 3581–3589).
16. Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273–297.
17. Zhou, D., Bousquet, O., Lal, T. N., Weston, J., & Schölkopf, B. (2004). Learning with local and global consistency. In *Advances in neural information processing systems* (pp. 321–328).
18. Sun, Y., Yen, G. G., & Yi, Z. (2019). Evolving unsupervised deep neural networks for learning meaningful representations. *IEEE Transactions on Evolutionary Computation*, 23(1), 89–103 February.
19. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958. ISSN 1532-4435.
20. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323).
21. Dai, B., & Wipf, D. (2018). Diagnosing and enhancing vae models. In *International Conference on Learning Representations*.
22. Sun, Y., Xue, B., Zhang, M., Yen, G. G., & Lv, J. (2020b). Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 50(9), 3840–3854.

23. Wang, B., Sun, Y., Xue, B., & Zhang, M. (2019). A hybrid ga-pso method for evolving architecture and short connections of deep convolutional neural networks. In *Pacific Rim International Conference on Artificial Intelligence* (pp. 650–663). Springer.
24. Wang, B., Sun, Y., Xue, B., & Zhang, M. (2018a). Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)* (pp. 1–8). IEEE.
25. Wang, B., Sun, Y., Xue, B., & Zhang, M. (2018b). A hybrid differential evolution approach to designing deep convolutional neural networks for image classification. In *Australasian joint conference on artificial intelligence* (pages 237–250). Springer.
26. Liu, Y., Sun, Y., Xue, B., & Zhang, M. (2020). Evolving deep convolutional neural networks for hyperspectral image denoising. In *2020 international joint conference on neural networks (IJCNN)* (pp. 1–8). IEEE.

Part III

Evolutionary Deep Neural Architecture Search for Supervised DNNs

In this part, the ENAS algorithms for CNNs are introduced, which are widely used for supervised learning. There are the ENAS for plain CNN method (EvoCNN) [1], the ENAS for RB and DB-based CNN method (AE-CNN) [2], the ENAS for skip connection-based CNN method (CNN-GA) [3], the ENAS using hybrid GA and PSO method (Hybrid GA-PSO) [4], the ENAS based on Internet protocol method (IPPSO) [5], the ENAS using DE method (DECNN) [6], and the ENAS for analyzing hyperspectral image (CNN-HID) [7].

References

1. Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G Yen. Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24(2):394–407, 2020.
2. Yanan Sun, Bing Xue, Mengjie Zhang, and Gary Yen. Completely automated cnn architecture design based on blocks. *IEEE Transactions on Neural Networks and Learning Systems*, PP:1–13, 06 2019. doi: 10.1109/TNNLS.2019.2919608.
3. Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Jiancheng Lv. Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 50(9):3840–3854, 2020.
4. Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. A hybrid ga-pso method for evolving architecture and short connections of deep convolutional neural networks. In *Pacific Rim International Conference on Artificial Intelligence*, pages 650–663. Springer, 2019.
5. Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2018.
6. Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. A hybrid differential evolution approach to designing deep convolutional neural networks for image classification. In *Australasian Joint Conference on Artificial Intelligence*, pages 237–250. Springer, 2018.
7. Yuqiao Liu, Yanan Sun, Bing Xue, and Mengjie Zhang. Evolving deep convolutional neural networks for hyperspectral image denoising. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

Chapter 6

Architecture Design for Plain CNNs



6.1 Introduction

Using GAs in the architecture design of CNNs directly presents a number of challenges. On the one hand, the suitable architecture cannot be determined unless its performance is analyzed and compared to other architectures. However, measuring the effectiveness of a sole individual consumes a long time, and the situation becomes even more challenging when population-based methods are adopted. To hasten the process of evolution, significantly more computational resources would be needed. On the other hand, because the appropriate depth of CNNs for a certain issue is not known, it is difficult to confine the range of search space for design optimization. Consequently, a variable-length gene encoding approach may be the best alternative. However, a newly found challenge is identifying how to determine the crossover operation for distinct building blocks. In addition, the weight initialization values have a significant impact on the validated architecture performance, while tackling this issue necessitates a fine strategy of gene encoding as well as the optimization of hundreds of thousands of decision variables.

Based on the above discussion, the goal of this chapter is to introduce EvoCNN, which is an ENAS approach to discovering excellent architectures of CNNs and appropriate values for initializing connection weights without the need for manual intervention. The following objectives have been set in order to reach this goal:

1. Establish an appropriate architecture gene encoding strategy that does not place any limits on the maximum length of the building blocks used in CNNs. It is anticipated that the newly developed architecture will be beneficial to CNNs in achieving good performance when addressing a variety of problems utilizing this gene encoding strategy.
2. It is important to investigate the encoding strategy for connection weight since it can effectively represent a large number of connection weights. It is anticipated that the recommended GA will successfully optimize the weight connection initialization problem in CNNs that make use of this encoding technique.

3. Develop associated mutation, crossover, and selection operators that can deal with both architectures and connection weights using the designed gene encoding methods.
4. Make a viable fitness metric for individuals representing diverse CNNs that will not necessitate extensive computational resources.
5. Examine whether or not the new strategy outperforms the existing methods in terms of accuracy of image classification tasks and quantity of weight values.

The remainder of this chapter is organized as follows: Sect. 6.2 describes the framework and specifics of each phase in EvoCNN. Sections 6.3 and 6.4 provide the experimental design and results, respectively.

6.2 Algorithm Details

In this section, the details of EvoCNN for image classification are documented, including the framework of EvoCNN and the design of its core components.

6.2.1 Algorithm Overview

Algorithm 1: Framework of EvoCNN

```

1  $P_0 \leftarrow$ Utilize the designed strategy of gene encoding for population initialization;
2  $t \leftarrow 0$ ;
3 while stopping condition is not met do
4   Evaluate the fitness value of each individual in  $P_t$  by utilizing the designed approach;
5    $S \leftarrow$  Utilize slack binary tournament selection to select parent solutions;
6    $Q_t \leftarrow$  Utilize the designed genetic operators to generate offspring individuals from  $S$ ;
7    $P_{t+1} \leftarrow$ Utilize the designed approach for environmental selection from  $P_t \cup Q_t$ ;
8    $t \leftarrow t + 1$ ;
9 end
10 Choose the best individual from  $P_t$  and decode it to the corresponding CNN.

```

Algorithm 1 explains how EvoCNN works. The population is initially seeded with the designed flexible strategy of gene encoding (line 1). After that, the process of evolution gets under way and continues on until it reaches a predefined condition for termination, such as the maximum generation number for evolution (lines 3–9). In the end, the individual having the best fitness value is chosen and decoded for one last round of training with the matching CNN (line 10). During the early stages of the evolutionary process, each individual is measured according to the efficient fitness measurement (line 4). After that, the built slack binary tournament selection (line 5)

choose which parent solutions to use, and new offspring are generated by making use of the designed genetic operators (line 6). Following that, representative individuals are chosen from the already existing parent individuals and the offspring individuals generated to comprise the population of the subsequent generation, which will take part in further evolution process (line 7). The following sections provide an in-depth description of the major phases of the Algorithm 1 process.

6.2.2 Strategy of Gene Encoding

As discussed, the convolutional layer, the pooling layer, and the fully connected layer are the three distinct building blocks contained in CNN designs. As a consequence of this, they need to be encoded on one chromosome so that evolution may take place. EvoCNN uses the variable-length gene encoding technique because it is exceptionally well-suited for the task at hand, and due to the fact that it is impossible to determine, prior to verifying the design of CNN, how deep a CNN should go in order to solve a particular problem. Furthermore, because the depths of CNNs have a large impact on their performance [1–4], this variable-length gene encoding technique gives EvoCNN a better chance of achieving the best result because there are no constraints imposed on the architecture search space.

In specifically, Fig. 6.1 depicts an example of three chromosomes of variable length from EvoCNN. The information encoded for a convolutional layer contains the width and height of the filter, the number of feature maps, the width and height of the stride, the type of the convolution, and the mean value and standard deviation of the elements in the filters. The information encoded for a pooling layer are the width and height of the kernel, the width and height of the stride, and the AVERAGE or the MAX. In addition, the number of neurons in the layer, and the mean value and the standard deviation of weights are the information encoded for a fully connected layer. In a convolutional or fully connected layer, hundreds of thousands of connection weights could exist, which cannot all be directly represented by a chromosome and

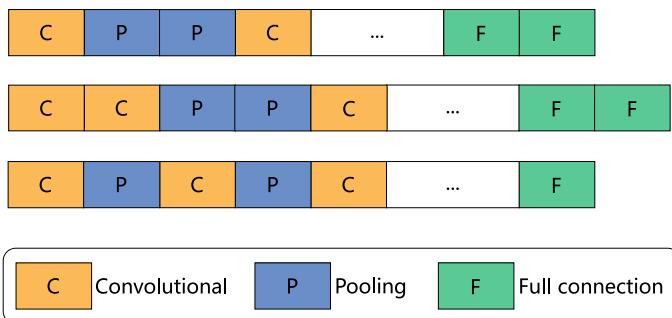


Fig. 6.1 An illustration of three chromosomes with varying lengths in EvoCNN

appropriately optimized by GAs. As a result, To represent the numerous weight parameters in EvoCNN, only two statistical measures in real numbers are employed to regard the connection weights: the standard deviation and the mean value, both of which are conveniently implemented by GAs. Following receipt of the optimal mean value and standard deviation, the connection weights are chosen from the matching Gaussian distribution.

6.2.3 Initialization of Population

For ease of discussions, each chromosome is divided into two parts. The first part consists of convolutional layers and pooling layers, whereas the second part consists of fully connected layers. According to CNN architecture convention, the first part begins with one convolutional layer. Only at the end of the first part may the second part be inserted. Furthermore, the length of each part is determined at random by selecting a number from a preset range.

Algorithm 2: Initialization of Population

Input: The size of population N , the maximal number of convolutional and pooling layers N_{cp} , and the maximal number of fully connected layers N_f .

Output: Initialized population P_0 .

```

1  $P_0 \leftarrow \emptyset;$ 
2 while  $|P_0| \leq N$  do
3    $part_1 \leftarrow \emptyset;$ 
4    $n_{cp} \leftarrow$  Uniformly sample an integer between  $[1, N_{cp}]$ ;
5   while  $|part_1| \leq n_{cp}$  do
6      $r \leftarrow$  Uniformly sample a number between  $[0, 1]$ ;
7     if  $r \leq 0.5$  then
8        $| l \leftarrow$  Initialize a convolutional layer with random settings;
9     else
10       $| l \leftarrow$  Initialize a pooling layer with random settings;
11    end
12     $part_1 \leftarrow part_1 \cup l;$ 
13  end
14   $part_2 \leftarrow \emptyset;$ 
15   $n_f \leftarrow$  Uniformly sample an integer between  $[1, N_f]$ ;
16  while  $|part_2| \leq n_f$  do
17     $| l \leftarrow$  Initialize a fully connected layer with random settings;
18     $| part_2 \leftarrow part_2 \cup l;$ 
19  end
20   $P_0 \leftarrow P_0 \cup (part_1 \cup part_2);$ 
21 end
22 Return  $P_0$ .

```

The primary steps involved in population initialization are shown in Algorithm 2, where $|\cdot|$ represents a cardinality operator, lines 3–13 and 14–19 show the production of the first and second parts of a particular chromosome, respectively. A convolutional layer is added first when initializing the first part. After that, a convolutional or pooling layer is selected and placed at the end using the probability of a single coin toss. This process is repeated until the length of this section meets the criteria that was established in the beginning. After selecting layers that are completely connected, new ones are added for the second phase. It is important to keep in mind that pooling layers, convolutional layers, and fully connected layers are all initialized with random settings. This indicates that the information that is them is selected at random before the model is trained. When these two parts have been finished, they will be combined into one chromosome and given to the user. The same process is used to establish a population consisting of individuals. It is important to take note that the initialized individuals in Algorithm 2 have lengths that vary, as opposed to the minimal lengths that were employed in previous papers [5, 6]. The following are some of the motivations behind this design: (1) CNNs represented by minimal-length individuals are incapable of dealing with the majority of large datasets, (2) it will take an excessive amount of time to evolve the individuals from the minimal lengths to the proper lengths; and (3) the designed mutation operator has functions that can shorten the length of an individual as necessary.

6.2.4 Evaluation of Fitness

Given that EvoCNN is primarily concerned with tasks of image classification, the classification error is the ideal statistic for determining their fitness level. The number of connection weights is chosen as an extra criterion to employ in judging the competence of individuals according to the notion of Occam’s razor [7], which states that complex explanations are preferable. The fitness of each represented CNN is evaluated with the use of a different dataset referred to as $D_{fitness}$ after it has been trained with the conventions on the training set D_{train} . Please keep in mind that the original training set is randomly divided into D_{train} and $D_{fitness}$, with $D_{fitness}$ being invisible to the CNN local training phase via SGD and therefore giving a more accurate measure of generalization ability on the unseen test set.

It is important to note that the test dataset, which is distinct from the D_{train} and $D_{fitness}$ datasets, will not be utilized during this phase of the process. The explanation for this is that the purpose of this stage is only to select the CNN from a group of candidates; it is not intended to determine which candidate has the highest accuracy of classification for the task. Furthermore, there is no reason for concern regarding the overfitting issue. Because D_{train} are used to train the CNNs, whereas $D_{fitness}$ is just used to access the individual fitness.

Due to the typically deep architectures of CNNs, training them to receive the final classification error will inevitably demand a substantial investment in computer resources as well as a lengthy period of time due to the large epoch number of training

Algorithm 3: Evaluation of Fitness

Input: The population P_t , the training epoch number k for measuring the accuracy tendency, the training set D_{train} , the fitness evaluation dataset $D_{fitness}$, and the batch size num_of_batch .

Output: The population with fitness P_t .

```

1 for each individual  $s$  in  $P_t$  do
2    $i \leftarrow 1$ ;
3    $eval\_steps \leftarrow |D_{fitness}|/num\_of\_batch$ ;
4   while  $i \leq k$  do
5     Train the connection weights of the CNN represented by individual  $s$ ;
6     if  $i == k$  then
7        $accy\_list \leftarrow \emptyset$ ;
8        $j \leftarrow 1$ ;
9       while  $j \leq eval\_steps$  do
10          $accy_j \leftarrow$  Evaluate the classification error on the  $j$ -th batch data from
11            $D_{fitness}$ ;
12          $accy\_list \leftarrow accy\_list \cup accy_j$ ;
13          $j \leftarrow j + 1$ ;
14       end
15       Calculate the number of parameters in  $s$ , the mean value and standard deviation
16         from  $accy\_list$ , assign them to individual  $s$ , and update  $s$  from  $P_t$ ;
17       end
18        $i \leftarrow i + 1$ ;
19   end
20 Return  $P_t$ .

```

(complete training in CNNs always covers the epoch number greater than 100). This will make it considerably more difficult here because of the fact GAs are based on population and take into account more than one generation, as it will take each individual a full training in each generation. Consequently, it will not be possible. In order to solve this issue in a sensible way, a solution is came up with. Each individual is trained using a limited epoch number, say five or 10, according to the architectures and the setting of weight initialization. After each individual has been trained, in the final epoch, the mean value as well as the standard deviation in terms of classification error are determined for each batch of $D_{fitness}$. When evaluating a single individual, both the mean value and the standard deviation are taken into consideration. It is self-evident that the individual is superior when the mean value is lower. When the mean values of the individuals being compared are identical, the deviation that uses less standard deviations is the more accurate one.

Mean value, standard deviation, and parameter number are the three indicators that are applied in the process of evaluating fitness. This technique for evaluating the fitness of individual was driven by a number of criteria including: (1) It is sufficient to study simply the performance trend. Individual which perform better in the initial few training epochs of CNNs would most likely perform better in the succeeding training epochs with more confidence. (2) Because the mean value and standard

deviation are useful indications of statistical significance, they are appropriate for examining this trend, and the final classification error can be obtained by optimizing just the best individual evolved using EvoCNN. (3) Manufacturers of smart devices prefer CNN models with fewer connection weights.

6.2.5 Slack Binary Tournament Selection

To pick parent individuals for EvoCNN crossover operations, one slack variation of the standard binary tournament selection is built, which is documented in Algorithm 4. In a nutshell, two sets of comparisons are used. Comparisons between the mean values of individuals involve a *alpha* threshold, while comparisons between parameter numbers involve another *beta* barrier. If the parent solution cannot be found using these comparisons, the individual with the lowest standard deviation is picked.

Algorithm 4: Slack Binary Tournament Selection

Input: Two individuals to be compared, the threshold of mean value α , and the threshold of parameter number β .

Output: The selected individual.

```

1  $s_1 \leftarrow$  The individual having larger mean value;
2  $s_2 \leftarrow$  The other individual;
3  $\mu_1, \mu_2 \leftarrow$  The mean values of  $s_1, s_2$ ;
4  $std_1, std_2 \leftarrow$  The standard deviations of  $s_1, s_2$ ;
5  $c_1, c_2 \leftarrow$  The parameter numbers of  $s_1, s_2$ ;
6 if  $\mu_1 - \mu_2 > \alpha$  then
7   | Return  $s_1$ .
8 else
9   | if  $c_1 - c_2 > \beta$  then
10    |   | Return  $s_2$ .
11    | else
12    |   | if  $std_1 < std_2$  then
13    |   |   | Return  $s_1$ .
14    |   | else if  $std_1 > std_2$  then
15    |   |   | Return  $s_2$ .
16    |   | else
17    |   |   | Return random one from  $\{s_1, s_2\}$ .
18    |   | end
19   | end
20 end
```

In practice, deep CNNs have a large number of parameters, which can easily lead to overfitting. As a result, when the discrepancy between mean values derived by two individuals is less than the α threshold, the weight number is examined. The performance of CNNs is unaffected by minor changes in parameter numbers. As a

result, *beta* is also introduced. Parent solutions are picked and stored in a mating pool by iteratively performing this selection. The mating pool size in EvoCNN is configured to be identical to the size of population.

6.2.6 Offspring Generation

The crossover and mutation operators in GAs produce offspring. Both of these operators contribute to the generally favorable performance of GAs. The following is a rundown of the procedures involved in producing offspring using EvoCNN:

- Step 1: choose two parent solutions at random from the mating pool;
- Step 2: apply the crossover operator to the chosen solutions to generate offspring;
- Step 3: perform the mutation operator on the resulting offspring;
- Step 4: after removing the parent solutions from the mating pool and keeping the offspring, proceed back to Step 1–3 and repeat the process until the mating pool is empty.

Figure 6.2 depicts the crossover procedure. A method termed UA is devised for individual recombination with varied lengths of chromosomes in order to accomplish crossover. The first phase of the crossover procedure is referred to as the unit collection (UC) phase. During this phase, three distinct units, which include the convolutional layer, the pooling layer, and the fully connected layer, are collected first and then separated into three distinct lists based on their positions in the relevant chromosome. The fully connected layer is the unit that has the most connections between its neighbors. After that, the tops of the three lists are connected, and crossovers are performed on the units that are still in the same locations. This part of the process is called the UA and crossover phase. After the crossover process has been completed, the process will continue with the unit restore (UR) phase. During this phase, the units contained in these lists are returned to their original positions on the chromosomes associated with them. Using these three distinct phases in sequence (i.e., the UC, the UA and crossover, and the UR), it is possible for two chromosomes of various lengths to easily exchange gene information and undergo crossover. Because it operates solely on unit lists that contain items of the same type, this UA crossover action is appropriate because they share the same ancestors. The rest units continue to remain the same location, because they are not involved in any crossover activities because they are not paired.

Mutation operations can be performed on every position of the units of a chromosome. A unit could be created, eliminated, or modified for a given mutation point, with a chance of 1/3. In the scenario of unit addition, there is a 1/3 chance of the creation of a convolutional layer, a pooling layer, or a fully connected layer. If the mutation modifies an existing unit, the specific modification will differ depending on the type of unit, and all of the information that is stored in the unit will be updated. EvoCNN uses the SBX [8] and polynomial mutation [9] because of their exceptional

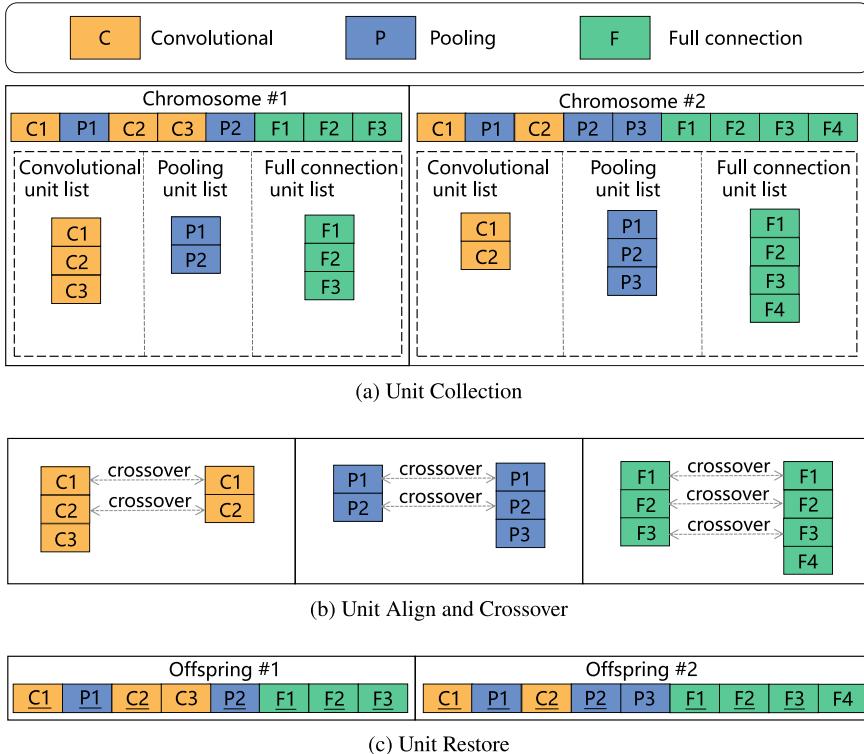


Fig. 6.2 An illustration of the complete crossover procedure

performance in gene representations that use real numbers. This is due to the fact that all encoded forms are denoted by real numbers.

6.2.7 Environmental Selection

Algorithm 5 depicts the environment selection. Elitism and diversity are addressed during the environment selection process. To elaborate, an initial selection is made of a subset of individuals whose mean values appear to have potential, and then the remaining individuals are selected using the modified binary tournament selection method outlined earlier. These two strategies take elitism and diversity into account at the same time, which is likely to increase EvoCNN performance.

Note that the chosen elites are eliminated before the tournament selection begins (line 3 of Algorithm 5), while the individuals picked for diversity are retained in the current population for the next round of tournament selection (lines 4–8 of Algorithm 5) based on community convention.

Algorithm 5: Environment Selection

Input: The elitism fraction γ , and the current population $P_t \cup Q_t$.
Output: The selected population P_{t+1} .

```

1  $a \leftarrow$  Compute the number of elites based on  $\gamma$  and the predefined population size  $N$  from
   Algorithm 13;
2  $P_{t+1} \leftarrow$  Choose  $a$  individuals that have the best mean values from  $P_t \cup Q_t$ ;
3  $P_t \cup Q_t \leftarrow P_t \cup Q_t - P_{t+1}$ ;
4 while  $|P_{t+1}| < N$  do
5    $s_1, s_2 \leftarrow$  Randomly select two individuals from  $P_t \cup Q_t$ ;
6    $s \leftarrow$  Utilize Algorithm 15 to select one individual from  $s_1$  and  $s_2$ ;
7    $P_{t+1} \leftarrow P_{t+1} \cup s$ ;
8 end
9 Return  $P_{t+1}$ .

```

6.2.8 Select and Decode Best Individual

At the end of the evolution process, multiple individuals may have promising mean results but varying architectures and connection weight initialization results. In this situation, there will be a number of different choices available for determining the best Individual. For instance, if the highest performance is only concerned, their architecture configurations can be ignored and pay the attention entirely on the accuracy of their classifications. In any other case, an analogous choice may be made if more of an emphasis on the fewer number of parameters. After determining the best individual, the matching CNN is decoded using the encoded architecture and connection weight initialization information, and the decoded CNN is extensively trained with a larger number of epochs for future use by SGD. Once this has been completed, the matching CNN will be used to generate a prediction for the new data.

6.3 Experimental Design

In order to evaluate how well EvoCNN performs, a number of different experiments are developed, implemented, and executed on certain benchmark datasets for image classification tasks including Fashion and MNIST variants. The results are then contrasted with those obtained from selected state-of-the-art peer rivals. In the following, the chosen peer competitors are first described, and then document the parameter settings.

6.3.1 Peer Competitors

The chosen peer competitors are 2C1P, 2C1P2F+Dropout, 3C2F, GRU+SVM+Dropout, 3C1P2F+Dropout, GoogleNet [10], SqueezeNet-200 [11], AlexNet [12], VGG16 [4] and MLP 256–128–64, which perform the experiments on the Fashion dataset. The peer competitors on other benchmarks are from the literature [13] recently published. There are TIRBM [14], CAE-2 [15], PGBM+DN-1 [16], RandNet-2 [13], ScatNet-2 [17], LDANet-2 [13], PCANet-2 (Softmax) [13], SVM+RBF [18], NNet [18], SAA-3 [18], SVM+Poly [18], and DBN-3 [18].

6.3.2 Parameter Settings

According to the Pareto principle, the proportion of elitism is fixed at 20%. It is important to note that in order to compile the fitness evaluation dataset, 20% of the data from the training images are selected at random. The width and height settings for filter, stride, and kernel are constrained in the EvoCNN implementation. Both the stride width and the height of the convolutional layer are set to one, both the stride width and the stride height of pooling layer are both set to the same value to its kernel, and the “SAME” convolutional type is used. The group of folks working with deep learning uses these parameters quite frequently. Furthermore, the CNN designs evolved in EvoCNN will strictly follow the regular CNN architecture rather than the mixed ones, despite the fact that the mixed ones may perform better, which is beyond the scope of this study.

TensorFlow [19] is utilized in the process of implementing EvoCNN, and each instance of the code is executed in a computer that possesses two GPUs that have the same models of GTX1080. The BN [20] is an exercise that is performed on each individual during the last portion of the training procedure, which aims to accelerate the process and the weight decay while maintaining a consistent number in order to avoid overfitting. Due to the heuristic nature of EvoCNN, 30 different runs are performed on each benchmark dataset. The mean results for the comparisons are then determined, unless something else is specified. In addition, each iteration of the trials takes four days to complete on the Fashion benchmark, whereas the other benchmarks take between two and three days.

6.4 Experimental Results and Discussion

In this section, the experimental results and analysis of EvoCNN in comparison to peer competitors are presented, and then, in the following section, the weight initialization process that is used in EvoCNN is analyzed.

Table 6.1 The classification errors of EvoCNN and chosen peer competitors on Fashion dataset

Classifier	# Parameters	Error(%)	# Epochs
2C1P2F+Drouout	3.27M	8.40(+)	300
3C2F	–	9.30(+)	–
2C1P	100K	7.50(+)	30
GRU+SVM+Dropout	–	10.30(+)	100
3C1P2F+Dropout	7.14M	7.40(+)	150
AlexNet [12]	60M	10.10(+)	–
GoogleNet [10]	101M	6.30(+)	–
MLP 256–128–64	41K	10.00(+)	25
SqueezeNet-200 [11]	500K	10.00(+)	200
VGG16 [4]	6.50(+)		26M200
EvoCNN (best)	6.68M	5.47	100
EvoCNN (mean)	6.52M	7.28	100

6.4.1 Overall Results

Table 6.1 shows experimental findings for the Fashion benchmark dataset, whereas Table 6.2 shows results for the MRD, MB, MBI, MRB, Rectangle, MRDBI, Convex and RI benchmark datasets. The last two rows of Tables 6.1 and 6.2 display the best and mean classification errors obtained using EvoCNN, respectively, while the other rows present the most accurate reports of classification errors made by peer competitors. To reiterate, comparing statistical results is a common practice among those who are interested in EC. On the other hand, the results of the peer competitors are taken directly from the articles in which they were originally published, and statistical results are not included. Indeed, in the deep learning community, only the top results are disclosed. In order to make it easier to examine the comparisons, the terms “+” and “–” have been provided to indicate whether the best result produced by EvoCNN is superior to or inferior to the best result acquired by the comparable peer competitor. This was done in order to facilitate examination of the comparisons. The indication that there are no results that can be accessed from the source is denoted by the symbol “–”. Due to the fact that the vast majority of information regarding the parameter number and training epochs from the chosen peer competitors on Fashion dataset can be accessed, similar information from EvoCNN has also been included in Table 6.1 for comparisons. On the other hand, similar information is not supplied for the benchmarks in Table 6.2 because the peer competitors do not make it available to public. On the benchmarks, the results from all compared algorithms are shown without any utilization of data augmentation or preprocessing. This is done to ensure that the comparison is fair.

Table 6.1 clearly shows that when the best performance is compared, EvoCNN exceeds all ten peer competitors. GoogleNet and VGG16, two state-of-the-art algo-

Table 6.2 The classification errors of EvoCNN and the chosen peer competitors on the benchmarks

Classifier	MRD	MB	MBI	MRB	MRDBI	RI	Convex	Rectangle
TIRBM [14]	4.20(-)	—	—	—	35.50(+)	—	—	—
CAE-2 [15]	9.66(+)	2.48(+)	15.50(+)	10.90(+)	45.23(+)	21.54(+)	—	1.21(+)
ScatNet-2 [17]	7.48(+)	1.27(+)	18.40(+)	12.30(+)	50.48(+)	8.02(+)	6.50(+)	0.01(=)
PGBM+DN-1 [16]	—	—	12.25(+)	6.08(+)	36.76(+)	—	—	—
PCANet-2 (Softmax) [13]	8.52(+)	1.40(+)	11.55(+)	6.85(+)	35.86(+)	13.39(+)	4.19(—)	0.49(+)
RandNet-2 [13]	8.47(+)	1.25(+)	11.65(+)	13.47(+)	43.69(+)	17.00(+)	5.45(+)	0.09(+)
SVM+RBF [18]	11.11(+)	3.03(+)	22.61(+)	14.58(+)	55.18(+)	24.04(+)	19.13(+)	2.15(+)
LDANet-2 [13]	7.52(+)	1.05(—)	6.81(+)	38.54(+)	12.42(+)	16.20(+)	7.22(+)	0.14(+)
NNet [18]	18.11(+)	4.69(+)	20.04(+)	62.16(+)	27.41(+)	33.20(+)	32.25(+)	7.16(+)
SVM+Poly [18]	15.42(+)	3.69(+)	16.62(+)	56.41(+)	24.01(+)	24.05(+)	19.82(+)	2.15(+)
DBN-3 [18]	10.30(+)	3.11(+)	6.73(+)	47.39(+)	16.31(+)	22.50(+)	18.63(+)	2.61(+)
SAA-3 [18]	10.30(+)	3.46(+)	23.00(+)	11.28(+)	51.93(+)	24.05(+)	18.41(+)	2.41(+)
EvoCNN (best)	5.22	1.18	4.53	2.80	35.03	5.03	4.82	0.01
EvoCNN (mean)	5.46	1.28	4.62	3.59	37.38	5.97	5.39	0.01

rithms, achieve classification error rates of 6.5% and 6.3%, respectively, with the difference being only 0.2%. EvoCNN lowers the error rate by 0.83% to 5.47%. In addition to this, the mean effectiveness of EvoCNN is superior to that of the best eight competitors, and it is only slightly lower than that of the best VGG16 and GoogleNet models. Furthermore, when compared to GoogleNet, which has 101 million connection weights, and VGG, which has 26 million connection weights, EvoCNN has significantly smaller number of connection weights at 6.52 million. Additionally, EvoCNN employs only fifty percent of the epoch number that VGG16 does. According to the findings on Fashion benchmark, the results show that EvoCNN provides higher performance in the designing architectures and initializing connection weights for CNNs.

The results of Table 6.2 demonstrate that out of the 13 options, EvoCNN is the most effective one. On MRB, MBI, MRDBI, Rectangle, RI datasets, EvoCNN achieves the greatest performance, and on the MRD, Convex, and MB datasets, TIRBM, PCANet-2 (Softmax), and LDANet-2 gain the best classification accuracy, while EvoCNN achieves the second best performance on these three datasets. In addition, when the mean performance of EvoCNN is compared to the best of the other 12 methods, EvoCNN also exhibits the best results on MBI, MRB, Rectangle, and RI, the second best results on Convex and MRD, the third and fourth best results on MB and MRDBI. On MBI and MRB, the classification error rates that are the lowest for the others are 11.5% and 6.08%, respectively, while the mean error rates for EvoCNN are 4.62% and 3.59%, respectively. In conclusion, the best classification error achieved by EvoCNN emerges victorious in 80 out of 84 trials when compared against the best results achieved by 12 peer competitors. The mean classification

error of EvoCNN is lower on 75 out of the 84 comparisons, making it better than the best error of any of the other 12 methods.

6.4.2 Performance of Weight Initialization

More experiments are being carried out to test the efficacy of the initialization method for connection weight designed in EvoCNN. Alternative weight initializers will be compared to see if the architecture or initialization values of connection weight effect the performance of classification. To accomplish this, a second set of CNNs with the identical architectures to the evolved EvoCNN are created, but their weight values are created with the commonly used Xavier initializer [21].

The comparisons are illustrated in Fig. 6.3, where the horizon axis represents the benchmark datasets and the vertical axis represents the classification error rates. Figure 6.3 demonstrates conclusively that the provided connection weight initialization technique in EvoCNN outperforms the widely utilized Xavier initializer in all of the benchmarks. To be more specific, the designed weight initialization technique improves classification accuracy by roughly 1.5% on the MB, Fashion, MBI, MRD, Convex, and RI benchmarks, and by 4.5% on the MRDBI benchmark. When the results in Tables 6.1 and 6.2 are compared, it is likewise possible to draw the conclusion that CNN designs contribute more to classification performance than connection weight initialization does. By automatically designing both the initial connection weights and CNN designs, EvoCNN is able to reach results that are rather encouraging.

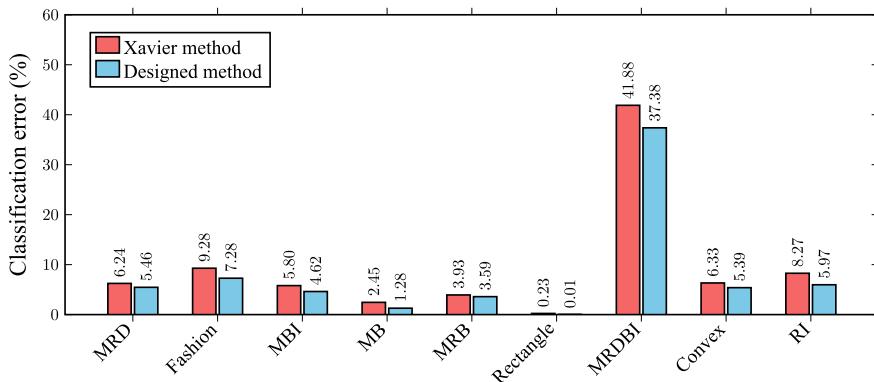


Fig. 6.3 Performance comparison between different weight initialization methods

6.4.3 *Discussion*

The core components designed in EvoCNN and the results of experiments will be detailed in this section, which may provide useful insights into the uses of EvoCNN.

The exploitation search, also known as the local search, is carried out by crossover operators in GAs, whereas the exploration search, also known as the global search, is carried out by mutation operators. Only well-designed iterations of both local and global search will successfully boost performance. This is because they should work together to complement each other. Crossover operators, on the other hand, work on chromosomes that are all the same length, whereas EvoCNN uses chromosomes of varying lengths. Furthermore, CNNs have numerous different fundamental layers, which makes constructing crossover operations in this context more complicated. This is due to the difficulty of performing crossover operations between genes from diverse origins. As a direct consequence of this development, EvoCNN possesses a whole new crossover function. In order to complete the crossover operation, UC, UA, and UR are developed. This enhances communications between encoded information and, as can be assumed, improves performance when searching for suitable CNN architectures.

The knowledge about gradients is utilized in the bulk of the procedures that are taken in order to optimize the weights of CNNs. It is a well-known fact that GD-based optimization methods are sensitive to the beginning location of the variables that are intended to be optimized. Without an appropriate starting position, algorithms that are based on gradients can easily become trapped in local minima. It would appear that using GAs to determine a better beginning place for the connection weights would be impossible. This is because of the high number of parameters, which cannot effectively represent a chromosomes through encoding nor can they be optimized to their full potential. EvoCNN uses an indirect encoding method, which encodes the weights in each layer terms of the mean values and standard deviations. This allows the network to learn more effectively. This design in EvoCNN was developed because, in reality, using the standard deviation as well as the mean to define the initial point of weights is a flawed method, which could be easily observed in open-source libraries of deep learning [19, 22, 23]. As a result, this design in EvoCNN was developed. This method requires only two real numbers to be used in order to accurately represent hundreds of thousands of parameters. As a result, the amount of computer resources needed for encoding and optimization could be significantly reduced.

The currently available methods for finding CNN architectures often regard the fitness of an individual fitness to be equivalent to their final classification accuracy. The process of achieving such a accuracy frequently needs many more epochs for training, which is a lengthy undertaking. It is natural to use a large amount of computer resources to accomplish the architecture design using this fitness evaluation approach in order to accelerate the design. Unfortunately, not every interested researcher has access to computational resources. In practice, a propensity of each individual that may anticipate future quality would suffice, and the final classification accuracy

would not be required. As a result, the limited epoch number are used for the training of these individuals. EvoCNN does not rely heavily on computational resources while calculating fitness. In conclusion, researchers with limited domain knowledge could create viable CNN architectures for handling particular problems in their (academic) environment, where computational resources are typically constrained, by using well-designed yet basic method for fitness evaluation and strategy for gene encoding.

Furthermore, when EvoCNN ends, surprising findings have been revealed, i.e., numerous individuals with identical performance but considerably varying numbers of connection weights, owing to the population-based character of EvoCNN. For example, on Fashion dataset, the architectures having similar classification accuracy values of 99, 98, 97, 96, and 95% are with the significantly parameter numbers of 569,250, 98,588, 7,035, 3,205, and 955, respectively. Recent years have seen the development of a wide range of applications, the most notable of which are the self-driving automobile and a number of fascinating real-time applications on mobile devices. Because of the restricted processing power and battery life of mobile devices, trained CNNs with the same end result yet fewer parameters are significantly more preferred, and trained CNNs require significantly less computing resource and energy consumption. In addition, the performance of CNNs is not affected by the length of the basic layers, and there are additional pieces of hardware that have been developed specifically to accelerate the calculations that are performed by CNNs. One example of this is hardware that is dedicated specifically to convolutional operations. In this sense, EvoCNN could provide device manufacturers a higher number of possibilities from which to choose in order to make judgments based on their own particular interests. If standard operating procedures are followed, it can be difficult to locate such models in a single search.

6.5 Chapter Summary

In this chapter, we introduced the automatic architecture design algorithm based on GAs for the plain CNN directly constructed by convolutional layers, pooling layers, and fully connected layers. In addition, the optimal weight initialization methods in terms of the two key parameters of Gaussian distribution are also considered in the designed algorithm for better weight initialization. Because the popular fixed-length encoding methods designed in the standard GAs cannot adapt to the unknown best depth of the CNNs, the variable-length encoding strategy, as well as the corresponding crossover operators, are additionally designed. The experiments on image classification tasks have demonstrated the effectiveness of the designed algorithm.

However, automatically designing CNN architectures directly based on convolutional layers, pooling layers, and the fully connected layers often result in the CNNs with poor performance, although the algorithm design is easy to understand and implement. In the next chapter, we will introduce the automatic CNN architecture design based on the building blocks of state-of-the-art CNNs, which could balance well the algorithm efficiency as well as the algorithm effectiveness.

References

1. Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1798–1828.
2. Bengio, Y., & Delalleau, O. (2011). On the expressive power of deep architectures. In *International Conference on Algorithmic Learning Theory* (pp. 18–36). Springer.
3. Delalleau, O., & Bengio, Y. (2011). Shallow vs. deep sum-product networks. In *Advances in neural information processing systems* (pp. 666–674).
4. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
5. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).
6. Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2), 99–127.
7. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, 24(6), 377–380.
8. Deb, K., & Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. *Complex Systems*, 9(2), 115–148.
9. Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms* (Vol. 16). Wiley.
10. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).
11. Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., Keutzer, K. (2016). SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size. [arXiv:1602.07360](https://arxiv.org/abs/1602.07360).
12. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), 84–90. <https://doi.org/10.1145/3065386>.
13. Chan, T.-H., Jia, K., Gao, S., Jiwen, L., Zeng, Z., & Ma, Yi. (2015). Pcanet: A simple deep learning baseline for image classification? *IEEE Transactions on Image Processing*, 24(12), 5017–5032.
14. Sohn, K., & Lee, H., (2012) Learning invariant representations with local transformations. <https://arxiv.org/abs/1206.6418>.
15. Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning* (pp. 833–840).
16. Sohn K., Zhou G., Lee C., & Lee, H. (2013). Learning and selecting features jointly with point-wise gated boltzmann machines. <https://dl.acm.org/citation.cfm?id=3042918>.
17. Bruna, J., & Mallat, S. (2013). Invariant scattering convolution networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1872–1886. <https://doi.org/10.1109/tpami.2012.230>.
18. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning* (pp. 473–480). ACM. <https://doi.org/10.1145/1273496.1273556>.
19. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. [arXiv:1603.04467](https://arxiv.org/abs/1603.04467).
20. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, Volume 37 of Proceedings of Machine Learning Research* (pp. 448–456). PMLR.

21. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
22. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., & Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (pp. 675–678). ACM.
23. Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., & Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. [arXiv:1512.01274](https://arxiv.org/abs/1512.01274).



7.1 Introduction

As mentioned in Part III, the research of CNN architectural design algorithms is now in the early phases, particularly for entirely automatic ones with great performance and using limited CPU resources. In this chapter, a novel GA-based method to automatically design CNN architectures named AE-CNN will be introduced with the following highlights. Firstly, the use of AE-CNN has no pre-conditions on any prior knowledge of basic CNN design, examined datasets, GA, pre-processing, re-composition, and post-processing. Secondly, the variable-length encoding technique is used to estimate the best CNN depth. The new operators for crossover and mutation are invented and included in for exploring and exploiting the search space in discovering the ideal CNN architectures in order to achieve variable-length encoding. Thirdly, for accelerating the architecture design, an efficient encoding approach according to RBs and DBs are designed, and limited computational resources are used, while AE-CNN achieves promising results. It is worth noting that, while the RBs and DBs are employed in AE-CNN, users are no need to be familiar with these blocks in order to use it.

The rest of the chapter will be organized as follows: Sect. 7.2 describes AE-CNN in detail. The experimental design and the experimental results are provided in Sects. 7.3 and 7.4, respectively, for the purpose of evaluating the performance of AE-CNN.

7.2 Algorithm Details

This section focuses on the framework of AE-CNN and its essential parts, which is designed also for image classification problems.

7.2.1 Algorithm Overview

Algorithm 1: The AE-CNN Framework

Input: The size of population N , the number of maximal generation T , the probability of crossover μ , the probability of mutation ν .

Output: The best CNN.

```

1  $P_0 \leftarrow$  Initialize a population having size of  $N$  with the encoding strategy;
2 Evaluate the fitness value of each individual in  $P_0$ ;
3  $t \leftarrow 0$ ;
4 while  $t < T$  do
5    $Q_t \leftarrow \emptyset$ ;
6   while  $|Q_t| < N$  do
7      $p_1, p_2 \leftarrow$  Using binary tournament selection to choose two parents from  $P_t$ ;
8      $q_1, q_2 \leftarrow$  Generate two offspring from  $p_1$  and  $p_2$  with  $\mu$  crossover  $\nu$  mutation;
9      $Q_t \leftarrow Q_t \cup q_1 \cup q_2$ ;
10  end
11  Evaluate the fitness value of each individual in  $Q_t$ ;
12   $P_{t+1} \leftarrow$  Choose  $N$  individuals with environmental selection from  $P_t \cup Q_t$  ;
13   $t \leftarrow t + 1$ ;
14 end
15 Choose the optimal individual from  $P_t$  and decode it to the corresponding CNN.

```

AE-CNN is designed with three parts as shown in Algorithm 1. To begin, the population is given a random size of N (line 1). After that, the fitness of the individuals is evaluated (line 2). Then, with the maximum generation number of T (lines 3–14), the GA involves all individuals in the population. In the end, the optimal individual selected from the last population on the basis of fitness is used to decode the optimal CNN architecture (line 15). An empty population is created for offspring throughout the evolutionary process, (line 5), then, utilizing crossover and mutation operations, new offspring are created from chosen parents, the parents, on the other hand, are chosen using a binary tournament selection (lines 6–10); following an assessment of the fitness of the offspring produced (line 11), new individuals are chosen from the current population containing the present individuals as well as the offspring produced using the environment selection operation (line 12) as the parent solutions that will survive into the next the next generation. On line 6, the symbol for $|\cdot|$ is a cardinality operator. Sections 7.2.2, 7.2.3, 7.2.4 to 7.2.5 will detail the stages of population initialization, fitness evaluation, offspring generation and environment selection, respectively.

7.2.2 Population Initialization

For the following evolutionary process, population initialization generates a basic population with many individuals. Individuals in GAs are typically initialized at random with a uniform distribution, and each individual indicates a potential result to the issue that has to be resolved. Since GAs are used to discover the optimum CNN architecture in the algorithm, a CNN architecture should be represented by each individual in the algorithm. The architecture of a CNN is made up of several pooling layers, convolutional layers, and fully connected layers in a certain sequence, along with their parameter values. CNNs are stacked utilizing RBs, DBs, and pooling layers in the algorithm, which is driven by the extraordinary performance of ResNet [1] and DenseNet [2], however fully connected layers are not taken into account. Because of their full-connection nature, overfitting may be readily caused by fully connected layers [3]. To reduce this issue, other strategies, such as Dropout [4], could be utilized. However, these strategies will introduce additional factors, so that must be carefully tuned to prevent increasing the computing complexity of the algorithm. The experimental results presented in Sect. 7.4 demonstrate that the promising performance of the algorithm may be obtained without requiring fully connected layers. Algorithm 2 summarizes the specifics of AE-CNN population initialization.

Algorithm 2: Initialize Population

Input: The population size N , the training instance dimension $d \times d$.

Output: The initialized population P_0 .

```

1  $P_0 \leftarrow \emptyset$ ;
2  $m_p \leftarrow$  Calculate the maximum number of pooling layers by  $\lfloor \log_2(d) \rfloor$ ;
3 for  $i \leftarrow 1$  to  $N$  do
4    $k \leftarrow$  Initialize a positive integer at random;
5    $a \leftarrow$  Initialize an empty array with the size of  $k$ ;
6   for  $j \leftarrow 1$  to  $k$  do
7      $u \leftarrow$  Select one from {RBU, DBU, PU} at random;
8     if  $u$  is a PU and the number of used PU is not less than  $m_p$  then
9       |  $u \leftarrow$  Select one from {RBU, DBU} at random;
10      end
11      Encode  $u$  and put the encoded information into the  $j$ -th position of  $a$ ;
12    end
13     $P_0 \leftarrow P_0 \cup a$ ;
14 end
15 Return  $P_0$ .

```

Then, the specifics of lines 8 and 11 are detailed since the rest of Algorithm 2 are direct. In particular, CNN pooling layers perform dimension reduction upon their input data, with the most typical pooling operation halving the input size, as shown in state-of-the-art CNNs [1, 2, 5–8]. Finally, the pooling layers used cannot be chosen at random, but it must adhere to the limitation calculated in line 2. For instance, if the input data is 32×32 , the amount of pooling layers employed cannot exceed six,

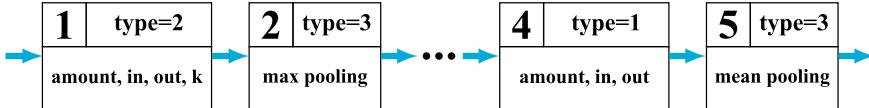


Fig. 7.1 An example of the encoding strategy

since six pooling layers lower the input data dimension to 1×1 , and on the 1×1 dimension, an additional pooling layer generates a logic error.

Encoding provides GAs with the capacity to mimic real-world problems, which may subsequently be solved directly by the GAs. The corresponding encoding strategy, which is the initial stage in using GAs, is used to encode information. There is no one-size-fits-all encoding approach that may be applied to all issues. A new encoding approach are designed for effectively modeling CNNs with various architectures in the algorithm. For the utilized RBs, the filter size of *conv2* is set to 3×3 , which is also utilized for the convolutional layers in the utilized DBs, based on the configuration of state-of-the-art CNNs [1, 9]. According to conventions, the stride of the employed pooling layers are set to 2×2 , which indicates that in the evolved CNN, a single pooling layer halves the input dimension for a single time. Finally, input and output spatial sizes are the unspecified parameter settings for RBs, those for DBs are the input and output spatial sizes, as well as k , and the unknown parameter settings for pooling layers are merely their types, for example, the MAX and AVERAGE. The amount of convolutional layers in a DB is recognized since it may be calculated from the input and output spatial sizes, as well as k . As a result, the encoding approach is according to three different kinds of units and their relative positions in CNNs. RB unit (RBU), DB unit (DBU), and pooling layer unit (PU) are the three units. An RBU and a DBU, for example, each include many RBs and DBs, whereas a PU only has one pooling layer. The justifications are that: (1) by inserting several RBs or DBs into an RBU or a DBU, the depth of the CNN may be adjusted greatly compared to stacking RBs or DBs one at a time, hence accelerating the heuristic search of the algorithm by simply modifying the depth of CNN; and (2) numerous pooling layers are less flexible than just one in PU, because numerous PUs can be stacked to achieve the result of multiple successive pooling layers. In addition, for the sake of the implementation of algorithm, one parameter is added to indicate the type of unit. The type, the amount of RBs, the input spatial size, and the output spatial size, which are represented as *type*, *amount*, *in*, and *out*, respectively, comprise the encoded information for an RBU. The encoded information for a DBU, on the other hand, is identical to that of an RBU, with the exception of the added parameter k . In a PU, encoding the pooling type requires only one parameter.

The algorithm is demonstrated in Fig. 7.1 encoding a CNN include nine units. Each of the numbers in the upper-left corner of the block represents the position of a CNN unit. If the *type* is 1, 2, or 3, the unit is either an RBU, a DBU, or a PU. Notably, the encoding approach does not restrict the maximum length of any

individual, the algorithm may use the supplied variable-length encoding approach to adaptively determine the optimal CNN architecture with the correct depth.

7.2.3 Fitness Evaluation

Individual fitness is computed according to the information encoded by these individuals and the problem at hand, and provides a quantifiable measure of how effectively they adapt to the environment. Individual fitness in AE-CNN is determined by the classification accuracy according to the architecture encoded by the individual and the associated fitness evaluation data. An individual having a higher fitness contains a greater probability of producing offspring having better fitness beyond themselves, according to EAs. In AE-CNN, each individual is decoded to the relevant CNN for fitness evaluation and then added to a classifier to be trained as a traditional CNN. The logistic regression for binary classification and the Softmax regression for multiple classification are the most commonly used classifiers. The decoded CNN is trained on the training data in AE-CNN, and the fitness is the optimal classification accuracy on the fitness evaluation data after CNN training.

Algorithm 3: Evaluate Fitness

Input: The population P_t for fitness evaluation, training data \mathcal{D}_{train} , fitness evaluation data

$\mathcal{D}_{fitness}$.

Output: The population P_t with fitness.

```

1 for each individual in  $P_t$  do
2    $cnn \leftarrow$  Transform the information encoded in individual to a CNN with the
     corresponding architecture;
3   Initialize the weights of  $cnn$ ;
4   Train  $cnn$  on  $\mathcal{D}_{train}$ ;
5    $acc \leftarrow$  Evaluate the classification accuracy of the trained  $cnn$  on  $\mathcal{D}_{fitness}$ ;
6   Assign  $acc$  as the fitness of individual;
7 end
8 Return  $P_t$ .

```

In Algorithm 3, the fitness of the algorithm is evaluated in the same way for each individual in the population. To begin, the architecture information of individual is converted into a CNN with the appropriate architecture (line 2), which is the opposite of the encoding approach described in Sect. 7.2.2. After that, the CNN is trained using the given training data (line 4) after being initialized with weights (lines 3) similar to a hand-crafted CNN. Note that the weight initialize approach and the training approach are, respectively, the Xavier initializer [10] and the SGD with momentum, that are widely utilized in the deep learning field. In the end, utilizing fitness evaluation data, the trained CNN is evaluated (line 5), and the evaluated classification accuracy is utilized to measure the fitness of the individual (line 6).

7.2.4 Offspring Generation

Parent individuals must be chosen ahead of time to establish a population of offspring. The generated offspring are expected to get higher fitness beyond their parents, according to the fundamental concept of evolutionary approaches, because they inherit the qualities from both parents. As a result, the individuals with the optimal fitness should be selected as the parent individuals. Nevertheless, selecting the optimal ones as parents can simply lead to a reduction in population diversity, which result in early convergence [11, 12] and, as a consequence, the maximum performance of population cannot be reached [13, 14] because of the trap in local minima [15, 16]. A general solution to this problem is to choose promising parents at random. Utilizing the binary tournament selection [17] for this purpose [17, 18] in AE-CNN, that is based on GA community conventions. The binary tournament picks two individuals at random from the population, with the one having the highest fitness being selected as one of the parent individuals. This process is repeated to choose another parent individual, and the crossover operation is performed by these two parent individuals. Noting that each crossover operation generates two offspring, and each generation produces N offspring, the crossover operation is executed $N/2$ times throughout each generation, where N represents the population size.

Algorithm 4: Crossover Operation of AE-CNN

Input: Two parent individuals, p_1 and p_2 , chosen according to the binary tournament selection, crossover probability μ .

Output: Two offspring solutions.

```

1  $r \leftarrow$  Uniformly generate a number from  $[0, 1]$ ;
2 if  $r < \mu$  then
3   Select a position from  $p_1$  and  $p_2$  at random, respectively;
4   Separate  $p_1$  and  $p_2$  according to the selected positions;
5    $q_1 \leftarrow$  Combine the first part of  $p_1$  and the second part of  $p_2$ ;
6    $q_2 \leftarrow$  Combine the first part of  $p_2$  and the first part of  $p_1$ ;
7 else
8    $q_1 \leftarrow p_1$ ;
9    $q_2 \leftarrow p_2$ ;
10 end
11 Return  $q_1$  and  $q_2$ .

```

Biologically evident, the crossover operation in conventional GAs is conducted on two individuals of the identical length. Individuals in the algorithm have varied lengths based on the encoding strategy, that is, the corresponding CNNs have varying depths. The traditional crossover operator cannot be utilized in this case. The crossover operator, on the other hand, usually relates to local search capability of GAs, which exploits the search space for an impressive performance. Due to the lack of a crossover operation in GAs, the final performance of solution may be harmed. The one-point crossover operator is used in the algorithm. This is due to the fact that

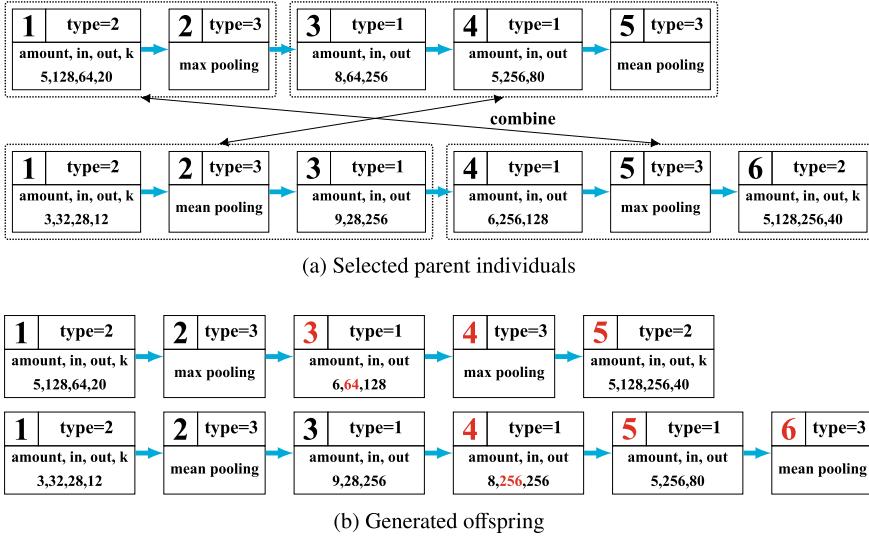


Fig. 7.2 The two selected parent individuals for the crossover operation (shown in Fig. 7.2a) and the generated offspring (shown in Fig. 7.2b). The numbers in each block reflect the corresponding configuration, while the red numbers in Fig. 7.2b indicate the required adjustments after the crossover operation

in GP [19], the one-point crossover has been broadly employed. GP is another essential form of EA, and its individuals commonly have varying lengths. Algorithm 4 depicts the crossover operation of algorithm.

Note that certain essential modifications are applied to the generated offspring automatically. For instance, the *in* of present unit ought to match the *out* of prior unit, as well as any further cascade adjustments generated by this change. An example of the crossover operation is illustrated in Fig. 7.2, with the two parent individuals displayed in Fig. 7.2a. If the separation positions of these two parent individuals are the 3rd and 4th units, after that Fig. 7.2b displays the produced offspring, with the red values indicating the relevant adjustments required for the representation of logic in a valid CNN following the crossover operation.

In most cases, the mutation operation conducts a global search in GAs. It operates on a single produced offspring with a predetermined probability and mutation types allowed. The encoding strategy is used to design the mutation kinds that are available. The following mutation types are possible in the algorithm:

- Adding (adding an RBU, adding a DBU, or adding a PU to the selected position);
- Removing (removing the unit at the selected position);
- Modifying (modifying the encoded information of the unit at the selected position).

Algorithm 5 describes the mutation operation in the algorithm. Since all of the produced offspring employ the identical mutation operation, Algorithm 5 simply displays the process of one offspring for simplicity. Note that if the offspring are not

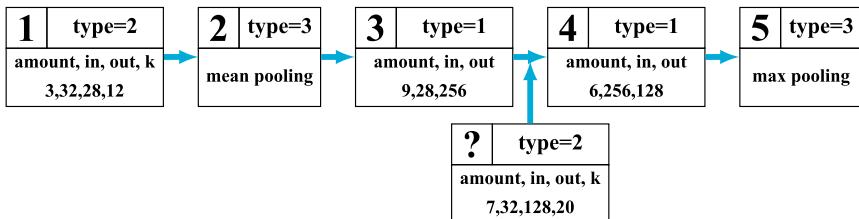
Algorithm 5: Mutation Operation of AE-CNN

Input: The offspring q_1 , mutation probability ν .
Output: The mutated offspring.

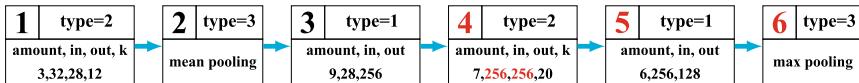
```

1  $r \leftarrow$  Uniformly generate a number from  $[0, 1]$ ;
2 if  $r < \nu$  then
3   Select a position from  $q_1$  at random;
4    $type \leftarrow$  Choose one from {Adding, Removing, Modifying} at random;
5   if  $type$  is Adding then
6     |  $mu \leftarrow$  Choose one from {adding an RBU, adding a DBU, adding a PU} at random;
7   else if  $type$  is Removing then
8     |  $mu \leftarrow$  removing a unit;
9   else
10    |  $mu \leftarrow$  modifying the encoded information;
11   end
12   Perform  $mu$  at the selected position;
13 Return  $q_1$ .

```



(a) The selected individual for mutation and the initialized RBU at random for the corresponding mutation



(b) Mutated individual

Fig. 7.3 An example of the “adding an RBU” mutation. Specifically, the first row and the second row in Fig. 7.3a indicate the chosen individual for the mutation and the initialized RBU at random for the “adding an RBU” mutation at the fourth position of the individual to be mutated. Figure 7.3b shows the mutated individual, and the red numbers represent the required adjustments after the mutation

modified, they will stay the same. In addition, according to the logic of composing a valid CNN as emphasized in the crossover operation, a sequence of necessary modifications will be made automatically. To get a deeper comprehension of the mutation, an instance of “adding an RBU” is provided in Fig. 7.3, where Fig. 7.3a depicts the selected individual for the mutation and the RBU which is initialized at random, and Fig. 7.3b displays the mutated individual. The red numbers in Fig. 7.3b also represent the essential changes after the mutation. All of these necessary adjustments are implemented automatically in the crossover and mutation operations.

7.2.5 Environmental Selection

Algorithm 6: Environmental Selection

Input: The population P_t , the generated offspring population Q_t , the population size N .
Output: The population P_{t+1} surviving in the next generation.

```

1  $P_{t+1} \leftarrow \emptyset;$ 
2 for  $j \leftarrow 1$  to  $N$  do
3    $p_1, p_2 \leftarrow$  Choose two individuals from  $P_t \cup Q_t$  at random;
4    $p \leftarrow$  Choose the one with higher fitness from  $\{p_1, p_2\}$ ;
5    $P_{t+1} \leftarrow P_{t+1} \cup p;$ 
6 end
7  $p_{best} \leftarrow$  Choose the one with the highest fitness from  $P_t \cup Q_t$ ;
8 if  $p_{best}$  is not in  $P_{t+1}$  then
9   | Choose one from  $P_{t+1}$  at random and then replace it by  $p_{best}$ ;
10 end
11 Return  $P_{t+1}$ .
  
```

A population of N individuals will be selected from the present population, i.e., $P_t \cup Q_t$, for the environmental selection to serve as the parents for the following generation. To avoid being trapped in local minima [15, 16] and premature convergence [11, 12], a good population should have both convergence and diversity [20]. Actually, the parent individuals should include those with the highest fitness for convergence and those with significant fitness differences from one another for diversity. To achieve this, the individual with the highest fitness will be chosen, as well as $N - 1$ individuals chosen using binary tournament selection [17, 18], as parent individuals who will produce offspring for the new population. Using the elitism mechanism [21] in GAs to explicitly select the optimal one to be the parent for the next generation could avoid the performance of population from deteriorating as evolution progresses.

The environmental selection of the algorithm is described in detail in Algorithm 6. With the binary tournament selection shown in lines 2–6, N individuals are selected from the current population P_t as well as the produced offspring population Q_t . The best individual p_{best} who has the best fitness is then chosen from $P_t \cup Q_t$ (line 7), followed by a check to see if p_{best} has been chosen for P_{t+1} . If p_{best} does not exist in P_{t+1} , it will be substituted by a random one chosen from P_{t+1} . Since the binary tournament selection is according to fitness, the offspring in Q_t should have been evaluated for fitness before environmental selection.

7.3 Experimental Design

The aim of the experiment is to see whether AE-CNN can obtain promising results on image classification problems. In this part, the peer competitors against whom the performance of the algorithm will be compared are explained in Sect. 7.3.1, and then highlight the benchmark datasets and parameter settings in Sect. 7.3.2.

7.3.1 Peer Competitors

A number of peer competitors are picked to conduct the comparison to illustrate the advantage of AE-CNN. The chosen peer competitors, in particular, can be split into three categories.

DenseNet [2], ResNet [1], Maxout [22], VGG [23], Network in Network [24], Highway Network [25], All-CNN [26] and FractalNet [27] are the state-of-the-art CNNs whose architectures have been hand-crafted by domain experts. Furthermore, because of prospective performance of ResNet, two distinct versions are used in the experiment: ResNet having 101 layers and ResNet having 1,202 layers, which are referred to as ResNet (depth = 101) and ResNet (depth = 1,202), respectively. The majority of peer competitors in this class are the championship of the large-scale vision challenge [28] in latest years due to their promising performance. The purpose of choosing these CNNs is to see whether the algorithm for automatically designing CNN architectures can outperform hand-crafted CNNs. The second category discusses semi-automated CNN architecture design algorithms, such as Genetic CNN [29], Hierarchical Evolution [30], EAS [31], and Block-QNN-S [32]. The third category includes algorithms such as Large-scale Evolution [33], CGP-CNN [34], NAS [35], and MetaQNN [36], which can design CNN architectures entirely automatically.

7.3.2 Parameter Settings

The results given by peer competitors are analyzed in their seminal papers. The reason for this is that the results that are reported are almost always the best. It is not necessary to set the parameters of peer competitors in this way. The principle of setting all parameters in the algorithm according to their generally utilized values in the algorithm is adhered to make it easier for researchers who want to utilize the algorithm to find the optimal CNN architectures for their researched data, even if they do not have any experience with GAs. The population size and maximum generation number are both set to 20, while the crossover and mutation probabilities are set at 0.9 and 0.2, accordingly. With a proportion of 1/5, the fitness evaluation

data is separated at random from the training data. To the end, all of the classification error rates are compared using the identical test data.

Each individual is trained using SGD with a batch size of 128 to evaluate their fitness. The parameter settings for SGD are influenced by traditions of peer competitors as well. In this case, the momentum is fixed at 0.9. The learning rate is set at 0.01 at the start, but warms up to 0.1 from the second to the 150th epoch, and is scaled by splitting 10 at the 250-th epoch. The weight decay is fixed at 5×10^{-4} . Furthermore, if the individual is out of memory during the training, the fitness will reset to zero. When the process of evolutionary is finished, the optimal individual is trained again using the same SGD settings on the original training data, as well as the error rate on the test data is presented for comparison. The optimal individual is trained for five independent runs due to the heuristic nature of the algorithm and the expensive computing cost. Since all of the peer competitors used in the comparisons simply indicate their optimal results regardless of the number of times they have competed, the optimal result of the algorithm from the five independent tests is offered here for a fair comparison.

Furthermore, CIFAR-10 and CIFAR-100 are enhanced by padding four zeros to either side of one image, then cropping it to its original size and flipping it horizontally at random, before being input to the algorithm, according to the traditions of the chosen peer competitors [1, 2, 29–36].

In addition, the maximum convolutional layers in a DB are given as 10 when $k = 12$ and $k = 20$ and 5 when $k = 40$, depending on the design of DenseNet. In a CNN, the maximum number of RBUs and DBUs is 4. In a DBU and an RBU, the number of DBs and RBs is specified between three and ten. It is worth noting that these options are primarily according to the computational resources, as any number higher than these will quickly run out of memory. If the computational platform of user has more powerful GPUs, the number may be changed to any value. The experiment is run on three Nvidia GeForce GTX 1080Ti GPU cards, using the codes written in PyTorch [37].

7.4 Experimental Results and Analysis

For an in-depth comparison to the selected peer competitors, the performance of the algorithm is analyzed that taking into account not only the classification error but also the amount of parameters and computing complexity in the experiments shown in Sect. 7.4.1. Because it is difficult to mathematically examine computational complexity of each peer competitor, the consumed “GPU Days” are utilized as a proxy for computational complexity. The amount of GPU Days is calculated by multiplying the number of GPU cards in use by the number of days the algorithms took to identify the optimum architectures. For the CIFAR-10 dataset, for example, the algorithm ran for nine days on three GPU cards, resulting in a GPU Days value of 27 when multiplying nine days by three used GPU cards. Therefore, the hand-crafted state-of-the-art CNNs do not contain data on the GPU Days. On the chosen

benchmark datasets, the evolution trajectories of the algorithm is also provided in determining the optimal architectures, which may help readers decide if the algorithm converges with the parameter values employed (shown in Sect. 7.4.2). To the end, in Sect. 7.4.3, the best architectures are presented, which may be beneficial information for researchers hand-crafting CNN architectures.

7.4.1 Performance Overview

The experimental results of AE-CNN, in addition to the selected peer competitors, are shown in Table 7.1. To make it simpler to compare the results, Table 7.1 is split into five rows by six horizontal lines. The first row indicates the column titles, while the second, third, and fourth rows relate to state-of-the-art peer competitors whose architectures are designed by hand, semi-automatically, and automatically, respectively. The result of the algorithm are shown in the fifth row, which is an automated algorithm for designing CNN architectures. In addition, the symbol “–” in Table 7.1 denotes that the peer competitor has not publicly released any results.

AE-CNN surpasses all state-of-the-art peer competitors designed by hand for CIFAR-10, as demonstrated in Table 7.1. AE-CNN has a classification error that is around 1.0% less than FractalNet and DenseNet ($k = 12$), 2.1% less than VGG, ResNet (depth = 101), and All-CNN, 3.5% less than Highway Network and ResNet (depth = 1,202), and even 5.0% less than Network in Network and Maxout. On CIFAR-100, AE-CNN has a much lower classification error than Maxout, Network in Network, Highway Network, and All-CNN, a slightly lower classification error than ResNet (depth = 101), DenseNet ($k = 12$), ResNet (depth = 1,202), and VGG, and a comparable but still superior classification error than FractalNet. On both CIFAR-10 and CIFAR-100, the amount of parameters of the CNN generated by AE-CNN is more than ResNet (depth = 101) and DenseNet ($k = 12$), but significantly lower than ResNet (depth = 1,202), FractalNet, and VGG.

On both CIFAR-10 and CIFAR-100, AE-CNN outperforms Genetic CNN when compared to semi-automated peer competitors. Despite the fact that Hierarchical Evolution outperforms AE-CNN on CIFAR-10, AE-CNN uses a fraction of the GPU Days that Hierarchical Evolution does. When in comparison to AE-CNN, Block-QNN-S performs marginally worse on CIFAR-10 but marginally better on CIFAR-100, whereas AE-CNN occupies 1/3 the GPU Days that Block-QNN-S consumes, and the best CNN identified by AE-CNN has a fewer amount of parameters than Block-QNN-S. Furthermore, EAS and AE-CNN have almost same classification errors on CIFAR-10, while the fact that the best CNN evolved by AE-CNN has just 2.0M parameters, it is only 1/11 of EAS. To conclude, when compared to semi-automated peer competition, AE-CNN outperforms them while having a substantially lower number of parameters. When employing the algorithms in this category, it is important to keep in mind that domain knowledge is still necessary. For example, on CIFAR-10, EAS only uses 10 GPU Days for the optimal CNN, which is according to a base CNN with known great performance. As a result, the comparison with

Table 7.1 The comparisons between the algorithm and the state-of-the-art peer competitors in terms of the classification error (%), amount of parameters and the consumed GPU Days on the CIFAR-10 and CIFAR-100 benchmark datasets

	CIFAR-10	CIFAR-100	# of Parameter	GPU Days	
DenseNet (k = 12) [2]	5.24	24.42	1.0M	–	hand-crafted architecture
ResNet (depth = 101) [1]	6.43	25.16	1.7M	–	hand-crafted architecture
ResNet (depth = 1,202) [1]	7.93	27.82	10.2M	–	hand-crafted architecture
Maxout [22]	9.3	38.6	–	–	hand-crafted architecture
VGG [23]	6.66	28.05	20.04M	–	hand-crafted architecture
Network in Network [24]	8.81	35.68	–	–	hand-crafted architecture
Highway Network [25]	7.72	32.39	–	–	hand-crafted architecture
All-CNN [26]	7.25	33.71	–	–	hand-crafted architecture
FractalNet [27]	5.22	22.3	38.6M	–	hand-crafted architecture
Genetic CNN [29]	7.1	29.05	–	17	semi-automated algorithm
Hierarchical Evolution [30]	3.63	–	–	300	semi-automated algorithm
EAS [31]	4.23	–	23.4M	10	semi-automated algorithm
Block-QNN-S [32]	4.38	20.65	6.1M	90	semi-automated algorithm
Large-scale Evolution [33]	5.4	–	5.4M	2,750	completely automatic algorithm
Large-scale Evolution [33]	–	23	40.4M	2,750	completely automatic algorithm
CGP-CNN [34]	5.98	–	2.64M	27	completely automatic algorithm
NAS [35]	6.01	–	2.5M	22,400	completely automatic algorithm
MetaQNN [36]	6.92	27.14	–	100	completely automatic algorithm
AE-CNN	4.3	–	2.0M	27	completely automatic algorithm
AE-CNN	–	20.85	5.4M	36	completely automatic algorithm

regard to GPU Days spent is unfair to AE-CNN, that is totally automated and does not require any human experience or additional resources.

On both the CIFAR-10 and CIFAR-100 datasets, AE-CNN had the optimal performance among the automatic peer competitors with regard to classification error, amount of parameters, and GPU Days used. AE-CNN obtains a classification error of 4.3% on CIFAR-10, while the best and worst classification errors of peer competitors are 5.4% and 6.92%, respectively. Furthermore, as compared to MetaQNN, AE-CNN has a smaller classification error. On the CIFA100, AE-CNN has a 2.15% lower classification error than Large-scale Evolution and has 5.4M number of parameters that is considerably less than Large-scale Evolution (40.4M). On both CIFAR-10 and CIFAR-100, NAS, Large-scale Evolution, and MetaQNN utilize much more GPU Days than AE-CNN. The comparison demonstrates that in the automatic peer competitors to which the algorithm belongs, the algorithm obtains the optimal performance.

The justification for AE-CNN excelling Large-scale Evolution, NAS, Meta-QNN, and CGP-CNN is as follows. To begin with, Large-scale Evolution does not use the crossover operator that allows for local search. As a result, the performance of the GA-based design suffers. Second, in order to develop the optimum CNN architecture, CGP-CNN uses a fixed-length encoding technique. CGP-CNN need to determine a maximum length of CNNs during the architecture design in order for the encoding strategy to work. The maximum length predefined by CGP-CNN is less than the optimal length determined by AE-CNN, as seen in [34]. Finally, RL is used to design NAS and Meta-QNN. RL methods need more computer resources than GA to achieve the same level of performance [38] since the fitness value is not calculated. Given the computational resources available, NAS and Meta-QNN would perform inferior to AE-CNN.

7.4.2 Evolution Trajectory

Typically, when employing EAs to solve real-world issues, whether or not they have converged is need to be determined. The plotting of evolution trajectories is a better approach to see this. In this part, the evolution trajectories of AE-CNN with regard to the benchmark datasets are presented and analyzed. To do so, the classification accuracy of each individual is gathered in each generation first, and next display the statistical results.

Figure 7.4a, b depict the evolution trajectories of AE-CNN on CIFAR-10 and CIFAR-100, respectively. The horizontal axis in Fig. 7.4 indicates the generation number, and the vertical axis indicates classification accuracy; the blue line reflects the average classification accuracy of individuals within the identical generation, and the light-red area is contoured by the highest and lowest classification accuracy of individuals within each generation.

The mean classification accuracy increases rapidly from generation one to three, and then improves in steady as the evolution process continues until the 14th gen-

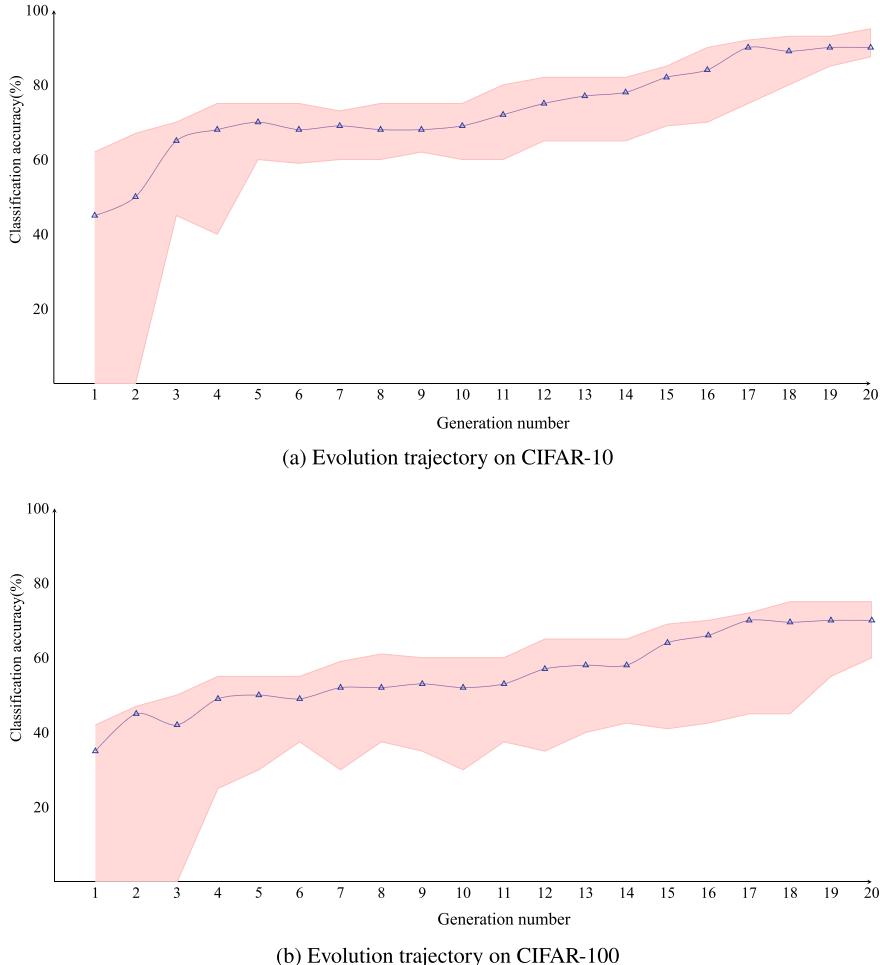


Fig. 7.4 Evolution trajectories of AE-CNN on CIFAR-10 and CIFAR-100

eration, when the mean classification accuracy increases significantly from approximately 75% to approximately 95%, and the algorithm finally converges when it terminates. When using the employed GPUs owing to an out-of-memory issue as shown by the lower boundary of the light-red area, the worst classification accuracy was present in the first two generations with a classification accuracy of zero; Out-of-memory architectures are excluded from the population because of their uncompetitive fitness starting at generation three, and classification accuracy progressively increases until the algorithm is terminated, although there is an exception at 4th generation. While the evolutionary process continues, the optimal performance almost maintains the same improvement as the mean classification accuracy, as may be observed from the upper boundary of the light-area. Furthermore, the difference between the best and

worst classification accuracy decreases, implying that the population is approaching a constant state.

Figure 7.4b shows a similar situation. From the 1-st generation to the 4th generation, the mean classification accuracy increased from approximately 30% to approximately 45%, with a small decrease at the 3-rd generation; the mean categorization accuracy has improved from the 4th generation through the 14th generation, after that, it rises to approximately 79% in the 17th generation, from approximately 50% at the 14th generation; and then the mean classification accuracy has continued to improve until the evolutionary process terminates. Due to the out-of-memory individuals who were initiated at random, the worst classification accuracy remains zero for the first three generations. However, beginning with the fourth generation, the worst classification accuracy increases until the 20th generation, excluding the 10 and 15th generations. The best classification accuracy improves nearly in lockstep with the mean classification accuracy, as shown by the evolution trajectories of the best classification accuracy, and also saves the converged performance from the 17th generation.

There will be no decline in the optimal classification accuracy indicated by the upper bounds of the light-red areas, as shown in Fig. 7.4a, b, which is accomplished by the elitism described in detail in Sect. 7.2.5, that is, the individual with the optimal fitness is retained without exception for the next generation. To conclude, AE-CNN converges with the default parameter settings in terms of GAs, that may allow users to utilize the algorithm to discover the optimal CNN architectures for their own data even if they are unfamiliar with GAs. If more computational resources are available, the maximum number of generation and population size may be increased to larger numbers.

7.4.3 Designed CNN Architectures

Figures 7.5 and 7.6 show the optimal CNN architectures discovered using the method on CIFAR-10 and CIFAR-100, respectively.

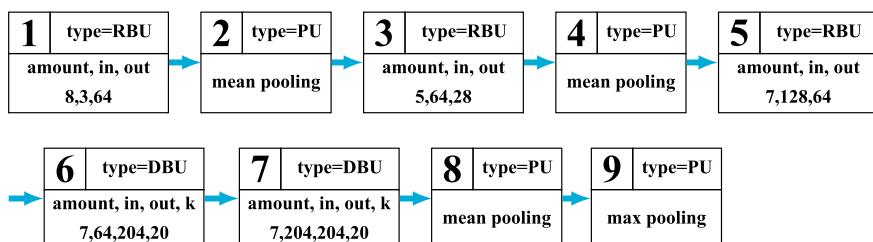


Fig. 7.5 Best architecture designed by AE-CNN on CIFAR-10

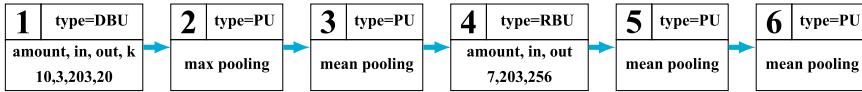


Fig. 7.6 Best architecture designed by AE-CNN on CIFAR-100

Tables 7.5 and 7.6 show that the optimal architecture on CIFAR-10 is made up of nine units which are designed using the encoding strategy in Sect. 7.2.2, and has a total of 38 layers (34 convolutional layers and four pooling layers), whereas the optimal architecture on CIFAR-100 is made up of six units with 21 layers (17 convolutional layers and four pooling layers).

The automatically found architectures according to both blocks have substantially less complex architectures and much better performance than the state-of-the-art CNNs that are only constructed on DBs or RBs. This could be used as previous knowledge when hand-crafting CNN architectures, indicating that ensemble blocks are further efficient. Furthermore, CIFAR-100 is widely regarded as a more challenging benchmark than CIFAR-10, and while working with CIFAR-100, researchers typically examine CNN architectures that have more layers than CIFAR-10. However, the optimum architecture for CIFAR-100 has a lower amount of layers beyond CIFAR-10, as seen in Figs. 7.5 and 7.6. Contrary to the common sense, finding the optimal architecture via evolution search can also result in the generation of valuable domain expertise.

7.5 Chapter Summary

In this chapter, we introduced and discussed the AE-CNN algorithm, which can automatically design the optimal architecture DNNs. The major difference of AE-CNN between the algorithm discussed in the previous chapter lies in the search space. In specific, the search space designed in AE-CNN is composed of the RBs and DBs, which are the building blocks of ResNet and DenseNet. These two CNNs are very popular in the machine learning community and are widely used as backbone architectures for related tasks. Through the comparison of the overall performance, and also the investigations on the evolution trajectory as well as the resulted CNN architectures, the effectiveness and the efficiency of AE-CNN are justified.

In essence, the RBs and DBs have some common characters although both blocks have different architectures, i.e., the skip connections. To some extent, the performance of the resulting CNNs may have a relationship to the skip connections. In the next chapter, we will introduce another architecture design algorithm based on this kind of connection to show the superiority in image classification tasks.

References

1. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
2. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
3. Cawley, G. C., & Talbot, N. L. C. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11, 2079–2107.
4. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958. ISSN 1532-4435.
5. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), 84–90. <https://doi.org/10.1145/3065386>.
6. Sainath, T. N., Mohamed, A.-R., Kingsbury, B., & Ramabhadran, B. (2013). Deep convolutional neural networks for lvcsr. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (pp. 8614–8618). IEEE.
7. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems* (pp. 3104–3112).
8. Clark, C., & Storkey, A. (2015). Training deep convolutional neural networks to play go. In *32nd International Conference on Machine Learning (ICML 2015)* (pp. 1766–1774).
9. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1–9).
10. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
11. Leung, Y., Gao, Y., & Zong-Ben, X. (1997). Degree of population diversity-a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5), 1165–1176.
12. Michalewicz, Z., & Hartley, S. J. (1996). Genetic algorithms + data structures = evolution programs. *Mathematical Intelligencer*, 18(3), 71.
13. Sun, Y., Yen, G. G., & Yi, Z. (2018). Improved regularity model-based EDA for many-objective optimization. *IEEE Transactions on Evolutionary Computation*. <https://doi.org/10.1109/TEVC.2018.2794319>.
14. Sun, Y., Yen, G. G., & Yi, Z. (2017). Reference line-based estimation of distribution algorithm for many-objective optimization. *Knowledge-Based Systems*, 132, 129–143.
15. Davis, L. (1991). *Handbook of genetic algorithms*. Van Nostrand Reinhold.
16. Goldberg, D. E., & Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine Learning*, 3(2), 95–99.
17. Miller, B. L., Goldberg, D. E. et al. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3), 193–212.
18. Zhang, G., Gu, Y., Hu, L., & Jin, W. (2003). A novel genetic algorithm and its application to digital filter design. In *Proceedings of 2003 Intelligent Transportation Systems* (Vol. 2, pp. 1600–1605). IEEE.
19. Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1998). *Genetic programming: An introduction* (Vol. 1). Morgan Kaufmann San Francisco.
20. Holland, J. H., et al. (1992). *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press.
21. Vasconcelos, J. A., Ramirez, J. A., Takahashi, R. H. C., & Saldanha, R. R. (2001). Improvements in genetic algorithms. *IEEE Transactions on Magnetics*, 37(5), 3414–3417.

22. Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning* (pp. 1319–1327).
23. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
24. Lin, M., Chen, Q., & Yan, S. (2014). Network in network. In *Proceedings of the 2014 International Conference on Learning Representations*.
25. Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015b). Highway networks. In *Proceedings of the 2015 International Conference on Learning Representations Workshop*.
26. Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for simplicity: The all convolutional net. In *Proceedings of the 2015 International Conference on Learning Representations*.
27. Larsson, G., Maire, M., & Shakhnarovich, G. (2016). Fractalnet: Ultra-deep neural networks without residuals. *The 5th International Conference on Learning Representations*. <https://openreview.net/forum?id=S1VaB4cex>.
28. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3), 211–252 ISSN 0920-5691, 1573-1405. <https://doi.org/10.1007/s11263-015-0816-y>.
29. Xie, L., & Yuille, A. L. (2017). Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017* (pp. 1388–1397). <https://doi.org/10.1109/ICCV.2017.154>.
30. Liu, H., Simonyan, K., Vinyals, O., Fernand C., & Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. [arXiv:1711.00436](https://arxiv.org/abs/1711.00436).
31. Cai, H., Chen, T., Zhang, W., Yu, Y., & Wang, J. (2018). Efficient architecture search by network transformation. In *Thirty-Second AAAI Conference on Artificial Intelligence*.
32. Zhong, Z., Yan, J., & Liu, C.-L. (2018). Practical network blocks design with q-learning. In *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*.
33. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).
34. Suganuma, M., Shirakawa, S., & Nagao, T. (2018). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 5369–5373). <https://doi.org/10.24963/ijcai.2018/755>.
35. Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. [arXiv:1611.01578](https://arxiv.org/abs/1611.01578).
36. Baker, B., Gupta, O., Naik, N., Raskar, R. (2016). Designing neural network architectures using reinforcement learning. [arXiv:1611.02167](https://arxiv.org/abs/1611.02167).
37. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in pytorch. <https://openreview.net/forum?id=BJjsrmfCZ>.
38. Sun, Y., Xue, B., Zhang, M., Yen, G. G. (2020a). Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24(2), 394–407.

Chapter 8

Architecture Design for Skip-Connection Based CNNs



8.1 Introduction

In this chapter, an efficient and effective algorithms employing GA is introduced, dubbed CNN-GA, to find the best CNN architectures for specific image classification tasks automatically, such that the found CNN can be directly employed without any need for manual tuning. CNN-GA is an algorithm for automating the architecture design of CNN. Please keep in note that the terms “automatic” and “automatic+manually+tuning” are discussed from the perspective of end-users, rather than developers. In developing high-performance CNN architecture design algorithms, however, adequate domain expertise should be promoted. This effort is not difficult to comprehend by comparing it to the design of the Windows Operating System by Microsoft scientists: to ensure the users could be able to effectively operate on computers even if they do not have considerable understanding of operating systems, the scientists should put as much of their professional knowledge as they possibly can while building a user-friendly operating system.

The following are the contributions of CNN-GA :

1. CNN-GA does not limit the depth of CNNs to a predetermined number; rather, the appropriate depth is identified throughout the process of evolutionary by identifying the CNN that achieves the highest classification accuracy on the provided data. Despite the fact that this design may generate the best CNN, due to individuals with unequal lengths, the crossover operation, which does exploitation search, is unable to work in this circumstance. In order to satisfy this need, a crossover operator has also developed to adapt for these individuals along the evolutionary process.
2. The skip connection, whose superiority in effectively training deep architectures has been theoretically and experimentally verified, is immediately incorporated into CNN-GA. By leveraging deep architectures and avoiding the vanishing gradient difficulties, the evolved CNNs are able to handle complex data [1]. In addition, the search space can be restricted by this design, allowing for the highest

performance in the shortest amount of time. Furthermore, as compared to previous models that have similar performance, the architectures of CNN-GA have a substantially fewer number of parameters.

3. CNN-GA finds the optimal CNN designs totally automatically, with no need for operator intervention during the process of evolutionary search. When the evolution process is complete, the resulting CNNs can be utilized to process input without further tuning, e.g., adding extra pooling or convolutional layers. In addition, other users can use CNN-GA without having to make any prior preparations, e.g., providing a network that has been manually tuned.
4. To speed up the fitness evaluation for individuals in CNN-GA with the same generation, an asynchronous computational component has been developed to utilize the available computing power to its full potential. Furthermore, a cache component is designed to minimize the amount of time required by fitness evaluation in the overall population

The rest of this chapter is laid out as follows. Firstly, Sect. 8.2 documents the details of CNN-GA. In Sects. 8.3 and 8.4, the experimental designs, results, and the analysis are presented.

8.2 Algorithm Details

In this section, the framework of CNN-GA is introduced first in Sect. 8.2.1, then detailed the essential steps in Sects. 8.2.2–8.2.5. In order to assist readers in better comprehending CNN-GA, each essential step will be documented in detail and the analysis for particular designs will be provided.

8.2.1 Algorithm Overview

The framework of CNN-GA is shown in Algorithm 1. Specifically, by providing the population size, the image classification dataset as well as the maximum generation number for the GA, and a set of CNN building blocks which is predefined, CNN-GA starts to work and eventually finds the optimal CNN architecture for the given image classification dataset via a sequence of evolutionary processes. A population is initialized at random during the evolution with a predetermined population size, and using the designed encoding approach to encode the specified building blocks (line 1). The counter of current generation is then reset to zero (line 2). The fitness of each individual, which encodes a specific CNN architecture, is tested on the given dataset (line 4) during evolution. After then, the individuals who will serve as parents are selected according to their fitness, and new offspring are produced through the use of genetic operators such mutation and crossover operators (line 5). The environmental

Algorithm 1: CNN-GA Framework

Input: The population size, a set of predefined building blocks, the maximum generation number, the image classification dataset.

Output: The optimal CNN architecture found by CNN-GA.

```

1  $P_0 \leftarrow$  Utilize the designed variable-length encoding strategy and the given population size to initialize a population;
2  $t \leftarrow 0$ ;
3 while  $t < \text{the maximum generation number}$  do
4   Utilize the designed acceleration components to conduct fitness evaluation for every individual in  $P_t$ ;
5    $Q_t \leftarrow$  Utilize the designed mutation and the crossover operators to produce offspring based on the chosen parent individuals;
6    $P_{t+1} \leftarrow$  Environmental selection from  $P_t \cup Q_t$ ;
7    $t \leftarrow t + 1$ ;
8 end
9 Return the individual with the highest fitness in  $P_t$ .
```

selection then selects a population of individuals who will survive into the following generation from the current population (line 6). The existing population is made up of two groups: the offspring population and the parent population. At last, the counter is incremented by one, and the development continues until the counter reaches the maximum generation value. CNN-GA follows the standard GA pipeline (line 5), and Algorithm 1 describes the selection phases, mutation phases, and crossover phases, as well as the offspring population.

With their inherent biological mechanism, GAs only give a unified framework for solving optimization problems. When GAs are employed in practice, their biological mechanism components must be created particularly for the problems that need to be handled. To ensure the efficiency and effectiveness of CNN-GA in creating CNN architectures, the acceleration components, the genetic operators, and the encoding method with variable length are carefully developed (in Algorithm 1, these are also emphasized in bold).

8.2.2 Population Initialization

A CNN is made up of various layers, including convolutional layers, pooling layers, and even fully connected layers at times. The depth of CNN is a key factor that influence its performance, and deep depth could become a reality by employing skip connections. In the designed encoding strategy, a new building block called the skip layer is created that replaces the convolutional layer when creating a CNN by directly employing the skip connections. Furthermore, the designed encoding approach ignores completely connected layers, and the reason for this will be explained later in this section. In the suggested encoding strategy, only the pooling layers and the skip layers are employed to form a CNN.

A skip layer, for example, is made up of one skip connection layers and two convolutional layers. Using a skip connection, the input of first convolutional layer is connected to the output of the second layer. Convolutional layers include a variety of parameters, including the stride size, the filter size, the type of convolutional operation, and the number of feature maps, as mentioned previously. The stride sizes, the filter sizes, and the convolutional operations are all set to the same values in the designed encoding strategy. The stride sizes and filter sizes are set to 1×1 and 3×3 , respectively, and the convolutional operation is utilized just once. Feature map numbers denoted by $F1$ and $F2$ represent parameters encoded for a skip layer in the two convolutional layers. Furthermore, for the kernel and stride sizes, the pooling layers employed in the designed encoding strategy are 2×2 . To this purpose, for a pooling layer, the sole parameter encoded is the pooling type denoted as $P1$. Later in this part, the reasons for this design and the adopted settings are going to be discussed.

Algorithm 2: Population Initialization

Input: The size of population T .
Output: The initialized population P_0 .

```

1  $P_0 \leftarrow \emptyset$ ;
2 while  $|P_0| < T$  do
3    $L \leftarrow$  Generating an integer at random which is greater than zero;
4    $list \leftarrow$  Creating a linked list that contains  $L$  nodes;
5   foreach node in  $list$  do
6      $r \leftarrow$  Generating a number from  $(0, 1)$  with uniform probability;
7     if  $r < 0.5$  then
8        $node.type \leftarrow 1$ ;
9        $node.F1 \leftarrow$  Randomly generating an integer which is greater than zero;
10       $node.F2 \leftarrow$  Randomly generating an integer which is greater than zero;
11    else
12       $node.type \leftarrow 2$ ;
13       $q \leftarrow$  Generating a number from  $(0, 1)$  with uniform probability;
14      if  $q < 0.5$  then
15         $| node.P1 \leftarrow max$ ;
16      else
17         $| node.P1 \leftarrow mean$ ;
18      end
19    end
20  end
21   $P_0 \leftarrow P_0 \cup list$ ;
22 end
23 Return  $P_0$ .
  
```

The details of the population initialization are shown in Algorithm 2. In a brief, the same approach is used to initialize each person in T , and then P_0 is used to store them. During the process of initializing an individual, the length L of that individual, which reflects the depth of the related CNN, is set based on a random value (line 3).

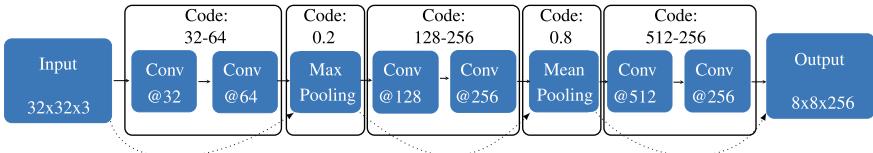


Fig. 8.1 An illustration of how a CNN is represented by the designed encoding approach

After that, a linked list with L nodes is generated (line 4). The linked list is then stored in P_0 (lines 21) after each node is configured (lines 5–20). A number, r , is randomly generated from $(0, 1)$ (line 6) during the configuration of each node. If $r < 0.5$, the *type* property of this node is set to 1, indicating that it is a skip layer. Otherwise, by setting *type* to 2, this node indicates a pooling layer. When it comes to a skip connection layer, the feature map numbers are generated at random and then assigned to *node.F1* and *node.F2* (lines 7–11), respectively. Otherwise, the likelihood of tolling a coin determines the pooling type. When the probability is less than 0.5, the pooling type, *node.P1*, is set to MAX, and otherwise to AVERAGE (lines 11–19).

Figure 8.1 demonstrates an example of the designed encoding strategy for encoding a CNN. Two pooling layers and four skip layers make up this CNN. One skip layer is represented by a string that contains the feature map numbers of all of the convolutional layers that are within the same skip layer, whereas a number reflecting the pooling type represents a pooling layer. A random number in the range of $(0, 0.5)$ denotes MAX layer, and one in the range of $[0.5, 1]$ denotes a AVERAGE layer. The code that represents the whole CNN is “32-64-0.2-128-256-0.8-512-256” to represent a CNN whose depth is eight, as illustrated in this example where the code for each layer is shown above itself.

The reasons for employing two convolutional layers in a skip layer, discarding fully connected layers, and the settings for the pooling and skip layers in CNN-GA will be discussed next. It is common practice to append the tail of CNN with several fully connected layers. Due to its extensive connection [2], the fully connected layer, on the other hand, easily results in the overfitting phenomenon [3]. Dropout [2] is a technique for reducing overfitting by randomly eliminating a portion of the connections. Each dropout, however, will create a new parameter. A parameter that has been properly specified is the only thing that can lead to a promising performance of the associated CNN. At the same time, the number of neurons in each fully connected layer and the number of fully connected layers are two characteristics that are difficult to adjust. If the fully connected layers are included in the designed encoding approach, the search space would significantly expand, making finding the optimal CNN design more challenging. The design of ResNet inspired the usage of the skip layer that contains two convolutional layers, and the effectiveness of such skip layers has been empirically proven in literature [4–7]. However, in each skip layer of ResNet, the sizes of feature maps are set to be identical. The suggested

encoding strategy allows for unequal feature map sizes, which is more flexible. Furthermore, using the 1×1 stride and setting the convolutional operation to *same* maintains the input data dimension constant, this allows for greater flexibility in the automatic design since changing the stride size from 1×1 does not impact the size of the image. For the pooling layers, the kernel and filter sizes, and the stride size are all based on existing hand-crafted CNNs [4, 5]. Another major reason for supplying such options is having extensive experience manually modifying CNN designs. Section 8.4 will demonstrate the effectiveness of such settings.

8.2.3 Fitness Evaluation

Algorithm 3: Fitness Evaluation of Population

Input: The image classification dataset , the population P_t containing all the individuals that are going to be evaluated.

Output: The population P_t P_t containing the individuals whose fitness has been evaluated.

```

1 if  $t = 0$  then
2   | Cache  $\leftarrow \emptyset$ ;
3   | Setting Cache to a global variable;
4 end
5 foreach individual in  $P_t$  do
6   | if the identifier of individual is in Cache then
7     |   |  $v \leftarrow$  Querying the fitness by identifier from Cache;
8     |   | Setting  $v$  to individual;
9   | else
10    |   | while GPU is available do
11      |   |   | Using an available GPU to asynchronously evaluate individual (Algorithm 4
12      |   |   | shows the details);
13    |   | end
14  end
15 Return  $P_t$ .

```

The fitness evaluation process of the individuals within the population P_t is detailed in Algorithm 3. Given the image classification dataset for finding the optimal CNN architecture, and the population P_t containing all the individuals that are going to be evaluated. Each individual of P_t is evaluated in the same way, and then the population P_t of the individuals with their fitness. In the event where the evaluation of fitness is performed on the initial population, i.e., P_0 , a global cache system referred to as *Cache* is built, which stores the individuals fitness with unknown architectures (lines 1–4). If *individual* is discovered in *Cache*, the fitness of that individual denoted by *individual* in P_t is obtained directly from *Cache* (lines 6–8). Otherwise, for its fitness evaluation, *individual* is asynchronously placed on an available

GPU (line 6–8). It is worth noting that an individual from *Cache* is queried based on their identifier. Any identifier can theoretically be used as long as it can identify individual encoding different designs. In CNN-GA, the 224-hash code [8] is used as the corresponding identifier in terms of the encoded architecture, which has been adopted by most programming languages. In addition, the individual is placed on an available GPU in an asynchronous manner, which means that there is no need to wait for the fitness evaluation of the next individual until the fitness evaluation of the individual currently evaluating is complete, but may move on to the next one right away.

Algorithm 4: Fitness Evaluation of Individual

Input: The available GPU, the individual *individual*, the training data D_{train} , the global cache *Cache*, the training epochs number, and the data for fitness evaluation $D_{fitness}$ from the investigated image classification dataset.

Output: The *individual* with its evaluated fitness.

- 1 Constructing a CNN with classifier according to the investigated image classification dataset and the encoded information in *individual*;
 - 2 $v_{best} \leftarrow 0$;
 - 3 **foreach** epoch in the given training epochs **do**
 - 4 Using the given GPU to train the CNN on D_{train} ;
 - 5 $v \leftarrow$ Calculating the classification accuracy on $D_{fitness}$;
 - 6 **if** $v > v_{best}$ **then**
 - 7 $v_{best} \leftarrow v$;
 - 8 **end**
 - 9 **end**
 - 10 Setting v_{best} as the fitness of *individual*;
 - 11 Putting the identifier of *individual* and v_{best} into *Cache*;
 - 12 **Return** *individual*.
-

Algorithm 4 shows the details of assessing the fitness of one individual. To begin, a CNN is decoded from *individual*, and a classifier (line 1) is added to this CNN according to the specified image classification dataset. A Softmax classifier [9] is employed in CNN-GA, and the categories number is decided by the investigated image classification dataset. When decoding a CNN, the output of the convolutional layer is added to a ReLU [10] with a BN operation [11] operation followed by, which is based on modern CNN [4, 12]. In situations that the spatial number of the skip layer does not equal to the input data, a convolutional layer will be added to the input data [4, 12], which possesses the unit filter as well as the unit stride, but the feature maps with special number. The CNN is then trained on the training data using the SGD [13] algorithm on the GPU (line 4), and the classification accuracy is evaluated on the fitness evaluation data (line 5). The employ of the SGD training method and the Softmax classifier is based on the common practices in deep learning community. When the training is completed, the fitness of *individual* (line 10) is chosen as the best classification accuracy on the fitness evaluation data. Finally, the *individual* identifier and fitness are connected and stored in *Cache* (line 11).

In the following, reasons for developing such an asynchronous and cache component are discussed. In conclusion, because CNN training takes a long time, depending on the architecture, the training can take times from several hours to several months, thus the asynchronous and cache component are developed for accelerating the fitness evaluation in CNN-GA. The asynchronous component, in particular, is a GPU-based parallel computation platform. Deep learning algorithms, because of the computational nature of computing gradients, are frequently deployed on GPUs to accelerate the training process [14]. Existing deep learning packages, such as TensorFlow [15] and PyTorch [16], do allow multi-GPU calculations. Their parallel calculations, on the other hand, are based on the pipelines of model-parallel and data-parallel. The input data is separated into numerous tiny groups in the data-parallel pipeline, and each group is assigned to one GPU for computation. This is due to inability of a single GPU which has the limited memory to process all of the data at once. A model is partitioned into multiple little models in the model-parallel pipeline, and each GPU holds one small model. The reason is that the computing capabilities of a single GPU is insufficient to execute an entire model. The designed parallel pipeline, on the other hand, obviously does not fit into either of the pipelines, but rather a higher level based on them. As a result, an asynchronous component like this is built to take advantage of the GPU computing power, in particular for algorithms that are based on population. Furthermore, when a complex problem can be broken into multiple distinct sub-problems, the asynchronous component is often used. By running these sub-problems in parallel on separate computer platforms, the overall amount of time it takes to solve the problem is reduced. EAs have traditionally been employed to tackle issues when the fitness evaluation is not time-consuming (despite the existence of a class of computationally expensive problems, their fitness is frequently estimated using a surrogate model to avoid a direct fitness evaluation) and there is little need to construct asynchronous components. On occasion, they just employ the built-in components that are based on the programming languages that have been adopted. However, practically all of these built-in components are based on CPU, and therefore cannot successfully train DNNs since the NN acceleration platform is based on GPU. In addition, the fitness of each individual is evaluated independently, which perfectly fits the setting of employing this technique. Because of the reasons stated above, CNN-GA includes an asynchronous component. The cache component is also employed to accelerate the fitness evaluation, with the following considerations: (1) There is no need to evaluate the fitness of those individuals surviving into the next generation again if their architecture is not changed, and (2) the evaluated architecture can be regenerated with mutation and crossover operations in a subsequent generation. It should be noted that the weight inheritance of Large-scale Evolution does not apply to the second consideration.

Algorithm 5: Offspring Generating

Input: The mutation probability p_m , the crossover probability p_c , the list of mutation operation l_m , the probabilities of selecting different mutation operations p_l , the population P_t containing individuals with evaluated fitness.

Output: The generated offspring population Q_t .

```

1  $Q_t \leftarrow \emptyset;$ 
2 while  $|Q_t| < |P_t|$  do
3    $p_1 \leftarrow$  Selecting two individuals from  $P_t$  at random, and then selecting the individual with the better fitness;
4    $p_2 \leftarrow$  Repeating Line 3;
5   while  $p_2 == p_1$  do
6     | Repeating Line 4;
7   end
8    $r \leftarrow$  Generating a number from  $(0, 1)$  at random;
9   if  $r < p_c$  then
10    | Choosing a point in  $p_1$  at random and dividing it into two parts;
11    | Choosing a point in  $p_2$  at random and dividing it into two parts;
12    |  $o_1 \leftarrow$  joining the first part of  $p_1$  and the second part of  $p_2$ ;
13    |  $o_2 \leftarrow$  joining the first part of  $p_2$  and the second part of  $p_1$ ;
14    |  $Q_t \leftarrow Q_t \cup o_1 \cup o_2$ ;
15  else
16    |  $Q_t \leftarrow Q_t \cup p_1 \cup p_2$ ;
17  end
18 end
19 foreach individual  $p$  in  $Q_t$  do
20    $r \leftarrow$  Generating a number from  $(0, 1)$  at random;
21   if  $r < p_m$  then
22     |  $i \leftarrow$  Choosing a point in  $p$  at random;
23     |  $m \leftarrow$  Selecting one operation from  $l_m$  according to the probabilities in  $p_l$ ;
24     | Performing the mutation  $m$  at the point  $i$  of  $p$ ;
25   end
26 end
27 Return  $Q_t$ .

```

8.2.4 Offspring Generating

Algorithm 5, which is divided into two parts, shows the details of generating offspring. The crossover (lines 1–18) is the first, and the mutation (lines 19–26) is the second. During the crossover operation, a total of $|P_t|$ offspring will be generated, where $|\cdot|$ is the collection size. To be more specific, two parents are chosen initially, and each parent is chosen from two individuals who are chosen at random, based on which individual possesses the better fitness (lines 3–7). The binary tournament selection [17], which is used in GAs for single-objective optimization, is a variant of this selection. After the parents have been chosen, a random number is generated (line 8), to determine whether or not the crossover will take place. These two parent individuals are placed in Q_t as offspring (line 16) if the generated number does not fall below the predefined probability of crossover. Otherwise, both these

parent individuals are divided into two parts at random, and the two parts divided from the parent individuals are switched to generate offspring (lines 10–14). During the mutation process, a number is generated at random first (line 20), and if the generated number is less than p_m , the mutation is executed on the current individual (lines 21–25). When modifying an individual, a location denoted by i is chosen at random from the current individual, and a mutation operation denoted by m is chosen from the specified list of mutation according to the probabilities defined in p_l . After that, m is applied to the position i . The available operations for mutation defined in the CNN-GA mutation list are:

- Adding a pooling layer with random settings;
- Adding a skip layer with random settings;
- Altering the parameter values of the building block in a random manner at the position that has been chosen.
- Removing the layer at the position that has been chosen;

The purpose of adopting a mutation operation from a probability list and designing such a crossover operator is explained next. To begin with, the designed crossover operator is modeled after the one-point crossover [18] found in traditional GAs. The one-point crossover, on the other hand, was designed only for individual of similar length. Individuals with unequal lengths are treated with the designed operator for crossover. Despite its simplicity, the suggested crossover operator improves the performance of discovering CNN architectures, as demonstrated empirically in Sect. 8.4. Second, existing algorithms choose the particular mutation operation with an equal probability. The provided mutation operations are chosen with different probability in CNN-GA. A larger probability for the “Adding a skip layer with random settings” mutation operator is provided, which will enhance the depths of CNNs with a higher probability. Equal probabilities for other mutation processes is still used. This design is based on the idea that a deeper CNN will be more powerful, as previously stated. Although “Adding a pooling layer with random settings” can increase the depth of CNNs, utilizing only one pooling layer reduces the dimension of input data by half, resulting in the unavailability of the found CNN. As a result, an increased probability is need not be given.

8.2.5 Environmental Selection

The details of the environmental selection are shown in Algorithm 6. Using the binary tournament selection, $|P_t|$ individuals are first selected from the current population ($Q_t \cup P_t$), and then these individuals will be incorporated into population of next generation denoted P_{t+1} (lines 2–6). Second, the individual with the best fitness is chosen and then checked to see if it has been placed to P_{t+1} . If not, the individual with worst fitness in P_{t+1} will be replaced (lines 7–10).

Only choosing the top $|P_t|$ individuals for the following generation creates the premature phenomenon [19], which leads to the algorithm trapping in a local opti-

Algorithm 6: Environmental Selection

Input: The offspring population Q_t , the parent population P_t
Output: The next generation population P_{t+1} .

```

1  $P_{t+1} \leftarrow \emptyset;$ 
2 while  $|P_{t+1}| < |P_t|$  do
3    $p_1, p_2 \leftarrow$  Selecting two individuals at random from  $Q_t \cup P_t$ ;
4    $p \leftarrow$  Selecting the individual with a better fitness from  $\{p_1, p_2\}$ ;
5    $P_{t+1} \leftarrow P_{t+1} \cup p$ ;
6 end
7  $p_{best} \leftarrow$  Finding the individual who has the best fitness from  $Q_t \cup P_t$ ;
8 if  $p_{best}$  is not in  $P_{t+1}$  then
9   | Replacing the individual with the worst fitness in  $P_{t+1}$  by  $p_{best}$ ;
10 end
11 Return  $P_{t+1}$ .

```

mum [20, 21]. The algorithm will not converge unless select the individuals with the best fitness for the following generation deliberately. In theory, an ideal population should include both excellent and poor individuals in order to increase diversity [22, 23]. In most cases, the binary tournament selection is utilized for this purpose [17, 24]. However, relying just on the binary tournament selection may miss the best individual, which would lead to the algorithm not moving in a better evolution direction. As a result, the individual with best fitness are added to the population of next generation explicitly, which is an elitism method used in EAs [25].

8.3 Experimental Design

A set of tests on image classification tasks are done to evaluate the performance of CNN-GA. In particular, Sect. 8.3.1 introduces the peer competitors who will be compared to CNN-GA. The parameter settings of CNN-GA are then provided in Sect. 8.3.2.

8.3.1 Peer Competitors

The state-of-the-art algorithms are chosen as peer competitors to CNN-GA in order to demonstrate its efficiency and effectiveness. The peer competitors, in particular, are chosen from three different categories.

The first category includes manually designed state-of-the-art CNNs such as ResNet [12], DenseNet [4], VGGNet [26], Maxout [27], Network in Network [28], Highway Network [29], and All-CNN [30]. The comparison is based on two versions of ResNet, namely the ResNet models with the depth of 1, 202 and 110. They are called ResNet (depth = 1, 202) and ResNet (depth = 110) for ease of reference.

ResNet (depth = 110) is the victor in terms of classification accuracy on CIFAR-10 among all the ResNet variants explored in the seminal paper [12], and ResNet (depth = 1, 202) is the most sophisticated version, which is the major rationale for choosing them for the comparison. For DenseNet, the DenseNet-BC version is used, which not only has the best classification accuracy but also the fewest parameters of all the variants [4]. It is worth noting that most algorithms in this category have recently won large-scale visual recognition challenges [31].

The architecture design algorithms for CNN from the “automatic+manually tuning” and “automatic” are found in the second and third categories, respectively. The second category includes Hierarchical Evolution [7], Block-QNN-S [32], NSANet [33], EAS [34], and Genetic CNN [35], while the third category includes NAS [34], MetaQNN [36], Large-scale Evolution [6], and CGP-CNN [37]. Two versions of NASNet are compared, NASNet-A+cutout and NASNet-B, because they have the best classification accuracy and the smallest number of parameters, respectively. The term “cutout” refers to a regularization method [38] employed in CNN training that has the potential to increase final performance.

Please note that CNN-GA is primarily focused on offering an “automatic” solution for researchers without extensive domain knowledge of CNN architecture tuning to design potential CNN architectures. According to the “No Free Lunch Theorems [39]”, CNNs with manual tuning will have higher classification accuracy than “automatic” CNNs like CNN-GA should be expected. Indeed, CNN-GA is only fair when compared to CNN architectural designs from the “automatic” category. It is still wants to compare CNN-GA to CNN architectures designed with domain knowledge in this experiment to demonstrate its efficiency and effectiveness in comparison to all existing state-of-the-art CNNs.

8.3.2 Parameter Settings

In terms of the datasets, the training set is divided into two distinct parts. The first comprises 90% of the total images and serves as a training set for individuals, whereas the rest 10% images serve as a fitness evaluation set for evaluating fitness. During the training phases, the images are also enhanced. To make a fair comparison, the same augmentation procedure as many peer competitors often used are applied, i.e., padding four zeros pixels to each direction of the image, and then a 32×32 image is arbitrarily cropped. Finally, performing a horizontal flip on the cropped image at random with a probability of 0.5 [4, 12].

To broaden the use of CNN-GA, it has been developed so that potential users need not any prior understanding of EAs. Hence, it is only necessary to adjust the parameters of CNN-GA according to the conventions. As recommended in [40], the probabilities of mutation and crossover are set to 0.2 and 0.9, respectively. The method in [12] is used to train each individual, i.e., the SGD is used to train 250 epochs with a momentum of 0.9 and a learning rate of 0.1, and the learning rate is decaying by a factor of 0.1 at the 1st, 149th, and 249th epochs. This training

regimen is followed by the majority of peer competitors. When CNN-GA finishes, the one with the best fitness value is selected and train it on the original training set for 350 epochs. Finally, the classification accuracy on testing set is tabulated for comparison with peer competitors. Furthermore, depending on the settings used by state-of-the-art CNNs, the available number of feature maps is set to 64, 128, 256. The normalized probability of increasing the depth is set to 0.7 for the four mutation operations shown in Sect. 8.2.4, while the others have equal probabilities. In theory, any probability for adding mutation can be set by keeping it greater than the others. In addition, the number of generations and the population size are all set to 20, and the peer competitors use comparable parameters. The larger maximum generation number and population size should theoretically result in better performance, but they will also consume more computational resources. Since the existing settings readily beat most of the peer competitors based on the results in Table 8.1, larger settings are not studied in this chapter.

8.4 Experimental Results and Analysis

In this section, an overview of the comparison findings between CNN-GA and the chosen peer competitors is presented first, as well as the performance of the generated CNN architecture transferred to ImageNet. The crossover operation and the acceleration components, two novel components of CNN-GA, are then explored and analyzed separately. Finally, the evolution trajectories of CNN-GA are displayed, which helps readers in better understanding CNN-GA during the process of determining the appropriate CNN design.

8.4.1 Overall Results

Due to the fact that the architectures of the state-of-the-art CNNs that fall into the first category of peer competitors are all manually created, both the parameter number and the classification accuracy are evaluated. For the algorithms in other two categories, the consumed “GPU days” for finding the optimal CNN is compared, in addition to the number of parameters and the classification accuracy. In particular, a GPU day indicates that the algorithm has completed one day on one GPU, which reflects the computational resource consumed by these algorithms. When comparing the number of parameters and the classification accuracy among peer competitors, the discovered CNN is named the same as the corresponding architecture discovering algorithm for the convenience of summarizing the comparison results. For example, when compared to the chosen state-of-the-art CNNs, the CNN architecture discovered by CNN-GA is titled CNN-GA.

Table 8.1 shows the comparison results between CNN-GA and the peer competitors. Table 8.1 shows the peer competitors divided into three blocks based on the

Table 8.1 The comparisons of CNN-GA and the chosen peer competitors on CIFAR-10 and CIFAR-100 benchmark

		CIFAR-10	CIFAR-100	# Parameters	GPU days	Manual assistance?
Manually designed	ResNet (depth = 110) [12]	93.57	74.84	1.7M	–	Completely needed
	ResNet (depth = 1,202) [12]	92.07	72.18	10.2M	–	Completely needed
	DenseNet-BC [4]	95.49	77.72	0.8M	–	Completely needed
	VGGNet [26]	93.34	71.95	20.04M	–	Completely needed
	Maxout [27]	90.70	61.40	–	–	Completely needed
	Network in Network [28]	91.19	64.32	–	–	Completely needed
	Highway Network [29]	92.40	67.66	–	–	Completely needed
	All-CNN [30]	92.75	66.29	1.3M	–	Completely needed
Automatic + manually tuning	Genetic CNN [35]	92.90	70.97	–	17	Partially needed
	Hierarchical Evolution [7]	96.37	–	–	300	Partially needed
	EAS [34]	95.77	–	23.4M	10	Partially needed
	Block-QNN-S [32]	95.62	79.35	6.1M	90	Partially needed
	NASNet-B [33]	96.27	–	2.6M	2,000	Partially needed
	NASNet-A + cutout [33]	97.60	–	27.6M	2,000	Partially needed
Automatic	Large-scale evolution [6]	94.60	–	5.4M	2,750	Completely not needed
	Large-scale Evolution [6]	–	77.00	40.4M	2,750	Completely not needed
	CGP-CNN [37]	94.02	–	1.68M	27	Completely not needed
	NAS [34]	93.99	–	2.5 M	22,400	Completely not needed
	Meta-QNN [36]	93.08	72.86	–	100	Completely not needed
	CNN-GA	95.22	–	2.9M	35	Completely not needed
	CNN-GA	–	77.97	4.1M	40	Completely not needed
	CNN-GA + cutout	96.78	–	2.9M	35	Completely not needed
	CNN-GA + cutout	–	79.47	4.1M	40	Completely not needed

categories, with the category titles in the first column. In addition, the last column includes information on how much manual assistance the associated CNN required when discovering the architectures of CNNs. The names of the peer competitors are also listed in the second column. The classification accuracy values on the CIFAR-10 and CIFAR-100 dataset are shown in the third and fourth columns, respectively, while the number of parameters in the corresponding CNNs is shown in the fifth column. The sixth column displays the number of GPU days used, which is solely relevant to architecture design algorithms. The symbol “–” denotes that the corresponding algorithm has no publicly known outcome. On both the CIFAR-10 and CIFAR-100 datasets, VGG, Block-QNN-S, EAS, All-CNN, ResNet (depth=110), ResNet (depth = 1,202), and DenseNet have the same amount of parameters, owing to the fact that on both datasets, the similar CNNs achieving the best classification accuracy. Notice that the results of peer competitors in Table 8.1 are all taken from their own seminal articles. It is notable for VGG because in its seminal publication, it did not conduct experiments on CIFAR-10 and CIFAR-100. The data shown in Table 8.1 are taken from [37]. The best CNN discovered by CNN-GA is chosen from the population in the previous generation, and it is then trained independently for five times. From the five outcomes, the best classification accuracy is chosen to be displayed in Table 8.1, following the conventions of its peer competitors [6, 7, 32, 34–37]. The results by executing CNN-GA with the cutout, which is labeled as “CNN-GA + cutout” in Table 8.1 are provided. This is inspired by the performance boost of NASNet provided by the cutout regularization. The top CNN-GA architectures discovered in CIFAR-10 and CIFAR-100 are specifically chosen, retrained them on the training set with the cutout, and then reported the classification accuracy on the testing set.

Every effort was made in this experiment to make a fair comparison by employing the same process of training and data augmentation strategy as the chosen peer competitors. However, because there are several peer competitors competing for the best classification accuracy, they use different training processes and data augmentation approaches. For example, to boost classification accuracy, NASNet used a new path drop technique. CNN-GA with the same training procedure and data augmentation methods as the most peer competitors are kept because the improved path drop methodology is not publicly available. As a result, comparing classification accuracy alone is not fair to CNN-GA must be remembered.

On the CIFAR-10 dataset, CNN-GA outperforms ResNet (depth = 1,202) and VGG by 3.15% and 1.88%, respectively, while using only 28 and 14% of their respective parameters. On the CIFAR-10 dataset, CNN-GA has a higher number of parameters than All-CNN and ResNet (depth = 110), but CNN-GA has the best classification accuracy. On CIFAR-10, CNN-GA performs slightly worse than DenseNet-BC, but significantly better on CIFAR-100, which is a more difficult benchmark dataset compared with CIFAR-10. Furthermore, on the CIFAR-10 dataset, CNN-GA outperforms Maxout, Network in Network, and Highway Network in terms of classification accuracy. For the results on CIFAR-100 dataset, CNN-GA uses 85%, 40%, and 52% fewer parameters than DenseNet, VGG, and ResNet (depth=1202), but it still improves classification accuracy by 1.39%, 6.02%, and 5.79%. Although it employs a larger network than All-CNN and RestNet (depth = 101), CNN-GA

surpasses Maxout, All-CNN, ResNet (depth = 110), Network in Network, and Highway Network in terms of classification accuracy. In conclusion, on the CIFAR-10 and CIFAR-100 datasets, CNN-GA achieves the best classification accuracy among the manually designed state-of-the-art CNNs. It also uses many less parameters than the majority of the first category CNNs.

The classification accuracy achieved by CNN-GA is better than that achieved by Genetic CNN of peer competitors in the second category. CNN-GA has a 1.15% poorer classification accuracy than Hierarchical Evolution, but it requires a tenth of the GPU time. On the CIFAR-10 and CIFAR-100 datasets, CNN-GA has slightly worse classification accuracy than Block-QNN-S, whereas CNN-GA uses around half the GPU days and the number of parameters as Block-QNN-S. In addition, the classification accuracy achieved by CNN-GA is competitive to that of EAS, while CNN-GA employs a significantly smaller number of parameters than that of EAS (CNN-GA employs 87% fewer parameters than that of EAS). Furthermore, NASNet-B performs somewhat better on CIFAR-10 than CNN-GA (96.20% vs. 95.22%), yet CNN-GA consumes only around 1/55 of the GPU days that NASNet-B consumes (35 GPU days v.s., 2,000 GPU days). When compared to NASNet-B, NASNet-A + cutout improves classification accuracy by 1.33% using a CNN with 27.6 million parameters and 2,000 GPU days; however, CNN-GA + cutout improves classification accuracy by 1.76% using the same CNN with 2.9 million parameters and 35 GPU days. Despite the fact that CNN-GA aims to design promising CNN architectures without any prior CNN domain expertise, it nevertheless outperforms its peers in this “automatic+manually+tuning” category. The major disadvantage of the algorithms in this category, in comparison to CNN-GA, is that they require extensive expertise when utilized to handle real-world applications. EAS, for example, requires a manually tuned CNN on the given dataset, which is subsequently refined using EAS. If the tuned CNN does not provide promising results, the generated EAS will not perform well in the end. Furthermore, CNNs discovered using Block-QNN-S and Hierarchical Evolution cannot be employed directly. They must be manually put into a larger CNN that has been pre-designed. If the larger network is badly built, Hierarchical Evolution and Block-QNN-S will perform poorly as well. To boost efficiency, NASNet used a redesigned path-dropping technique in addition to the CNN architecture, which is required by all versions of NASNet.

On the CIFAR-10 dataset, CNN-GA outperforms Large-scale Evolution and CGP-CNN, and much outperforms Meta-QNN and NAS in terms of classification accuracy. On the CIFAR-100 dataset, CNN-GA also outperforms Meta-QNN and Large-scale Evolution in terms of classification accuracy. Furthermore, on the CIFAR-10 dataset, CNN-GA has just 2.9M parameters, which is about half of what Large-scale Evolution has. CNN-GA contains 4.1M parameters on the CIFAR-100 dataset, saving 90% parameters compared to Large-scale evolution, which requires 40.4M parameters [6]. In addition, on the CIFAR-10 dataset and CIFAR-100 dataset, CNN-GA requires just 35 GPU days and 40 GPU days, respectively, but Large-scale Evolution requires 2,750 GPU days on the CIFAR-10 dataset and another 2,750 GPU days on the CIFAR-100 dataset. Furthermore, on the CIFAR-10 dataset, NAS uses 22,400 GPU days. According to the GPU days and the number of parameters used, CNN-

GA has a promising performance by using 90% simpler architectures with 99% less computing resource than the average number of competitors in this category (The number is calculated by first adding up the relevant numbers of all of the peer competitors in this category, and then normalized by the number of classification results that are available. E.g., the number of classification results is six, and 28, 027 is the total consumed GPU days of the peer competitors. As a result, the average consumed GPU days are 4, 671. The average GPU days consumed by CNN-GA are calculated to be 37.5 in the same way).

In terms of not only the classification accuracy, but also the processing resource used and the number of parameters, CNN-GA beats most state-of-the-art CNNs manually created and all automatic architecture discovering algorithms. Although the classification accuracies achieved by the state-of-the-art “automatic+manually-tuning” architecture discovering algorithms are similar (to slightly better) than CNN-GA, CNN-GA is automatic and does not need users to have any prior knowledge of CNNs when solving real-world tasks, which is the main goal of this chapter.

8.4.2 Transferable Performance on ImageNet

The experiment by classifying ImageNet [41] using the CNN architecture discovered by CNN-GA on a small-scale dataset is performed in order to explore the transferable performance. The best CNN architecture found by CNN-GA on CIFAR-10 is employed and then train it from scratch, following the conventions of CNN architecture algorithms.

In this experiment, the ImageNet-2012 version are used, i.e., ILSVRC2012, and each image is randomly cropped to 224×224 with a randomly horizontal flip as the input data, following the common practice in [4, 12, 33, 35]. Peer competitors are chosen from the CNN architecture design algorithms who have also explored the transferable performance on ILSVRC2012 with the same training routine to keep the comparison as fair as possible. Note only Genetic CNN2 and NASNet3 fulfill this need. While both Hierarchical Evolution [7] and Block-QNN-S [32] performed similar research, Hierarchical Evolution employed images with a dimension of 219×219 , whereas Block-QNN-S only used a subset of ILSVRC2012. As a result, Hierarchical and Block-QNN-S are not included in this analysis. On top of the transferred architecture, a convolutional layer is added with a kernel size of 7×7 , followed by a MAX layer, and the number of the feature map for the final convolutional layer is changed to 512, as in [35]. The transferred architecture is trained on the training set, and the classification accuracy is reported on the validation set, according to the norms of researching ILSVRC2012. The top-1 and top-5 outcomes are presented in this comparison, as stated in [35].

When the transferred architectures from Genetic-CNN, NASNet, and CNN-GA are performed on the ILSVRC2012 dataset, Table 8.2 presents the top-1 and top-5 classification accuracies. Because NASNet did not report the top-5 classification accuracy of the transferred architecture, the corresponding cell in Table 8.2 is indi-

Table 8.2 The top-1 and top-5 classification accuracy values of the transferred architectures on the ILSVRC2012 dataset

	Top-1	Top-5
Genetic CNN	72.13	90.26
NASNet	74.0	—
CNN-GA	74.82	92.33

cated by “—”. Furthermore, NASNet ran numerous tests on ILSVRC2012 with various settings, with the result displayed in Table 8.2 being the only one that used the same input dimension as Genetic CNN and CNN-GA for a fair comparison. When their designed CNN architectures are transferred to the large-scale ILSVRC2012 dataset, the CNN architecture designed by CNN-GA on CIFAR-10 has the best top-1 and top-5 classification accuracies among the chosen peer competitors, as shown in Table 8.2.

8.4.3 Performance of Crossover Operator

CNN-GA is re-run on the CIFAR-10 dataset with the crossover operator disabled to verify the effectiveness of the designed crossover operator, and the comparison results are displayed in Figs. 8.2 and 8.3. Figure 8.2 depicts the best classification accuracy in each generation, whereas Fig. 8.3 depicts the standard classification accuracy deviations in each generation. The line with “diamond” markers in both figures refers to CNN-GA with crossover operator, whereas the line with “solid circle” markers refers to the algorithm without crossover operator. This experiment is done with the same initial population to ensure a fair comparison. Furthermore, a line connects the

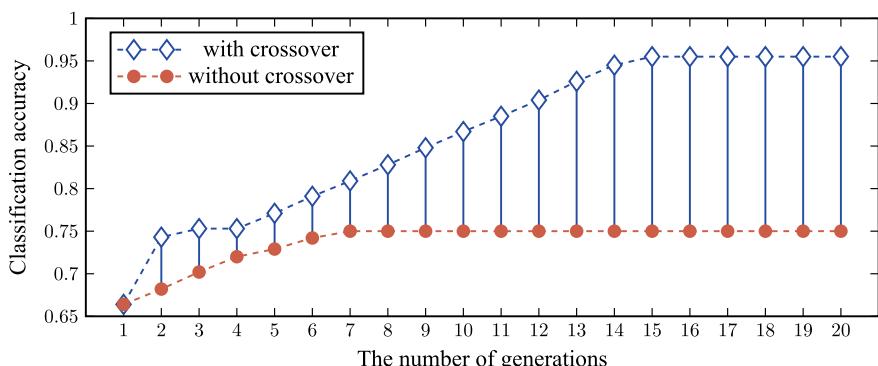


Fig. 8.2 The comparisons of the best classification accuracy regarding whether using the designed crossover operation in CNN-GA algorithm on the CIFAR-10 dataset

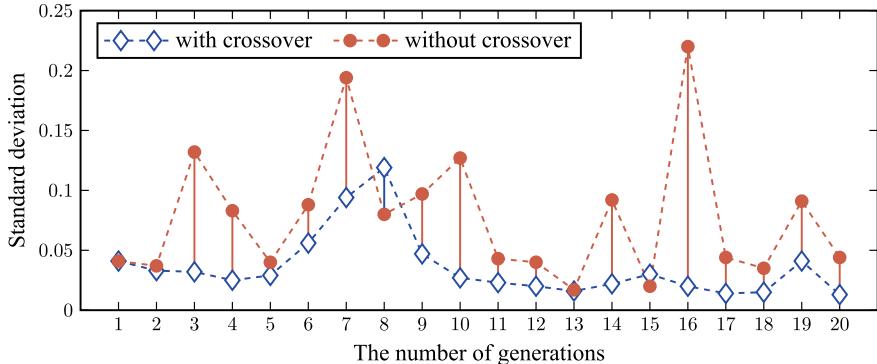


Fig. 8.3 The standard deviation comparisons of the best classification accuracy regarding whether using the designed crossover operation in CNN-GA on the CIFAR-10 dataset

comparison findings from the same generation. If the line is blue, it means value of CNN-GA is higher than CNN-GA without the crossover operator. Otherwise, the line is represented in red, implying that the value obtained by disabling the crossover operator is greater than that obtained by CNN-GA. Figure 8.2 shows that the higher the number, the better, which is the reverse of Fig. 8.3 (i.e., the smaller, the better).

As shown in Fig. 8.2, the best classification accuracy of CNN-GA gradually improves until the 15th generation, when it reaches around 95% classification accuracy. When CNN-GA is run without the crossover operation, it converges after the seventh generation and only obtains a classification accuracy of roughly 75%. The mutation probability is actually not set too high. As a result, the architectures of CNNs will not vary significantly over time. As a result, unless the crossover operation is used, the final classification accuracy is not very promising unless the process is controlled for a long time. The Large-scale Evolution algorithm, for example, does not execute any crossover operations and uses a total of 5, 500 GPU days on the two datasets [6]. CNN-GA, on the other hand, requires only 75 GPU days and achieves greater classification accuracy than Large-scale Evolution.

In addition to the first generation, the standard deviations of classification accuracy are not significantly different at the 2, 5, 13, and 15th generations, as shown in Fig. 8.3. This number is better without the genetic operator at the eighth generation in CNN-GA than it is with it. In all other generations, however, the standard deviations of CNN-GA are better than those without the crossover operator, which account for 14/20. This outcome can be simply explained using the crossover operator theory of local search. When the crossover operator is turned off, the algorithm just performs a global search, resulting in higher standard deviations.

In conclusion, the crossover operator of CNN-GA improves performance while using significantly less computational resources.

8.4.4 Performance of Acceleration Components

Two components have been designed in CNN-GA to speed up the process of fitness evaluations. The first is the evaluation of asynchronous fitness on available GPUs. The cache component is the other. Both have the same goal, yet they operate at different levels.

Comparison tests will be not performed in this regard because it is evident that the asynchronous fitness evaluation designed in CNN-GA is capable of improving the fitness evaluation. With one more GPU available for concurrently evaluating fitness, the “GPU day” might be cut in half.

The number of individuals, whose fitness values are derived from the cache in each generation, is counted and compared with those whose fitness values are gained through GPU training. The findings of the comparison are given in Fig. 8.4, where the horizontal axis represents the number of generations and the vertical axis represents the number of individuals who obtained fitness from cache and training. Individuals who get their fitness from the intended cache component appear in the second generation, as seen in Fig. 8.4. At the 2, 4, 7, 8, 9, 12, 13, 15, and 16th generations, approximately three individuals get their fitness from the cache. This number is increased to six for the 5, 17, and 18th generations. Furthermore, at the 5 and 18th generations, there are more than eight individuals that do not use GPUs for their fitness evaluations. In conclusion, the cache provides fitness to 92 individuals, or about 25% of the overall population. This means that with the designed cache component, the GPU days required to execute the fitness evaluation are cut one quarter.

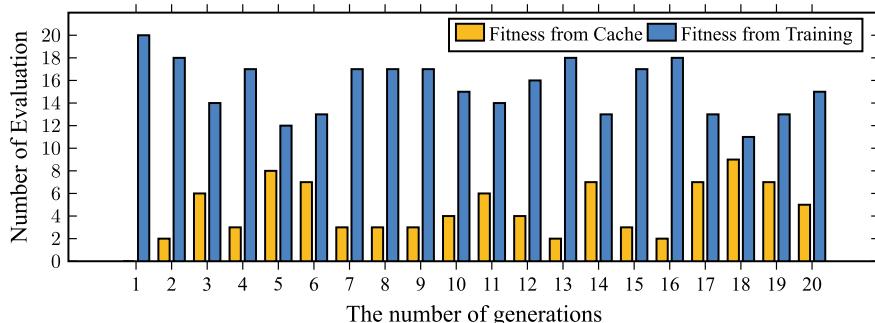


Fig. 8.4 The comparisons in terms of the numbers of fitness evaluations which are from the designed cache component and trained from scratch in each generation on the CIFAR-10 dataset

8.4.5 Evolution Trajectory

The evolution trajectory of CNN-GA on the CIFAR-10 dataset is illustrated in Fig. 8.5 to better understand the details of CNN-GA in finding the CNN architectures. To do so, the individuals chosen through environmental selection are collected in each generation, then the boxplot [42] is used to display the statistics in terms of classification accuracy. At the same time, both a solid line and a dashed line are used to connect the median and best classification accuracy during each generation. The number of generations is represented on the horizon axis, while the classification accuracy is represented on the vertical axis in Fig. 8.5.

Shown in Fig. 8.5, as the evolution progresses, both the best and median categorization accuracy improve. The fluctuation in classification accuracy with each generation becomes smaller and lower as the height of each box is investigated, implying that the evolution towards a steady state in finding the CNN architectures on the CIFAR-10 dataset. Furthermore, the classification accuracy rises dramatically from the first to the second generation, which is owing to the population is randomly initialized at the start of development. From the second to the fourth generations, the classification accuracy improves significantly less than in prior generations, then it rapidly increases until the 15-th generation. Since then, the classification accuracy has been relatively constant until the evolution ends, implying that the 20 generation setting is suitable because CNN-GA converges with this value.

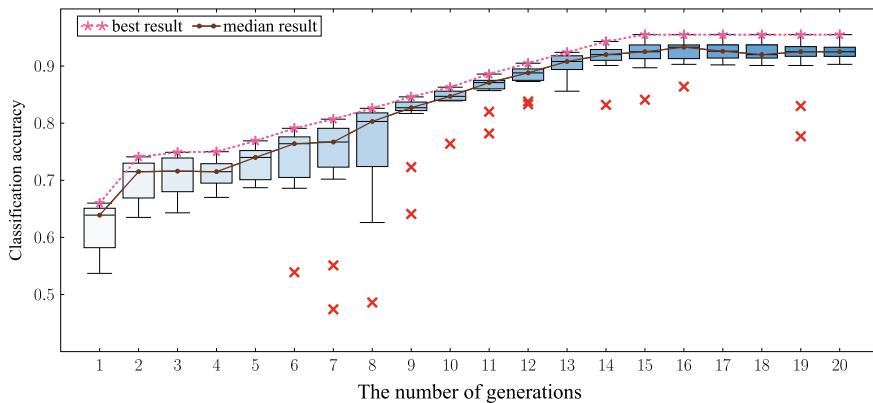


Fig. 8.5 The evolution trajectory of CNN-GA in finding the best CNN architecture on the CIFAR-10 dataset

8.5 Chapter Summary

In this chapter, we introduced the architecture design algorithm CNN-GA, which is based on skip connections. In particular, there are two types of skip connections based on the addition or concatenation of the resulting feature maps. Commonly, the skip connections of addition are much more popular because this will not cause significant computation cost, where the number of resulting feature maps is kept unchanged. Consequently, CNN-GA is just designed with this kind of skip connection. Like other algorithms discussed in this paper, in addition to the evaluation of the overall performance, the core components of CNN-GA, and also the evolutionary trajectory, the architecture resulting from CNN-GA is also transferred to ImageNet for validation.

We have introduced multiple architecture design algorithms solely upon GAs. In the next chapter, we will introduce another architecture design algorithm by hybrid using GA and PSO.

References

1. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>.
2. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958. ISSN 1532-4435.
3. Hawkins, D. M. (2004). The problem of overfitting. *Journal of Chemical Information and Computer Sciences*, 44(1), 1–12.
4. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
5. He, K., Zhang, X., Ren, S., & Sun, J. (2016b). Identity mappings in deep residual networks. In *Lecture Notes in Computer Science* (pp. 630–645). Springer.
6. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).
7. Liu, H., Simonyan, K., Vinyals, O., Fernand C., & Kavukcuoglu, K. (2015b). Hierarchical representations for efficient architecture search. [arXiv:1711.00436](https://arxiv.org/abs/1711.00436).
8. Housley, R. (2004). A 224-bit one-way hash function: Sha-224. RFC 3874.
9. Nasrabadi, N. M. (2007). Pattern recognition and machine learning. *Journal of Electronic Imaging*, 16(4), 049901.
10. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323).
11. Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in match-normalized models. In *Advances in Neural Information Processing Systems* (pp. 1945–1953). Curran Associates, Inc.
12. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).

13. Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade* (2nd ed.) (pp. 421–436). Springer. ISBN 978-3-642-35289-8. https://doi.org/10.1007/978-3-642-35289-8_25.
14. Helfenstein, R., & Koko, J. (2012). Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, 236(15), 3584–3590.
15. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. [arXiv:1603.04467](https://arxiv.org/abs/1603.04467).
16. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A. (2017). Automatic differentiation in pytorch. <https://openreview.net/forum?id=BJJsrmfCZ>.
17. Miller, B. L., Goldberg, D. E., et al. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3), 193–212.
18. Srinivas, M., & Patnaik, L. M. (1994). Genetic algorithms: A survey. *Computer*, 27(6), 17–26.
19. Michalewicz, Z., & Hartley, S. J. (1996). Genetic algorithms + data structures = evolution programs. *Mathematical Intelligencer*, 18 (3), 71.
20. Goldberg, D. E., & Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine Learning*, 3(2), 95–99.
21. Davis, L. (1991). *Handbook of genetic algorithms*. Van Nostrand Reinhold.
22. Anderson-Cook, C. M. (2005). *Practical genetic algorithms*.
23. Malik, S., & Wadhwa, S. (2014). Preventing premature convergence in genetic algorithm using dgca and elitist technique. *international Journal of Advanced Research in Computer Science and Software Engineering*, 4(6).
24. Zhang, G., Gu, Y., Hu, L., & Jin, W. (2003). A novel genetic algorithm and its application to digital filter design. In *Proceedings of 2003 Intelligent Transportation Systems* (Vol. 2, pp. 1600–1605). IEEE.
25. Bhandari, D., Murthy, C. A., & Pal, S. K. (1996). Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence*, 10(06), 731–747.
26. Simonyan, K.,& Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
27. Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning* (pp. 1319–1327).
28. Lin, M., Chen, Q., & Yan, S. (2014). Network in network. In *Proceedings of the 2014 International Conference on Learning Representations*.
29. Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015b). Highway networks. In *Proceedings of the 2015 International Conference on Learning Representations Workshop*.
30. Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for simplicity: the all convolutional net. In *Proceedings of the 2015 International Conference on Learning Representations*.
31. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115 (3), 211–252. ISSN 0920-5691, 1573-1405. <https://doi.org/10.1007/s11263-015-0816-y>.
32. Zhong, Z., Yan, J., & Liu, C.-L. (2018). Practical network blocks design with q-learning. In *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*.
33. Zoph, B., Vasudevan, V., Shlens, J., Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 8697–8710).
34. Zoph, B., & Le, WQ. V. (2016). Neural architecture search with reinforcement learning. [arXiv:1611.01578](https://arxiv.org/abs/1611.01578).
35. Xie, L.,& Yuille, A. L. (2017). Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017* (pp. 1388–1397). <https://doi.org/10.1109/ICCV.2017.154>.

36. Baker, B., Gupta, O., Naik, N., & Raskar, R. (2016). Designing neural network architectures using reinforcement learning. [arXiv:1611.02167](https://arxiv.org/abs/1611.02167).
37. Suganuma, M., Shirakawa, S., & Nagao, T. (2018). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 5369–5373). <https://doi.org/10.24963/ijcai.2018/755>.
38. DeVries, T., & Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. [arXiv:1708.04552](https://arxiv.org/abs/1708.04552).
39. Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), 67–82.
40. Back, T. (1996). *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press.
41. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6), 84–90. <https://doi.org/10.1145/3065386>.
42. Williamson, D. F., Parker, R. A., & Kendrick, J. S. (1989). The box plot: a simple visual method to interpret data. *Annals of Internal Medicine*, 110(11), 916–921.

Chapter 9

Hybrid GA and PSO for Architecture Design



9.1 Introduction

In this chapter, a new approach based on EC is introduced for automatically searching for the optimal CNN architecture and determining whether or not to use shortcut connections between one layer and its forward layer. After that, a two-level encoding strategy is applied to a hybrid EC methodology that is composed of a GA and a PSO. This allows for the generation of both the network architecture and the shortcut connections within it. The technique is referred to as *DynamicNet* because to the fact that during the course of the evolutionary process, both the architecture and the shortcut connections are determined dynamically without any involvement from a human being. On three widely used datasets that have differing degrees of complexity, *DynamicNet* will be evaluated in comparison with one method that is based on EC and 12 methods that are considered to be state-of-the-art. The following is a list of the specific goals and contributions:

- Develop a new encoding technique that makes use of both the CNN architecture and shortcut connections in order to meet the challenge. A two-level encoding is utilized since the accuracy of the classification is dependent on the CNN architecture, which in turn is determined by the shortcut connections, and the shortcut connections influence how effectively the CNN can be trained. The CNN architecture is shown at the first-level, while the shortcut connections are shown at the second-level. These two levels are represented by means of a decimal value vector and a binary value vector, respectively.
- Design a hybrid algorithm that is capable of two-level encoding so that it can be used. In order to evolve CNN architectures, a variable-length PSO method is utilized. This is due to the fact that PSO performs well while continuously optimizing. On the other hand, GA are utilized to evolve shortcut connections due to the fact that GA is effective when applied to optimization problems involving binary values.

- A new method for fitness evaluation is designed to enhance the effectiveness of encoded CNN. The fitness value of the method is represented by classification accuracy. Each evaluation necessitates the costly process of training the encoded CNN. In order to accelerate the training, a limited number of training epochs are utilized. To improve the accuracy of classification, an automated method is set up to search through a series of learning rates for the best one.

9.2 Algorithm Details

9.2.1 Overall Structure of the System

The framework of the system is depicted in Fig. 9.1. The dataset is partitioned into two sets, i.e. training and test set, with the training set itself being further subdivided into two parts, the training part and test part. The training and test parts are handed over to the EC process, which is the HGAPSO algorithm. The NN is trained using the training part, whereas the test part, which is distinct from the training part, is used to compute the fitness value. EC produces optimal individual in the process of evolution. The CNN evaluation technique concludes with optimal CNN architecture being trained on the entirety of the training set. Following this step, the test accuracy of the trained CNN model is achieved, which serves as the final output of the system.

9.2.2 The Evolved CNN Architecture-DynamicNet

When looking at the figures for ResNet and DenseNet side by side, it is clear that each layer in ResNet only has a maximum of two connections to the layers below it. DenseNet is characterized by its highly interconnected structure; hence, the number

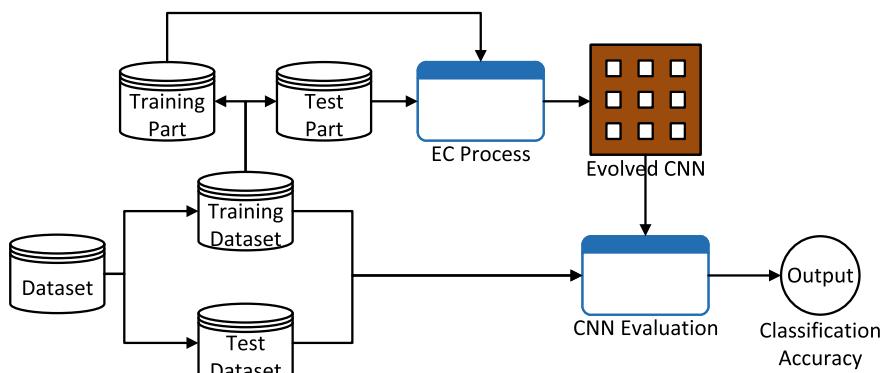


Fig. 9.1 The flowchart of the experimental process

of layers that are represented by the connections that each subsequent layer receives from its predecessors is equal to the number of layers that came before it. As a consequence of this, the number of feature maps in the input layer is equal to the sum of the numbers of feature maps in all of the layers that came before it. This leads to an exponential increase in the number of feature maps, particularly in layers that are relatively close to the layer that will be output. The strategy utilized in DenseNet involves sectioning off the entirety of the CNN into a number of different blocks that are referred to collectively as DBs. The transition layer, which consists of a convolutional layer and a pooling layer, is then applied to each block. This allows for a reduction in the amount of feature maps to half of what was originally input. The filter size and stride size of convolutional layer are set to three and one. Fixed hyperparameters, such as 2×2 for the kernel size and two for the stride size, are also utilized in the pooling layer. Because DynamicNet is so interconnected, the amount of feature maps may face the same problem of exponential expansion. As a result, DynamicNet employs the similar method of DBs.

A number of convolutional layers are contained within each block, each of which has a filter size of 3×3 , and a stride size of one. The *growth rate* refers to the pace at which the total number of input feature maps increases after each subsequent layer. Because the number of *blocks*, *convolutional layers*, and *growth rate* are all manually designed in DenseNet, it is necessary to have in-depth knowledge of the domain in addition to performing a large number of manual experiments in order to find an architecture that is suitable. In HGAPSO, the design of these three hyperparameters will likewise be accomplished automatically.

9.2.3 HGAPSO Encoding Strategy

Within DynamicNet, several blocks of the network are connected to one another via transition layers and shortcut connections that are constructed between the layers of network blocks. The pattern used to construct network can be used to group the architecture-related hyperparameters into shortcut connections and architecture. When it comes to the architecture of the network, there are a few different hyperparameters that need to be developed. These include the block number, the convolutional layer contained inside each block, and the convolutional layer growth rate contained within each block. In addition to the densely connected architecture in DenseNet, HGAPSO will investigate the various topologies of shortcut connections, also known as the various combinations of partial shortcut connections in each block. In this way, the effective features can be preserved and the useless ones are eliminated.

According to the design and the results of the hyperparameter analysis, the process of encoding could be broken down into two distinct processes. The first that needs to be done is the encoding of hyperparameters of the CNN architecture. As can be seen in Fig. 9.2a, each hyperparameter corresponds to a different dimension of the architecture encoding. Because there are two dimensions associated with the vector, the first dimension is the number of blocks, and the number of convolutional layers

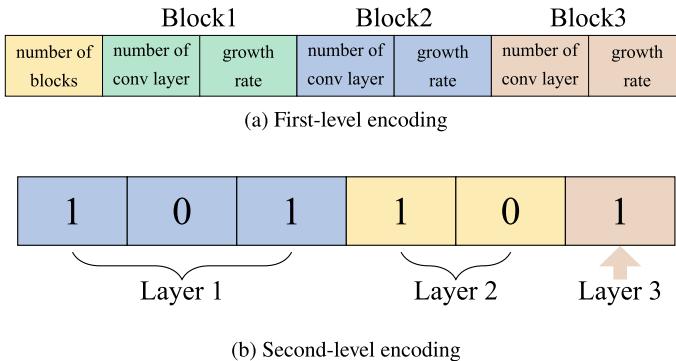


Fig. 9.2 HGAPSO encoding

and the growth rate are the two hyperparameters that are associated with each block. The encoding process goes from the first-level to second-level, so the former is the basis and the latter can transform the encoding into a binary vector. One example of a single block that consists of five layers is presented in Fig. 9.2. Each dimension denotes a shortcut connection that can be made quickly between two layers that are not adjacent to one another. However the layers that are adjacent to one another are always connected. The shortcut connections of first layer to the third, fourth, and fifth layers are denoted by the three binary digits—101, where a value of 1 indicates that the connection is present while a value of 0 is absent. As shown in Fig. 9.2, the full binary vector that represents the shortcut connections of the entire block is composed of a number of binary vectors that are identical to one another.

9.2.4 HGAPSO Search

9.2.4.1 Overview

As stated in Algorithm 1, the method consists of two levels of evolution based on the two-level encoding strategy. The design of the first-level encoded CNNs is evolved using first-level evolution, while the best combination of shortcut connections is determined using second-level evolution. There are various reasons to distinguish architecture from the evolution of shortcut-connection combinations. First, the architecture and the shortcut connections have different roles in the evolution process, as the former represents the capacity of the network, while the latter assists the network to achieve better performance during training. In addition, encoding hyperparameters of both into a vector can introduce noise into the search space.

Although the computational cost of two-level evolution is high, its search space can be effectively mitigated. Because the second-level strategy divides the large search space into two smaller search spaces, the interference of the large search

Algorithm 1: Framework of HGAPSO

```

1  $P \leftarrow$  Initialize the population with first-level encoding;
2  $P_{best}, G_{best} \leftarrow$  Empty PSO Personal Best and global best;
3 while first-level termination criterion is not satisfied do
4    $P \leftarrow$  Update the population with first-level PSO evolution detailed in Sect. 9.2.4.2;
5   for particle ind in population  $P$  do
6      $P_{sub} \leftarrow$  Initialize the population with second-level encoding;
7     while second-level termination criterion is not satisfied do
8        $P_{sub} \leftarrow$  Update the population with second-level GA evolution detailed in
      Sect. 9.2.4.3;
9       evaluate the fitness value of each individual;
10       $P_{subbest} \leftarrow$  retrieve the best individual in  $P_{sub}$ ;
11    end
12    Update  $P_{best}$  if  $P_{subbest}$  is better than  $P_{best}$ ;
13  end
14   $G_{best} \leftarrow$  retrieve the best individual in  $P$ ;
15 end

```

space is greatly attenuated. In addition, the second-level evolution only depends on the network architecture obtained in the first-level evolution, so when the first-level evolution outputs the architecture, the two can be processed in parallel in the subsequent optimization process, which will significantly speed up the process.

9.2.4.2 HGAPSO First-Level PSO Evolution

The first-level evolution process shown in Algorithm 2, i.e., using the PSO search architecture. PSO has been shown to be efficient and successful in optimization tasks using decimal values, so it makes sense to use PSO to operate on the encoding vectors of the first-level encoding. Since the dimensionality of the encoding vector is not fixed, a variable-length PSO is used to solve the problem. The premise of using the EC operator is to find matching blocks of the specific same input feature map in two different individuals. This is because for each block, the corresponding input feature map size is different, and it makes the most sense if the input of the feature map can be matched to a specific block. In the phase of HGAPSO proceeding to PSO evolution, the present particle is distinct from both the personal best and the global best in terms of their length, which means that it is necessary to find blocks that match the personal best and the global best, i.e., these blocks have the same output feature.

The first dimension of the vector corresponds to the number of blocks. The depth of the CNN architecture changes as the number of blocks changes, allowing the CNN architecture to adapt while the PSO population remains diverse. However, changing the number of blocks causes a significant change in the CNN architecture, and if it changes too frequently, each CNN architecture evolution may be too short to achieve good performance, so it is better to wait until other hyperparameters have

Algorithm 2: HGAPSO first-level PSO evolution

Input: The current particle ind , the personal best P_{best} , the global best G_{best} , the rate of changing the number of blocks r_{cb} .

- 1 $rnd \leftarrow$ Generate a random number;
- 2 locate the matched blocks of the particle ind ;
- 3 Alter the velocity and position of the matched blocks of the particle ind ;
- 4 **if** $rnd < r_{cb}$ **then**
- 5 Alter the velocity and position of the dimension of number of blocks of the particle ind ;
- 6 Randomly cut or generate the blocks to the value of the number of blocks;
- 7 **end**

been optimized given the specific number of blocks before proceeding. The rate of changing the number of blocks in the vector, which is a real value between $[0, 1]$, is introduced to maintain the diversity of the number of blocks and to reduce the disturbance caused by changing the number of blocks often. As a result, the rate of the number of blocks can be changed depending on the tasks to control the preference for diversity or stability.

If the number of blocks is increased or decreased, then a certain number of blocks will need to be arbitrarily sliced or generated in order to satisfy the prerequisite for the first dimension. If the number of blocks is increased from three to four, the hyperparameters of fourth block must be generated at random using the first-level encoding strategy and appended to the vector of three blocks. If the number of blocks is lowered from four to three, the final block is removed. When removing a block (or blocks) in HGAPSO, the process is always started from the most recent layer. This is due to the fact that the removal of a block does not affect the sizes of feature maps affiliated to remaining blocks.

9.2.4.3 HGAPSO Second-Level GA Evolution

As soon as the CNN architecture is created from the first-level evolution, the dimension of the second-level encoding is fixed. This makes it possible for the encoded vector to be represented by a binary vector with a fixed length, as is demonstrated in Sect. 9.2.3. Because GAs are good at optimizing binary issues, they were chosen as the second-level evolution algorithm.

9.2.5 HGAPSO Fitness Evaluations

It is shown in Algorithm 1 that the fitness evaluation is performed only at the second-level evolution, and the optimal individual fitness value in the GA is also used to represent the fitness of the optimal particle in the PSO. In addition, the Adam optimizer [1] is used for backpropagation and trained a certain number of epochs. The

accuracy of the trained CNN for the test part of the training data determines the value assigned to the individual fitness.

The number of epochs and the initial learning rate of Adam Optimizer are the two hyperparameters used in fitness evaluations. Because of the hardware available and the relatively short experimental time, five epochs are employed in the experiment. After deciding on the number of epochs, The baseline network DenseNet is utilized in the process of determining an initial learning rate for the purpose of optimizing a CNN with the specified depth and width, i.e., after determining the architecture of the CNN, the network with fully connected blocks is used to find the best initial learning rate among 0.9, 0.1, and 0.01.

In order to speed up the overall process of evolution, a portion of the dataset that was used for training is now being put to use in the second level of evolution. This is done due to the fact that second-level evolution requiring the highest amount of computing resources. Because the computing cost is modest, the entire training dataset is employed for the first-level evolution aiming at achieving a more constant performance given the architecture of a CNN. It is anticipated that doing so can generate a more accurate result.

9.3 Experimental Studies

9.3.1 Parameter Settings

All of the parameters are set based on PSO [2] and GAs [3] community conventions, with the complexity of the search space and the computational cost taken into consideration. Specifically, the range of layer number in each block is defined in [4, 8], the range of growth in each block is defined in [8, 32], the number of generations and the size of population are specified as 10 and 20, respectively. In addition, in PSO, c_1 and c_2 are both specified as 1.49618, and w is specified as 0.7298. Furthermore, in GA, the mutation rate and crossover rate are set as 0.01 and 0.09, and the elitism rate is set as 0.1.

9.3.2 State-of-the-Art Methods Versus HGAPSO

The results of the experiment are presented in Table 9.1, along with a comparison of HGAPSO to the state of the arts. The terms (– and (+) are used to indicate whether the HGAPSO result is significantly worse or better than the best result achieved by the corresponding peer competitor in order to clearly emphasize the comparative results. This is done so that the reader can make an informed decision. The term (–) denotes that the results of provider are either unobtainable or cannot be taken into account in any way.

Table 9.1 Classification errors of HGAPSO and the chosen peer competitors

	MDRBI	MB	CONVEX
CAE-2	45.23 (+)	2.48 (+)	–
SVM+Poly	54.41 (+)	3.69 (+)	19.82 (+)
RandNet-2	43.69 (+)	1.25 (+)	5.45 (+)
NNet		62.16 (+)	4.69 (+) 32.25 (+)
SAA-3		51.93 (+)	3.46 (+) 18.41 (+)
SVM+RBF	55.18 (+)	30.03 (+)	19.13 (+)
TIRBM	35.5 (+)	–	–
LDANet-2	38.54 (+)	1.05 (+)	7.22 (+)
ScatNet-2	50.48 (+)	1.27 (+)	6.50 (+)
PCANet-2 (Softmax)	35.86 (+)	1.40 (+)	4.19 (+)
DBN-3	47.39 (+)	3.11 (+)	18.63 (+)
PGBM+DN-1	36.76	–	–
HGAPSO (best)	10.529	0.744	1.032
HGAPSO (mean)	12.230	0.838	1.240
HGAPSO (standard deviation)	0.859	0.069	0.103
DECNN (mean)	37.551	1.457	11.192
DECNN (standard deviation)	2.447	0.113	1.943
P-value	0.0001	0.0001	0.0001

When compared to the error rates presented in Table 9.1, it is clear that HGAPSO accomplishes a significant amount of progress. In each of the three benchmark datasets, HGAPSO gets results that are significantly superior to those of its other peer competitors. The error rate is reduced on the MDRBI, MB, and CONVEX datasets by 10%, 1%, and 5%, respectively, when compared to the best competitor.

When the results of HGAPSO and DECNN are compared, Table 9.1 demonstrates that HGAPSO has a lower mean error rate as well as a lower standard deviation in comparison to DECNN. This indicates that the classification accuracy of HGAPSO has significantly improved.

9.3.3 Evolved CNN Architecture

By evolving the CNN architecture, it can be found that HGAPSO is able to achieve two stages of optimization, which are architecture optimization and shortcut connection optimization. This becomes apparent when one looks at the evolving CNN architectures. An evolved CNN design, for instance, consists of three blocks. There are four convolutional layers in the first block, with [0, 0, 0, 0, 1], [0, 1, 0, 1], [0, 0,

[1], [0, 0] and [1] representing connections from the input layer plus the subsequent three layers to the following layers, with 1 indicating a connection and 0 indicating no connection; the second block contains eight layers with a growth rate of 34 and corresponding connections [1, 0, 1, 0, 1, 0, 1, 0], [0, 1, 1, 1, 1, 0, 1], [1, 1, 1, 1, 1, 1, 0], [1, 1, 1, 0, 1], [1, 0, 0, 0], [0, 0, 0], [1, 1] and [0]; the third block contains 5 layers with corresponding connections [0, 0, 1, 1, 0], [0, 0, 0, 0], [1, 0, 0], [0, 1] and [0] while a growth rate is 39.

9.3.4 One-Run Result on CIFAR-10 Dataset

As mentioned earlier, testing HGAPSO was extremely computationally expensive. On a single GPU card, this experiment took one week to complete the experiment. In addition, it took 12 h to complete the run of the optimal architecture, which had a classification accuracy of 95.75%, ranking second among the state-of-the-art DNNs compiled on the rodrigob website, with classification accuracy values ranging from 75.86 to 96.53%. However, in order to manually fine-tune their performance, all state-of-the-art DNNs require incredibly specific domain knowledge in addition to a massive amount of experimentation. The fact that HGAPSO, on the other hand, can automatically grow the CNN architecture without the need for any kind of intervention from a human being is widely regarded as the most significant benefit.

9.4 Chapter Summary

In this chapter, we introduced a hybrid method based on both the GA and the PSO for the automated architecture design of CNNs. In particular, a two-level encoding method is designed in the hybrid method for both the architecture and the shortcut connections. This will collectively contribute to the performance of the resulting CNN architectures. The experiments against peer competitors demonstrated the effectiveness of this design. In addition, the architectures resulting from the hybrid method and the results only run on CIFAR-10 for once are also explored for the demonstration.

In principle, the encoding strategy bridges the utilization of EC methods and the real-world problem to be solved. When EC methods are used to design the CNN architectures automatically, the particular encoding strategy should be carefully treated. In the next chapter, we will introduce an algorithm with the encoding strategy focusing on the protocol of internet addresses.

References

1. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
2. Van den Bergh, F., & Engelbrecht, A. P. (2006). A study of particle swarm optimization particle trajectories. *Information Sciences*, 176(8), 937–971. <https://doi.org/10.1016/j.ins.2005.02.003>.
3. Digalakis, J. G., & Margaritis, K. G. (2000). An experimental study of benchmarking functions for genetic algorithms. *Proceedings of 2000 IEEE International Conference on Systems, Man and Cybernetics*. <https://doi.org/10.1109/icsmc.2000.886604>.

Chapter 10

Internet Protocol Based Architecture Design



10.1 Introduction

There has been a large number of research done to enhance utilizing EC for an evolved CNN architecture, but there has not been much study on using other EC approaches to develop CNN architectures, as a result, some other significant EC approaches are anticipated to be investigated for evolving CNN architectures without involving humans. PSO is chosen in this chapter because it has the benefits over traditional implementation, reduced computing cost, less parameters to tweak, and it has never been used to create the architectures of CNNs. However, because the best CNN architecture differs for different problems, the fixed-length encoding of particles in conventional PSO poses a significant problem for evolving CNN architectures, therefore a unique flexible encoding method is designed to overcome the fixed-length limitation, it is the most basic aspect of the work in this chapter.

The encoding method in this design, in particular, is based on the internet protocol (IP) address, which will be described briefly for a deeper understanding of the algorithm design. An IP address is a number identification provided to every device connected to a computer network that communicates with it [1]. The subnet is introduced to determine the network of an IP address, which is commonly defined in classless inter-domain routing (CIDR) style [2] by integrating the initial IP address with the size of the subnet mask. A network interface has both the IP address required to identify the host and its accompanying subnet utilized to distinguish distinct networks. For example, a standard 32-bit IP address may be 222.222.31.250, and a normal subnet could carry the beginning IP address and the size of the subnet mask as 222.222.31.0/8, indicating that the IP address in the subnet begins from 222.222.31.0 and the subnet mask has a size of 8, an IP range definition from 222.222.31.0 to 222.222.31.255.

The binary string clearly meets the criteria of encoding CNN layers into particles. Nevertheless, in this case, a big integer transformed from the binary string must be used as one dimension of the particle vector, this could result in a terrible PSO search

time. In contrast, the IP structure separates a big integer indicating the identification of a device in a wide network into multiple decimal values smaller than 256, each is kept in a single byte of the IP address, to make the IP address readable and memorable. The binary string may thus be broken into many bytes, with a single dimension of the particle vector being represented by each byte. PSO convergence may be aided by dividing one dimension of a huge amount into numerous dimensions of small numbers since all dimensions can be continuously learned in each round of particle updates and the search space of a single split dimension is significantly smaller. This principle is used in the intended encoding approach to achieve the flexibility of encoding numerous sorts of layers into a particle and substantially shorten the learning period.

The overarching purpose in this chapter is to introduce an effective and efficient PSO approach, dubbed IPPSO, for automatically discovering appropriate CNN architectures. This precise goals are as follows:

1. Build a new PSO algorithm on top of a new particle encoding approach capable of successfully encoding a CNN architecture.
2. Develop a way to overcome the fixed-length encoding constraint of classical PSO to design variable-length CNN architectures. To achieve a variable-length particle, a new layer named disabled layer will be presented.
3. A fitness evaluation approach uses an incomplete dataset rather than the entire dataset to considerably speed up the evolutionary procedure.

10.2 Algorithm Details

The IPPSO approach for evolving deep CNNs are going to be detailed in this section.

10.2.1 Algorithm Overview

Algorithm 1: IPPSO Framework

```

1  $P \leftarrow$  Initialize the population with the particle encoding strategy;
2  $P_{id} \leftarrow empty;$ 
3  $P_{gd} \leftarrow empty;$ 
4 while termination requirement is not met do
5   update velocity and position of every particle depicted in Algorithm 3;
6   evaluate the fitness value of every particle;
7   update  $P_{id}$  and  $P_{gd}$ ;
8 end

```

The structure of IPPSO is described in Algorithm 1. There are three key steps: initializing the population utilizing the particle encoding approach outlined in Sect. 10.2.2, position and velocity updating, and the termination requirement determination.

10.2.2 Encoding Strategy of Particle

The IP address served as the inspiration for the IPPSO encoding method. Despite the fact that the CNN architecture consists of three distinct layers (convolutional layer, pooling layer, and fully-connected layer), the encoded information of each type of layer differs with regard to both the amount of parameters and the range of each parameter. It is possible to generate a network IP address with a specified length and sufficient capacity to support all forms of CNN layers, the IP may then be partitioned into various subsets, each of which is utilized to construct a particular type of CNN layer.

To begin, the size of the binary string under the IP-based encoding structure must be determined. In terms of the convolutional layers, the three important parameters—filter size, number of feature maps, and stride size—are the primary elements influencing the performance of CNNs. Second, according to the size of the benchmark datasets, the parameter ranges are set to [1, 8], [1, 128], and [1, 4] for the three previously listed parameters. Third, using a CNN architecture with a filter size of two, an amount of feature maps of seven, as well as a stride size of two, the decimal values may be represented as binary strings of 010, 011 1111, and 11, respectively, where the decimal value is converted to binary form and the binary string is filled with zeros until it reaches the desired number of bits. Finally, the total amount of bits is 12 and the example binary string is 010 011 1111 11, which is obtained by combining the binary strings of the three parameters. With regard to pooling layers and fully-connected layers, the entire amount of bits and the sample binary string may be acquired by repeating the procedure of the convolutional layer. Because the maximum amount of bits to indicate a layer is 12, and the unit of an IP address is one byte—eight bits, the 12 bit IP address are going to need two bytes.

Furthermore, the subnets over all kinds of CNN layers must be specified base on the amount of bits for every layer, as well as the subnet are going to be represented using the CIDR style. Three subnets, which has sufficient capacity to simulate all three types of CNN layers, have to be created because there are three types of CNN layers. Beginning at the convolutional layer, the beginning IP of the subnet is set to 0.0; additionally, because the designed 2-byte IP address has a total length of 16 bits and the convolutional layer requires a total of 12 bits, subtracting the entire amount of bits from the length of the IP address yields the subnet mask length, with a range of 0.0 to 15.255, this yields a subnet representation of 0.0/4. The initial IP address of the pooling layer is 16.0, which is acquired by adding one to the final IP address of the convolutional layer, and the length of the subnet mask is 5, which is calculated in the identical manner as the convolutional layer. As a consequence, the subnet of

Table 10.1 The three kinds of CNN layers, as well as the disabled layer are divided into four subnets

IP Range	Subnet (CIDR)	Layer type
0.0–15.255	0.0/4	Convolutional layer
24.0–31.255	24.0/5	Pooling layer
16.0–23.255	16.0/5	Fully connected layer
32.0–39.255	32.0/5	Disabled layer

Table 10.2 An example of IP addresses—one for each kind of CNN layers

IP address	Binary (filled to 2 bytes)	Layer type
4.255	(0000) 010 011 1111 11	Convolutional layer
15.63	(0000) 11 11 0 011 1111	Pooling layer
7.255	(00000) 111 11111111	Fully connected layer
7.255	(00000) 111 11111111	Disabled layer

the pooling layer is represented by 16.0/5 with a range of 16.0 to 23.255. In the same way, the subnet 24.0/5 of the fully connected layer, which has a range of 24.0 to 31.255, is designed. All of the subnets are illustrated in Table 10.1 to make the subnets apparent.

Because the particle length of PSO is constant after initialization, an effective method of deactivating parts of the layers in the encoded particle vector will be employed to accomplish this goal. As a result, the introduction of a new layer type known as disabled layer and a related subnet known as disabled subnet. In order to attain an equivalent probability for the disabled layer, the disabled subnet is 32.0/5 and a range of 32.0–39.255, as indicated in Table 10.1, with each layer encoded as a 2-byte IP address. Table 10.2 indicates how the example is converted to IP addresses by merging all of the binary strings of the parameters of each layer into one binary string, zeros are appended to the combined binary string until its length reaches two bytes, the subnet mask is applied to the binary string, and transforming the final binary string to an IP address with a single byte as a unit separated by periods. For example, the example binary string for the convolutional layer is 010 011 1111 11, which is packed to 0000 010 011 1111 11 to achieve the length of two bytes; and then 2-byte binary strings—0000 0100 and 1111 1111, can be achieved by utilizing the subnet mask, which adds the beginning IP address of the subnet to the binary string. Eventually, the IP address 4.255 is obtained by translating the first byte to the decimal value of 4 and the second byte to the decimal value of 255.

The velocity and position of PSO can be determined when each layer is converted to a 2-byte IP address. Nonetheless, there are some parameters that must take into consideration first. They are `max_length` denoting the maximum number of CNN layers and `max_fully_connected` denoting the maximum fully connected layers with the restriction of at least one fully connected layer. The position and velocity data

4.255 (C)	15.63 (P)	4.255 (C)	7.255 (D)	7.255 (F)
-----------	-----------	-----------	-----------	-----------

Fig. 10.1 An illustration of IP addresses in a particle that has five CNN layers

4	255	15	63	4	255	7	255	7	255
---	-----	----	----	---	-----	---	-----	---	-----

Fig. 10.2 An illustration of a particle vector encoded with five CNN layers

will be encoded as a byte array with a constant length of $\text{maximum_length} \times 2$, with every byte representing one particle dimension.

CNN architecture is encoded using a particle vector as an example to illustrate how it handles CNN architecture with varied length. Assuming that the maximum number of CNN layers (i.e., `max_length`) is five, a series of IP addresses indicating a CNN architecture with the maximum number of 5 layers may be encoded into five IP addresses in Fig. 10.1 utilizing example IP addresses in Table 10.2, where C, P, F, and D represent a convolutional layer, a pooling layer, a fully connected layer, and a disabled layer, respectively. Figure 10.2 shows the appropriate particle vector with a dimension of 10. Because one of the layers in the example is disabled, the total amount of layers is four. Nevertheless, the seventh and eighth dimensions of the particle vector can change to 18 and 143 after a few PSO updates, accordingly, turning the third IP address denoting a disabled layer into a pooling layer, resulting in a five-layer CNN architecture in the updated particle. In contrast, the fifth and sixth dimensions of the particle vector can change to 35 and 255 after a few updates, accordingly, causing the third IP address to drop into the disabled subnet, resulting in a total of layers to three. To summarize, the particle using the IPPSO encoding technique can represent variable-length CNN architectures from three to five as illustrated in this example.

10.2.3 Initialization of Population

When the population size is specified, individuals are generated at random till the population size is reached with regard to population initialization. An empty vector is created for each individual, and each element is utilized to save a network interface comprising the IP address and subnet information. Every element from the second to $(\text{max_length} - \text{max_fully_connected})$ layer may be filled by a convolutional layer, pooling layer, or disabled layer; till the first fully-connected layer is added, it may be filled with any of the four sorts of layers from $(\text{max_length} - \text{max_fully_connected})$ to $(\text{max_length} - 1)$ layers, only fully connected layers or disabled layers are permitted after that. The final element is always a fully connected layer of the same size as the class number. Furthermore, every layer may be created using random configuration - a randomly generated IP address within an effective subnet.

10.2.4 Evaluation of Fitness

Algorithm 2: Fitness Evaluation

Input: The population P , the training epoch number k , the training set D_{train} , the fitness evaluation dataset $D_{fitness}$, the batch size $batch_size$.

Output: The population with fitness P .

```

1 for individual  $s$  in  $P$  do
2   |  $i \leftarrow 1$ ;
3   | while  $i <= k$  do
4     |   | Train the connection weights of the CNN
        |   | represented by individual  $s$ ;
5   | end
6   |  $accy\_list \leftarrow$  Batch-evaluate the trained model on the dataset  $D_{fitness}$  which has the
      | batch size  $batch\_size$  and save the accuracy for each batch;
7   |  $mean \leftarrow$  Calculate the mean value of  $acc\_list$ ;
8   |  $fitness \leftarrow mean$ ;
9   |  $P \leftarrow$  Update the fitness of the individual  $ind$  in the population  $P$ ;
10 end
11 Return  $P$ 
```

Before completing the fitness evaluation, it is necessary to select a suitable weight initialization approach, and Xavier weight initialization [3] is picked because it has been proven to be a successful method and is implemented in most deep learning libraries. Every individual is transformed into a CNN architecture, complete with its own set of parameters for fitness evaluation as shown in Algorithm 2. The first portion of the training dataset will be used to train it for k epochs. The second portion of the training dataset will be used to batch-evaluated the partly trained CNN, resulting in a set of accuracy values. Eventually, for each individual, the mean value of the accuracy is determined, which is kept as the individual fitness.

10.2.5 Update Particle with Velocity Clamping

Because each layer is encoded as a two-bit interface in the particle vector and the acceleration coefficients for every byte in Algorithm 3 would like to be regulated, the two acceleration coefficients must be used to implement the two float arrays of size two, as indicated in Eq. 10.1. P_{id} and P_{gd} refer to the i th byte of the IP address of the local best and global best, accordingly, which are decimal values, and w, r_1, r_2 are set the same as classical PSO. The main difference is in the implemented way of the acceleration coefficients. In traditional PSO, each of the acceleration coefficients has a singular value, whereas $c_1[i]$ and $c_2[i]$ are the acceleration coefficients for the i th byte of the IP address, where in the case of 2-byte IP encoding, i is either one or two. Because various parameters might fall into distinct bytes of the IP address, the

Algorithm 3: Update Particle with Velocity Clamping

Input: particle individual vector ind , acceleration coefficient array for P_{id} c_1 , acceleration coefficient array for P_{gd} c_2 , inertia weight w , max velocity array v_{max} .

Output: individual vector ind .

```

1 for element interface in ind do
2   |   i  $\leftarrow 0$ ;
3   |   for i < number of bytes of IP address in interface do
4     |     |   x  $\leftarrow$  the ith byte of the IP address in the interface;
5     |     |   ( $r_1, r_2$ )  $\leftarrow$  uniformly generate  $r_1, r_2$  between [0, 1];
6     |     |    $v_{new} \leftarrow$  Update velocity based on Eq. 10.1;
7     |     |    $v_{new} \leftarrow$  Apply velocity clamping using  $v_{max}$ ;
8     |     |    $x_{new} \leftarrow x + v_{new}$ ;
9     |     |   if  $x_{new} > 255$  then
10    |       |     |    $x_{new} \leftarrow x_{new} - 255$ ;
11   |   end
12 | end
13 end
14 fitness  $\leftarrow$  evaluate the updated individual ind;
15 ( $P_{id}, P_{gd}$ )  $\leftarrow$  Update pbest and gbest by comparing their fitness;
16 Return ind
```

acceleration coefficients must be divided for each byte of the IP address, and fine-tuning the learning process may require the ability to investigate a single parameter more than others.

Following the definition of the coefficients, the velocity and position of each byte are updated in the particle by utilizing the corresponding coefficients for that byte. Because every interface in the particle vector has some constraints based on its position in the particle vector, for example, only a convolutional layer, pooling layer, or disabled layer may be used as the second interface. If the new interface does not belong to a valid subnet, it must be replaced by a valid subnet interface with a random IP address. The new particle is evaluated when all the bytes have been updated. In addition, the fitness value is compared to the local best and global best for their updating if necessary.

$$v_{new} = w * v + c_1[i] * r_1 * (P_{id} - x) + c_2[i] * r_2 * (P_{gd} - x) \quad (10.1)$$

10.2.6 Selection and Decoding of Best Individual

The best individual in PSO is regarded as the global best. It is possible to get a list of network interfaces from the global best particle vector by decoding every two bytes from left to right. The type of layer may be determined based on the subnets in Table 10.1, and the IP address can then be decoded into multiple sets of binary strings that represent the parameter values of the layers. After decoding all of the interfaces

in the particle vector in the global best, the final CNN design may be achieved by linking all of the decoded layers in the same order as the particle vector interfaces.

10.3 Experimental Design

The peer competitors and the parameter settings of IPPSO are going to be provided in this section.

10.3.1 Peer Competitors

The considered peer competitors are state-of-the-art algorithms which have demonstrated potential classification errors on the selected benchmarks. Based on the literature [4], TIRBM [5], CAE-2 [6], ScatNet-2 [7], PGMB+DNI [8], PCANet-2 (Softmax) [4], RandNet-2 [4], SVM+RBF [9], LDANet-2 [4], NNet [9], SVM+Poly [9], DBN-3 [9] and SAA-3 [9] are chosen as the peer competitors in the experiments.

10.3.2 Parameter Settings

Based on the conventions, all the parameter settings are specified following the communities of deep learning [10] as well as PSO [11]. Specifically, the maximum length of CNN layers `max_length` is set to nine, the maximum fully connected layers provided at least there with one fully connected layer `max_fully_connected` is set to three. The population size N and the number of training epochs before evaluating the trained CNN k are set to 30 and 10, respectively. The size of batch used to evaluate the CNN is set to 200. The array of acceleration coefficient are all set with the values of 1.49618, the weight of inertia for updating velocity is set to 0.7289.

TensorFlow [12] is used to implement IPPSO, and the code executes on a GPU computer having two GTX1080 cards. Because of the stochastic nature of IPPSO, each benchmark dataset is subjected to 30 independent runs, with the mean results utilized for comparisons unless otherwise noted. On every execution of the benchmark dataset, the trials take about two and a half hours. Section 10.4.2 depicts the CNN architectures evolved for each benchmark.

10.4 Experimental Results and Analysis

The classification performance, as well as the comparison to peer competitors, the CNN architectures produced by IPPSO, and the accompanying visualization, will be provided in this section.

10.4.1 Overall Performance

Table 10.3 shows the results of the experiment for each of the three benchmark datasets, with the final three rows indicating classification errors gained by IPPSO from the 30 executions in terms of best, mean, and standard deviations, meanwhile the remaining rows indicating the best classification errors from peer competitors. This follows the convention of deep learning community reporting the best results. The symbols “(+)” and “(−)” are supplied to show if the result produced by IPPSO is better or worse than the best result produced by the comparable peer competitor. The symbol “−” denotes that the result of the corresponding algorithm is unavailable or cannot be counted.

Table 10.3 clearly shows that when the mean classification errors of IPPSO are compared to the best result of the peer competitors, IPPSO comes in second place on the MB dataset, merely slightly behind LDANet-2. On the MDRBI dataset, which is the most difficult of the three benchmarks, IPPSO gets the best result, and the fifth best on the CS dataset, that is not perfect but quite competitive.

10.4.2 Evolved CNN Architectures

Despite the fact that IPPSO is conducted on every benchmark with 30 separate executions, just one is selected for this description. In particular, on the MB benchmark, the first four are the convolutional layers, which have the file size of 2, 6, 8, and 7, respectively, have the stride size of 1, 3, 4, and 4, respectively, and have the number of feature maps of 26, 82, 114, and 107, respectively. Following that are two fully connected layers having the neuron numbers of 1,686 and 10, respectively. On the MDRBI benchmark, the first four and the sixth are all convolutional layers, the fifth is a “AVERAGE” pooling layer having the kernel size and stride size of 5 and 3, respectively. Specifically, the five convolutional layers have the filter size of 2, 6, 7, 7, and 5, respectively, have the stride size of 1, 3, 4, 4, and 3, respectively, and have the number of feature maps of 32, 90, 101, 97, and 68, respectively. On the CS benchmark, the first four are also convolutional layers having the filter size of 1, 7, 1, and 6, respectively, the stride size of 1, 4, 1, and 3, respectively, and the number of feature maps of 11, 108, 8, and 92, respectively. After that, the remaining two are both fully connected layers with the respective neuron numbers of 906 and

Table 10.3 The classification errors of IPPSO and the chosen peer competitors on CS, MDRBI and MB datasets

Classifier	CS	MDRBI	MB
TIRBM	–	35.50(+)	–
ScatNet-2	6.50(–)	50.48(+)	1.27(+)
CAE-2	–	45.23(+)	2.48(+)
PGBM+DN-1	–	36.76(+)	–
PCANet-2 (Softmax)	4.19(–)	35.86(+)	1.40(+)
RandNet-2	5.45(–)	43.69(+)	1.25(+)
SVM+RBF	19.13(+)	55.18(+)	3.03(+)
LDANet-2	7.22(–)	38.54(+)	1.05(–)
NNet	32.25(+)	62.16(+)	4.69(+)
SVM+Poly	19.82(+)	56.41(+)	3.69(+)
DBN-3	18.63(+)	47.39(+)	3.11(+)
SAA-3	18.41(+)	51.93(+)	3.46(+)
IPPSO (best)	8.48	34.50	1.13
IPPSO (mean)	12.06	33	1.21
IPPSO (standard deviation)	2.25	2.96	0.103

2. The disabled layers are not visible in the resulting CNN designs since they were eliminated during the decoding process. As a result, it appears that IPPSO may learn a variable-length CNN architecture, as shown by the architectures stated above: six CNN layers for the MB and CS benchmarks as well as eight CNN layers for the MDRBI benchmark.

10.4.3 Trajectory Visualization

The PSO trajectory of the procedure of evolution is shown to better comprehend IPPSO.

In terms of the trajectory, the local best particle results in every generation and the global best particle in every generation from one round of the trials are achieved and shown in Fig. 10.3. Evidently, the global best is identified after only just few generations, and the particles continue to fly through the search space, but none of them can achieve a higher accuracy, indicating that PSO has found the optimum just after a few of steps. Despite the complexity of the optimization challenge, the PSO approach with only 30 particles may swiftly reach the optimum, demonstrating that PSO is effective and efficient on optimization tasks.

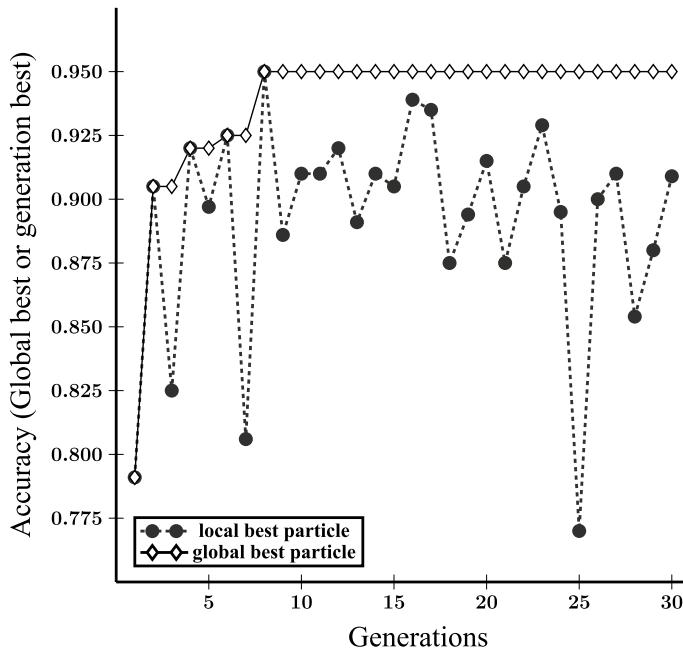


Fig. 10.3 PSO trajectory of the local best particle and the global best particle

10.5 Chapter Summary

In this chapter, we introduced the CNN architecture automation method with the novel encoding strategy, which is motivated by the design of the IP address. In addition, the concept of “subnet” often appearing in the IP address is mainly for determining the corresponding network. A standard IP address often has 32 bits, and a formal subnet typically takes the beginning IP address and the size of the subnet mask. In addition, the IP address naturally has the format of binary, which can be directly explored by the standard EC methods. Furthermore, the particular EC method used in this algorithm is PSO, and the overall performance along with the resulting architectures as well as the evolutionary trajectory are provided for justifying the effectiveness and efficiency of the designed algorithm.

GA and PSO are the representatives of EC methods and have been widely used to design the corresponding algorithms, such as the architecture design algorithm introduced in the previous chapters. In the next chapter, we will introduce a DE-based architecture algorithm, which also has similar merit to both GAs and PSO algorithms.

References

1. Postel, J. (1980). Dod standard internet protocol. *ACM SIGCOMM Computer Communication Review*, 10(4), 12–51.
2. Fuller, V., Li, T., Yu, J., & Varadhan, K. (1993). Classless inter-domain routing (cidr): An address assignment and aggregation strategy.
3. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
4. Chan, T.-H., Jia, K., Gao, S., Jiwen, L., Zeng, Z., & Ma, Y. (2015). Peanet: A simple deep learning baseline for image classification? *IEEE Transactions on Image Processing*, 24(12), 5017–5032.
5. Sohn, K., & Lee, H. (2012). Learning invariant representations with local transformations. <https://arxiv.org/abs/1206.6418>.
6. Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning* (pp. 833–840).
7. Bruna, J., & Mallat, S. (2013). Invariant scattering convolution networks. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1872–1886. <https://doi.org/10.1109/tpami.2012.230>.
8. Sohn K., Zhou G., Lee C., & Lee H. (2013). Learning and selecting features jointly with point-wise gated boltzmann machines. <https://dl.acm.org/citation.cfm?id=3042918>.
9. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning* (pp. 473–480). ACM. <https://doi.org/10.1145/1273496.1273556>.
10. Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade* (pp. 599–619). Springer.
11. Van den Bergh, F., & Engelbrecht, A. P. (2006). A study of particle swarm optimization particle trajectories. *Information Sciences*, 176(8), 937–971. <https://doi.org/10.1016/j.ins.2005.02.003>.
12. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M. et al. (2016). *Tensorflow: Large-scale machine learning on heterogeneous distributed systems*. [arXiv:1603.04467](https://arxiv.org/abs/1603.04467).

Chapter 11

Differential Evolution for Architecture Design



11.1 Introduction

The general goal of this chapter is to explore the capacity of DE, named DECNN, to evolve deep CNN architectures and parameters automatically. Designing new crossover and mutation operators of DE, as well as an encoding scheme, and a second crossover operator will help to achieve the goal. DECNN will be evaluated on six datasets of various complexity that are widely used and compared to 12 state-of-the-art methods. The objectives are as follows:

- Breaking the limitation of predefining CNN maximal depth by refining the existing effective encoding scheme which is employed by IPPSO;
- Developing and designing novel crossover and mutation operators for DECNN, which may be used on vectors with variable length to overcome the fixed-length constraint of standard DE method;
- Designing and incorporating a second crossover operator for DECNN to generate offspring in the following generations that represent the CNN architectures which differ from their parents in length.

11.2 Algorithm Details

To meet the need of evolving variable-length CNN architectures, DECNN employs the DE as the core EA and designs a second crossover operator to produce offspring with lengths different from their parents..

11.2.1 DECNN Algorithm Overview

The framework of DECNN is stated in Algorithm 1. First of all, the population P is initialized by the designed population initialization method, which will be discussed in Sect. 11.2.3, and then the a set P_{best} storing the individuals with best fitness values is initialized. After that, the algorithm starts the evolution, until the termination criterion is met. In each evolutionary generation, the refined crossover and mutation of DE are applied first, after that, the designed second crossover is utilized to generate two offspring, and the best one between the two offspring and their parents is chosen. next, the fitness values of the individuals newly generated are evaluated, and the individuals having best fitness in the population are placed into P_{best} . In the following, the core components in DECNN are discussed.

Algorithm 1: Framework of DECNN

```

1  $P \leftarrow$  Initializing the population detailed in Sect. 11.2.3;
2  $P_{best} \leftarrow empty;$ 
3 while the termination criterion has not been met do
4   Applying the refined crossover and mutation of DE detailed in Sect. 11.2.5;
5   Applying the designed second crossover to generate two offspring, and the best one
      between the two offspring and their parents is selected as detailed in Sect. 11.2.6;
6   Evaluating each individual to get the fitness value;
7    $P_{best} \leftarrow$  Retrieving the individual with best fitness value in the population;
8 end
```

11.2.2 Adjusted IP-Based Encoding Strategy

The layer of DNNs is represented by a IP address in the designed IP-based encoding strategy, and then the IP address is pushed into a series of interfaces. Each of them contains an IP address in addition to the subnet that corresponds to it, in the same way that the layers in DNNs are ordered. CNNs, for example, are made up of three types of layers: the convolutional layer, the pooling layer, as well as the fully connected layer. The initial stage in the encoding process is to determine the range that can be used to indicate each attribute of each CNN layer type. Although the attributes of CNN layers have no specified boundaries, for the purpose of applying optimization algorithms to the task, it is necessary to provide a range with sufficient capacity so the optimal accuracy on the classification tasks can be achieved. The restrictions for each attribute in this chapter are designed with the goal of achieving a low mistake rate. To be more specific, there are three attributes for the convolutional layer: the number of feature maps that is ranging from 1 to 128, the size of stride that is ranging from 1 to 4, and the size of filter that is ranging from 1 to 8. Because these three

Table 11.1 The three attributes of CNN layers with their corresponded range

Layer type	Range	# of bits	Parameter
Convolutional layer	[1,8]	3	Filter size
	[1,128]	7	# of feature maps
	[1,4]	2	Stride size
		12	Total
Pooling layer	[1,4]	2]	Kernel size
	[1,4]	2	Stride size
	[1,2]	1	Type: 1(MAX), 2(AVERAGE)
		5	Total
Fully connected layer	[1,2048]	11	# of Neurons
		11	Total

attributes must be concatenated into a single value, a bits binary string can hold all three convolutional layer attributes: 7 bits to hold the number of feature maps, 2 bits to hold the stride size, and 3 bits to hold the filter size. The fully connected layer and the pooling layer can be hold in binary strings with 11 bits and 5 bits, respectively, in a similar way. Table 11.1 shows the range of each feature in detail.

After determining the total number of bits included in a binary string, a particular CNN layer is then easily capable of being converted into a binary string. If a convolutional layer with the feature maps number of 32, the stride size of 2, and the filter size of 2, it may calculate the binary strings [000 1111], [01], and [001] through transforming the decimal numbers. This is because the binary string begins with 0, and the decimal attribute value of CNN layers begins with 1, 1 is deducted from the decimal number before conversion. By combining the binary strings of each attributes, the final binary string [001 000 1110 01] that represents the supplied convolutional layer is formed. Figure 11.1 shows the details of the example.

In the same way that a subnet must be defined before assigning an IP address to the interface, such as a desktop or laptop, a subnet need to be designed by the IP-based encoding strategy for each CNN layer type. Because the size of the search space is determined by the amount of bits used by each layer type, and the pooling layer uses far less bits compared with the other two, the probabilities of choosing a pooling layer are significantly lower than the other two. To equalize the probabilities of choosing each layer type, the binary string corresponding to the pooling layer occupies 11 bits after having a place-holder occupies 6 bits appended to it, bringing the probabilities of choosing a pooling layer to the identical level as choosing a fully connected layer. Because there are three layer types, each with a maximum bit count of 12, hence a binary string of 2 bytes can carry the encoded CNN layers. The IP address of convolutional layer will be represented by the 2-byte binary starting with [0000 0000 0000 0000], and the binary representation of the finishing IP address is going to be [0000 1111 1111 1111]. By adding 1 to the last IP address of the

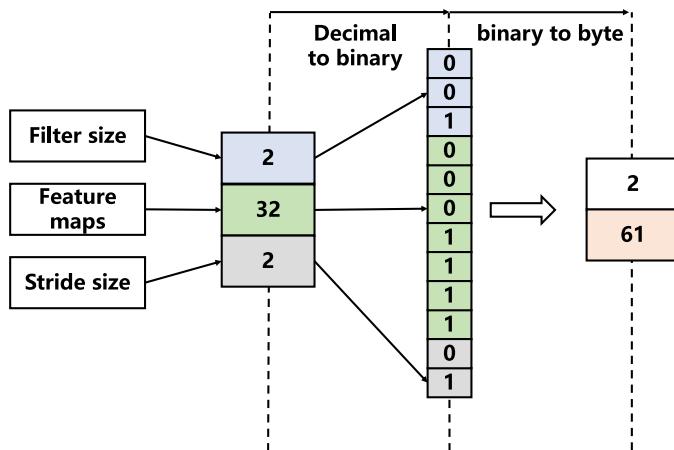


Fig. 11.1 An illustration of how a convolutional layer is encoded by a byte array

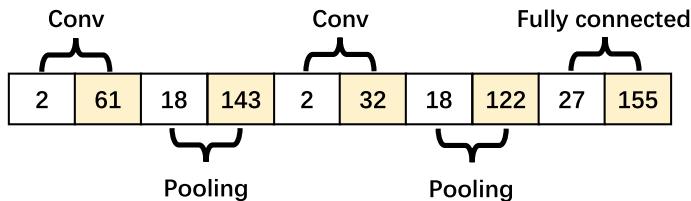


Fig. 11.2 An example of how a vector is encoded from a CNN architecture

convolutional layer, the starting IP of the 11 bits fully connected layer is [0001 0000 0000 0000], and the finishing IP address is [0001 0111 1111 1111]; likewise, the IP address of the pooling layer starts from [0001 1000 0000 0000] and ends with [0001 1111 1111 1111]. In conclusion, the 2-byte style IP ranges for every subset can be summarized as follows: the IP address of the convolutional layer ranges from 0.0 to 15.255, the IP address of the fully connected layer ranges from 16.0 to 23.255, and the IP address of the pooling layer ranges from 24.0 to 31.255, which are calculated by translating the binary strings mentioned previously to binary strings of 2 bytes. Now a CNN layer is ready to be encoded into an IP address, the convolutional layer depicted in Fig. 11.1 is used as an illustration here. The IP address is represented by the binary string [0000 0010 0011 1001], which may be transformed to a 2-byte form IP address [2.61] by adding the starting IP of the convolutional layer subnet and the binary string of the convolutional layer together. An example vector can be seen in Fig. 11.2, which is encoded from a CNN architecture consist of two convolutional layers, two pooling layers, and one fully connected layer.

11.2.3 Population Initialization

Since lengths of individuals must be diverse, the population initialization begin by generating individuals with random length. The length is picked at random from a Gaussian distribution whose standard deviation ρ equals 1 and center μ is set to a predetermined length according to how complex the classification task are, as indicated in Eq. (11.1). After determining the length of candidate, the attribute values and layer type for each layer in the candidate can be produced at random. To complete the population initialization, the process is repeated until the population size is reached.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2} \quad (11.1)$$

11.2.4 Fitness Evaluation

Algorithm 2 illustrates the process of evaluating fitness. First, the fitness evaluation process receives four arguments, the candidate solution for representing an encoded architecture of CNN, the training set for training the CNN architecture, the number of training epoch to train the CNN architecture which is decoded from the candidate solution, and the dataset for fitness evaluation, which is employed to get the fitness value by testing accuracy of the trained model on the dataset. Second, the process of fitness evaluation is straightforward, it employs the back propagation as the training algorithm and a predetermined training epoch number to train the CNN architectures on the training set, and then tests the trained architectures on the testing set to get the accuracy as their fitness value. Only partial dataset is used to train the candidate CNN with a restricted epoch number to reduce computational cost, which are regulated by the arguments of fitness function— k , D_train and $D_fitness$.

Algorithm 2: Fitness Evaluation

Input: The training set D_train , the training epoch number k , the candidate solution c , the dataset of fitness evaluation $D_fitness$.

Output: The evaluated fitness value $fitness$.

- 1 Training the weights of the CNN which is represented by the candidate c for k epochs on the training set D_train ;
 - 2 $acc \leftarrow$ Evaluating the trained CNN on the dataset $D_fitness$ of fitness evaluation;
 - 3 $fitness \leftarrow acc$;
 - 4 **Return** $fitness$.
-

11.2.5 DECNN DE Mutation and Crossover

The mutation and crossover operations of the DECNN are analogous to the crossover and mutation operations of standard DE, but an extra step is added for trimming the vectors with longer length before carrying out any operation. This is due to the fact that the DECNN candidates are of varying lengths and the traditional operations of DE in Eqs. (1.4) and (1.3) are only applicable to vectors of a fixed length. To be more specific, the three random vectors that are used for the mutation have their lengths trimmed down to the shortest length of the three. Moreover, if the length of the trial vector that is produced by the mutation has longer length compared to the length of its parent, it will be shortened until it is equal to the length of its parent.

11.2.6 DECNN Second Crossover

By slicing the vector at the cutting points, each individual of the two parents is separated into two parts, and then one of those parts is exchanged with the other, similar to the crossover of GAs. To regulate population variation, the cutting point is chosen at random using a Gaussian distribution in which a hyperparameter ρ serves as the standard deviation and the middle point serves as the center. Figure 11.3 depicts how the second crossover works.

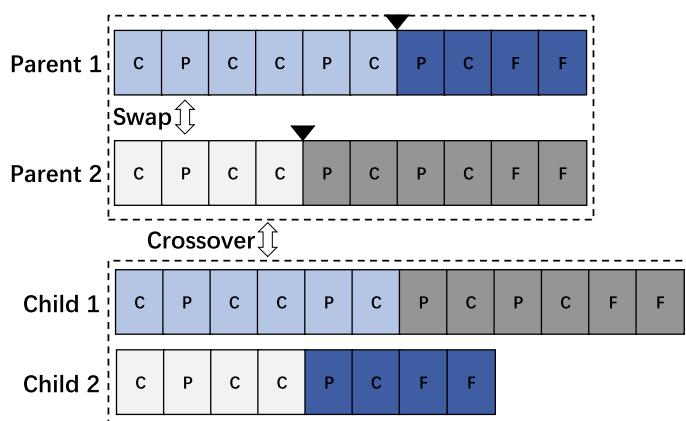


Fig. 11.3 An example of the second crossover in DECNN

11.3 Experimental Design

11.3.1 State-of-the-Art Competitors

On the benchmark datasets that are mentioned before in the literature [1], six state-of-the-art algorithms have been reported to have obtained promising performances. As a result, they are chosen as the peer competitors of DECNN. They are RandNet-2 [1], LDANet-2 [1], SAA-3 [2], ScatNet-2 [3], CAE-2 [4], PCANet-2 (Softmax) [1], TIRBM [5], SVM+Poly [2], SVM+RBF [2], DBN-3 [2], PGBM+DN1 [6], and NNet [2].

11.3.2 Parameter Settings

All of the parameters are set up in accordance with community rules of DE [7], with a small population factored in to reduce time for computation and search space complexity. The population size is set to 30 and the number of generations is set to 20 for the evolutionary process. Only 10% of the training dataset is used as evaluation dataset in the fitness evaluation, and the training epochs is set to 5. The differential rate and crossover rate are set to 0.6 and 0.45, respectively, in the DE parameters. DECNN performs 30 independent runs on each of the benchmark datasets with the hyperparameters ρ for second crossover and μ for population initialization set to 2 and 10 respectively.

11.4 Experimental Results and Analysis

Due to the stochastic nature of DE, a test of statistical significance is necessary in order to produce a compelling result for the comparison. When DECNN is compared with the state-of-the-art methods, One sample T-Test is employed to test if DECNN achieves better results; when comparing error rates between the peer EC competitor dubbed IPPSO [8] and DECNN, and two sample T-Test is employed for the purpose of determining if the difference is statistically significant. Table 11.2 displays the classification results of DECNN against state-of-the-art algorithms, whereas Table 11.3 compares IPPSO with DECNN.

11.4.1 DECNN Versus State-of-the-Art Methods

Table 11.2 shows the experimental results and a comparison of DECNN with state-of-the-art methods. The terms (+) and (-) are used to represent whether the result

Table 11.2 The classification errors of DECNN in comparison to the chosen peer competitors

classifier	MB	CONVEX	MDRBI	MBI	MRD	MRB
TIRBM	–	–	–	35.5 (–)	4.2 (–)	–
CAE-2	2.48 (+)	–	45.23 (+)	15.5 (+)	9.66 (+)	10.9 (+)
ScatNet-2	1.27 (–)	6.5 (–)	50.48 (+)	18.4 (+)	7.48 (+)	12.3 (+)
PGBM+DN-1	–	–	36.76 (=)	12.15 (+)	–	6.08 (=)
PCANet-2 (Softmax)	1.4 (–)	4.19 (–)	35.86 (–)	11.55 (+)	8.52 (+)	6.85 (+)
RandNet-2	1.25 (–)	5.45 (–)	43.69 (+)	11.65 (+)	8.47 (+)	13.47 (+)
SVM+RBF	30.03 (+)	19.13 (+)	55.18 (+)	22.61 (+)	11.11 (+)	14.58 (+)
LDANet-2	1.05 (–)	7.22 (–)	38.54 (+)	12.42 (+)	7.52 (+)	6.81 (+)
NNet	4.69 (+)	32.25 (+)	62.16 (+)	27.41 (+)	18.11 (+)	20.04 (+)
SVM+Poly	3.69 (+)	19.82 (+)	54.41 (+)	24.01 (+)	15.42 (+)	16.62 (+)
DBN-3	3.11 (+)	18.63 (+)	47.39 (+)	16.31 (+)	10.3 (+)	6.73 (+)
SAA-3	3.46 (+)	18.41 (+)	51.93 (+)	23 (+)	10.3 (+)	11.28 (+)
DECNN (best)	1.032	7.992	32.852	5.666	4.066	3.458
DECNN (mean)	1.457	11.192	37.551	8.685	5.533	5.558
DECNN (standard deviation)	0.113	1.943	2.447	0.447	1.405	1.711

Table 11.3 Classification rates of IPPSO and DECNN

	MB	CONVEX	MDRBI	MBI	MRD	MRB
IPPSO (standard deviation)	0.170	2.135	5.380	1.835	0.712	1.543
IPPSO (mean)	1.558	12.645	38.791	9.857	6.072	6.255
DECNN (standard deviation)	0.113	1.943	2.447	1.405	0.447	1.711
DECNN (mean)	1.457	11.192	37.551	8.685	5.533	5.558
P-value	0.01	0.01	0.2554	0.01	0.001	0.1027

achieved by DECNN is superior to or inferior to the best result achieved by the comparable peer rival, in order to clearly present the comparison results. The term (=) represent the mean error rate achieved by DECNN is slightly lower or higher compared to the competitor, but the significance difference is not supported from a statistical standpoint. The term (–) indicates that the results of provider are not available or cannot be counted.

Table 11.2 shows that the results achieved by DECNN in terms of error rates are encouraging. To be more specific, on both the MB and CONVEX benchmark datasets, DECNN ranks fifth. On the MBI benchmark, DECNN outperforms all state-of-the-art methods. On the MDRBI dataset, DECNN achieves the fourth best mean error rate. However the P-value from one sample T-Test between the third best and DECNN is 0.0871, indicating that the difference is not statistically significant, hence DECNN is tied for the third with PGBM+DN-1. On the MRB benchmark, DECNN achieves mean error rate that is lower compared to all other methods. However, there is not a significant difference between the second best algorithm and DECNN given the calculated P-value of 0.1053, hence DECNN is tied for first with PGBM+DN-1. On the MRD benchmark, DECNN outruns all other methods except TIRBM. Furthermore, when comparing the best results of DECNN with the 12 state-of-the-art algorithms on five out of the six datasets, DECNN obtains the lowest error rates: 32.852% on MDRBI, 4.066% on MRD, 3.458% on MRB, 1.032% on MB, and 5.666% on MBI. This demonstrates that DECNN is able to enhance the results obtained by the state of the arts.

11.4.2 DECNN Versus IPPSO

Table 11.3 shows that when the results of DECNN is compared with IPPSO, the mean error rates achieved by DECNN are lower on all six benchmark datasets, as well as the standard deviations of DECNN are lower on five of the six datasets. This indicates that DECNN outperforms IPPSO in terms of the overall performance. Since a kind of local search is performed in terms of the parameters of CNN architectures and their depth between the two parents and their offspring, the second crossover operator increases DECNN performance.

11.4.3 Evolved CNN Architecture

After the evolved architectures of CNN being evaluated, it is discovered that DECNN is capable of evolving the length of architecture. Individual lengths are approximately 10 when the evolutionary process begins, but the evolved CNN architectures lengths drop to between 3 and 5 according to how complex the datasets are, demonstrating that DECNN can effectively evolve CNN architectures of any length.

11.5 Chapter Summary

In this chapter, we introduced the DECNN method, which is a DE-based NAS algorithm. In particular, a new genetic operation including the crossover operator and the mutation operator is designed in DECNN. In addition, an encoding scheme, as well as a second crossover operator, are also developed in DECNN. These designs can collectively help DECNN to perform well on image classification tasks. DECNN is firstly compared with state-of-the-art peer competitors and then compared with IPPSO which was introduced in the previous chapter. Both quantitative comparisons demonstrated the effectiveness of DECNN. In addition, the architecture resulting from DECNN in the experiments has also been described.

All the algorithms introduced in this part concern the algorithmic designs, and the experiments are all about general image classification tasks. In the next chapter, we will introduce an architecture design algorithm focusing on the applications to the images of the hyper spectrum.

References

1. Chan, T.-H., Jia, K., Gao, S., Jiwen, L., Zeng, Z., & Ma, Y. (2015). Pcanet: A simple deep learning baseline for image classification? *IEEE Transactions on Image Processing*, 24(12), 5017–5032.
2. Larochelle, H., Erhan, D., Courville, A., Bergstra, J., & Bengio, Y. (2007). An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th International Conference on Machine Learning* (pp. 473–480). ACM. <https://doi.org/10.1145/1273496.1273556>.
3. Bruna, J., & Mallat, S. (2013). Invariant scattering convolution networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1872–1886. <https://doi.org/10.1109/tpami.2012.230>.
4. Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning* (pp. 833–840).
5. Sohn, K., Lee, H. (2012). *Learning invariant representations with local transformations*. <https://arxiv.org/abs/1206.6418>.
6. Sohn K., Zhou G., Lee C., & Lee, H. (2013). *Learning and selecting features jointly with point-wise gated boltzmann machines*. <https://dl.acm.org/citation.cfm?id=3042918>.
7. Gamperle, R., Muller, S. D., & Koumoutsakos, A. (2002). A parameter study for differential evolution. *NNA-FSFS-EC 2002*, 10, 293–298.
8. Wang, B., Sun, Y., Xue, B., & Zhang, M. (2018a). Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)* (pp. 1–8). IEEE.

Chapter 12

Architecture Design for Analyzing Hyperspectral Images



12.1 Introduction

Denoising images is a key part of the process of images. The hyperspectral image (HSI) has three dimensions in addition to the natural 2D image to display spectral and spatial information. In forestry [1, 2], agriculture, and urban planning, HSIs are commonly used. However, there are impacts with the multi-detector utilized to create the HSIs because of the harsh space environment, resulting in the noise of HSIs. The accuracy of consequent work, such as classification tasks [3], will be affected by the corrupted hyperspectral data with noise. As a result, in recent years have been an increase in interest in HSI denoising [4, 5]. Numerous methods, for example the Tenser-SVD [6] as well as the K-singular value decomposition [7], have been proposed. In general, the HSI denoising techniques may be categorized into three major groups.

- (1) **Filter-Based Methods:** The filter-based approaches are predicated on the employment of filtering processes using various filters, such as the wavelet transform and the Fourier transform. A gray-scale image may be regarded as one of the hundreds of channels of an HSI in particular. As a result, gray-level image denoising approaches that have been used for a long time, such as block-matching 3-D filtering (BM3D) [8], may be applied directly to each channel. Because of the fact that these filtering approaches need the parameters to be adjusted manually, they are still susceptible to being affected by the transform function. Furthermore, it is also possible that the poor performance of the spectral bands is due to the ignoring of the correlations that exist between them.
- (2) **Optimization-Based Methods:** Utilizing acceptable assumptions or priors, these methods retain spatial and spectral characteristics, such as low-rank (LR), Non-Local, total variation, and sparse representation models. Owing to the strong spectral correlations and high-dimensional feature set in HSIs, the LR regularization has been frequently utilized in HSI denoising due to its capacity to effectively disclose the low-dimensional architecture from the high-dimensional data. Due

to the high similarity between each band and the significant spectral correlation between neighboring bands, LR matrix recovery (LRMR) is proposed by Zhang et al. [9] for HSIs restoration by translating a 3-D cube to a 2-D matrix. Tensor-based approaches have recently been developed [10, 11] to better spectral information and combine spatial, and they may accomplish remarkable results when operating on robust computing platforms. As a result, despite their success in HSI denoising, these approaches fail miserably when applied to mixed noise.

- (3) **Learning-Based Methods:** A number of deep learning approaches for denoising HSIs [12] have been presented and proven useful in recent years. Natural image denoising [13–15] is a common application of CNNs, one of the most popular deep learning-based approaches. Nevertheless, when the problem changes, the architecture of CNN must be modified. In fact, the appropriate network design for a certain task is unknown, making it challenging to build an ideal CNN, which means the parameters of each layer, the number of various kinds of layers, and the depth of the CNN must all be determined. Meanwhile, the weights of the network, which are critical to its performance [16], must be retrained using a gradient-based method in order to achieve the promised performance, which depends heavily on the initialization process [17].

In order to further investigate the benefits of CNNs in image denoising and to eliminate the requirement for professional knowledge during architectural design and weight initialization, this chapter proposes a unique algorithm (referred to as HID-CNN) according to a GA [18, 19] to create an appropriate architecture and weight initialization for CNNs in order to effectively and efficiently solve the denoising problem of HSI. In conclusion, HID-CNN has made the following contributions::

1. Individuals for automated CNN architecture design are encoded using a new gene encoding strategy of GA. The encoded individual holds the info of the relevant CNN architecture. For HSI denoising, evolved CNNs may be created by a large number of such individuals.
2. Exploration and exploitation searches benefit from effective genetic operators. The fixed-length chromosome in GA may easily carry out genetic operators. The only way to successfully explore network architectures is to use a variable-length encoding strategy, yet there is no commonly acknowledged method for playing the role of genetic operators. On the encoding strategy, the operators may execute effectively.
3. An enhanced slack binary tournament is offered as a method for selecting potential individuals for offspring generation. In most cases, vast computational resources are necessary to train CNNs. In the environmental selection process, the complexity is considered to be a significant evaluation criterion. The enhanced technique may aid in our quest for a high-performance, low-complexity CNN.

The rest of this chapter is laid out as follows: The algorithm is detailed in Sects. 12.2. The experimental design and results are presented in Sect. 12.3 and 12.4, respectively.

12.2 Algorithm Details

12.2.1 Algorithm Overview

The HID-CNN framework is detailed in Algorithm 1, with contributions underlined in italics and bold. To begin, The population is initialized at random, as well as every individual is created at random using a gene encoding strategy (line 1). Following that, parent selection is carried out on the initial population (line 2). The evolution then takes effect till the termination requirement is met (lines 4–9). At last, decoding the optimal individual resulted in anticipated CNN, which is now prepared for its last training (line 10).

Throughout evolution, the fitness of all of the initialized individuals is evaluated. To accelerate evolution, the individuals are trained for a single epoch on the training dataset, while evaluating their fitness on the evaluation dataset. The improved slack binary operation then selects two parent individuals from the population, and the offspring is produced using genetic operation (line 6). This procedure will be detailly demonstrated in Sect. 12.2.3. After offspring generation and fitness evaluation (line 7), the process of environmental selection begins to select the next generation from current existing individuals and newly generated offspring (line 8). The subsequent generation then continues the following round of evolution.

Algorithm 1: HID-CNN Framework

```

1  $P_0 \leftarrow$  Utilized the gene encoding strategy, initiate the population at random;
2 Evaluate fitness of  $P_0$ ;
3  $k \leftarrow 0$ ;
4 while the halt condition is not met do
5    $k \leftarrow k + 1$ ;
6    $O_k \leftarrow$  Select a parent individual from  $P_{k-1}$  using the improved slack binary operation
     to create the offspring using the genetic operation;
7   Evaluate fitness of  $O_k$ ;
8    $P_k \leftarrow$  Environment selection from  $P_{k-1} \cup O_k$ ;
9 end
10 Return  $P_k$ .

```

12.2.2 Gene Encoding Strategy

In general, it is difficult to find the best CNN architecture without prior knowledge. The performance of CNNs [20–23] is significantly influenced by their architecture, particularly their depth. The depth of traditional CNN designs is determined by the expertise in the field, which is not always allotted appropriately to each work

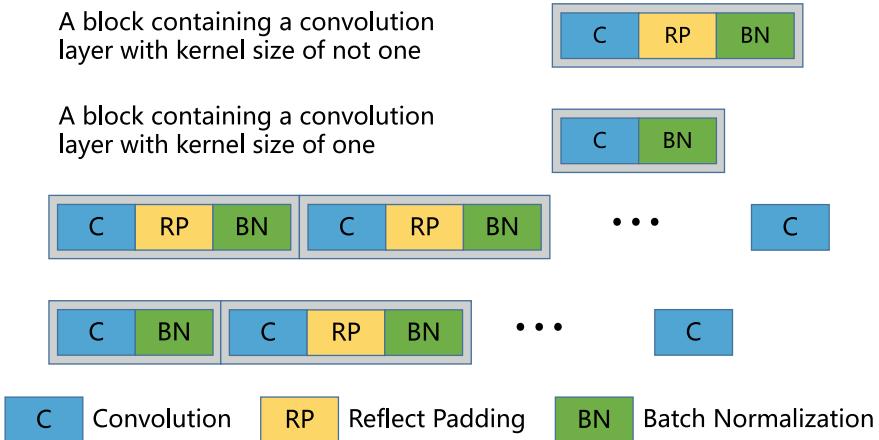


Fig. 12.1 An example of shows two chromosomes with varied lengths and two kinds of blocks from HID-CNN

at hand. The variable-length gene encoding strategy addresses this problem in the algorithm since it can automatically determine the ideal depth without requiring expert knowledge. Multiple consecutive blocks are used to build the CNN in the encoding strategy, each of which has a ReLU activation function, a BN layer, and a convolutional layer. Furthermore, after the convolutional layer, an reflect padding layer is added with a non-zero kernel size to prevent the image size from decreasing. The final block is set as a convolutional layer with fixed kernel size and feature map size since the output image is close to the real one.

As previously stated, HID-CNN architectures contain three distinct layers: the reflect padding layer, the BN layer, and the convolutional layer. The information from the convolutional layer is utilized to encode the architecture onto a single chromosome for the evolution, as the reflect padding layer can be regarded as an appendage of the convolutional layer and the BN layer works well with the default settings. Figure 12.1 shows an example of two chromosomes with varied lengths and two kinds of blocks from HID-CNN. On average, a CNN may contain hundreds of thousands of weights, which cannot all be initialized explicitly. Mean and standard deviation are employed to effectively initialize the weights of CNN via the Gaussian initialization mechanism in the gene encoding strategy. In addition, the chromosome encodes the significant parameters of the convolutional layer, such as the amount of feature maps, the filter height, and the filter width.

Algorithm 2 details the initial process of the population via genetic encoding strategy, where $|\cdot|$ is a cardinality operator. To begin, a population with an empty size N is created (line 1). In addition, individuals are generated at random to fill the population (lines 2–19) till the population its predetermined size N is reached. At last, the P_0 population is returned as initialized (line 20). Based on whether or not there are fixed parameters, the CNN architecture encoded by the individuals consists

Algorithm 2: Population Initialization

Input: the minimal number of blocks, N_{min} ; the maximal number of blocks, N_{max} ; the population size N

Output: Initialized population, P_0

```

1  $P_0 \leftarrow \emptyset$ 
2 while  $|P_0| \leq N$  do
3    $start \leftarrow \emptyset$ ;
4    $k \leftarrow$  create evenly a random number between  $[N_{min}, N_{max}]$ ;
5   while  $|start| < k$  do
6      $c \leftarrow$  create a convolutional layer using the settings of random;
7      $start \leftarrow start \cup c$ ;
8      $s \leftarrow$  the kernel size of the convolutional layer;
9     if  $s \neq 1$  then
10        $c \leftarrow$  create a reflect padding layer which has  $\frac{s-1}{2}$  size;
11        $start \leftarrow start \cup c$ ;
12     end
13      $c \leftarrow$  create a batch-normalization layer using the settings of random;
14      $start \leftarrow start \cup c$ ;
15   end
16    $c \leftarrow$  create a convolutional layer using predefined kernel size and number of feature maps;
17    $start \leftarrow start \cup c$ ;
18    $P_0 \leftarrow P_0 \cup start$ ;
19 end
20 return  $P_0$ 
```

of two parts, the start and the final convolutional layer, and the initialization of the individuals is also divided into two parts. The start of the first part is generated using random settings (lines 3–15), while the second part adds a convolutional layer with partial random initialization (lines 16 and 17).

It is worth to note that line 4 determines the upper and lower boundaries of the depth since a shallow CNN might perform poorly on HSI denoising and a deep CNN will occupy a significant amount of GPU resources. Furthermore, while the CNN output size is set, the filter size and feature map size of the final convolutional layer are also fixed, whereas the mean and standard deviation of filter elements are varied (line 16). To find an appropriate reflect padding layer, there needs to be an odd number of the kernel size of the convolutional layer.

12.2.3 Offspring Generation

The fitness evaluation criteria of this work is the MSE, which measures the pixel value difference between the original image and the denoised image. In particular, Eq. (12.1) is utilized to determine the MSE

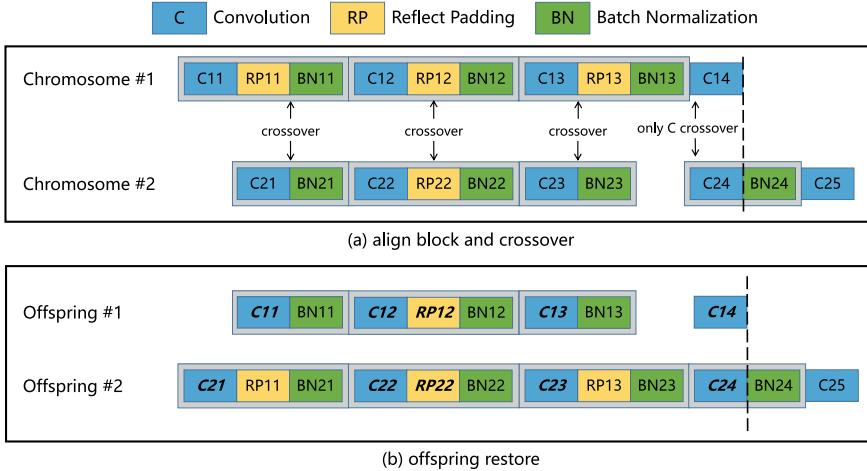


Fig. 12.2 A crossover example

$$\text{MSE} = \frac{\|D - O\|^2}{M \times N} \quad (12.1)$$

where $\|\cdot\|$ represents the L_2 norm of a matrix, denoised and original image pixel values are represented by D and O , accordingly, the length and width of the image that is represented by M and N . The lower the MSE, the greater the denoising impact, as well as the higher the fitness level of the individual. In fact, multiple CNN architectures with almost identical MSE will exist, but the one with the least number of parameters should be the most promising because a CNN with fewer parameters is less complex and may have greater generalization ability. Therefore, the amount of CNN parameters, which utilized as the fitness of the individuals, is also indicates the complexity of the CNN.

As with conventional GAs and as described in Sect. 12.2.1, the offspring will be generated using genetic operations following the initialization and fitness evaluation of the original population. Traditionally, the crossover operator and the mutation operator are two common genetic operators that are used to create offspring. The following are the stages involved in producing offspring:

1. Algorithm 3 will be used to select two parents (Sect. 12.2.4 will discuss this algorithm in detailed);
2. Generate the offspring by performing mutation and crossover on the selected parent individuals.
3. Save the offspring and repeat Step 1 till the amount of offspring reaches the predetermined size.

The mutation action is split into two steps by the algorithm. The first step allows the polynomial mutation operator [24] to mutate the encoded variable information in a given range. The depth of the CNN is randomly increased and decreased in the

second stage. This step may be performed on every position of the CNN, with each position being chosen at random. Then, a single peculiar mutation operation is chosen at random from: 1) adding a randomly initialized block, 2) deleting a block with the exception of the last layer, and 3) maintaining the depth. After that, the operation is executed in the chosen position. Every type of operation has a 1/3 possibility of being chosen at random. In the crossover operation, the SBX [25] is employed because of its potential performance on real numbers, such as, the encoded variable information in the encoding strategy.

Figure 12.2 shows the process of crossover, with RP representing the reflect padding layer, BN representing the BN layer, as well as C representing the convolutional layer. The first chromosome in this example comprises three blocks, whereas the second contains four. To begin, the blocks of every chromosome line up according to the sequence as illustrated in Fig. 12.2.a. In addition, the matched blocks then perform the crossover operation. When performing the crossover operation, there are two distinct circumstances: the first is for the identical architectures when both blocks are composed of C+BN or C+RP+BN. The second is for distinct architectures with one block is C+BN and another is C+RP+BN. In the first instance, the crossover operation is performed by simply exchanging the encoded variable information. However, in the second instance, in addition to the encoded variable information, the reflect padding is also moved to the side with no reflect padding. Notably, whenever the lengths of the chromosomes differ, the crossover operation is carried out on the final layer of the shorter chromosome as well as the convolutional layer corresponding to the longer chromosome. So as to obtain a valid image size, the amount of feature maps in the last layer will not be included in this operation. At last, the offspring is generated as illustrated in Fig. 12.2.b, where italics and bold show that this layer has been changed.

12.2.4 Environmental Selection

Maintaining a population with potential convergence and variety, the “slack binary tournament selection” elite mechanism is utilized in the developed environment selection. The design details are shown in Algorithm 3. To begin, two individuals from the population are chosen at random (line 1). In addition, the individual has the lower MSE is assigned to l_1 , while another individual is assigned to l_2 (lines 2 and 3). Furthermore, their MSE values and complexity are associated with their related parameters (lines 4 and 5). Finally, in order to choose the better individual, the proportion (line 6) and the difference (line 9) are compared to the specified threshold. Specifically, if the performance of a CNN is significantly better than another (lines 6 and 7) the better one is selected; if their performances are similar, the one which has lower complexity is chosen (lines 9 and 10). When performance and complexity are about equivalent, performance is the better option (lines 11 and 12).

Algorithm 3: Slack Binary Tournament Selection

Input: β ; The population, α ; complexity threshold, MSE threshold
Output: The chosen individual

```

1 Choose two individuals at random from the population
2  $l_1 \leftarrow$  the individual has lower MSE;
3  $l_2 \leftarrow$  another individual;
4  $e_1, e_2 \leftarrow$  the MSE of  $l_1, l_2$ ;
5  $c_1, c_2 \leftarrow$  the complexity of  $l_1, l_2$ ;
6 if  $\frac{e_2 - e_1}{e_1} > \alpha$  then
7   | return  $l_1$ 
8 else
9   | if  $c_1 - c_2 > \beta$  then
10    |   | return  $l_2$ 
11   | else
12    |   | return  $l_1$ 
13   | end
14 end

```

Note that a proportion rather than a simple difference is selected as a threshold since the MSE value of all individuals is quite large during the early stages of evolution, but in the latter stages of evolution, the majority of individuals have excellent performance of denoising, such as, the MSE value is usually small. As a result, the proportion is optimal to keep the selection operation consistently. When the process of evolution is complete, the optimal individual from the last generation is chosen to execute the last training of CNN.

12.3 Experimental Design

On the selected benchmark dataset, an experiment is built to compare HID-CNN to that of state-of-the-art peer competitors in order to quantify its performance. Then, the benchmark dataset, as well as how to construct the evaluation, test, and training datasets, are discussed first. After that, the peer competitors are listed. In the following, the parameter settings are discussed, which include the parameters in training for the algorithm.

12.3.1 Benchmark Dataset

The benchmark dataset is the Indian Pines dataset [26], which is commonly utilized in HSI denoising [12, 27]. Over the Indian Pines test site in northern Indiana, the AVIRIS sensor collected the data. The Indian Pines dataset, in particular, has two HSIs with varying sizes. The first has a size of $614 \times 2678 \times 220$, whereas the

second has a size of $1848 \times 614 \times 220$. To make a fair comparison, Gaussian noise with an intensity level of $\sigma = 0.003$ is added to the original images to create the data which is used for comparison of denoising algorithms.

According to the conventions, for experiments, the dataset has been separated into three parts, namely the test dataset, the evaluation dataset, and the training dataset, which are chosen at random and contain 4,838, 4,000, and 17,535 images, accounting for 18.3%, 15.2%, and 66.5%, respectively. To get enough images to train the CNN, the original clean images and produced noise images are trimmed with a size of 30×30 and a sampling stride of 10.

12.3.2 Peer Competitors

For peer competitors, several state-of-the-art HSI denoising techniques are selected. In addition to the concentration of algorithm on CNN approaches, the algorithm of Chang [12] is selected, which lately published potential performance on HSI denoising, as one of the peer competitors in this experiment.

The following is a list of the chosen peer competitors: (1) the LR tensor approximation (LRTA) approach [28]; (2) the BM3D approach [8]; (3) the tensor dictionary learning approach (TDL) [29]; (4) the total-variation-regularized LR matrix factorization approach (LRTV) [30]; (5) the intrinsic tensor sparsity regularization approach (ITSReg) [31]; (6) the block matching 4-D filtering approach (BM4D) [32]; (7) the approach of Chang (Artificial-CNN) [12]; (8) the LRMR approach [9].

Note that the algorithm selects the optimal CNN according to the evaluation dataset after being trained on the training dataset. After the process of evolution is complete, the optimal performance is obtained by training it on both the training and evaluation datasets according to deep learning community conventions. During this process, the test dataset is hidden.

12.3.3 Parameter Settings

The parameters of the evolution in the algorithm, as well as the parameters of the peer competitors, are the parameters set in the experiment.

All evolution parameter settings are established according to GA community conventions [24, 25]. The depth of CNN ranges between four to eight depending on performance and complexity. The filter size of the convolutional layer, which corresponds to the filter width and height, is selected at random from {1, 3}. The feature map sizes of all layers range between 128 to 512, with the exception of the last convolutional layer, which has a fixed size. The mean range between -0.8 to 0.8, and the std range between 0 to 0.5, according to preliminary experiment and experience. Furthermore, the total generation number and population size are likewise set at 10 and 30, accordingly. According to conventions [24, 25], the distribution index of

the polynomial mutation and SBX are likewise set to one. According to the Pareto principle, the rate of elitism in the environment selection is set at 20%. The complexity and MSE are utilized in the fitness function in the evolutionary stage, as mentioned in Sect. 12.2.3.

Additionally, since their default settings provide the optimal performance, all peer method parameters are set to default.

12.3.4 Training Details

The training details for both the evolution process and the final training are covered in this section.

During the evolution stage, every individual must first be trained in order to be evaluated for fitness. For this stage, the Adam method [33] is utilized as the optimizer, with the exponential decay rate set to the default value and the learning rate set to 0.004. To guarantee that every individual receives consistent training, the training dataset will not be shuffled during the evolution stage.

The Adam is used as the CNN optimizer again in the final train, with momentum parameters of 0.9, 0.999, and 10^{-8} , accordingly, with the starting learning rate set at 0.001. Notably, the training and evaluation datasets are both utilized to train the CNN, and they are shuffled at this stage.

Besides its approach for weight initialization, which utilizes the Xavier initializer [17], the training of Artificial-CNN and HID-CNN employs the identical settings. On an Ubuntu server equipped with an NVIDIA 2080 Ti GPU card, PyTorch is used to implement both CNNs.

Paying attention that, the batch sizes for both the training and test stages of HID-CNN and Artificial-CNN are set to 100. Additionally, the training epoch is not restricted but instead stop the training and test by investigating whether or not performance improves.

12.4 Experimental Results and Analysis

The quantitative evaluation indicators and visual comparisons are utilized to analyze the experimental results in order to obtain an integrated comparison for all peer competitors and HID-CNN. The four image quality measurements (IQMs) utilized as quantitative evaluation indicators are the MERGA [34], the mean structural similarity index (MSSIM) [35], the mean peak signal to noise ratio (MPSNR), and the mean feature similarity index (MFSIM) [36]. The perceptual consistency, structural consistency, and MSE value are used by MFSIM, MSSIM, and MPSNR to evaluate how similar the target and reference images are. The bigger they are, the more similar the two images are, implying that the approach is more effective. MERGA, unlike

Table 12.1 Comparison of Average Denoising Performance Eight Competing Methods Regarding Four PQIs of INDIAN PINE Under $\sigma = 0.003$

	Nosiy	LRTA	BM3D	TDL	LRTV	ITSReg	BM4D	Artificial-CNN	LRMR	HID-CNN
MERGA	29.857	5.877	7.567	4.483	5.929	4.906	5.575	4.095	5.333	3.976
MSSIM	0.98997	0.99937	0.99905	0.99967	0.99954	0.99956	0.99951	0.99974	0.99958	0.99977
MPSNR	50.311	64.939	62.881	67.248	64.852	66.979	65.122	69.317	65.115	70.051
MFSIM	0.96443	0.99030	0.97201	0.98883	0.98072	0.98829	0.98107	0.99197	0.98599	0.99213

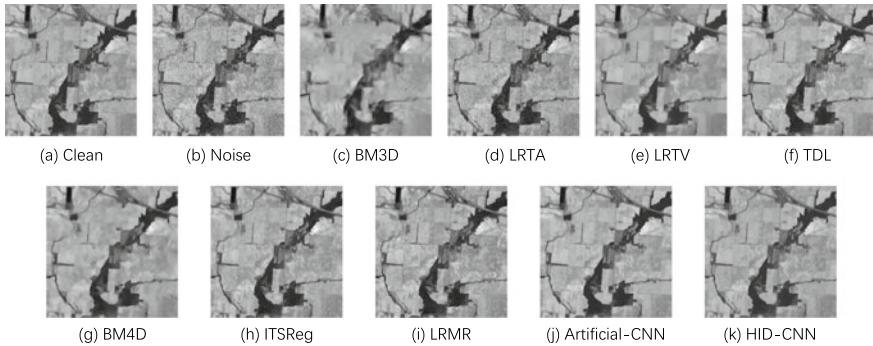


Fig. 12.3 Visual comparison in Band 187. **a** Clean. **b** Noise. **c** BM3D. **d** LRTA. **e** LRTV. **f** TDL. **g** BM4D. **h** ITSReg. **i** LRMR. **j** Artificial-CNN. **k** HID-CNN

the other three indicators, measures fidelity, and the lower the MERGA, the better the approach.

12.4.1 Overall Results

Table 12.1 shows the average results of the four performance evaluation indicators. Band 187 was selected at random for visual comparison in Fig. 12.3.

The optimal performance of every evaluation indicator is highlighted in bold in Table 12.1. Table 12.1 shows that among the compared algorithms, CNN-based approaches are unquestionably the most effective. HID-CNN, in particular, provides the highest MFSIM, MSSIM, and MPSNR values, as well as the lowest MERGA, and becomes the sole approach with MPSNR greater than 70. In Fig. 12.3, the BM3D was unable to handle the noise adequately, and the BM4D produced over-smoothing as a result. In LRTA and LRMR, residual Gaussian noise is readily visible, and the left approaches have high fidelity.

Table 12.2 Comparison With Artificial CNN. The symbols “+”, “-”, and “=” indicate if the IQMs results of HID-CNN are better than, worse than or equal to that of the Artificial-CNN with a substantial degree 1%

	HID-CNN	Artificial-CNN
total parameters	1,838,603	32,254,420
depths	5	16
+/-=		2/0/2
train time	15 GPUhs	54 GPUhs

12.4.2 Comparisons with Artificial-CNN

To tell the effect of the automatic architecture selection mechanism, the CNN-based approaches are compared independently. The amount of total parameters, depth, performance derived from evaluation indicators, and training time are all shown in Table 12.2. To measure training time, the GPUh is used, which is one hour on one GPU service.

Artificial-CNN is far more difficult than HID-CNN, as seen in Table 12.2. On the other hand, Artificial-CNN takes three times longer to train than HID-CNN. Despite the fact that both CNNs have a great denoising effect, the shorter CNN performs better with regard to evaluation indicators. Furthermore, the HID-CNN model is twice as good as Artificial-CNN with regard to IQMs results on the four measurements employed.

12.5 Chapter Summary

In this chapter, we introduced the architecture design algorithm for CNNs concerning the denoising tasks for HSI. Most natural images are two-dimensional, while the HSIs are with three dimensions with extra information for the objects. In practice, the HSIs are generated with multiple detectors in a harsh environment. This will often cause the generated HSIs with noises, which will affect the performance of the downstream tasks. Currently, the well-performing models for image denoising are CNNs. Because the performance of CNNs highly depends on their architecture which is often manually designed with expertise, the introduced algorithm in this chapter can automatically generate the optimal CNN architecture performing well on the HIS for denoising tasks.

This chapter and the previous chapters focused on the automated design of supervised DNNs, mainly the CNNs, for image tasks. In particular, the previous chapters are for classifying images that are popular in the general image processing tasks, such as CIFAR-10 and CIFAR100, while this chapter is for denoising HSIs. In addition to the algorithms for automated DNN architectures in this part and the previous part, we will also introduce some recent advances regarding ENAS, which are shown in Part IV.

References

1. Zhang, L., Zhang, L., Tao, D., & Huang, X. (2011). On combining multiple features for hyperspectral remote sensing image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 50(3), 879–893.
2. Thenkabail, P. S., & Lyon, J. G. (2016). *Hyperspectral remote sensing of vegetation*. CRC Press.
3. Li, J., Zhang, H., Huang, Y., & Zhang, L. (2013). Hyperspectral image classification by non-local joint collaborative representation with a locally adaptive dictionary. *IEEE Transactions on Geoscience and Remote Sensing*, 52(6), 3707–3719.
4. Kong, Z., & Yang, X. (2019). Color image and multispectral image denoising using block diagonal representation. *IEEE Transactions on Image Processing*, 28(9), 4247–4259.
5. He, W., Yao, Q., Li, C., Yokoya, N., & Zhao, Q. (2018). Non-local meets global: An integrated paradigm for hyperspectral denoising. [arXiv:1812.04243](https://arxiv.org/abs/1812.04243), <https://arxiv.org/abs/1812.04243>.
6. Zhang, Z., Ely, G., Aeron, S., Hao, N., & Kilmer, M. (2014). Novel methods for multilinear data completion and de-noising based on tensor-svd. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3842–3849).
7. Elad, M., & Aharon, M. (2006). Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Transactions on Image processing*, 15(12), 3736–3745.
8. Dabov, K., Foi, A., Katkovnik, V., & Egiazarian, K. (2007). Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, 16(8), 2080–2095.
9. Zhang, H., He, W., Zhang, L., Shen, H., & Yuan, Q. (2013). Hyperspectral image restoration using low-rank matrix recovery. *IEEE Transactions on Geoscience and Remote Sensing*, 52(8), 4729–4743.
10. Fan, H., Li, C., Guo, Y., Kuang, G., & Ma, J. (2018). Spatial-spectral total variation regularized low-rank tensor decomposition for hyperspectral image denoising. *IEEE Transactions on Geoscience and Remote Sensing*, 56(10), 6196–6213.
11. Xie, Q., Zhao, Q., Meng, D., & Zongben, X. (2017). Kronecker-basis-representation based tensor sparsity and its applications to tensor recovery. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(8), 1888–1902.
12. Chang, Y., Yan, L., Fang, H., Zhong, S., & Liao, W. (2018). Hsi-denet: Hyperspectral image restoration via convolutional neural network. *IEEE Transactions on Geoscience and Remote Sensing*, 57(2), 667–682.
13. Jain, V., & Seung, S. (2009). Natural image denoising with convolutional networks. In *Advances in neural information processing systems* (pp. 769–776).
14. Lefkimiatis, S. (2017). Non-local color image denoising with convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3587–3596).
15. Xie, J., Xu, L., Chen, E. (2012). Image denoising and inpainting with deep neural networks. In *Advances in neural information processing systems* (pp. 341–349).
16. Sun, Y., Xue, B., Zhang, M., & Yen, G. G. (2020a). Evolving deep convolutional neural networks for image classification. *IEEE Transactions on Evolutionary Computation*, 24(2), 394–407.
17. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).
18. Ashlock, D. (2006). *Evolutionary computation for modeling and optimization*. Springer Science & Business Media.
19. Back, T. (1996). *Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press.
20. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.

21. Bengio, Y., & Delalleau, O. (2011). On the expressive power of deep architectures. In *International Conference on Algorithmic Learning Theory* (pp. 18–36). Springer.
22. Bengio, Y., Courville, A., & Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8), 1798–1828.
23. Delalleau, O., & Bengio, Y. (2011). Shallow versus deep sum-product networks. In *Advances in Neural Information Processing Systems* (pp. 666–674).
24. Deb, K. (2001). *Multi-objective optimization using evolutionary algorithms* (Vol. 16). Wiley.
25. Deb, K., & Agrawal, R. B. (1995). Simulated binary crossover for continuous search space. *Complex Systems*, 9(2), 115–148.
26. Aviris, N. W. (2012). Indiana's indian pines 1992 data set. <https://purr.purdue.edu/publications/1947/1>.
27. Zhang, Q., Yuan, Q., Li, J., Liu, X., Shen, H., & Zhang, L. (2019). Hybrid noise removal in hyperspectral imagery with a spatial-spectral gradient network. *IEEE Transactions on Geoscience and Remote Sensing*.
28. Renard, N., Bourennane, S., & Blanc-Talon, J. (2008). Denoising and dimensionality reduction using multilinear tools for hyperspectral images. *IEEE Geoscience and Remote Sensing Letters*, 5(2), 138–142.
29. Peng, Y., Meng, D., Xu, Z., Gao, C., Yang, Y., & Zhang, B. (2014). Decomposable nonlocal tensor dictionary learning for multispectral image denoising. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2949–2956).
30. He, W., Zhang, H., Zhang, L., & Shen, H. (2015). Total-variation-regularized low-rank matrix factorization for hyperspectral image restoration. *IEEE Transactions on Geoscience and Remote Sensing*, 54(1), 178–188.
31. Xie, Q., Zhao, Q., Meng, D., Xu, Z., Gu, S., Zuo, W., Zhang, L. (2016). Multispectral images denoising by intrinsic tensor sparsity regularization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1692–1700).
32. Maggioni, M., Katkovnik, V., Egiazarian, K., & Foi, A. (2012). Nonlocal transform-domain filter for volumetric data denoising and reconstruction. *IEEE Transactions on Image Processing*, 22(1), 119–133.
33. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
34. Wald, L. (2002). *Data fusion: Definitions and architectures: Fusion of images of different spatial resolutions*. Presses des MINES.
35. Wang, Z., Bovik, A. C., Sheikh, H. R., Simoncelli, E. P., et al. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4), 600–612.
36. Zhang, L., Zhang, L., Mou, X., & Zhang, D. (2011). Fsim: A feature similarity index for image quality assessment. *IEEE Transactions on Image Processing*, 20(8), 2378–2386.
37. Irwin-Harris, W., Sun, Y., Xue, B., & Zhang, M. (2019). A graph-based encoding for evolutionary convolutional neural network architecture design. In *2019 IEEE Congress on Evolutionary Computation (CEC)* (pp. 546–553). IEEE.
38. Sun, Y., Wang, H., Xue, B., Jin, Y., Yen, G. G., & Zhang, M. (2019a). Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. *IEEE Transactions on Evolutionary Computation*, 24(2), 350–364.
39. Fernandes, F. E., Jr & Yen, G. G. (2019). Pruning deep convolutional neural networks architectures with evolution strategy. *Information Sciences*, 552, 29–47.
40. Zhou, Y., Yen, G. G. & Yi, Z. (2019). A knee-guided evolutionary algorithm for compressing deep neural networks. *IEEE Transactions on Cybernetics*, 1–13.
41. Ye, Q., Sun, Y., Zhang, J., & Lv, J. C. (2020). A distributed framework for ea-based nas. *IEEE Transactions on Parallel and Distributed Systems*.
42. Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2017). A downsampled variant of imagenet as an alternative to the cifar datasets. [arXiv:1707.08819](https://arxiv.org/abs/1707.08819).

43. Pham, H., Guan, M., Zoph, B., Le, Q., & Dean, J. (2018). *Efficient Neural Architecture Search Via Parameters Sharing, Volume 80 of Proceedings of Machine Learning Research* (pp. 4095–4104). Stockholmsmssan. PMLR.
44. Cheng, Y., Wang, D., Zhou, P., & Zhang, T. (2018). Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1), 126–136.

Part IV

Recent Advances in Evolutionary Deep Neural Architecture Search

This part will mostly focus on the recent advances in ENAS, including the investigation on novel encoding space based on directed acyclic graph (DAG) [1], the end-to-end performance predictor (E2EPP) [2], deep neural architecture pruning (DeepPruningES) [3], deep neural architecture compression (KGEA) [4], and distribution training framework for ENAS [4]. In addition to the graph-based encoding space work, the other advances, in particular, can be divided into two categories. The first is to figure out how to cut down on the massive amount of computation resources and computation time needed to achieve ENAS. The second is to figure out how to prune the NN without sacrificing its performance too much.

Although past experiments have demonstrated that the ENAS outperforms some other approaches (e.g., RL-based approaches) for searching DNNs with greater precision, the searching cost of ENAS is substantially higher, and the key reasons are shown as follows. First, the accuracy of the DNN is always used to describe the fitness value of an individual, which will be accumulated to alter the search policies, thus a DNN must be evaluated independently on the target dataset. However, the training and validation of a DNA often take a long period, and a single generation typically contains a dozen of individuals. Second, following evaluation, some individuals with exceptional performance are chosen as parents, and by performing mutation and crossover procedures in each new generation, numerous new offspring requiring evaluation are formed. Third, a large population size and evolution durations are always chosen in order to seek for a neural architecture with greater performance as much as feasible, which implies hundreds of individuals must be decoded, trained, and verified on the target dataset completely.

The methods to alleviate the time-consuming challenge of the ENAS approaches discussed in this part fall into two primary categories. In the literature, numerous DNN assessment acceleration options are designed, including low fidelity [5], weight sharing [6], and prediction predictor. The performance predictor, in particular, has the potential to significantly speed up the search process yet without much side effect; E2EPP, a novel method that falls into this category, will be discussed. Second, various well-known distributed training approaches, including data parallelism and model parallelism, have been created to speed up the training of a large-scale DNN. None of the available distributed training approaches, however, is appropriate for ENAS methods. As a result, a novel distributed training strategy will also be introduced.

The second frontier issue, in addition to the time-consuming problem of the existing ENAS, is to reduce the number of parameters. A vast number of parameters are often required to assure sufficient capacity of the network, which is a common design guideline for network architectures. As a result, the achievement of DNNs usually necessitates a lot of computing resources and a lot of computer memory. For example, the VGG-19 [7] model is over 500 MB in size and costs roughly + floating-point operations (FLOPs) to compute for a single input sample. Intelligent mobile devices (such as smartphones, tablets, and wearable devices) are becoming increasingly common, creating a unique demand for real-time applications that incorporate DNNs for intelligent information processing. However, in comparison to the costly inference cost of deep models, those platforms have significantly limited computational power and storage capacity, making deployment difficult. As a result, compressing DNNs is required to make deep models suitable for platforms with limited resources. The core idea is to speed up inference by pruning parameters, with the goal of obtaining a compressed model with the best accuracy and the smallest model size possible. As a result, performance and parameter scaling are two essential enabling elements for such applications. Because reducing parameters can result in a loss of performance, there appears to be a sensitive balance between performance and parameter scale to consider when compressing DNNs. Although manually searching the pruning parameter space to establish a fair balance between them is difficult, it has piqued the interest of both academia and industry [8]. In this part, two automatic CNN pruning and compression techniques will be introduced.

References

- 1 William Irwin-Harris, Yanan Sun, Bing Xue, and Mengjie Zhang. A graphbased encoding for evolutionary convolutional neural network architecture design. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 546–553. IEEE, 2019.
- 2 Yanan Sun, Handing Wang, Bing Xue, Yaochu Jin, Gary G Yen, and Mengjie Zhang. Surrogate-assisted evolutionary deep learning using an end-to-end random forest-based performance predictor. *IEEE Transactions on Evolutionary Computation*, 24(2):350–364, 2019.
- 3 Francisco E Fernandes Jr and Gary G Yen. Pruning deep convolutional neural networks architectures with evolution strategy. *Information Sciences*, 552: 29–47, 2019.
- 4 Yao Zhou, Gary G. Yen, and Zhang Yi. A knee-guided evolutionary algorithm for compressing deep neural networks. *IEEE Transactions on Cybernetics*, pages 1–13, 2019.
- 5 Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv e-prints*, art. arXiv:1707.08819, Jul 2017.

- 6 Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. volume 80 of *Proceedings of Machine Learning Research*, pages 4095-4104, Stockholm, Sweden, 10-15 Jul 2018. PMLR.
- 7 Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*, Lille, France, 2015.
- 8 Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1):126-136, 2018.

Chapter 13

Encoding Space Based on Directed Acyclic Graphs



13.1 Introduction

Although CNN-GA [1] is totally automated, the generated architectures have a restricted connectional structure since it employs an encoding strategy that encodes the building blocks into a linked list that can be extended to any depth during the process of evolution. Each linked list encoding building block is a “skip layer” or a pooling layer, where the skip layer containing a skip connection and two convolutional layers [1]. This design is modeled on ResNet [2]. Good results have been achieved by this search space, but it means that the architectures of CNN with arbitrary graph structure are unable to be generated by CNN-GA [1]. The Block-QNN-S cell produced by Block-QNN [3] and DenseNet [4] are two famous architectural designs that employ a complicated graph structure which is different from the architecture of ResNet.

The purpose of this chapter is to overcome these restrictions by introducing an innovative encoding strategy for representing the architecture of CNN with the unlimited depth and any graph structure. The following are the main contributions of this chapter:

1. A DAG is utilized to represent the architecture of CNN. Compared to other representations that were created for the purpose of evolutionary search, the restrictions placed on the search space is much fewer. By doing so, search algorithms will be able to generate novel architectures, such as new skip connection applications.
2. Based on the designed encoding, a method for randomly producing architectures is presented. As a result, the method is able to be employed as a population initialization component of EAs or for random search on its own. The goal of the initialization method is to place minimal limits on the search space while avoiding the production of invalid or clearly suboptimal architectures.
3. A simple random search is employed to demonstrate the efficiency of this ENAS encoding strategy and the related initialization method, in the search phase, just one-tenth of the final training data is used. It demonstrates the possibility for the

representation to reach promising accuracy despite the presence of the limited training data throughout the search, a straightforward search algorithm, and a huge search space.

13.2 Algorithm Details

The CNN encoding method is discussed in this section, as well as an algorithm that random initializing the architectures of CNN which is specified by the encoding.

13.2.1 Encoding Strategy Overview

A DAG can be used to represent the components of the CNN architecture, as previously discussed. In this section, the designed graph-based encoding strategy is detailed. An ordered pair $G = (V, E)$ of edges E and vertices V can be used to represent a general graph. Each edge in a DAG has a corresponding direction that it points in, and there is a limitation that no way exists from a node v to itself. Notice the further limitation of encoding strategy that there is always *precisely one* node r which has no edges that terminate at this node (but it is still possible that exist edges that begin at it). This node is termed as root node by us, as it reflects the output of CNN.

In this designed encoding strategy, every node in the graph represents either a reference to the input data, an operator, or a building block. Every node has a type assigned to it, which is one of AVERAGE, CONCAT, INPUT, CONV, MAX and SUM. Furthermore, a *feature_maps* attribute is included in the CONV nodes, this attribute indicates the channel number that will be output by the convolutional layer. A convention is employed that the edge from node n_1 to node n_2 indicates n_1 receiving its input data from n_2 in this encoding. This is *opposite* to the data flow direction in the network, i.e., the edges from the output layer of network are directed toward the input data (can be seen in the left side of Fig. 13.1). This convention makes it easier to describe algorithms that working on this structure although it is arbitrary.

A fixed attribute *arity* is included in every node type, that defines the necessary number of edges originating from that node. Notice that the ending edge number of that node is arbitrary. The arity of INPUT nodes is 0 since the INPUT nodes are leaf nodes; Assign arity 2 to the CONCAT nodes and SUM nodes; and arity 1 for all other nodes. Chaining SUM and CONCAT nodes can still represent concatenation and summation of any numbers of arrays.

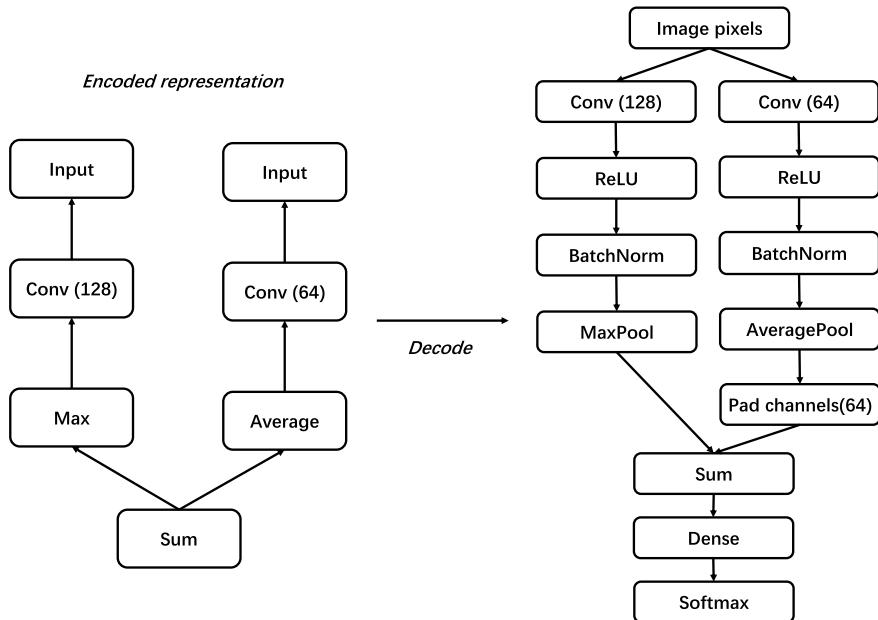


Fig. 13.1 An illustration of an encoded representation along with the decoded architecture of CNN that corresponds to it

13.2.2 Representation and Decoding Details

In this section, the graph structure will be introduced in further detail, and depict the decoding method that is utilized to convert the encoded representation into the corresponding final architecture of CNN (phenotype). Figure 13.1 shows an illustration of the encoded representation, and depicts the decoded architecture of CNN that corresponds to it.

Firstly, the node types which are available are described in detailed and justifying these design choices. In the final CNN, a “convolutional building block” is represented by a CONV node, which is defined as consisting of a convolutional layer whose stride size is 1, a ReLU activation function [5], and a batch normalization [6], in that order. The ConvBlock which is employed in the CGP-CNN method [7] inspired this design, and the usage of BN and ReLU are also are in line with recent hand-crafted CNNs [2, 4]. To ensure that the dimension of image do not vary once convolution has been applied, the 1×1 stride and the padding mode of “same” are used. The amount of feature maps is a CONV node parameter that can be configurated. On the contrary, a filter (kernel) with fixed 3×3 size is used. This decision is made since 3×3 is a common kernel size in CNNs like ResNet [2] and DenseNet [4] which are manually designed; also, it has been suggested that stacks of 3×3 filters can successfully substitute kernels of varying sizes without compromising their capacity

for representation [8]. As a result, adopting a fixed value for the filter size narrows the search space.

AVERAGE and MAX layers are represented by the AVERAGE and MAX node types, respectively. For both the AVERAGE and MAX layers, stride size of 2 and a kernel size of 2 are used to ensure that the spatial dimensions of the input data are always halved. By stacking these pooling layers, the dimension can be greater reduced. The node type of INPUT indicates the input image (or minibatch of images) selected from the training data or test data being applied to the CNN. Although only one INPUT node is necessary conceptually, it may be more convenient for the implementation of algorithm to enable several INPUT nodes, all these INPUT nodes are deemed equivalent.

The concatenation and summation operations are represented by the CONCAT and SUM nodes, respectively. If the two inputs to either CONCAT or SUM nodes are not equivalent in the dimensions during the decoding step, the larger feature map is downsampled by MAX layer, bringing the two feature maps to become equivalent in the dimensions. This follows the procedure employed in [1, 7].

If the channel number between the two inputs differs when a SUM node is decoding, to make the channel number equal, the input with a smaller channel number will be added with the padding channels with all zeros (see Fig. 13.1). This method is derived from a method that is mentioned in the ResNet paper [2], as an alternative, the convolutions of 1×1 are also suggested to alter the dimensionality of channel. Zero-padded channels for this decoding method are chosen since it reduces the number of parameters than 1×1 convolutions.

To build the final architecture of CNN according to the encoded representation, every node in the graph is transformed into the CNN layers it corresponded to, as mentioned above. Finally, at the end of the CNN, a fully-connected (dense) layer with the same number of output nodes as the number of image classes is added. As the final component of the CNN, a Softmax activation function is added.

13.2.3 Initialization Algorithm Overview

In this section, an algorithm for randomly generating an architecture of CNN represented by the encoding strategy is presented. Algorithm 1 gives the framework of this initialization method.

To begin, an integer depth denoted by d is chosen at random (line 1), within this initialization algorithm, this depth is referred as the *target depth*. When generating a population, this enables for CNNs of various depths, perhaps increasing the probability of obtaining an architecture that has the appropriate amount of depth for the investigated dataset. The graph is then built by the algorithm in a breadth-first order using a standard first-in, first-out queue. The graph is iteratively 'grown' in depth, starting with a random generated node as the root (line 6). The algorithm generates several "child" nodes that same with the arity of each node in the queue (lines 15–32). A node as well as its level ℓ are contained in each item in the queue; the level

Algorithm 1: Framework of the Initialization Method

```

1  $S \leftarrow$  Initializing a new multiset which is empty
2  $Q \leftarrow$  Initializing a new first-in, first-out queue
3  $d \leftarrow$  Generating a integer depth that is larger than zero in a random manner
4 Generating a node  $r$  which is not null as the root in a random manner, and adding the pair
    $(r, 0)$  to  $Q$ 
5  $nonleaf \leftarrow$  False
6  $level \leftarrow 0$ 
7 while  $Q$  is not empty do
8    $(n, \ell) \leftarrow$  Dequeueing a node and its level from  $Q$ 
9    $a \leftarrow$  Getting the arity of  $n$ 
10  if  $\ell \neq level$  then
11     $nonleaf \leftarrow$  False
12     $level \leftarrow \ell$ 
13  end
14   $i \leftarrow 0$ 
15  while  $i < a$  do
16    if  $\ell + 1 = d$  then
17      Adding an edge from  $n$  to a new INPUT node
18    else
19       $R \leftarrow$  Computing a set of node types that are disallowed
20       $c \leftarrow$  Generating a node limited by  $R$  at random
21      if  $c = null$  then
22         $S \leftarrow S \cup \{n\}$ 
23      else
24        Adding an edge from  $n$  to  $c$ 
25        if  $type(c) \neq INPUT$  then
26          Enqueueing the pair  $(c, \ell + 1)$  to  $Q$ 
27           $nonleaf \leftarrow$  True
28        end
29      end
30    end
31     $i \leftarrow i + 1$ 
32  end
33 end
34 Adding missing edges to all nodes in  $S$ , with root  $r$ 
35 Remove excess pooling nodes, starting from root  $r$ 
36 return  $r$ 

```

is equivalent to the minimal number of edges from the node to the root. The level information is utilized to prevent producing nodes that are deeper than the target depth which is determined previously. Nodes with a level $\ell = d - 1$ are restrained to have only INPUT nodes as offspring; INPUT nodes are leaf nodes, preventing the further growth of the graph (lines 16–17). Otherwise, the algorithm is going to generate additional nodes at random for the parent node for $\ell < d - 1$. (lines 19–29). To avoid constructing invalid or inefficient architectures, the set R (line 19) is utilized to specify which types of node are disallowed (see Algorithm 2 in Chap 3).

It has come to attention that a straightforward breadth-first algorithm, in which new child nodes are produced for each existing node, can only construct trees. To allow generating the DAGs, the process that produces random nodes to return NULL is permitted (lines 21–22). Instead of adding an edge to a new node, the NULL value indicates that an edge that references an existing node should be added, resulting in a DAG. Delay this process by adding the node to a multiset S for the subsequent processing because the graph has not been fully built at this phase in the algorithm (line 22). As a result, the overall algorithm can be stated as building a tree first, then the multiset S is used to add additional edges to generate a DAG. Else, if returning a non-NUL node which is generated at random, the new edge will be added and enqueued.

Following the primary part of the algorithm, the “missing” edges are added to the tree in order to construct a graph in the further processing (line 34); which is described in Algorithm 3 in Chap 3. Additionally, validation of the number of pooling nodes requires postprocessing. Since every pooling node halves the spatial dimensions, there is a limit on how many pooling nodes may be used before the dimensions of image changed to 1×1 , making further pooling invalid. Since the kernel size and stride size of the pooling nodes are both 2, the base-2 logarithm of the size of the input image can be used to determine the maximum number of pooling nodes that can be contained within a linear sequence for a square image. The problem is solved in implementation by calculating the pooling nodes number and using a convolutional node that is generated at random to replace the redundant pooling nodes (line 35). As employing pooling nodes before other operations (convolution, etc.) may result in the loss of important image data as an outcome of the reduction in dimensionality, the pooling nodes closest to the input data are replaced first.

13.2.4 Initialization Algorithm Details

In this section, the components that are employed within the overall framework of the previously mentioned initialization algorithm will be introduced. The process for determining the set R of limited node types, which is employed when producing additional nodes for the purpose of expanding the graph, is presented in Algorithm 2.

The target depth d for the graph, as well as the current node n (to which the new nodes are going to be attached) and its level ℓ , are the inputs to Algorithm 2. The boolean value *nonleaf* is calculated in Algorithm 1 and indicates if a non-leaf node (i.e. one that is not INPUT) is added at the current level.

R is first initialized as an empty set by the algorithm (line 1). If an edge from node n to an INPUT node already exists (condition 1), or if there has not been any non-leaf node added at the current level (condition 2), then INPUT will be added as a limited node type (lines 2–5). The first condition is there because that the nodes with arity $a > 1$ are CONCAT and SUM, respectively, and summing or concatenating the input image data with itself would be redundant. The second condition ensures that the DAG can grow continually in terms of depth: if no non-leaf nodes have been added

Algorithm 2: Determining Disallowed Node Types

```

Input: The boolean nonleaf; a depth  $d$ ; a node  $n$  with level  $\ell$ 
1  $R \leftarrow \emptyset$ 
2  $b \leftarrow$  True if there exists an edge from  $n$  to an INPUT node; else False
3 if  $b$  or not nonleaf then
4   |  $R \leftarrow R \cup \{\text{INPUT}\}$ 
5 end
6 if  $\ell + 2 = d$  or type( $n$ )  $\in \{\text{SUM, CONCAT}\}$  then
7   |  $R \leftarrow R \cup \{\text{BINARY}\}$ 
8 end
9 return  $R$ 

```

to this level yet, adding an INPUT (i.e., leaf) node may result in the graph generation being terminated before the target depth is reached.

In the second primary stage of Algorithm 2 in Chap. 3, it is decided whether or not nodes for binary operators (CONCAT and SUM) are allowed (lines 6–8). Binary operators can only be permitted if the level of ‘parent’ node meets $\ell < d - 2$, according to the first condition. This condition is added because if the level of node meets $\ell = d - 2$, a new CONCAT or SUM node is only be able to take input from the pixels of input image. If this happens, the image pixels will be redundantly summed or concatenated, as in the same situation previously discussed. Chains of CONCAT and SUM nodes are prohibited by the second condition. That is, no edge is allowed to exist between the two binary nodes. This constraint is not strictly required, unlike the preceding cases mentioned in Algorithm 2 in Chap. 3. Instead, this condition is employed as a search space constraint. Allowing arbitrary binary operator sequences can rapidly increase the branches number produced by the algorithm, potentially resulting in suboptimal and overly complex CNNs. It is worth noting that the CNN encoding is still universal, and since this encoding is designed to cooperate with EAs, it is possible to build genetic operators in such a way that binary operators are allowed to be represented by the edges between nodes.

The procedure of adding new edges according to the multiset S of “missing” edges is detailed in Algorithm 3. In other words, the algorithm adds the edges indicated in S to the tree, transforming it into a DAG.

The algorithm chooses a node n in the graph at random for each node p in the set (line 6). Because there is a possibility that n is not a valid option (will be discussed next), the algorithm repeatedly loops until it finds a node n that is valid, up to some maximal number of iterations. If this maximal number is passed, and edge is added from p to an INPUT node. There are two possible explanations for why the addition of an edge between nodes p to n is invalid: one is that this could form a cycle, or that an edge from p to n already exists (line 8). Furthermore, unless the maximal number is exceeded without a valid n being identified, an INPUT node is not considered a valid choice for n ; this is due to the fact that the purpose of this method is to generate new connections between nodes inside the graph (analogous to the idea of skip connections, as mentioned before).

Algorithm 3: Adding Missing Edges

```

Input: The root node  $r$  of the DAG; the multiset  $S$  holding every node  $p$  who has a missing
      edge
1 for  $p$  in  $S$  do
2   |  $t \leftarrow \text{NULL}$ 
3   |  $valid \leftarrow \text{False}$ 
4   |  $k \leftarrow 0$ 
5   | while  $k$  is below the count of maximum iteration do
6     |   |  $t \leftarrow$  A node chosen at random from the graph with root  $r$ 
7     |   |  $b_1 \leftarrow$  True if a path exists in the directed graph that begins  $t$  and ends at  $p$ ; else False
8     |   |  $b_2 \leftarrow$  True an edge from  $p$  to  $t$  exists; else False
9     |   | if  $\text{type}(t) \neq \text{INPUT}$  and not  $b_1$  and not  $b_2$  then
10    |   |   |  $valid \leftarrow \text{True}$ 
11    |   |   | break
12    |   | end
13    |   |  $k \leftarrow k + 1$ 
14  | end
15  | if  $valid$  then
16    |   | Adding an edge from  $p$  to  $t$ 
17  | else
18    |   | Adding an edge from  $p$  to a new INPUT node
19  | end
20 end

```

The method to generate a node and the corresponding parameters at random is detailed in Algorithm 4; this method is based on the method employed in the CNN-GA paper [1], but the set R is introduced to specify those disallowed node types. The set R is utilized to create a set P of allowed node types (lines 1–2), from which a node type is selected at random using a uniform distribution (line 3). Notice that there are more different values in the set U than just the node types that are available in the representation of graph. That is, instead of containing AVERAGE and MAX nodes in U , a POOL type is simply used; similarly, use a BINARY type instead of SUM and CONCAT types. This is because that every type in P is assigned with same chance. If, for instance, the pooling types AVERAGE and MAX are included separately in P , the likelihood of producing a pooling node would be higher than that of generating a convolutional node, which should be prevented since pooling layers reduce dimensionality and can lead to information loss if employed too excessively. Similarly, if a binary operation (SUM or CONCAT) has a higher probability than a convolutional node, this may lead to overly complex architectures of CNN.

If the node type BINARY is chosen, a number q with random value between 0 and 1 is generated (with uniform distribution). If the value is lower than 0.5, then returning a new SUM node; else, a CONCAT node is generated. If CONV is chosen as the node type, a new CONV node is produced, with the feature maps number specified at random. To be specific, the feature maps number is uniformly selected from {64, 128, 256}, which are the numbers used by the CNN-GA approach [1]. Furthermore, having a fixed set of options for the feature maps number helps to reduce the search space

Algorithm 4: Generating Random Node

```

Input: A set  $R$  of those node types that are restricted
1  $U \leftarrow \{\text{POOL, CONV, REFERENCE, INPUT, BINARY}\}$ 
2  $P \leftarrow U - R$ 
3  $n \leftarrow \text{A new node}$ 
4  $t \leftarrow \text{A node type randomly chosen in } P$ 
5 if  $t = \text{BINARY}$  then
6    $q \leftarrow \text{Generating a number between 0 and 1 with uniform distribution}$ 
7   if  $q < 0.5$  then
8     |  $n.type \leftarrow \text{CONCAT}$ 
9   else
10    |  $n.type \leftarrow \text{SUM}$ 
11  end
12 else if  $t = \text{CONV}$  then
13   |  $n.type \leftarrow \text{CONV}$ 
14   |  $n.feature\_maps = \text{Random choice in } \{64, 128, 256\}$ 
15 else if  $t = \text{POOL}$  then
16   |  $q \leftarrow \text{Uniformly generate a number between 0 and 1}$ 
17   | if  $q < 0.5$  then
18     |   |  $n.type \leftarrow \text{MAX}$ 
19   | else
20     |   |  $n.type \leftarrow \text{AVERAGE}$ 
21   | end
22 else if  $t = \text{INPUT}$  then
23   |  $n.type \leftarrow \text{INPUT}$ 
24 else if  $t = \text{REFERENCE}$  then
25   |  $n = \text{NULL}$ 
26 end
27 return  $n$ 

```

(hence contributing to an increase in the effectiveness of the search). If the node type POOL is chosen, there is a 50% chance that an AVERAGE node will be produced; otherwise, a MAX node will be produced.

A new INPUT node will be simply returned by the method if the specified type is INPUT. The special REFERENCE type is utilized to indicate the circumstance that no new node is produced but an edge point to a node that exists in the graph is inserted instead. The NULL will be simply returned in this case by Algorithm 1 in Chap. 4. This value is subsequently utilized as an indicator by the initialization procedure (Algorithm 1 in Chap. 3), which, as previously explained, updates the set S of “missing” edges to be processed.

13.3 Experimental Studies

13.3.1 Overview

On the benchmark data set named CIFAR-10 [9], the suggested encoding and initialization method is tested. To improve the efficiency of the architecture search, only take 10% of the 50,000 training images during the architecture search process. Specifically, 4,500 images are utilized as the training set and 500 images are used as the validation set throughout the architecture search. In order to preserve a balance throughout the training set and validation set regarding the class distribution, a stratified sampling process is used to select the subsets of the whole 50,000 training images.

A simple random search is employed to show the effectiveness of the encoding, in which 200 CNN architectures are randomly produced with the initialization method. The 4,500 images are used to train each CNN, and the performance is evaluated after each epoch on the validation set. For each architecture, the obtained highest validation accuracy is recorded. the three architectures with the best accuracy are selected after all models are trained using the reduced data set and re-train them with the complete training set that contains 5,000 validation images and 45,000 training images. The objective for choosing three architectures instead of one optimal architecture is to increase the chances of selecting one that generalizes from the reduced training set to the complete training set. For each architecture, recording the epoch with the best validation accuracy. Finally, each architecture is re-trained another time, using the whole training set (contains 50,000 images) *without* a validation set, up to the maximum epoch already determined; On the test set, calculating the accuracy of the CNN which is re-trained.

13.3.2 Parameter Settings

A learning rate schedule and training routine are employed according to the settings employed in [10] for evaluating the performance of every CNN architecture. That is, training up to 200 epochs using stochastic gradient descent [11] with an initial learning rate of 0.1, a momentum of 0.9, and a minibatch size of 128. After the 60th, 120th, and 160th epochs, the learning rate decays by a factor of 0.1. The Softmax cross-entropy loss is employed as the loss function. The CNN architecture was simply discarded in circumstances when an error occurred during the training of CNN, e.g., the program obtains a NaN (not-a-number) loss result or runs out of the available GPU memory.

The weight decay with the coefficient of 0.5×10^{-3} is employed, following the setting employed in [10], to reduce overfitting. Furthermore, a data augmentation technique is adopted according to the method employed in [1, 7]: performing a

random horizontal flip, after each side of the image being padded with 4 pixels, a random 32×32 crop is selected.

For the CNN population initialization algorithm, a range of depths must be specified for the random search. The maximum depth is set to 12 and the minimum depth is set to 6 in this implementation. Although this limits the greatest depth of a random search, The genetic operators like mutation and crossover could be used by an EA-based search to expand the representation to any depth; this is going to be explored in the following work.

13.3.3 *Experimental Results*

The accuracy gained on the test set of CIFAR-10 (contains 10,000 images) by a random search utilizing the encoding and by 10 state-of-the-art peer competitors, involving both CNNs built by automatic design algorithms and CNNs that are hand-crafted, is shown in Table 13.1. All random search accuracies on the test set are achieved after the found architecture being re-trained using the entire 50,000 training images. The amount of trainable parameters (in millions), the GPU days consumed to search the architecture, and the classification accuracy are also displayed for each model; this table format is similar to that used in [1]. The “GPU days” column of hand-crafted architectures is marked “–”.

Two separate trials of the random search are shown for designed method. The results of the CNN which contains the fewest parameters and the CNN which achieves the best accuracy are displayed for each trial. The architecture search consumes roughly 10 h utilizing two GTX 1080 Ti GPUs; the three final architectures are re-trained with one GPU and consumes about 12 h, for a total of 1.33 GPU days.

Even though random search achieves classification accuracies that are generally lower than those CNNs built by automatic design algorithms and CNNs that are hand-crafted, the computational resources required by this method is significantly less and the results it achieved are promising despite using the reduced data during architecture search. In addition, the method is capable of producing models of various complexities (i.e., numbers of parameters); for instance, after being re-trained with 50,000 images, One generated CNN which contains only 750k parameters could achieve an accuracy of 91.44%.

The graph representation corresponding to the best architecture from the random search (trial #2) is shown in Fig. 13.2. The “AP” and “MP” nodes in the diagram represent AVERAGE and MAX, respectively; the “Conv” nodes indicate convolutional building blocks, with the number of feature mappings in parenthesis.

Table 13.1 The results obtained by using the encoding in random searches compared to the peer competitors

	Accuracy %	Parameters	GPU Days
ResNet (depth = 101) [2]	93.57	1.7 M	–
ResNet (depth = 1,202) [2]	92.07	10.2 M	–
VGG [12]	93.34	20.04 M	–
DenseNet [4]	94.17	27.2 M	–
Hierarchical Evolution [13]	96.37	–	300
Block-QNN-S [3]	95.62	6.1 M	90
Genetic CNN [14]	92.90	–	17
Large-scale Evolution [15]	94.60	5.4 M	2,750
CGP-CNN [7]	94.02	1.68 M	27
CNN-GA [1]	95.22	2.9 M	35
Random search with the encoding (trial #1, best accuracy)	92.45	1.45 M	1.33
Random search with the encoding (trial #1, fewest parameters)	90.98	0.83 M	1.33
Random search with the encoding (trial #2, best accuracy)	92.57	1.14 M	1.33
Random search with the encoding (trial #2, fewest parameters)	91.44	0.75 M	1.33

13.4 Chapter Summary

The methods introduced in Part III utilize the encoding strategy where each individual is linear, and implemented the corresponding algorithms using the data structure of the linked list. Given the fact that existing state-of-the-art CNNs are almost not linear, such as with skip connections, using a linked list for the representation may not be straightforward, since additional efforts are required for the corresponding transformation. Recent related works showed that the architecture of DNNs can also be represented by a DAG, based on which the encoding strategy can be reasonably designed. The algorithm introduced in this chapter just followed this motivation and designed the DAG-based encoding strategy for the automatic architecture design of DNNs. The designed encoding strategy can achieve very promising performance on CIFAR-10 only with a random search for a glance of quick verification. This undoubtedly demonstrates the superiority of the GAG-based encoding strategy and

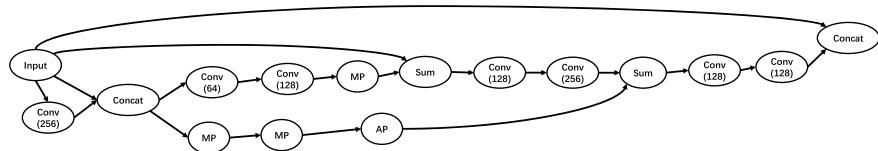


Fig. 13.2 After being re-trained with the whole 50,000 training images, an accuracy of 92.57% is achieved by the random search produced best CNN architecture

inspires further investigation of the graphs based, in contrast to the traditional linear based.

Although NAS could automate the architectures of DNNs without or with only rare human efforts, the computation cost of the NAS algorithm is often prohibitive high. This issue is caused by the training of DNNs during the search process, which need the intensive computational resource. In the next chapter, we will introduce another kind of recent advance in NAS algorithms, i.e., E2EPP, which could greatly alleviate this critical issue.

References

1. Sun, Y., Xue, B., Zhang, M., Yen, G. G., & Lv, J. (2020b). Automatically designing cnn architectures using the genetic algorithm for image classification. *IEEE Transactions on Cybernetics*, 50(9), 3840–3854.
2. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
3. Zhong, Z., Yan, J., & Liu, C.-L. (2018). Practical network blocks design with q-learning. In *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*.
4. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
5. Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (pp. 315–323).
6. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, Volume 37 of Proceedings of Machine Learning Research* (pp. 448–456). PMLR.
7. Suganuma, M., Shirakawa, S., & Nagao, T. (2018). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 5369–5373). <https://doi.org/10.24963/ijcai.2018/755>.
8. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of CVPR* (pp. 2818–2826).
9. Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical Report, University of Toronto. <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
10. Zhu, Y., Yao, Y., Wu, Z., Chen, Y., Li, G., Hu, H., Xu, Y. (2018). Gp-cnas: Convolutional neural network architecture search with genetic programming. [arXiv:1812.07611](https://arxiv.org/abs/1812.07611).

11. Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade* (2nd ed.) (pp. 421–436). Springer. ISBN 978-3-642-35289-8. https://doi.org/10.1007/978-3-642-35289-8_25.
12. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
13. Liu, H., Simonyan, K., Vinyals, O., Fernand C., & Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. [arXiv:1711.00436](https://arxiv.org/abs/1711.00436).
14. Xie, L., & Yuille, A. L. (2017). Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017* (pp. 1388–1397). <https://doi.org/10.1109/ICCV.2017.154>.
15. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).

Chapter 14

End-to-End Performance Predictors



14.1 Introduction

In fact, common optimization problems in ENAS are computationally expensive and are usually handled using surrogate-assisted EAs(SAEAs) [1], employing inexpensive approximation regression and classification models, such as the Gaussian process model [2], radial basis network (RBN), etc., to replace the costly fitness evaluation [3]. SAEAs have proven to be useful and efficient in a variety of practical optimization applications [4]. Typically, the model is trained using a small number of samples with target fitness values, and then the trained model is used to evolve predictors of fitness values for the search process to speed up optimization [5]. Based on the literature [6], SAEAs are classified as online or offline algorithms based on whether or not expensive fitness evaluations are employed to increase the training data throughout the optimization phase. While online algorithms are better in terms of model quality and performance, these are based on additional training data, and new data is very expensive to train, so it is more practical to use offline algorithms. The offline algorithm runs very fast, and since the algorithm is independent in both data collection and optimization search, there is no need to collect new data independently during the training process. So far, the common offline SAEA algorithms broadly contain data pre-processing [7], data mining [6], and ensemble learning [8], which can effectively solve the problem of insufficient data.

The purpose of this chapter is to design an innovative and efficient random forest-based [9] E2EPP. Random forest is used because of its effectiveness on a small number of samples and its robustness in the application of discrete variables as well as in parameter adjustments [10, 11]. When E2EPP examines new CNN architectures, it is able to immediately determine the performance of a CNN, which naturally speeds up the process of ENAS. The contributions of this chapter are listed below:

1. The train data in the random forest are presented as data pairs, each pair consisting of the CNN architecture and the corresponding performance values. However, the CNN architecture is simulated by the describing language and cannot be fed

directly into the random forest model. So, an efficient encoding approach is provided to be able to directly transform the CNN architecture into the corresponding data, which can be used as samples for the random forest to recognize.

2. The samples (i.e., data pairs, including architecture and performance) involved in the training of random forest model may not necessarily cover the entire search space, so if only the tree with the best performance is used for prediction, then it will lead to biased final results. Therefore, a selective ensemble technique is used to pick various trees to synthesize local regions for each generation.
3. Extensive experiments are carried out to validate the two goals of using the performance predictor: the efficiency in accelerating the evolutionary process and the efficacy in discovering the ideal CNN design.

The rest of this chapter is structured as follows. Section 14.2 introduces the relevant work. Section 14.3 then covers the E2EPP algorithm specifics. Sections 14.4 and 14.5 offer the experimental designs and results, respectively, to validate the efficacy and efficiency.

14.2 Related Work

There are two main types of performance predictors that are most popular now. The first type is learning curve performance predictors such as freeze-thaw Bayesian optimization (FBO) algorithm [12] and the E2EPPs mentioned in this chapter, with representatives, such as the Peephole algorithm [13]. Both of predictors are subject by the training-prediction computational paradigm.

The FBO algorithm uses a Bayesian optimization-based regression model to predict the performance of architecture. The principle is to train the regression model by learning data from the learning curve of the first t epochs by using Bayesian optimization, and finally predicting the performance of the network architecture at the T -th epoch ($T > t$). The Peephole algorithm uses end-to-end performance prediction, which is trained by combining multiple CNN architectures and their performance values as data pairs, using long-short time memory NN [14]. The trained model can predict performance directly based on the CNN architecture, so the end-to-end mechanism has raw data on the input end and classification accuracy on the output end. The Peephole algorithm applies word vector techniques to the network architecture which makes it possible to be used directly as input.

The FBO algorithm essentially benefits from its trained-CNN-free nature, which means it does not require any trained CNNs in preparation. The FBO technique is efficient since training a CNN takes time, which can range from many days to weeks. If the learning curve is not smooth, this method will be ineffective because curve fitting relies on smoothness as a premise. Because a schedule of learning rates is commonly utilized in modern deep learning applications, the learning curve is typically not smooth. The learning curve will have a non-smooth portion if the learning rate is altered. Another shortcoming of the FBO algorithm is its non-end-to-end nature

(i.e., in order to forecast the performance of each CNN, a portion of the training data for this CNN must be acquired in order to train the predictor), which necessitates significantly more labor work when employed. Owing to the end-to-end nature, the Peephole algorithm is more realistic and easier to use. However, its disadvantage is also very obvious that it requires a large number of training samples. Compared to ENAS, which does not use a predictor, the Peephole algorithm has greater computational complexity in collecting training data, requiring 8,000 CNN architectures. ENAS, on the other hand, consistently achieves encouraging results despite examining only hundreds of individuals. If there are enough computing resources to train 8000 architectures, then there is no need for a performance predictor. This limitation is partly due to the NN-based method utilized in its regression mode, which normally relies heavily on a significant amount of labeled training data.

E2EPP provides the advantages of being end-to-end and depending on limited training data, which solve both of the drawbacks noted above for existing performance predictors.

14.3 Algorithm Details

The main challenge of ENAS, as previously stated, is the prohibitive cost of evaluating a single CNN design. Random forest as the fitness predictor is used, inspired by offline SAEAs, to replace the costly fitness measurement. Figure 14.1 depicts the framework of the designed performance predictor as well as the accompanying ENAS for a better understanding. Many CNN architectures that have been trained on specific tasks are applied to the training of random forest-based performance predictors. The building

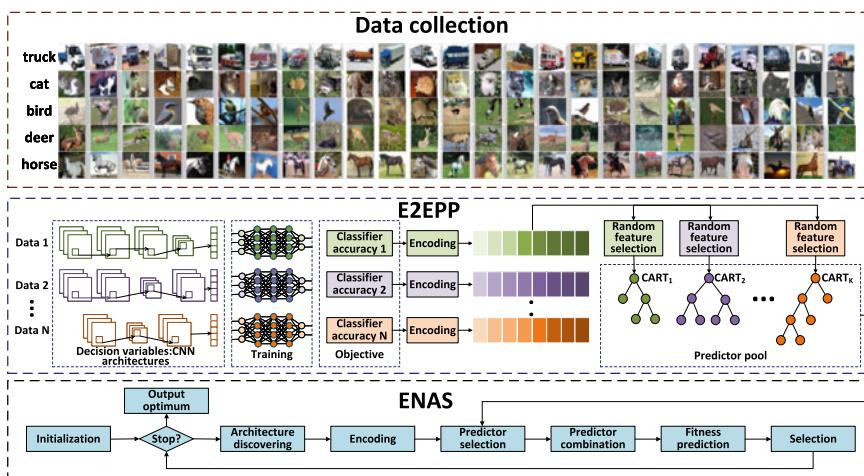


Fig. 14.1 Main framework of E2EPP

and prediction procedure of random forest are computationally cheaper than the CNN training process and can be utilized again during the evolutionary optimization. As a result, the significant computational cost of ENAS can be reduced.

E2EPP consists of three blocks, as shown in Fig. 14.1. As mentioned above, a series of data pairs are needed as samples before the predictor is trained. So firstly, the image data are obtained through the data collection block and trained using ENAS, during which no E2EPP is involved. After training, each data sample containing the architecture with corresponding performance is obtained. Second, the architecture is encoded into discrete form and used to construct a random forest-based prediction pool, which contains a large number of regression trees(say K) (denoted as CARTs). The newly constructed CNN architecture is encoded as input to a random forest during each generation of ENAS optimization and its performance is predicted using an adaptive combination of CARTs from a pool of predictors. When the optimization is complete, ENAS outputs the optimal network structure. Note that there is no additional training process during this period.

Since the CNN architecture is first transformed into discrete variables as data samples, random forest is chosen as the fitness predictor in E2EPP because it is more suitable for discrete regression problems. A fixed random forest cannot guarantee accuracy in local areas where the population is dispersed, hence a random forest-based predictor is adaptively updated during the generations of evolutionary search. Encoding the CNN architectures into the appropriate format for the random forest, training random forest, and predicting fitness using learned random forests are all covered in the following sections. Then, the strength and weakness of E2EPP are summarized.

14.3.1 Encoding

The encoding is performed on trained CNNs with randomly initialized architectures. To gather this information, AE-CNN is used to construct a collection of valid CNN architectures at random. A valid CNN architecture is one that can be trained using a prescribed training process with no exceptions, such as out-of-memory failures that result in a zero-classification accuracy. The acquired training data are described below.

1. RBs and DBs: Each CNN architecture consists of as many as four RBs and four DBs, while the output channels of each block are chosen between [32, 512], which is specified by the state-of-the-art CNN [15, 16].
2. PBs: Each generated CNN architecture has a maximum of four pooling layers. MAX and AVERAGE are the two categories of PBs.

In general, a CNN is encoded into a chromosome using $3N_b + 2N_p$ discrete variables, where N_b is the maximum number of RBs and DBs and N_p is the maximum number of PBs. RBs or DBs are encoded into a triplet, i.e. $[type, out, amount]$,

Algorithm 1: Encoding A CNN Architecture

Input: The CNN architecture \mathcal{A} , the maximal number B_n of DBs and RBs, the maximal number N_p of PBs.

Output: The encoded architecture information.

```

1  $b\_list \leftarrow \emptyset;$ 
2  $p\_list \leftarrow \emptyset;$ 
3  $l \leftarrow$  Calculate the number of blocks in  $\mathcal{A}$ ;
4 for  $i \leftarrow 1$  to  $l$  do
5    $block \leftarrow$  Get the  $i$ -th block of  $\mathcal{A}$ ;
6   if  $block$  is a RB then
7     | Put 1 into  $b\_list$ ;
8     | Put the values of the  $out$  and  $amount$  of  $block$  into  $b\_list$ ;
9   else if  $block$  is a DB then
10    | Put the values of the  $k$ ,  $out$  and  $amount$  of  $block$  into  $b\_list$ ;
11   else
12     | if  $block$  is a maximal pooling layer then
13       |   | Put 1 to  $p\_list$ ;
14     | else
15       |   | Put 0 to  $p\_list$ ;
16     | end
17     | Put  $i$  to  $p\_list$ ;
18   end
19 end
20 Put zero to  $b\_list$  until  $|b\_list| = 3N_b$ ;
21 Put zero to  $p\_list$  until  $|p\_list| = 2N_p$ ;
22 Return  $b\_list \cup p\_list$ .

```

which make up the first $3N_b$ variables. RBs have only one block type and set to 1, while DBs have three block types and the value depends on k , corresponding to 12, 20, and 40. It should be noted that the variable parameter in is not set in the triples because: (1) the in can be determined by the out of the previous block; (2) when the amount of data is relatively small, reducing the decision variables can effectively improve the performance of the regression model. Each pooling layer is encoded into a pair as [pooling type, layer position] for the following $2N_p$ variables, with the MAX and AVERAGE types represented by one and zero, respectively. If there are p PBs and the same number b of RBs and DBs in the CNN architecture, then the variables at from the $(3b + 1)$ th position to the $3N_b$ th position and those at from the $(3N_b + 2p + 1)$ th position to the $(3N_b + 2N_p)$ th position are specified as zeros. As a result, both the input and output of the random forest-based performance predictor have been determined; the input is $(3N_b + 2N_p)$ decision variables, while the output is a continuous value between $[0, 1]$. Finally, the concrete implementation of encoding the CNN architecture into numerical data that can be directly input is shown in Algorithm 1, where $|\cdot|$ denotes the countable operator.

14.3.2 Training of the Random Forest

A large number of CARTs can be generated in the prediction pool, while each CART is trained on the entire training set as well as a subset of random features (i.e., discrete variables) with a probability of 0.5 for each discrete variable, thus increasing the diversity of the entire prediction pool [17]. In the decision space, each node of CART represents a rectangular region, and the MSE of the samples in that region determines whether the node needs to be split (i.e., whether the MSE is less than a predefined threshold T_s [18]. If yes, the node is denoted as a leaf node). Once the K CARTs are trained, the prediction pool is available to be used as a predictor. Algorithm 2 demonstrates the entire process described above.

Algorithm 2: Performance Predictor Training

Input: The K CARTs, the encoded training data \mathcal{D}_{train} , the feature number m .

Output: The K trained CARTs and their selected feature ids.

```

1  $I \leftarrow \emptyset$ ;
2 for  $i \leftarrow 1$  to  $K$  do
3    $CART \leftarrow$  Select the  $i$ th CART from CARTs;
4    $v \in R^m \leftarrow$  Randomly generated a vector from  $[0, 1]$ ;
5    $I_i \leftarrow$  Collect the position of the elements whose values are greater than 0.5 in  $v$ ;
6   Train  $CART$  on the features whose ids are in  $I_i$ ;
7    $I \leftarrow I \cup I_i$ ;
8 end
```

Output: The K trained CARTs and the corresponding selected feature ids I .

14.3.3 Performance Prediction

It is very difficult to provide new data samples for the predictor pool because of the high cost of network training. Therefore, efficiently tuning and evaluating predictors is very difficult to do in SAEAs [4]. To compensate for the lack of data samples when training predictors, a large number of surrogate models have been used as ensemble members in recent offline SAEA [8], which has also proven to be feasible and effective. In addition, at each iteration, these ensemble members can be adaptively collected and combined to provide local information about the present population. Inspired by our previous work [8] that first proposed a combination technique for dealing with continuous values with decision variables, in the technique of this chapter, it will be used to adjust the random forest-based predictor at each iteration, i.e., to randomly select Q CARTs and use their average as the final fitness value. Although the decision variables of E2EPP are discrete, the combinatorial technique is also experimentally proven to work very well.

All of the K trained CARTs re-estimated their performance on the CNN \mathcal{A}^b with the best-predicted fitness value in each generation, and then Q of the K CARTs are uniformly picked from the K ranked CARTs depending on their prediction values on \mathcal{A}^b . To analyze both the parent and offspring populations, the Q CARTs are concatenated as the ensemble performance predictor. This selection is made regarding the performance diversity of CARTs based on the existing best CNN architecture \mathcal{A}^b . The generated CNN architecture \mathcal{A} is evaluated using an ensemble predictor consisting of Q CARTs. Thus, for the final fitness landscape, the adaptive ensemble predictor is able to balance the global tendency and local information, where K CARTs are used to evaluate the global average landscape, while Q of them explore the local information of the landscape. The prediction process for a generation is described in detail in Algorithm 3.

14.3.4 Strength and Weakness of E2EPP

As discussed in Sect. 14.2, the current performance predictor suffers from several shortcomings: (1) non-end-to-end nature, (2) setting strong assumptions on the learning curve to make it more consistent with the ideal state, and (3) dependence on large training samples. The E2EPP method mentioned in this section can effectively address the above issues.

The end-to-end nature is more adaptable during practical applications, firstly it does not need to consider whether the learning curve is smooth or not, and secondly it no longer needs to predict each CNN performance during the optimization process. Therefore, E2EPP can outperform the existing learning curve-based algorithms. According to the universal approximation principle [19], if a smooth curve exists during training, the desired performance can be achieved by the technique of learning curve, but in practice, the curve is often not so ideal.

E2EPP, on the other hand, does not necessitate a huge number of training samples. Most present algorithms that use deep learning techniques work if a large amount of training data is available. Such training data is collected in CNN performance prediction by training a large number of CNNs from scratch. However, even when running on GPUs, training still takes prohibitive computation time, and the primary objective of developing performance predictors is to cut down on the total amount of time spent training CNNs. If sufficient time is spent to training a large number of DNN architecture, the purpose of designing a performance predictor will lose its essence. The relationship between CNN architectures and their performance is established in E2EPP by random forests. The reason for doing so is that random forests can obtain desirable results using only a small number of training data, which is also proven in theory and practice [9, 10].

Unfortunately, the shortcoming of E2EPP is the uncertain of the least number of training samples required to achieve promising performance, which varies every task. In practice, training samples must be examined incrementally until the target performance is obtained.

Algorithm 3: Performance Predicting

Input: The K trained CARTs, the selected features ids I of each CART, the current best CNN \mathcal{A}^b , the number of most diverse prediction Q , the generated architectures \mathcal{A} to be evaluated.

Output: The fitness values of \mathcal{A} .

- 1 $Y \leftarrow \emptyset;$
- 2 $\mathcal{A}_{encoded}^b \leftarrow$ Encode \mathcal{A}^b based on the details shown in Sect. 14.3.1;
- 3 **for** $i \leftarrow 1$ to K **do**
- 4 | $CART \leftarrow$ Select the i -th CART from CARTs;
- 5 | $x \leftarrow$ From $\mathcal{A}_{encoded}^b$ select the elements whose ids are in I_i ;
- 6 | $y \leftarrow$ Use $CART$ predict the classification accuracy on x ;
- 7 | $Y \leftarrow Y \cup x$;
- 8 **end**
- 9 $Y \leftarrow$ Order the elements in Y ;
- 10 $I_{selected}^{CART} \leftarrow$ Uniformly select Q CARTs based on Y ;
- 11 **for** $i \leftarrow 1$ to $|\mathcal{A}|$ **do**
- 12 | $F_i \leftarrow$ the mean prediction of Q selected CARTs ($I_{selected}^{CART}$) on $|\mathcal{A}_i|$;
- 13 **end**
- 14 **Return** F .

14.4 Experimental Design

A number of studies are carefully prepared and carried out to validate the usefulness and efficiency of E2EPP. Although E2EPP intends to accelerate ENAS fitness evaluation, the ultimate goal is to identify the optimum CNN architecture that offers excellent classification performance on the given images. As a result, two tests are carried out: (1) analyzing classification performance of E2EPP alongside AE-CNN, and (2) inspecting the efficiency of E2EPP. Based on the conventions, the CIFAR-10 and CIFAR-100 are adopted as the benchmark datasets. In the following, the selected peer competitors and the corresponding parameter settings are detailed.

14.4.1 Peer Competitors

At this stage, there already exist many excellent algorithms that show excellent performance on classification tasks. Therefore, in the experiments, the peer competitors are classified into three categories, which are hand-designed network architecture, CNN network architecture obtained based on other search strategies (mainly RL), and ENAS-related algorithms. Among them, the first category contains DenseNet [16], ResNet [15], VGG [20], Maxout [21], Highway Network [22], Network in Network [23], FractaNet [24], All-CNN [25]. Specifically, considering the different performance of the algorithms, two ResNet-related variants are introduced, ResNet(depth = 101) and ResNet(depth = 1202). The second category also belongs to the algorithms of NN architecture search, mostly using RL methods for

search, mainly including NAS [26], MetaQNN [27], ENS [26] and Block-QNN-S [28]. The third category includes Large-scale Evolution [29], Genetic CNN [30], Hierarchical Evolution [31] and CGP- CNN [32], etc. Given that E2EPP was tested in a case study of AE-CNN, AE-CNN was still chosen to combine E2EPP (AE-CNN + E2EPP) during this comparison. The peer competitors chosen for the second experiment are the existing performance predictors [13, 33] discussed above.

14.4.2 Parameter Settings

The parameter settings for creating the training data from AE-CNN are given first in this section, followed by those of E2EPP. It should be noted that these parameter settings are used in both studies.

According to accepted deep learning training methods, it is most common to use an SGD optimizer equipped with Xavier method [34] to initialize the weights at the same time. In addition, the learning rate is set dynamically to 0.01 for the first epoch and each epoch from 151–249th, 0.1 for the second epoch to the 150th epoch, and 0.001 for the remaining epochs. The learning rate decay rate is set to 5×10^{-4} , the specified batch size is 128, and 350 epochs for training. The fitness evaluation dataset is used to assess classification accuracy. Because the utilized benchmark datasets lack a corresponding validation dataset, 20% images from the relevant training dataset are chosen at random as the fitness evaluation dataset. All of the CNNs are trained on three GPUs, all of which are NVIDIA 1080TI. Furthermore, due to the restricted GPU capacity, the maximum number of RB and DB are set to 4. The number of pooling layers in the CNN network architecture is related by the input size, and the image size in both CIFAR-10 and CIFAR-100 is 32×32 , so the pooling layers are set to 4. In addition, according to the DenseNet standard, the number of DUs is determined by the parameter k . When $k = 12$ or $k = 20$, the number of DUs in DB is 10, and when $k = 40$, the DUs are 5. The number of RUs in RB can be arbitrarily chosen as a number no greater than 10. For the parameter settings in the training data, N_p and N_b are 4 and 8, respectively, so according to the calculation, the length of the corresponding decision variables should be $32 (8 \times 3 + 4 \times 2)$. In addition, 1000 CARTs are generated as the performance prediction pool. When expanding each CART, the threshold for terminating node splitting is set to $1e - 4 \times \sigma^2$ (σ^2 is the variance of the training data). The random forest-based predictor is combined with 100 CARTs in each iteration. Table 14.1 summarizes all of the parameter settings.

When designing a CNN architecture with AE-CNN and the E2EPP method, the maximum generation number and population size are set 20, and the crossover and mutation probabilities are 0.9 and 0.1, respectively, as advised by AE-CNN. In addition, each generated CNN architecture is trained with an epoch in the GPU for detecting whether it causes memory overflow before it is evaluated by E2EPP so that the architecture can be evaluated by E2EPP in general. This is also for E2EPP to maintain a consistent data distribution between training and test data. After completing the evolutionary process, the individual with the highest classification accuracy

Table 14.1 A summary of the parameter settings

Parameter name	Parameter value
Learning rate	0.01 for 1, and 151–249 epochs; 0.1 for 2–150 epochs; 0.001 for 251–350 epochs
Weight decay	5×10^{-4}
Batch size	128
Training epochs	350
k of the DB	{12, 20, 40}
Maximal number of RUs in a RB	10
Maximal number of DUs in a DB	10 when $k = 12$ and $k = 20$; 5 when $k = 40$
Maximal number of RBs (N_b)	8
Maximal number of DBs (N_p)	4
Population size	20
Generation number	20
Crossover probability	0.9
Mutation probability	0.1
Number of CARTs	1000
Threshold of stopping node splitting	$1e - 4 \times \sigma^2$ (σ^2 is the variance of the training data)
Number of selected CARTs	100

is selected and run five times independently using the training pattern with the same settings as collecting training data to give the best results.

14.5 Experimental Results

This section presents and analyzes the results of the designed experiments. Section 14.5.1 delves into the classification results of AE-CNN+E2EPP in terms of classification accuracy and GPU days consumed. To thoroughly study E2EPP, its efficiency and effectiveness are individually investigated, and the relevant experimental results are provided in Sects. 14.5.2 and 14.5.3, respectively. In addition, a comparison was carried out between the random forest used in E2EPP and the RBN in order to highlight the potentially advantageous performance of random forest.

14.5.1 Overall Results

Table 14.2 shows the experimental findings of the comparing methods in terms of classification accuracy and GPU days utilized. Specifically, according to the descrip-

Table 14.2 The comparison of AE-CNN+E2EPP and the peer competitors in terms of the classification accuracy (%) and the consumed GPU days on the CIFAR-10 and CIFAR-100 benchmark datasets

	Peer Competitors	CIFAR-10	CIFAR-100	GPU Days
State-of-the-art CNNs manually designed	DenseNet [16]	94.76	75.58	–
	ResNet (depth=101) [15]	93.57	74.84	–
	ResNet (depth=1,202) [15]	92.07	72.18	–
	Maxout [21]	90.70	61.40	–
	VGG [20]	93.34	7.95	–
	Network in Network [23]	91.19	64.32	–
	Highway Network [22]	92.28	67.61	–
	All-CNN [25]	92.75	66.29	–
CNN architecture design algorithms based on non-evolutionary methods	FractalNet [24]	94.78	77.70	–
	NAS [26]	93.91	–	22,400
	MetaQNN [27]	93.08	27.14	100
	EAS [26]	95.77	–	10
CNN architecture design algorithms based on evolutionary methods	Block-QNN-S [28]	95.62	79.35	90
	Genetic CNN [30]	92.90	70.95	17
	Large-scale Evolution [29]	94.60	77.00	2,750
	Hierarchical Evolution [31]	96.37	–	300
	CGP-CNN[32]	94.02	–	27
	AE-CNN + E2EPP	94.70	77.98	8.5

tion above, the algorithms involved in the comparison are divided into three categories, corresponding to the second, third and fourth rows, respectively. And the last row shows the results obtained by AE-CNN+E2EPP. In addition, from the perspective of columns, except for the second column which is the name of algorithm, the

last three columns all need to be obtained experimentally, which are the experimental results of CIFAR-10 and CIFAR-100, and the corresponding GPU days required for the corresponding network training. These architectural design methods typically indicate the number of GPUs and days consumed. They are combined for ease of comparison by using “GPU days” as a metric of computing resource utilization. One GPU day, for example, signifies that a GPU is entirely utilized in one day to attain classification accuracy. The results of the peer competitors in the table are taken from the corresponding seminal publications. In contrast, GPU Days for all algorithms (except CNNs manually designed) are calculated by multiplying the number of GPUs by the number of running days. The “–” indicates that the metric does not exist in the seminal paper. It is worth noting that the GPU Days corresponding to AE-CNN+E2EPP is not the time spent in the evaluation with E2EPP but the time spent to acquire training samples upfront.

On two different datasets, AE-CNN+E2EPP both achieved relatively good results. On CIFAR-10, the accuracy is up to 4% higher compared to the manually designed network. And compared with the optimal algorithm FractalNet, it is just less than 1% lower. On the other more challenging CIFAR-100 dataset, AE-CNN+E2EPP beats all the manually designed networks. Among peer competitors in this category, AE-CNN+E2EPP shows promising performance. In addition, AE-CNN+E2EPP achieves a huge advantage in GPU Days consumption when compared to non-manually-designed algorithms. On CIFAR-10, AE-CNN+E2EPP is only less accurate than Hierarchical Evolution, and on CIFAR-100, AE-CNN+E2EPP beats all non-EC methods as well as ENAS. It is also worth noting that Hierarchical Evolution, EAS, Block-QNN-S, and CGP-CNN are all semi-automatic CNN architecture search algorithms, which means that human design expertise was introduced into the algorithm design process. In summary, AE-CNN+E2EPP won 26 out of 30 comparisons, while winning the GPU Days comparison in a heartbeat.

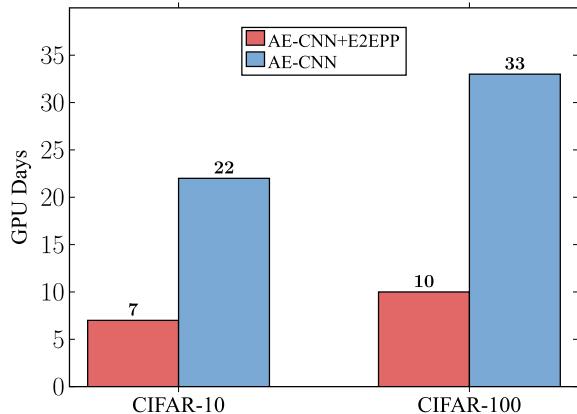
14.5.2 Efficiency of E2EPP

According to Table 14.2, it can be found that the use of E2EPP gives hope to AE-CNN to obtain classification accuracy quickly, which is the role of E2EPP. Therefore, in this section, to further verify the conjecture, whether to activate E2EPP in AE-CNN will be explored. The experimental setting is the same as AE-CNN+E2EPP to ensure fairness. In addition, the GPU consumption days are recorded of the newly generated architecture of AE-CNN when its fitness value satisfies that shown in Eq. (14.1).

$$\left| \frac{f_1 - f_2}{f_2} \right| \leq 0.01 \quad (14.1)$$

where f_1 represents the fitness of the CNN architecture newly generated by AE-CNN and f_2 represents the fitness of the AE-CNN+E2EPP displayed in Table 14.2.

Fig. 14.2 The number of GPU days required by AE-CNN and AE-CNN+E2EPP when attaining the same accuracy of classification on CIFAR-10 and CIFAR-100



The GPU Days consumed by AE-CNN when Equation (14.1) is satisfied on the CIFAR-10 and CIFAR-100 datasets are depicted in Fig. 14.2. Bars with red indicate that E2EPP is used, while bars with blue indicate that E2EPP is not used, and the number corresponding to the top of each bar indicates GPU Days. It can be clearly seen that E2EPP significantly improves the overall performance of AE-CNN. On the two datasets, 15 and 23 GPU Days were saved, respectively. Thus, the goal of developing a predictor capable of accelerating the evaluation has been achieved.

14.5.3 Effectiveness of E2EPP

In the previous section, it was verified that E2EPP plays a very crucial role in AE-CNN. And in this section, to further verify the effectiveness of E2EPP, E2EPP with existing performance predictors, namely FBO and Peephole, are compared. The metrics used in this experiment are MSE, Kendall's Tau (KTau) [35], and the coefficient of determination (CoD) [36] as recommended in [13]. Specifically, KTau indicates the relationship between the predicted ranks and their true ranks, and a larger value of KTau indicates a higher correlation, which takes values in the range $[-1, 1]$. CoD is also an indicator used to represent the relationship between predicted and true values, and it indicates the closeness between predicted and true values, with larger values indicating closer, and its value range is $-\infty$ to 1. In addition, after the quantitative analysis, the qualitative analysis is also carried out by plotting the relationship between the true value and the predicted value through a coordinate diagram, where the horizontal coordinate indicates the true value and the vertical coordinate indicates the predicted value. The parameters of FBO and Peephole are specified depending on the suggestions in [13]. Furthermore, the experiment is repeated 10 times on each benchmark dataset.

Table 14.3 The MSE, KTau, CoD and the used time of FBO, Peephole, and E2EPP algorithms on the CIFAR-10 dataset

	MSE	KTau	CoD	Time (s)
FBO	0.0077	0.2170	-4.346	3.702
Peephole	0.0019	0.5324	0.3765	359.296
E2EPP	0.0006	0.6604	0.5184	3.813

Table 14.4 The MSE, KTau, CoD and the used time of FBO, Peephole, and E2EPP algorithms on the CIFAR-100 dataset

	MSE	KTau	CoD	Time (s)
FBO	0.0078	0.1923	-2.631	3.725
Peephole	0.0024	0.5412	0.3238	458.652
E2EPP	0.0015	0.6501	0.3872	3.514

Tables 14.3 and 14.4 exhibit the quantitative comparison results for CIFAR-10 and CIFAR-100, respectively. As shown in Table 14.3, in terms of MSE, E2EPP is only 0.0006, which is one order of magnitude smaller than the other two predictors. This indicates that E2EPP may outperform FBO and Peephole in terms of prediction results. E2EPP also achieved the best results for the remaining two metrics. The KTau and CoD values of E2EPP reached 0.6604 and 0.5184, respectively, which means that the predicted value of E2EPP is closest to the corresponding true value. Furthermore, E2EPP takes 3.813 s to predict, which is little slower than FBO, which takes 3.702 s. However, it is still significantly faster than Peephole, which takes 359.296 s. Table 14.4 also contains the results of similar comparisons. Specifically, the MSE, KTau, and CoD values of E2EPP are the best among the three predictors, which proves that E2EPP is very advantageous in handling the CIFAR-100 dataset. In addition, E2EPP is also the fastest in time consuming.

In Figs. 14.3 and 14.4, the qualitative comparison results of the different predictors on CIFAR-10 and CIFAR-100 are documented. In each illustration, the horizontal and vertical axes indicate the true and predicted values, respectively, and in addition, the line $y = x$ are drew for visual comparison. If the predicted value is greater than the true value, it is above the line and vice versa, so that more points on the line indicate a better correlation performance of predictor. On CIFAR-10, it can be observed through Fig. 14.3a, b that only half of the values obtained from the FBO are close to the true values, while Peephole gets superior outcomes by providing more points close to the $y = x$. In contrast, in Fig. 14.3c, almost all predicted values obtained by E2EPP are close to the true values. This shows that the prediction effect of E2EPP has the least error and also validates the quantitative analysis in Table 14.3. In addition, the results in Fig. 14.4a, b, c regarding CIFAR-100 show that the data points of FBO and Peephole are more widely distributed, with a larger error between the true and predicted values, while the data distribution obtained by E2EPP is plotted

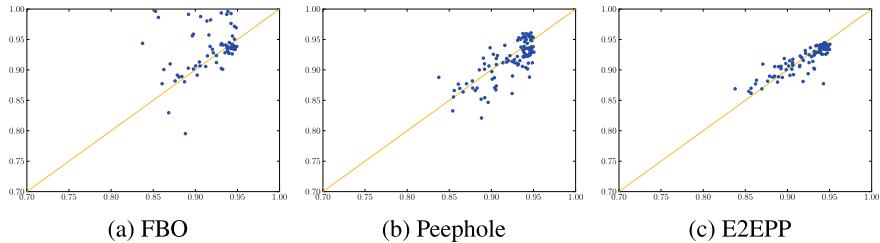


Fig. 14.3 The prediction and the true values of EBO, Peephole and E2EPP on the CIFAR-10 dataset. In each subfigure, the horizon axis denotes the true values, while the vertical axis denotes the corresponding prediction

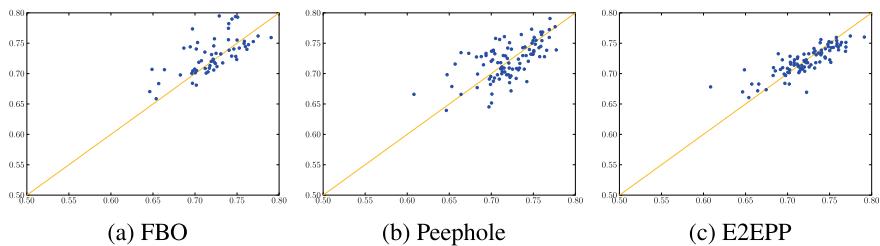


Fig. 14.4 The prediction and the true values of EBO, Peephole and E2EPP on the CIFAR-100 dataset. In each subfigure, the horizon axis denotes the true values, while the vertical axis denotes the corresponding prediction

more closely to $y = x$. This confirms the findings of the quantitative analysis shown in Table 14.4.

14.5.4 Comparison to Radial Basis Network

The comparison to RBN is carried out in order to emphasize the potentially performance of the random forest that is utilized by E2EPP; the results are displayed and analyzed in this section of the chapter.

The RBN is a three-layer network characterized by the use of a Gaussian function as the activation function and is commonly used in regression tasks. The RBN is selected for comparison in this section because: (1) its role is similar to that of the random forest; (2) it is the backbone of existing work [8] and provides us with ideas for designing E2EPP. For a fair comparison, the experimental setup is the same as described in Sect. 14.5.1, with only the random forest changed to RBN, and in addition, the changed algorithm is referred to as AE-CNN+RBN for distinction.

The results are shown in Table 14.5, where the accuracy information is shown without GPU Days, because both algorithms do not rely on GPUs during the performing process. The GPU Days required for the preliminary network training process can

Table 14.5 The comparison of AE-CNN+E2EPP and AE-CNN+RBN in terms of the classification accuracy (%) on the CIFAR-10 and CIFAR-100 benchmark datasets

	CIFAR-10	CIFAR-100
AE-CNN + E2EPP	94.70	77.98
AE-CNN + RBN	82.33	70.20

be viewed in Table 14.2. Furthermore, based on practice, the number of hidden units of the RBN is adjusted to be the same as the input size.

As shown in Table 14.5, AE-CNN+E2EPP is about 12 and 7.7% higher than AE-CNN+RBN on CIFAR-10 and CIFAR-100, respectively, which also indicates the superiority of using random forest in E2EPP. In fact, since the activation function of RBN uses Gaussian distribution, its mean and standard deviation are obtained from the training data, while for E2EPP, the training data is encoded with certain numerical numbers that do not correspond to the true meaning. On the other hand, RBN is known for dealing with continuous functions, while E2EPP is for discrete data. In conclusion, AE-CNN+E2EPP shows better performance than AE-CNN+RBN on both CIFAR-10 and CIFAR-100 benchmark datasets.

As mentioned above, four groups of experiments were conducted to verify the performance of E2EPP. The first is to test whether E2EPP can help ENAS achieve the desired performance in creating CNN architectures. The second is whether E2EPP acts as an acceleration performance. The third is to dig deeper into E2EPP from both qualitative and quantitative perspectives, and the fourth is to test whether the random forest used in E2EPP can outperform another widely used regression model, RBN. All but the third of the four groups of experiments use AE-CNN in conjunction with it. All experimental results can prove that E2EPP is effective and has excellent performance in accelerating ENAS. In addition, based on the end-to-end nature, E2EPP is able to minimize manual intervention. On the other hand, the random forest method used in the algorithm is able to achieve high performance relying only on a small number of samples, which provides a strong support for ENAS methods where it is difficult to obtain enough training samples. Furthermore, the random forest might take the discrete number as its input directly, which is another strong feature for performance prediction of CNNs when architecture information is employed as input data. This means that the ensemble approach relying on discrete data may be applicable not only to ENAS, but also to other deep learning model selection algorithms.

14.6 Chapter Summary

In this chapter, we introduced the E2EPP algorithm, which could greatly reduce the heavy cost of NAS algorithms. In the traditional NAS algorithms, a large number DNNs are trained for obtaining their performance values, and then the corresponding search strategies can be effectively proceeded based on the values. However, this will

inevitably incur a prohibitive computation cost. E2EPP treated this training process as a learning process, where the DNN architectures trained in advance serves as the training data, and then a random forest model is used to learn the relation between the architectures and their respective performance values. In the process of NAS, this trained random forest was used directly to predict the performance of DNN architectures, instead of being trained, thus the computation budget is saved. For a case study, E2EPP is verified with AE-CNN, and the experimental results showed that E2EPP can save half of the computation cost compared with AE-CNN.

DNNs are often powerful, as can be evidenced by their various real-world applications. However, the DNNs are also often with a large number of parameters, which requires their running or training highly depend on intensive computation resources. Some literature has shown that not all the parameters of the DNNs play the same role, and some parameters can be disabled. To this end, pruning the unnecessary weights of DNNs is an important research topic. In the next chapter, we will introduce Deep-PruningES in this regard.

References

1. Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2), 61–70.
2. Jeong, S., Murayama, M., & Yamamoto, K. (2005). Efficient optimization design method using kriging model. *Journal of Aircraft*, 42(2), 413–420.
3. Wang, H., Jin, Y., & Doherty, J. (2017). Committee-based active learning for surrogate-assisted particle swarm optimization of expensive problems. *IEEE Transactions on Cybernetics*, 47(9), 2664–2677.
4. Jin, Y., Wang, H., Chugh, T., Guo, D., & Miettinen, K. (2018). Data-driven evolutionary optimization: An overview and case studies. *IEEE Transactions on Evolutionary Computation*. <https://doi.org/10.1109/TEVC.2018.2869001>.
5. Zhou, Z., Ong, Y. S., Nair, P. B., Keane, A. J., & Lum, K. Y. (2007). Combining global and local surrogate models to accelerate evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(1), 66–76.
6. Wang, H., Jin, Y., & Janson, J. O. (2016). Data-driven surrogate-assisted multi-objective evolutionary optimization of a trauma system. *IEEE Transactions on Evolutionary Computation*, 20(6), 939–952.
7. Chugh, T., Chakraborti, N., Sindhya, K., & Jin, Y. (2017). A data-driven surrogate-assisted evolutionary algorithm applied to a many-objective blast furnace optimization problem. *Materials and Manufacturing Processes*, 32, 1172–1178.
8. Wang, H., Jin, Y., Sun, C., & Doherty, J. (2018). Offline data-driven evolutionary optimization using selective surrogate ensembles. *IEEE Transactions on Evolutionary Computation*. <https://doi.org/10.1109/TEVC.2018.2834881>.
9. Ho, T. K. (1995). Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition* (Vol. 1, pp. 278–282). IEEE.
10. Liaw, A., Wiener, M., et al. (2002). Classification and regression by random forest. *R News*, 2(3), 18–22.
11. Barandiaran, I. (1998). The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8).
12. Swersky, K., Snoek, J., & Adams, R. P. (2014). Freeze-thaw bayesian optimization. [arXiv:1406.3896](https://arxiv.org/abs/1406.3896).

13. Deng, B., Yan, J., & Lin, D. (2017). Peephole: Predicting network performance before training. [arXiv:1712.03351](https://arxiv.org/abs/1712.03351).
14. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/neco.1997.9.8.1735>.
15. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
16. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
17. Martínez-Muñoz, G., & Suárez, A. (2010). Out-of-bag estimation of the optimal sample size in bagging. *Pattern Recognition*, 43(1), 143–152.
18. Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and Regression Trees*. CRC Press.
19. Park, J., & Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. *Neural computation*, 3(2), 246–257.
20. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
21. Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., & Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning* (pp. 1319–1327).
22. Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015b). Highway networks. In *Proceedings of the 2015 International Conference on Learning Representations Workshop*.
23. Lin, M., Chen, Q., & Yan, S. (2014). Network in network. In *Proceedings of the 2014 International Conference on Learning Representations*.
24. Larsson, G., Maire, M., & Shakhnarovich, G. (2016). Fractalnet: Ultra-deep neural networks without residuals. *The 5th International Conference on Learning Representations*. <https://openreview.net/forum?id=S1VaB4cex>.
25. Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for simplicity: The all convolutional net. In *Proceedings of the 2015 International Conference on Learning Representations*.
26. Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. [arXiv:1611.01578](https://arxiv.org/abs/1611.01578).
27. Baker, B., Gupta, O., Naik, N., & Raskar, R. (2016). Designing neural network architectures using reinforcement learning. [arXiv:1611.02167](https://arxiv.org/abs/1611.02167).
28. Zhong, Z., Yan, J., & Liu, C.-L. (2018). Practical network blocks design with q-learning. In *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*.
29. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., & Kurakin, A. (2017). *Large-scale evolution of image classifiers* (pp. 2902–2911).
30. Xie, L., & Yuille, A. L. (2017). Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017* (pp. 1388–1397). <https://doi.org/10.1109/ICCV.2017.154>.
31. Liu, H., Simonyan, K., Vinyals, O., Fernand C., Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. [arXiv:1711.00436](https://arxiv.org/abs/1711.00436).
32. Suganuma, M., Shirakawa, S., & Nagao, T. (2018). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 5369–5373). <https://doi.org/10.24963/ijcai.2018/755>.
33. Domhan, T., Springenberg, J. T., & Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Vol. 15, pp. 3460–8).
34. Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (pp. 249–256).

35. Sen, P. K. (1968). Estimates of the regression coefficient based on kendall's tau. *Journal of the American Statistical Association*, 63(324), 1379–1389.
36. Ozer, D. J. (1985). Correlation and the coefficient of determination. *Psychological Bulletin*, 97(2), 307.

Chapter 15

Deep Neural Architecture Pruning



15.1 Introduction

The majority of NAS algorithms are intended to identify the optimal CNN architectures for a specific task. CNNs utilized for image classification and recognition problems demand strong hardware, such as data centers as well as GPU workstations, for training and inference. As a consequence, many models are incompatible with consumer hardware like portable appliances and smartphones. Nowadays, if a developer wants to execute computer vision problems on these devices, they must have a constant internet connection, which may not always be available or dependable enough to perform such tasks effortlessly for users. As a result, it is extremely desired to develop algorithms and techniques that lower the complexity of computing in CNN models.

Researchers in machine learning are increasingly interested in lowering the complexity of computing in CNNs while retaining their performance. To achieve this goal, there are three main methods: (1) taking into consideration the restrictions of the intended platform when creating CNN systems from the ground up; (2) utilizing dedicated mobile hardware that may efficiently run CNNs; and (3) modifying unnecessary parameters from current CNN architectures. The next sections go over each of these strategies in depth.

In the first method, innovative CNN architectures and operations may be designed to make use of the distinctive features of embedded and mobile systems. MobileNets [1] and CondenseNets [2], for example, are two customized CNN architectures designed for usage on mobile systems like smartphones. Both architectures utilize specific convolutional operations, such as group and depth-wise convolutions, to reduce the complexity of computing while maintaining efficiency. NAS may also be utilized to locate CNN architectures that are appropriate for embedded and mobile devices. The differentiable architecture search method created by Liu et al. is an example of a NAS that can generate compact CNNs utilizing a continuous search space [3]. The NetAdapt algorithm, created by Yang et al., is another

notable example, which can generate acceptable CNN architectures given a specified resource budget. NetAdapt may locate CNN architectures that are suited to a specific device [4]. The FastDeepIoT algorithm, which was created by Yao et al., is another example, it creates a very accurate initial model and then compresses it by considering the execution duration in a certain embedded and mobile device [5].

The second method for satisfying the needs of computing in CNN models is to create embedded and mobile hardware with particular features. Jetson embedded platform developed by Nvidia is one of the most prominent instances, it has a GPU that can conduct real-time inference utilizing complex CNN models [6, 7]. Another hot topic is the utilize of field-programmable gate array (FPGA) for CNN model inference. For FPGA systems, several researchers have changed deep learning libraries and frameworks such that inference speeds are equivalent to those of GPUs [8, 9]. The usage of such platforms, on the other hand, does not answer the question of how to minimize the complexity of computing in CNN models. In this regard, Ding et al. created an approach for reducing the complexity of computing in CNN operations by implementing them in FPGAs using block-circulant weight matrices [10]. The weight matrices of each layer are transformed into block-circulant matrices and fast Fourier transform multiplications are used to do inference, as a result, the complexity of computing and use of memory will be lowered.

The third method for lowering the complexity of computing in CNN models is pruning. In this situation, it is hypothesized that several human-created CNN models have an excessive number of superfluous parameters and that identifying and eliminating these parameters may decrease model complexity without affecting performance. Using L_2 regularization [11], Ding et al. created a approach called Auto-balanced Filter Pruning, in which whole convolutional filters¹ may be deleted from several layers according to their significance. Strong regularization is required for the first CNN model, since it must be trained using both weak and strong filters that are regularized based on the positive and negative factors, respectively. This first training step makes it simpler to determine which filters should be eliminated during the pruning process, but it also necessitates retraining the original CNN model, which may take a considerable amount of time. In addition, Luo et al. [12] pruned CNN models via deleting complete filters from many layers. ThiNet is their approach, and instead of pruning information from the current layer, it uses statistics from the next layer. By calculating the total of the absolute values of the filters in a particular layer and excluding those with the least sum [13], Li et al. developed a method that eliminates complete filters. EAs have also been used to prune the data. Two recent instances of EAs being utilized to prune CNN models with competing performance are Zhou et al. [14] and Wang et al. [15].

Other methods of dealing with the high computing needs of CNN models are also worth discussing. Weight quantization, for example, can be used to lower the amount of bits needed to encode the weights of network [16]. All weights are indicated by

¹ The trainable weights of a convolutional layer are referred to as convolutional convolutional filters or filters in recent work, whereas the output tensor of a convolutional layer is referred to as feature maps.

one bit in binarized networks, which is an extreme form of weight quantization [17, 18]. The other method is to employ ResNets [19] to create CNN models that calculate the residual mapping of a function, commonly known as a skip connection or a shortcut. Residual mapping avoids the vanishing gradient problem when creating models with hundreds or thousands of layers. Although residual mapping does not directly lower the computational cost of a CNN, it does aid in reaching better performance by allowing smaller filter sizes in convolutional layers. Instead of utilizing the output of just one layer like ResNet, DenseNets [20] employ the output of all preceding levels as the input to the present one as well as use shortcut connections from preceding layers. To put it another way, all subsequent convolutional layers in a DenseNet are completely connected. The utilization of many shortcut connections between layers, similar to ResNets, does not reduce the complexity of computing in the network, nonetheless, it does allow for the usage of smaller networks with improved performance compared to CNNs without shortcut connections.

When searching for models with lower computational complexity, the effort done to build CNN models with the highest potential performance ought to not be overlooked. In this chapter, an innovative pruning strategy for eliminating unneeded parameters from high-performing models is provided. To lower the complexity of computing in the model, removing unnecessary filters from convolutional layers are recommended. Since it removes the demand for special hardware to run these models, this method is considered the most suited for most applications. It also explains the overall architecture of CNNs, as well as how most models may be built with less parameters. As a result of this, pruning a CNN model may be thought of as a combinatorial optimization. The goal is to discover the best combination of convolutional filters in various layers of a given model. Also expressed as a MOP with two competing objectives: (1) to the greatest extent feasible, minimize the amount of parameters in a model while (2) ensuring that the accuracy of the model is as high as feasible. In addition, it is hoped that the model can be pruned without previous knowledge of them, so that several CNN can be pruned with a single algorithm. For this case, using EC algorithms is appropriate. These algorithms are appropriate for combinatorial optimization with conflicting goals. In particular, the evolution strategy (ES) [21] is a random parameter perturbation algorithm that evolves a population of candidate solutions.

The major goal of this chapter is to suggest using the ES to prune CNN models, which refers to as DeepPruningES. The reason for us to select ES is due to its ease of use along with its capacity to search for suitable solutions. Note that the reasonable answers are found rather than optimum ones. As a result, when pruning CNN models, solutions that include a trade-off between performance and the complexity of computing can be only identified. The optimal solutions are found on the Pareto frontier, where no single solution is clearly superior to the others. Identifying the real Pareto frontier for a particular model pruning task, on the other hand, is not an easy process. As a result, listed below are contributions of DeepPruningES:

- Pruning CNN architectures is seen as a two-objective optimization task that is tackled utilizing ES, with the complexity of computing and training error as competing

goals. In addition, the algorithm conducts progressive pruning of a selected CNN model, in which CNN model filters are gradually eliminated over generations.

- The algorithm removes the necessity to determine the true Pareto frontier by providing three solutions with distinct trade-offs at the conclusion of the pruning phase. A decision-maker (DM) may select the best options that match his/her demands in this regard. The following are some of the solutions:
 - The knee solution is the candidate solution has the optimal trade-off between the complexity of computing and training error;
 - The boundary light solution is the candidate solution has the minimum complexity of computing;
 - The boundary heavy solution is the candidate solution has the minimum training error in the population.
- A changed ES selection operator according to the minimum Manhattan distance (MMD) method [22] is designed to select these three solutions from a population of candidate solutions. Using this method, the algorithm does not need the usage of any trade-off parameters.

As detailed below, the algorithm differs from prior pruning algorithms that used evolutionary approaches:

- Wang et al. [15] designed an EA that resembles the DeepPruningES at first glance. The algorithm in [15], on the other hand, does not use a multi-criteria decision making (MCDM) framework, no matter how accurate the classification is, it can only come up with one solution with the low complexity of computing, while the algorithm may find three possible solutions with various trade-offs in classification performance and the complexity of computing. Unlike in [15], the DM does not have to put up any trade-off parameters in the algorithm. The pruning algorithm is a multi-objective optimization algorithm that simultaneously optimizes two competing objective functions, whereas the algorithm provided in [15] only optimizes one. Furthermore, the authors in [15] do not specify whether their algorithm prunes the whole layers of ResNet or only the layers among shortcut connections. In contrast, the approach given in this chapter may prune the whole layers of DenseNets and ResNet by utilizing two binary strings to encode their filters. As a result, it is possible to prune not just the intermediate layers of a ResNet or DenseNet model, but also the whole model.
- Zhou et al. [14] created the knee-guided EA (KGEA) for compressing DNNs appears to be comparable to the algorithm at first glance. DeepPruningES differs from KGEA in that it may prune numerous CNN topologies including DenseNets, ResNets, and VGGs, whereas KGEA may just prune basic CNN models like VGG networks [23]. Another difference is that KGEA uses a mechanism to maintain a diversified Pareto front, while the goal of DeepPruningES is to identify great candidate solutions as quickly as feasible. As a result, during the pruning phase, DeepPruningES does not need the maintenance of a diversified Pareto front. Moreover, because only maintaining three potential solutions are considered, an

ES framework, such as in [14], is more appropriate than a GA framework, which results in a population with fewer individuals than other evolutionary methods.

The algorithm may prune DenseNets, ResNets, and CNNs, which no other algorithm in the literature can do. In general, it can reduce the amount of FLOPs by up to 75% while maintaining the classification performance of the model.

The remainder of this work is divided into the following sections. Section 15.2 provides a detailed overview on DNNs models just like DenseNet, ResNet, and CNN, as well as convolutional filter pruning. Section 15.3 presents the algorithm. Section 15.4 discusses the experimental design utilized to verifying the algorithm. In Sect. 15.5, the experimental results are discussed.

15.2 Background

A four-dimensional tensor K with dimensions equal to $(O \times D \times W \times H)$ may be used to describe the weights of a convolutional layer, commonly known as kernels or filters, where O is the whole amount of kernels, D represents the depth of each kernel, W represents its width, and H represents its height. If necessary, every O kernel may contain one parameter to describe the bias of the O kernels. One way to lower the cost of computing in one convolutional layer is to remove some of these O kernels entirely. Filter pruning is one of the most prevalent approaches for reducing the complexity of computing in CNN models. This procedure is depicted in Figs. 15.1 and 15.2, where a filter is selected to be removed first, as illustrated by the dashed lines in Fig. 15.1, and then the complete CNN model is changed, resulting in a pruned model, as shown in Fig. 15.2. Notably, removing filters from layer i requires also eliminating the relevant depths of all filters in layer $i + 1$, if a suitable architecture is need to be achieved.

There are numerous approaches to choose which filters to discard in each convolutional layer in the literature, a process called filter selection. To identify which filters to remove, a number of statistics from the present layer are necessary. Some of the most commonly utilized decision criteria in the literature [24] are as follows:

- Mean activation: Molchanov et al. in [25] used it to compute the average activation of each filter in a particular layer and keep only those filters with the greatest average activation.
- $l1$ -norm: Li et al. utilized it in [13], where the $l1$ -norm determines the importance of a filter. Filters with a smaller $l1$ -norm are less important than those with a large $l1$ -norm.
- Average percentage of zeros (APoZ): Hu et al. suggested it in [26], and it removes filters that have a large number of zero activations throughout inference time.
- Entropy: Wu and Luo suggested it in [27]. In this scenario, the entropy of one filter is determined by feeding input images through the network and sorting the outputs into various bins. The basic assumption is that irrelevant filters will yield extremely similar outputs when given varied inputs.

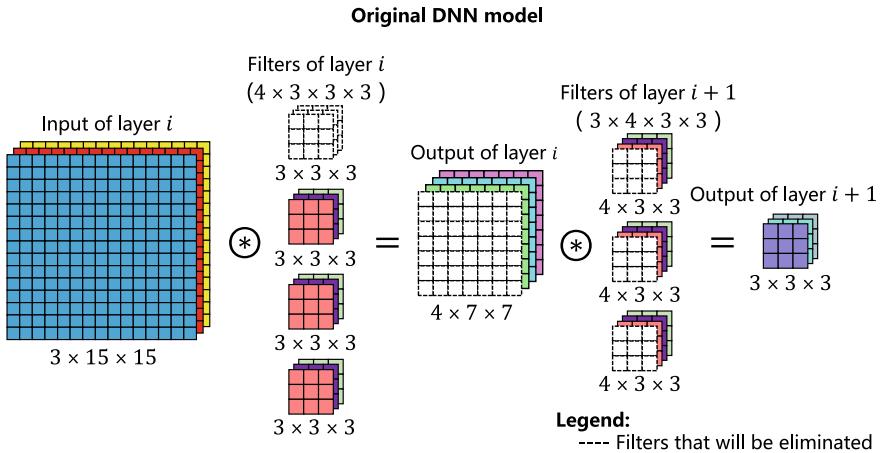


Fig. 15.1 Example of convolutional filter pruning where the first filter of the current convolutional layer is chosen to be eliminated

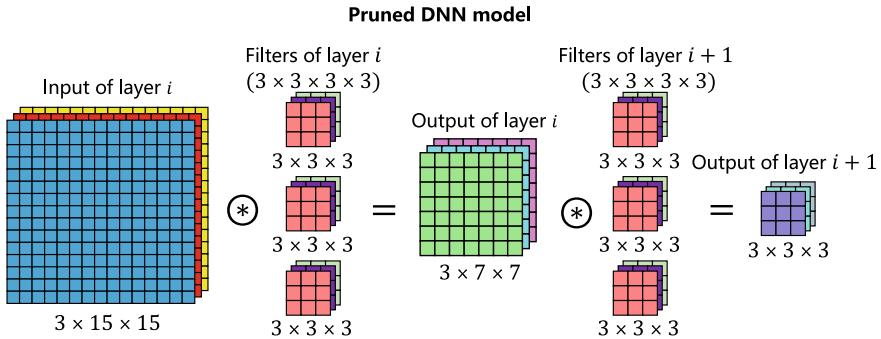


Fig. 15.2 Pruned CNN model after the first filter is removed

- Random: Without previous knowledge of the CNN architecture or filters, it chooses filters to be removed at random. Mittal et al. indicated in [24] that randomly removing filters yielded equivalent results to other selection criteria.

15.3 Algorithm Details

Algorithm 1 depicts the general framework of DeepPruningES. It primarily consists of the regular ES plus version that has been modified to do CNN pruning. gen represents the max amount of generations that the algorithm will run, λ_{size} represents the size of the offspring population, e_{eval} represents the amount of epochs to be utilized for an individual evaluation, $model$ represents the original CNN model that

Algorithm 1: DeepPruningES

Input : gen represents the max amount of generations that the algorithm will run, λ_{size} represents the size of the offspring population, e_{eval} represents the amount of epochs to be utilized for an individual evaluation, $model$ represents the original CNN model that will be pruned, α_{eval} represents the learning rate to be utilized in an individual evaluation, α_{fine} represents the learning rate and e_{fine} represents the amount of epochs.

Output: boundary light solution ($\mu.light$), boundary heavy solution ($\mu.heavy$) and knee solution ($\mu.knee$) which are three CNN models.

```

1  $\mu, \lambda \leftarrow Create\ An\ Initial\ Population(\lambda_{size}, model);$ 
2 for  $g = 1$  to  $gen$  do
3    $\mathbf{P} \leftarrow \mu + \lambda$ 
4    $\mu \leftarrow Selection\ of\ Knee\ Boundary(\mathbf{P}, model, e_{eval}, \alpha_{eval});$ 
5    $\lambda \leftarrow Generation\ of\ Offspring(\mu, \lambda_{size}, p_m, model);$ 
6 end
7  $Fine-tuning(\mu, model, e_{fine}, \alpha_{fine});$ 
8 return  $\mu.light, \mu.heavy, \mu.knee;$ 
```

will be pruned, α_{eval} represents the learning rate to be utilized in an individual evaluation, α_{fine} represents the learning rate and e_{fine} represents the amount of epochs are all inputs. DeepPruningES will produce three CNN models using different trade-offs, referred to as boundary light, boundary heavy, and knee solutions.

Algorithm 1 has four major components: Create An Initial Population (line 1), Selection of Knee Boundary (line 4), Generation of Offspring (line 5), and Fine-Tuning of the best solutions found (line 7). In the next sections, each of them will be detailed , including the genetic representation of an individual.

15.3.1 Genetic Representation of an Individual

Representation is one of the most crucial parts of any population-based algorithm. In DeepPruningES, the filters of a CNN model are represented by two binary strings. It's worth noting that only convolutional layers are pruned by DeepPruningES, which incur much more computational cost than fully connected ones [13]. Each bit in this representation corresponds to one filter. For instance, suppose two convolutional layers are indicated, one using 64 filters and another using 32 filters, a 96-bit binary string with a bit set to zero indicating that the appropriate filter has been deleted is required.

The representation, however, is dependent on the kind of CNN model being pruned. Only one binary string is utilized for pruning basic CNN models, with every bit representing a CNN model filter, as shown in Figs. 15.3 and 15.4. In ResNet models, the filters are encoded using two binary strings: one for the first convolutional layer and another for the second convolutional layer of a residual block. The second

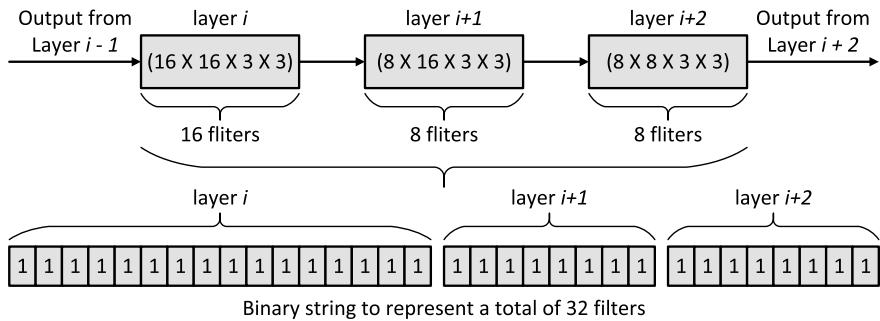


Fig. 15.3 Binary representation of a CNN model. Every layer is a convolutional layer

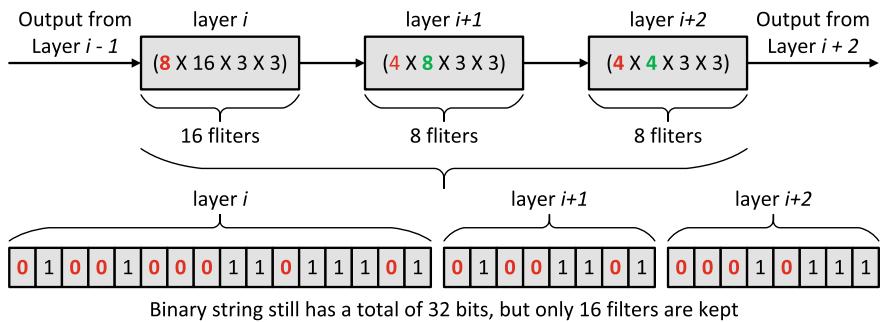


Fig. 15.4 After eliminating half of the filters from every layer, a CNN model is represented as binary information

layer of all residual blocks needs to be pruned simultaneously in order to maintain a constant tensor size throughout the model, since a tensor entering a residual block must be added with a tensor leaving the residual block. This process is depicted in Fig. 15.5, where one binary string encodes the first layers of every residual block on the left side and one binary string encodes the second layers of numerous residual blocks on the right side. It's worth noting that the second layers of blocks with the identical amount of filters, such as 16 filters, are encoded together. For instance, in Fig. 15.5, encoding the green layers (indicated by the bars on the right side of the box) requires just 16 bits, therefore removing four filters from the first green layer will necessitate removing four filters from all each subsequent green layer. Finally, DenseNet models require two binary strings in their binary representation: one for the bottleneck layers and another for the convolutional layers, according to Fig. 15.6. Since DenseNets employ concatenation rather than addition, every layer in the model can be pruned individually, making them easier to express than ResNets.

Fig. 15.5 Binary representation of a ResNet which has 20 layers

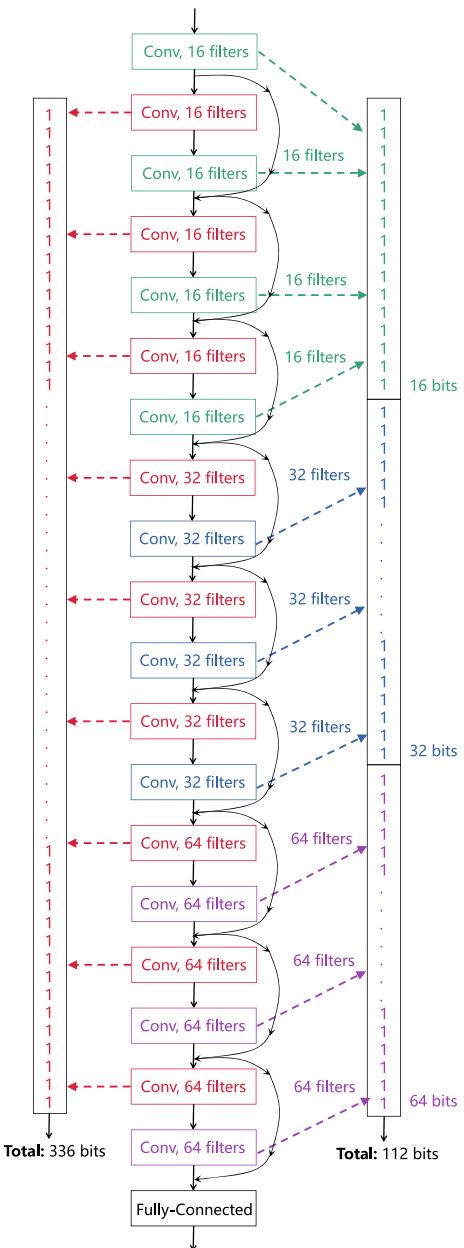
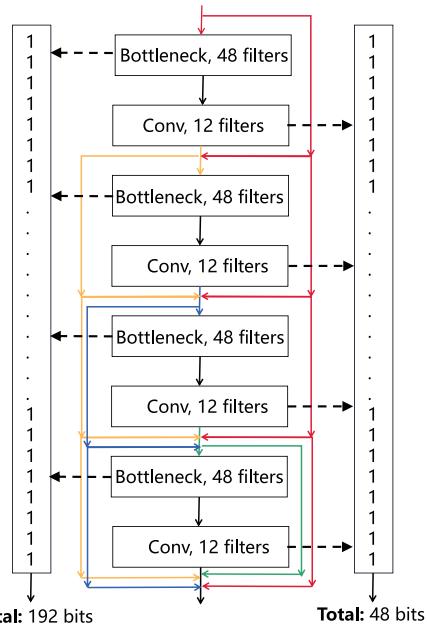


Fig. 15.6 DenseNet employs a binary encoding to represent a single dense block



15.3.2 Population Initialization

DeepPruningES begins with the initialization of a population of $3 + \lambda_{size}$ individuals. To begin, each individual is started as the same duplicate of the CNN model being pruned. In addition, with probability p_m , to create a pruned version of the original model, their genes are mutated. At last, the major for-loop can be executed, as illustrated in Algorithm 1, line 2.

The main advantage of using this method over a fully initialized randomly is that the population will be similar to that of the original CNN model, enabling us to get pretty decent results even with fewer generations and smaller populations.

15.3.3 Individual Evaluation

It is necessary to clarify how the algorithm evaluates an individual before discussing how the knee and boundary solutions are chosen. To begin, this is accomplished by removing the filters specified in the gene of the individual, then fine-tuning the resultant pruned model utilizing SGD for the learning rate of α_{eval} and the epochs of e_{eval} in a tiny sample of the dataset utilized to train the original model. This permits the individual to recover a portion of their previous performance prior to the selection

operation. Section 15.4.1 contains more information on how the dataset is selected and how long it takes to fine-tune the pruned model.

Individuals are evaluated using both the training error in the selected dataset and the amount of FLOPs for one input. These are incompatible goals because a tiny training error cannot be maintained using a limited amount of floating-point operations. The amount of FLOPs are chosen rather than the entire amount of parameters or filters since the time it takes to calculate the outputs of every given layer is affected not only by the number of parameters but also by the size of its inputs. As a result, the total amount of FLOPs of a model is regarded as a better indicator of the complexity of computing in it.

15.3.4 Selection of Knee and Boundary

Numerous real-world issues require DM to simultaneously optimize numerous factors in order to achieve a desired behavior. MCDM issues are those in which no one desirable or ideal solution for a particular situation. Frequently, they are MOPs in which changing one parameter in a potential solution has an impact on the outputs of several other objective functions. In MCDM situations, nevertheless, not all potential solutions are attractive. Important is just the solutions at the so-called Pareto optimal front in the objective space. In all objective functions, this region contains solutions that are no worse than others. In the Pareto optimal front, the knee solution is always discovered. In MCDM situations, the capability to discover the knee solution is critical, since it is the solution that may improve any of the objectives with the most with minimal modifies in its parameters [22].

In the case of CNNs, the parameters are the amount of filters remaining in the architecture, as well as the objective functions which wish to improve are the accuracy in a certain dataset and the complexity of computing in CNN. Therefore, one model may be superior to another depending on the sort of equipment used to deploy the CNN.

Before the knee can be identified, all of the solutions in the Pareto optimal front need to be found. The shortest Manhattan distance from the origin point of the objective space point to the Pareto optimal front solution is the knee, according to Chiu et al. in [22]. Searching all of the Pareto optimal front, on the other hand, is a difficult process. As a result, to find the knee solution, the modification of the MMD algorithm established by Chiu et al. are suggested. Since there is no interest in locating the entire Pareto optimal front, the geometric position of all possible solutions in the objective space can be used to ease the searching for the knee solution. This is accomplished by calculating the Manhattan distances of the whole possible solutions and choosing the one that has the shortest distance.

As a result, Algorithm 2 shows the algorithm for performing Knee and Boundary Selection, in which three individuals, referred to as boundary light, boundary heavy, and knee solutions, are usually chosen at each generation. To begin, \mathbf{P} must be determined, which indicates the individuals in the population, and determine f_2 and

Algorithm 2: Knee and Boundary Selection

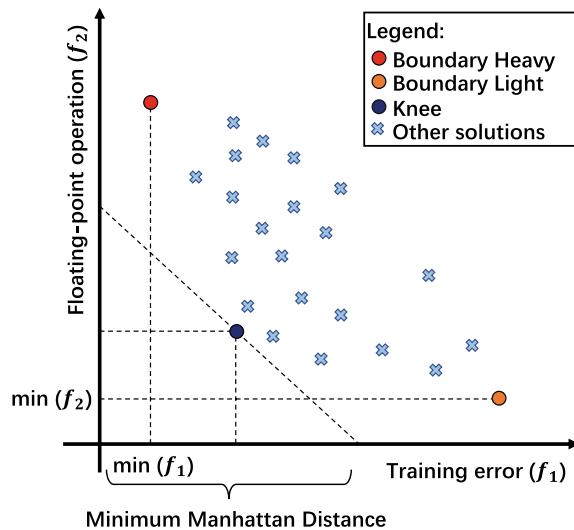
Input : $model$ represents the original CNN model, \mathbf{P} represents the individuals in the population, α_{eval} represents the learning rate of individual evaluation, e_{eval} represents the amount of epochs for individual evaluation.

Output: Boundary light solution ($\mu.light$), boundary heavy solution ($\mu.heavy$) and knee solution ($\mu.knee$) which are three individuals.

```

1 Evaluate Population( $\mathbf{P}$ ,  $model$ ,  $e_{eval}$ ,  $\alpha_{eval}$ );
2 Find  $\min(f_1)$ ,  $\min(f_2)$ ,  $\max(f_1)$ ,  $\max(f_2)$ ;
3  $\mu.heavy \leftarrow P_i$ , where  $f_1(P_i) = \min(f_1)$ ;
4  $\mu.light \leftarrow P_j$ , where  $f_2(P_j) = \min(f_2)$ ;
5 for  $k = 1$  to  $\text{len}(\mathbf{P})$  do
6    $| dist(k) = \frac{f_1(P_k) - \min(f_1)}{\max(f_1) - \min(f_1)} + \frac{f_2(P_k) - \min(f_2)}{\max(f_2) - \min(f_2)}$ ;
7 end
8  $\mu.knee \leftarrow P_k$ , where  $P_k$  has the min  $dist(k)$ ;
9 return  $\mu.light$ ,  $\mu.heavy$ ,  $\mu.knee$ ;
```

Fig. 15.7 Example of the boundary light, boundary heavy, and knee solutions



f_1 , which indicates their FLOPs and training error. In addition, the min and max values of population for both objectives must be evaluated (Algorithm 2, line 2). In the population, the individual with the minimum training error (that is the individual with f_1 equal to $\min(f_1)$) will be the boundary heavy solution. While the individual with the min FLOPs in the population, whose f_2 is equal to $\min(f_2)$, will be the boundary light solution. At last, utilizing the equation from Algorithm 2 (line 6), the Manhattan distance for each individual in the population is calculated and choose the individual which has the MMD as knee solution. Figure 15.7 depicts the process of selection, at the conclusion of the operation only these three individuals are returned.

15.3.5 Offspring Generation

The algorithm will use the three individuals from the knee and borders to generate new offspring when they have been chosen. To begin, one of these three individuals is chosen at random to be the parent of one offspring. In addition, with a probability of p_m , each of its gene elements is flipped at random. This operation is repeated until the population of offspring equals λ_{size} . It's also worth noting that the standard ES algorithm is used to generate a new offspring population, which only uses the knee and boundary solutions. As a result, a new production offspring is never utilized to create further offspring. This approach guarantees that new offspring are always created nearer to the boundary or knee solutions, enabling the algorithm to utilize the regions of interest effectively.

15.3.6 Fine Tuning

In the last stage of the algorithm, the knee and boundary solutions from the previous generation are fine-tuned. SGD is used to refine them, with a learning rate of α_{fine} during e_{fine} epochs. The fine-tuning process, unlike the Individual Evaluation, makes use of the full training set to enhance the three solutions as much as feasible. Fine-tuning uses a larger number of epochs than individual evaluation. The three solutions are stored on disk for later utilized at the end of this operation.

15.4 Experimental Design

The experimental design utilized to evaluate DeepPruningES is detailed in this section. To begin, the CNN architectures experimented with this work are presented. In addition, the algorithm settings that are utilized to prune such CNN architectures are then analyzed. Finally, all of the results in this chapter are generated on a laptop computer which has a NVIDIA GTX 1060 6GB GPU, 16 GB of RAM, and Core i7-8750H CPU running PyTorch on Windows 10 Pro 64-bits.

15.4.1 Chosen CNN Architectures for Pruning

DeepPruningES is evaluated using state-of-the-art CNN architectures. As mentioned in the introduction of this work, the algorithm is capable of pruning three distinct CNN architectures: DenseNets, ResNet, and CNN. Given a difficult dataset, and putting the algorithm to the test using CNNs from these architectures.

Table 15.1 Overview of the CNN architectures utilized to evaluate DeepPruningES

CNN	# Layers	# FLOPs	Test error on CIFAR-10 (%)	Test error on CIFAR-100 (%)
VGG16	16	3.15×10^8	6.06	32.33
VGG19	19	4.01×10^8	6.18	32.57
ResNet56	56	1.27×10^8	6.63	41.94
ResNet110	110	2.57×10^8	6.2	50.47
DenseNet50	50	0.93×10^8	6.92	41.25
DenseNet100	100	3.05×10^8	5.66	33.06

The CIFAR-10 and CIFAR-100 datasets established by Krizhevsky [28] are selected to be utilized in the experiments since apart from being a difficult dataset, there are numerous state-of-the-art CNNs results have been extensively reported, enabling us to remain focused on quantifying the quality of the pruning task.

The following networks are used to evaluate and compare with the algorithm: VGG19, VGG16, ResNet110, ResNet56, DenseNet100, and DenseNet50. The VGG19 and VGG16 [23] are two CNNs developed in 2014 that performed well in difficult image classification datasets including ImageNet and CIFAR-10. According to the name, VGG16 contains 16 layers, whereas VGG19 has 19 layers. The computational complexities of the VGG16 and VGG19 are 3.15×10^8 and 4.01×10^8 FLOPs, respectively. On the CIFAR-10 dataset, their original and fully trained architectures have test errors of 6.06% and 6.18%, respectively, while on the CIFAR-100 dataset, they have test errors of 32.33% and 32.57%, respectively. The authors in [19] first proposed ResNets such as ResNet56 and ResNet110 [19] and utilizing them in the CIFAR-10 dataset. Their complexity of computing are 1.27×10^8 and 2.57×10^8 FLOPs, accordingly. ResNet56 and ResNet110, when completely trained, can obtain test errors of 6.63 and 6.2% on the CIFAR-10 dataset, accordingly, and 41.94 and 50.47% on the CIFAR-100 dataset, accordingly. There are 56 layers in ResNet56, and 110 layers in ResNet110. Finally, the authors of DenseNet have proposed the DenseNet50 and DenseNet100 [20] architectures for classifying the CIFAR-10 dataset. These two versions of DenseNet contain 50 and 100 layers, accordingly, with the FLOPs of 3.05×10^8 and the complexity of computing of 0.93×10^8 , accordingly. Meanwhile, on the CIFAR-10 dataset, the test error is 6.92 and 5.66%, as well as 41.25 and 33.06% on the CIFAR-100 dataset. Table 15.1 summarizes the CNN architectures utilized to evaluate the method.

15.4.2 Algorithm Parameters

Table 15.2 shows the values of the parameters utilized to evaluate DeepPruningES. The parameters from Table 15.2 are used to get all of the results in this study that

Table 15.2 Parameters utilized to evaluate DeepPruningES

Parameter	Value
λ_{size} represents the offspring size	20
gen represents the maximum number of generations	10
p_m represents the mutation probability	0.1
e_{eval} represents the amount of epochs for individual evaluation	5
α_{eval} represents the learning rate for individual evaluation	0.1
e_{fine} represents the amount of epochs for fine-tuning	50
α_{fine} represents the learning rate for fine-tuning	0.01

are relevant to the algorithm. If some results are achieved with a different set of parameters, the differences are discussed. The impact of every parameter on the last output of DeepPruningES will be discussed in the following paragraphs.

The number of offspring generated each generation is determined by the size of the offspring (λ_{size}). More offspring allows the algorithm to cover a larger portion of the objective space, increasing the possibility of obtaining excellent solutions. Similarly, having a high amount of generations (gen) enhances the chances of obtaining excellent solutions. Nevertheless, when the number of offspring and/or generations grows, so does the expense of computing in executing the algorithm. In the experiments, 20 offspring and 10 generations are shown to be sufficient for achieving acceptable computing complexity.

Mutation probability (p_m) is the possibility that a single bit of the gene of an individual will become inverted. A low mutation probability would need too much time to produce major population changes, needing more individuals and generations to accomplish the desired result, whereas a high mutation probability would result in an excessive number of population changes, leading in almost no improvement. In the experiments, a probability of 0.1 is considered the optimal value heuristically that leads individuals to modify in a way that increases the overall quality of the population.

During an individual evaluation, e_{eval} and α_{eval} represent the amount of epochs and learning rate, respectively, which are used to determine how long an individual will be fine-tuned before the amount of FLOPs and training error are considered. The values for these two parameters are determined by the computing capability of the user. If an individual may be fine-tuned for a longer period of time, the overall fitness of the individual will be better sense. In order to be compatible with other EC-based algorithms used for CNN architecture generation and pruning [12, 27, 29], the learning rate of 0.1 and epochs of 5 are used. In this case, a high learning rate is used to increase the accuracy of the solutions more quickly.

Finally, fine-tuning the final solutions (e_{fine} , α_{fine}) using the amount of epochs and learning rate will reduce the training error of the three final solutions. They will be fine-tine by using epochs of 50 and a learning rate of 0.01, it is consistent with the work of others [11, 13]. In order to minimize training instability caused by excessively deleted filters, a lower learning rate than during an individual evaluation will be used in this case. If the solutions of algorithm are fine-tuned for more than 50 epochs, they can be improved even more. However, due to the restricted processing capability, it's now unable to conduct extensive fine-tuning sessions.

As previously stated, DeepPruningES does CNN filter pruning using the ES plus version, which is an elitist version of ES in which the best individuals are never lost. An experiment utilizing the ES comma version is provided in Sect. 15.5, and since the algorithm eliminates too many filters, even from the boundary heavy and knee solutions, producing poor results.

15.5 Experimental Results and Discussion

In this part, pruning results are shown first, followed by a comparison to those of peer competitors. After that, the generated results are discussed in detail.

15.5.1 Experimental Results

The CNN architectures shown in Table 15.1 are pruned 10 times to evaluate DeepPruningES. Table 15.3 shows the results using the CIFAR-10 dataset, where the best and mean test error as well as the amount of FLOPs for the knee, boundary heavy, and boundary light solutions after fine-tuning are presented for 50 epochs for each CNN architecture. Table 15.4 shows the results using the CIFAR-100 dataset, with the three selected solutions fine-tuned for 100 epochs. In addition, the results of peer competitors who used the same CIFAR-10 dataset are presented in Table 15.5. Other works in the literature are not included since they used the ImageNet dataset, which is too large for the equipment to handle and does not provide comparable results with other datasets.

The pruning algorithm achieves similar results on the CIFAR-10 dataset for the VGG16 and VGG19. The number of FLOPs is reduced by an average of 20% in boundary heavy solutions, 70% in boundary light solutions, and 57% in knee solutions. The mean test errors for the knee solutions, boundary light, and boundary heavy, are roughly 9.58% ~ 9.87%, 11.51% ~ 12.01% and 8.6% ~ 8.77%, respectively. These are the expected results, with the boundary heavy solutions having the least pruning and the maximum accuracy. The boundary light solutions, on the other hand, were the most pruned and had the lowest accuracy. Lastly, the knee solutions obtained pruned results and accuracy in the middle.

Table 15.3 On the CIFAR-10 dataset, pruning results are generated using DeepPruningES

CNN 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	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	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	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	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	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

The boundary heavy solutions reduce the quantity of FLOPs on the CIFAR-10 dataset by an average of 15% for the ResNet56 and ResNet110, an average of 77% reduction of the boundary light solutions and an average of 59% reduction of the knee solutions. The knee solutions, boundary light, and boundary heavy had test errors of 9.42% ~ 9.98%, 12.9% ~ 13.46%, and 7.93% ~ 8.77%, accordingly, after fine-tuning. ResNets are smaller by nature than traditional CNNs like VGG16 and VGG19, but these results indicate that they may still be pruned without sacrificing too much accuracy.

The average reduction in the number of FLOPs for the knee solutions, boundary light, and boundary heavy on the CIFAR-10 dataset is roughly 50%, 16%, and 73%, accordingly, for the DenseNet50 and DenseNet100. The average test errors for the knee solutions, boundary light, and boundary heavy, on the other hand, are

Table 15.4 On the CIFAR-100 dataset, pruning results are generated using DeepPruningES

CNN Model	DeepPruningES						
	Solution	Test error (best) (%)	Test error (mean) (%)	# FLOPs (best)	# FLOPs (mean)	% Pruned (best) (%)	% Pruned (mean) (%)
VGG16	Knee	34.27	34.41	1.289×10^8	1.411×10^8	59.19	55.37
	Boundary heavy	32.94	33.08	2.531×10^8	2.577×10^8	19.93	18.47
	Boundary light	35.68	36.36	6.786×10^7	7.015×10^7	78.53	77.81
VGG19	Knee	34.6	35.09	1.652×10^8	1.882×10^8	58.82	53.08
	Boundary heavy	33.11	33.41	3.246×10^8	3.277×10^8	19.09	18.31
	Boundary Light	37.05	37.41	8.640×10^7	8.970×10^7	78.46	77.64
ResNet56	Knee	53.98	55.57	1.484×10^7	1.681×10^7	88.40	86.86
	Boundary heavy	42.19	42.49	1.072×10^8	1.096×10^8	16.19	14.26
	Boundary light	53.89	56.70	1.421×10^7	1.587×10^7	88.89	87.61
ResNet110	Knee	59.06	64.09	3.068×10^7	3.633×10^7	88.08	85.89
	Boundary heavy	50.97	51.49	2.117×10^8	2.175×10^8	17.73	15.63
	Boundary light	62.07	64.93	3.069×10^7	3.365×10^7	88.07	86.93
DenseNet50	Knee	42.09	62.01	1.606×10^7	2.643×10^7	82.82	71.73
	Boundary heavy	41.59	42.02	7.569×10^7	7.815×10^7	19.05	16.42
	Boundary Light	63.44	65.06	1.606×10^7	1.705×10^7	82.82	81.76
DenseNet100	Knee	50.46	49.13	6.351×10^7	6.503×10^7	79.18	78.69
	Boundary heavy	35.344	34.41	2.469×10^8	2.50×10^8	19.05	18.04
	Boundary light	50.24	49.24	6.185×10^7	6.316×10^7	79.73	79.29

9.37% ~ 10.43%, 11.9% ~ 14.8%, and 8.39% ~ 9.26%, accordingly. These results are comparable to those obtained with ResNet pruning, which is impressive given that DenseNets are considerably smaller.

15.5.2 Result Discussion

According to Tables 15.3 and 15.4, the pruning algorithm may drastically decrease the amount of FLOPs in CNN architectures while maintaining great test errors. When comparing the results to those obtained by Li et al. in [13], as shown in Table 15.5, DeepPruningES gets higher pruning ratios with slightly worse test errors on CIFAR-

Table 15.5 Using the CIFAR-10 dataset to prune the results from peer competitors

Method	CNN Model	% FLOPs Pruned (%)	Test error (%)
Li et al. [13]	VGG16	34.2	6.60
	ResNet56	27.6	6.94
	ResNet110	38.6	6.70
Ding et al. [11]	VGG16	81.39	7.56
	ResNet56	66.88	9.43

10. The pruning ratios and test errors obtained by Ding et al. in [11] were better than those obtained by the algorithm. The CNNs, on the other hand, should be trained with robust and specialized regularization in their algorithm. To prune the CNN, DeepPruningES employs solely information on its complexity of computing and classification performance. As a consequence, even without any regularization or additional knowledge about the CNN being pruned, DeepPruningES is capable of generating results equal to those of Ding et al. in [11]. Table 15.6 also displays the test errors of multiple efficient mobile CNNs trained for 150 epochs on the CIFAR-100 dataset, and the results indicate that, in comparison to their mobile counterparts, the pruned version of the full-fledged CNN architecture demonstrates competitive classification performance. In addition to the qualitative results, DeepPruningES can identify pruned CNN models in a few hours on an Nvidia GTX 1060 6GB GPU. Nevertheless, the bottleneck occurs when the pruned models are fine-tuned (Algorithm 1, line 7) utilizing the entire dataset, which could take several days according to the CNN model being pruned.

It is also worth noting that the primary objective of the algorithm is to find a set of three trimmed CNN models. As a result, a DM has four options to consider: the original CNN model, the knee solution, the boundary light solution, or boundary heavy solution. Thus, while the boundary heavy solution may appear undesirable at first glance, it can be a significant candidate solution that a DM can utilized according to his or her hardware.

Given more computational resources, the test errors shown in Tables 15.3 and 15.4 might be improved much more. To show this, one pruned solution from VGG16 on the CIFAR-10 dataset from scratch are presented, which implies that all weights are reinitialized to random values and the network is trained from there. A knee solution with a pruning ratio of 58.64% and a test error of 9.94% is select to be retrained from scratch for more than 200 epochs. Figure 15.8 depicts the training curves for

Table 15.6 Efficient mobile deep CNNs trained from scratch results

	Test error on CIFAR-100 (%)
EfficientlineNet-B0	48.02
SqueezeNet	44.53
MobileNet v2	30.91

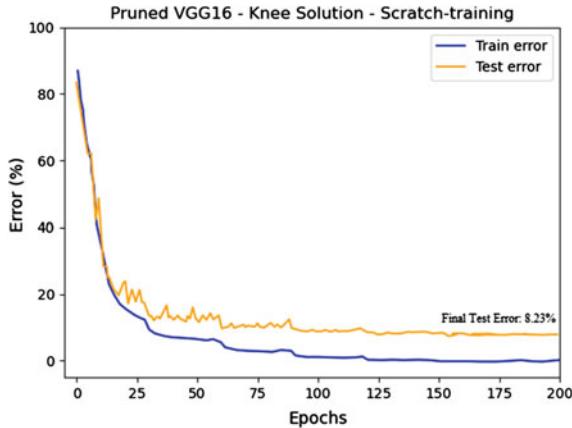


Fig. 15.8 Knee solution from the pruned VGG16 network trained from scratch for more than 200 epochs

this knee solution. Its final test error is 8.23%, which is significantly lower than the test error acquired after 50 fine-tuning epochs.

Because six different CNN models with three different architectures and amount of layers are evaluated, the results are more comprehensive than those of peer competitors. According to the knowledge, no work has tried to prune DenseNet architectures in the literature. Despite the fact that DenseNets already have lower complexity of computing and higher classification performance than regular CNNs, the results show that they may still be pruned without sacrificing classification performance.

A non-elitist common version with DeepPruningES, which is derived from the elitist ES plus version, are compared. Figure 15.9 depicts the population evolution over 10 generations for the plus version, whereas Fig. 15.10 depicts the population evolution over 10 generations for the comma version. In the plus version, it is simpler to retain the individual with the best training error over generations than in the comma version. Furthermore, the last solutions obtained utilizing the comma version are quite close with regard to error and FLOPs, defeating the objective various solutions with distinct trade-offs. If DeepPruningES is to be used for pruning CNN architectures, the plus version of the algorithm is recommended.

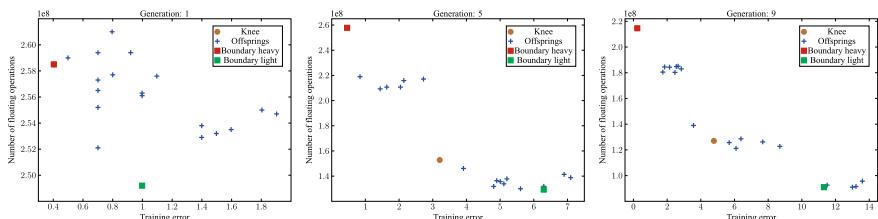


Fig. 15.9 Evolution of the population when pruning VGG16 utilizing the plus version of DeepPruningES

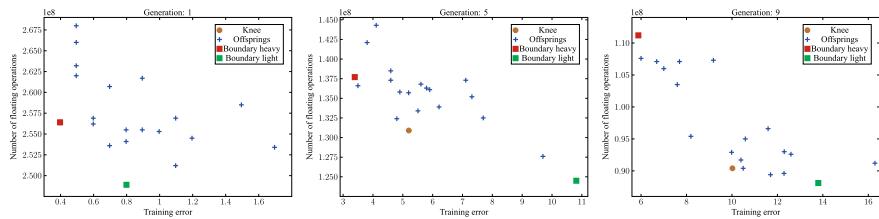


Fig. 15.10 Evolution of the population when pruning VGG16 utilizing the comma version of DeepPruningES

Finally, why an ES framework was selected over a traditional GA framework is discussed. Several candidate solutions are provided for DMs to pick from, rather than just one. As a result, the selection operator is compatible with an ES framework without the need to develop a fully-fledged multi-objective GA. In addition, while ES was developed to work using real-valued vectors, which is equally well suited to dealing with binary vectors [21].

15.6 Chapter Summary

In this chapter, we introduced a pruning method named DeepPruningES for DNNs, where the unimportant weights are reasonably removed while the performance is not significantly deteriorated. Specifically, the pruning strategy in DeepPruningES is treated as a two-objective optimization problem because the number of weights is often positively related to the performance. DeepPruningES is achieved by using ES but differs from the traditional multi-objective optimization algorithm where the Pareto front is the goal in the solving. Particularly, DeepPruningES concerns more upon the DM side and the optimal solution is directly obtained based on the knee solution which has the sound theoretical advantage in balancing the two objectives.

Although DeepPruningES directly focused on the optimal balanced solution, finding a Pareto front including multiple well-distributed solutions is also a popular strategy. In the next chapter, we will introduce an optimization algorithm specifically designed to address this multi-objective problem.

References

- Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. [arXiv:1704.04861](https://arxiv.org/abs/1704.04861).
- Huang, G., Liu, S., van der Maaten, L., & Weinberger, K. Q. (2018). Condensenet: An efficient densenet using learned group convolutions. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 2752–2761). IEEE. ISBN 978-1-5386-6420-9. <https://doi.org/10.1109/CVPR.2018.00291>, <https://ieeexplore.ieee.org/document/8578389/>.

3. Liu, H., Simonyan, K., & Yang, Y. (2019). DARTS: Differentiable architecture search. In *International Conference on Learning Representations*.
4. Yang, T.-J., Howard, A., Chen, B., Zhang, X., Go, A., Sandler, M., Sze, V., & Adam, H. (2018). NetAdapt: Platform-aware neural network adaptation for mobile applications. In V. Ferrari, M. Hebert, C. Sminchisescu, & Y. Weiss (Eds.), *Computer Vision—ECCV 2018* (Vol. 11214, pp. 289–304). Springer International Publishing. ISBN 978-3-030-01248-9 978-3-030-01249-6. https://doi.org/10.1007/978-3-030-01249-6_18.
5. Yao, S., Zhao, Y., Shao, H., Liu, S., Liu, D., Su, L., & Abdelzaher, T. (2018). Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems—SenSys ’18* (pp. 278–291). ACM Press. ISBN 978-1-4503-5952-8. <https://doi.org/10.1145/3274783.3274840>.
6. Halawa, H., Abdelhafez, H. A., Boktor, A., & Ripeanu, M. (2017). Nvidia jetson platform characterization. In F. F. Rivera, T. F. Pena, & J. C. Cabaleiro (Eds.), *Euro-Par 2017: Parallel Processing* (Vol. 10417, pp. 92–105). Springer International Publishing. ISBN 978-3-319-64202-4 978-3-319-64203-1. https://doi.org/10.1007/978-3-319-64203-1_7.
7. Su, M., Tan, J., Lin, C.-Y., Ye, J., Wang, C.-H., & Hung, C.-L. (2015). 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* (pp. 1854–1858). IEEE. ISBN 978-1-4799-8937-9. <https://doi.org/10.1109/HPCC-CSS-ICESS.2015.212>, <https://ieeexplore.ieee.org/document/7336442/>.
8. Danopoulos, D., Kachris, C., & Soudris, D. (2018). Acceleration of image classification with Caffe framework using FPGA. In *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)* (pp. 1–4). IEEE. ISBN 978-1-5386-4788-2. <https://doi.org/10.1109/MOCAST.2018.8376580>.
9. Bottleson, J., Kim, S., Andrews, J., Bindu, P., Murthy, D. N., & Jin, J. (2016). clCaffe: Opencl accelerated caffe for convolutional neural networks. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 50–57). IEEE. ISBN 978-1-5090-3682-0.
10. Ding, C., Yuan, G., Ma, X., Zhang, Y., Tang, J., Qiu, Q., Lin, X., Yuan, B., Liao, S., Wang, Y., Li, Z., Liu, N., Zhuo, Y., Wang, C., Qian, X., & Bai, Y. (2017). 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* (pp. 395–408). ACM Press. ISBN 978-1-4503-4952-9. <https://doi.org/10.1145/3123939.3124552>.
11. Ding, X., Ding, G., Han, J., & Tang, S. (2018). Auto-balanced filter pruning for efficient convolutional neural networks. In *AAAI Conference on Artificial Intelligence*. <https://aaai.org/ojs/index.php/AAAI/AAAI18/paper/view/16450>.
12. Luo, J.-H., Wu, J., & Lin, W. (2017). ThiNet: A filter level pruning method for deep neural network compression. In *2017 IEEE International Conference on Computer Vision (ICCV)* (pp. 5068–5076). IEEE. ISBN 978-1-5386-1032-9. <https://doi.org/10.1109/ICCV.2017.541>, <http://ieeexplore.ieee.org/document/8237803/>.
13. Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. (2017). Pruning filters for efficient convnets. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
14. Zhou, Y., Yen, G. TG., & Yi, Z. (2019). A knee-guided evolutionary algorithm for compressing deep neural networks. *IEEE Transactions on Cybernetics*, 1–13.
15. Wang, Y., Xu, C., Qiu, J., Xu, C., & Tao, D. (2018e). Towards evolutionary compression. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 2476–2485). ACM. ISBN 978-1-4503-5552-0. <https://doi.org/10.1145/3219819.3219970>.
16. Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

17. Courbariaux, M., Bengio, Y., & David, J.-P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28* (pp. 3123–3131). Curran Associates, Inc.
18. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized neural networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 4107–4115).
19. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
20. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).
21. Beyer, H.-G., & Schwefel, H.-P. (2002). Evolution strategies—A comprehensive introduction. *Natural Computing*, 1(1), 3–52. ISSN 15677818. <https://doi.org/10.1023/A:1015059928466>.
22. Chiu, W.-Y., Yen, G. G., & Juan, T.-K. (2016). Minimum manhattan distance approach to multiple criteria decision making in multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 20(6), 972–985. ISSN 1089-778X, 1089-778X, 1941-0026. <https://doi.org/10.1109/TEVC.2016.2564158>, <http://ieeexplore.ieee.org/document/7465803/>.
23. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
24. Mittal, D., Bhardwaj, S., Khapra, M. M., & Ravindran, B. (2019). Studying the plasticity in deep convolutional neural networks using random pruning. *Machine Vision and Applications*, 30(2), 203–216. ISSN 0932-8092, 1432-1769. <https://doi.org/10.1007/s00138-018-01001-9>.
25. Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2017). Pruning convolutional neural networks for resource efficient transfer learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
26. Hu, H., Peng, R., Tai, Y.-W., & Tang, C.-K. (2016). Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. [arXiv:1607.03250](https://arxiv.org/abs/1607.03250).
27. Luo, J.-H., & Wu, J. (2017). An entropy-based pruning method for CNN compression. [arXiv:1706.05791](https://arxiv.org/abs/1706.05791).
28. Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical Report, University of Toronto. <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
29. Fernandes, F. E., Jr. & Yen, G. G. (2019). Particle swarm optimization of deep neural networks architectures for image classification. *Swarm and Evolutionary Computation*, 49, 62–74. ISSN 22106502. <https://doi.org/10.1016/j.swevo.2019.05.010>, <https://linkinghub.elsevier.com/retrieve/pii/S2210650218309246>.

Chapter 16

Deep Neural Architecture Compression



16.1 Introduction

Various techniques have been proposed that take into account both the scale of parameters and performance [1–3]. Analysis of sensitivity or testing in a variety of configurations are commonly used to obtain a suitable compromise between model size and performance. Meanwhile, because convolutional operations take the majority of computing time, it has been investigated that pruning convolutional filters is a potentially promising method for striking a balance between the performance and the scale of parameters. As a result, several criteria for pruning filters in CNNs have been suggested [4–6] and implemented [7].

Even if the methods described above have significantly reduced the overhead of memory consumption and computing cost, deep model compression research needs to be pursued further. To begin with, obtaining a suitable trade-off between parameters reduction and performance degradation [1, 3, 6] usually necessitates significant human effort. Especially, it is required to manually determine either the performance guarantee or the compression rate, making the methods prohibitive to use and limiting their practical applicability. Furthermore, the performance measure of pruned filters is typically a discontinuous and non-differentiable function that is not able to be optimized in a direct manner using common gradient descent techniques. Greedy strategies are obviously sub-optimal [8] because when the parameters are pruned, they cannot be recovered in the later phase. In addition, the magnitude of parameters are commonly assumed to be able to estimate their importance [6, 9], while this assumption may not be a reliable way to characterize redundancy and needs to be theoretically validated. The key observation is that the information inherited from the original model is crucial in maintaining performance, which is often decided before fine-tuning. Besides, striking a suitable tradeoff between parameter reduction and information preservation may be a hopeful way to reduce the amount of human effort required for the DNN compression.

To compress DNN, a multi-objective optimization model is built in this chapter, which employs the commonly used population-based meta-heuristics [10–12] to optimize the objectives of performance and model size simultaneously. Whereas, to obtain the optimal solution, a posterior step of MCDM is required, which typically includes the participation of DMs in an interactive manner and imposes a significant burden cognitively in finding a compromise between two objectives involved. Generally, the optimal tradeoff between multiple objectives is chosen as the best solution. It equals to the optimum tradeoff between performance and parameter scale in the CNN compression problem. In the community of EC, this optimal solution is known as the knee solution because the knee region of the Pareto front is where it is positioned, where additional progress toward one goal leads to substantial backward progress toward another.

Figure 16.1 shows an example of this tradeoff, with the knee solution in the bottom left. The knee solution achieves less accuracy degradation and far fewer parameters compared with the solution which is shown in the top left. The knee solution, on the other hand, is substantially more accurate and has a model size that is slightly bigger

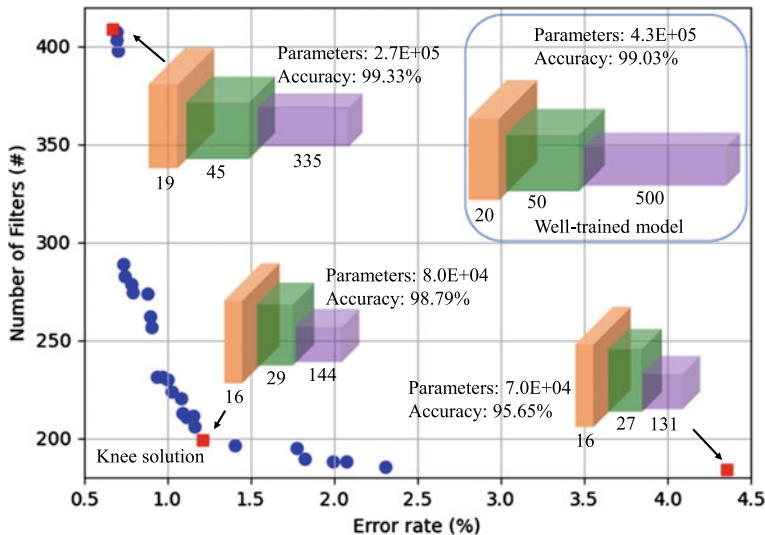


Fig. 16.1 An example of the knee solution generated when compressing LeNet on the MNIST dataset using the algorithm. The performance and the amount of filters of CNNs are represented by the vertical and horizontal axes, respectively. The number of retained filters in each of the convolutional layers is represented by three numbers below each CNN. The top left solution, which reduced the amount of filters by roughly half while performance is improved by 0.3%. The bottom right solution, on the other hand, retained far fewer filters (only 10%), however, the performance significantly deteriorated (almost 3.14% accuracy drop). The bottom left knee solution obtains a satisfactory compromise between performance (just 0.24% accuracy reduction) and the amount of filters (over 5× compression). There is no fine-tuning involved

than the bottom right solution. As a result, the highlighted knee solution, obtains a satisfactory tradeoff between the error rate and the size of parameters among all Pareto optimal solutions. Further reduction of parameters on the knee solution will result in substantial deterioration of performance, and further performance improvement will necessitate the retention of a larger amount of parameters. As can be observed, the knee solution can be quite helpful in balancing the performance and the size of parameters in the CNN compression problem.

In comparison to existing compression approaches, DNN compression is formulated as a multi-objective optimization problem and a knee guided EA (KGEA) is designed correspondingly to optimize the objectives of the performance and size of parameters simultaneously. Particularly, the minimum Manhattan distance (MMD) method for identifying the knee solution, can guide the direction of search to knee region to search solutions with a satisfactory compromise. Because of this characteristic, the method does not require an accuracy guarantee or a pruning ratio before compression. Furthermore, tremendous potential is offered by the derivative-free characteristics of KGEA for resolving DNN compression problems. The following are the main contributions of this chapter.

1. To formally formulate the CNN compression problem, a multi-objective optimization model is established, in which the objectives of the performance and size of parameters are optimized simultaneously to obtain a good tradeoff solution.
2. A KGEA is designed according to the multi-objective optimization model for alleviating the MCDM in CNN compression from burdensome human labor. Search direction can be effectively guided by the algorithm with the introduced MMD approach and finding a solution with a satisfactory tradeoff between the performance and size of parameters for deep models.
3. The importance evaluation of convolutional filters has direct correlation to the performance in KGEA, which does not employ heuristic criteria and is designed to characterize the redundancy of parameters in deep models robustly.
4. On fully convolutional LeNet and VGG-19, KGEA is empirically evaluated. The results of the experiments show that a quality tradeoff between compression ratio and performance can be achieved. Surprisingly, KGEA can generate a slightly compressed model with improved performance while no fine-tuning involved.

The rest of the chapter is laid out as follows. The motivation for compressing DNNs and multi-objective optimization is discussed in Sect. 16.2. Then Sect. 16.3 presents the details of KGEA for compressing DNNs. The experimental results achieved by using KGEA on commonly used deep models, and the corresponding analysis of results, are presented in Sect. 16.4.

16.2 Related Work and Motivation

16.2.1 Convolutional Neural Network Compression

In the field of CNN compression [13], numerous efforts have been made, and they can be divided into five categories in general. *Matrix factorization based methods* [1, 14] are the first, in which the weight matrices are decomposed by singular value decomposition, afterwards, approximate values of the parameters can be utilized as compressed weights. The tradeoff between performance and size of parameters can be altered based on the singular values which are ranked using their magnitudes. Though the approximation usually needs tuning from diverse configurations to balance speed gain and performance loss, it can be used as a posterior step in any compression method. The *quantization based methods* [15, 16] are the second category, which clusters the original parameters to represent them with lower precision. Although parameter quantization can efficiently reduce parameter storage, performance will surely suffer as a result, even if the network structure remains intact. Quantization, on the other hand, can be used as an extra method for CNN compression. *Knowledge distillation based methods* [17, 18] are the third category. In which a teacher network is compressed by using softened Softmax to train a student network that is both efficient and compact to learn the class distributions output of the teacher, where the teacher network is the large scale network that need to be compressed. However, knowledge distillation can only be used for classification tasks with the Softmax loss function, which limits its practical application [19]. *Pruning based methods* [20–22] make up the fourth group. Filter level pruning, in particular, has received a lot of attention because it is able to effectively decrease both computation and parameters at the same time, and off-the-shelf libraries can support the compressed CNNs perfectly. However, determining which filters may be safely pruned without performance degradation is difficult. There are a number of criteria that can be used to identify the redundant filters along this line of thought. In [6], the importance of a filter is evaluated by computing sum of its absolute weight, after that, a satisfactory compromise is accomplished by ranking them and a predefined threshold is used to keep the top-ranked filters. The criterion for pruning in is the APoZ of each filter [4]. Activations are seen to be redundant if the majority neurons outputs are near to zero. Methods based on architecture design make up the fifth group. It avoids model parameter compression by directly designing efficient and compact network architectures. 1×1 convolution is used in [23] to expand the capacity of network while maintaining computational complexity small. Additionally, the channel shuffle operations are used in Shufflenet [24] to enhance the information change within multiple groups, reducing computation cost while maintaining accuracy.

The study is classified as filter pruning because it has the benefit of simultaneously reducing computation and parameters overhead, and it may be further improved utilizing techniques including LR approximation and quantization. In addition, a multi-objective optimization paradigm is used for convenience of the conflicting objectives optimization in order to find a trade-off solution.

16.2.2 Evolutionary Algorithms and MMD

As a type of efficient approaches for solving MOPs, multi-objective EAs (MOEAs) [10, 25] are designed with the goal of identifying a group of tradeoff solutions [26, 27]. However, because the final solution is unique, the DM is rarely concerned with the whole optimal Pareto front in real-world applications. Particularly, in the CNN compression problem [19], it is preferable to discover the knee solution with a satisfactory compromise between each objective. On the other hand, for current networks, the amount of filters is so large, and performance is sensitive to filter changes, resulting in an incredibly large search space. It is difficult for conventional MOEAs to discover the knee regions in a fair amount of time. A single objective EA for compressing DNNs has been presented in a heuristic search based work [28]. Particularly, for the weighting scale of parameters and performance, a manually specified parameter is employed. It reduces the disadvantage of finding a suboptimal solution as a result of greedy search, but it also limits the ability to find an optimal trade-off solution. In this regard, using the multi-objective optimization paradigm, which has received far less attention, presents an interesting approach for dealing with this difficulty.

To define the knee for MCDM, a theoretical framework known as MMD approach [29] has recently been designed, which has rich geometric interpretations and may characterize the knee globally. Formally, for a transition from solution \mathbf{x}_i to \mathbf{x}_j , the improvement percentage in the m th objective is defined as

$$IP_m(\mathbf{x}_i \rightarrow \mathbf{x}_j) = \frac{f_m(\mathbf{x}_i) - f_m(\mathbf{x}_j)}{L_m} \quad (16.1)$$

where $L_m = \max f_m(\mathbf{x}) - \min f_m(\mathbf{x})$.

Then the improvement percentage of net is defined as

$$IP(\mathbf{x}_i \rightarrow \mathbf{x}_j) = \sum_{m=1}^M IP_m(\mathbf{x}_i \rightarrow \mathbf{x}_j) \quad (16.2)$$

As a result, a preference model for pairwise comparison is defined as

$$\mathbf{x}_j \text{ is preferred to } \mathbf{x}_i \text{ if } IP(\mathbf{x}_i \rightarrow \mathbf{x}_j) > 0 \quad (16.3)$$

A divide and conquer (D&C) approach based on Eqs. (16.1), (16.2) and (16.3), can be employed to locate the knee solution by repeatedly contrasting each pair of possible solutions. Furthermore, the following steps can be used to obtain an equivalent condition from Eq. (16.3)

$$IP(\mathbf{x}_i \rightarrow \mathbf{x}_j) > 0 \quad (16.4)$$

$$\Leftrightarrow \sum_{m=1}^M \frac{f_m(\mathbf{x}_i)}{L_m} > \sum_{m=1}^M \frac{f_m(\mathbf{x}_j)}{L_m} \quad (16.5)$$

$$\Leftrightarrow \mathbf{x}^* = \arg \min_{\mathbf{x}} \sum_{m=1}^M \frac{f_m(\mathbf{x})}{L_m} \quad (16.6)$$

The MMD approach can be represented by further incorporating the ideal vector \mathcal{Z}^{min} into Eq. (16.6) and utilizing the fact that $f_m(x) > \mathcal{Z}_m^{min}$

$$\mathbf{x}^* = \arg \min_{\mathbf{x}} \left\| \frac{f(\mathbf{x})}{L} - \frac{\mathcal{Z}^{min}}{L} \right\|_1 \quad (16.7)$$

where x^* indicates the identified knee solution, $\|\cdot\|_1$ represents the Manhattan norm, and $\mathcal{Z}^{min} = \min f(\mathbf{x})$.

The knee solution that is in the closest proximity to the normalized ideal vector is chosen by the MMD approach, according to an intuitive explanation. In [29], more details about MMD can be found.

16.3 KGEA for Compressing DNNs

KGEA for CNN compression is described in this section. For filter pruning, KGEA uses the GA paradigm. The operation of convolutional filter pruning, in particular, is introduced first. The developed multi-objective optimization model for CNN compression is shown in the subsection below. Finally, the details and design principles of the KGEA are provided, along with a discussion.

16.3.1 Convolutional Filter Pruning

Formally, the input feature maps and convolutional filters in the i th layer of a CNN with L convolutional layers are represented as F_i and W_i , respectively. For the i th convolutional layer, the number of input channels is C_i , and the number of filters in that layer is N_i . The produced feature maps and each of those filters will then have channels with the numbers of N_i and C_i , respectively. After compression, if T filters are removed in the i th layer, the number of channels in the produced feature maps is decreased to $N_i - T$, similarly, the amount of filters is decreased to $C_i - T$. It is worth noting that in the next layer, the corresponding channels of filters need to be pruned as well. As a result, the filter pruning is always carried out within two convolutional layers. Figure 16.2 depicts how filters are pruned in the convolutional layer.

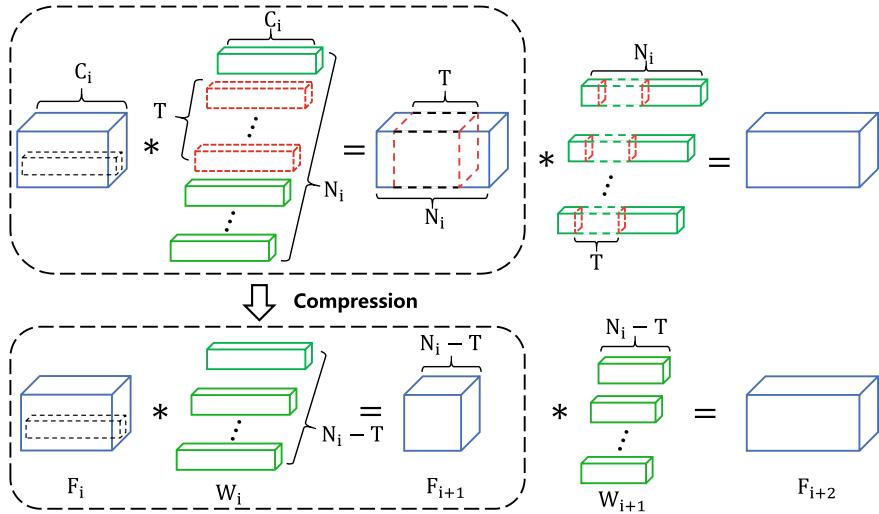


Fig. 16.2 When filters are pruned in one convolutional layer, the number of filter channels in the following convolutional layer is reduced, where T is the number of pruned filters and N_i is the number of filters in the i th layer. The input feature map F_i has channels with the number of C_i , and the filters in the i th layer is W_i . The number of filters in W_i is reduced to $T - N_i$ after compression, and the number of filter channels in W_{i+1} is also reduced to $T - N_i$

It is a combinatorial optimization problem to figure out which filters to prune. In ideal conditions, pruning a subset of filters is assumed that will not lead to significant performance degradation. However, the cost function need to be evaluated for $2^{\sum_{i=1}^L N_i}$ times on a chosen subset of data, where N_i is the number of filters in the i th layer and L is the number of layers. For the most common CNNs (most CNNs contain over 1,000 filters), it appears to be an NP-hard problem that cannot be solved apparently in a affordable time by evaluating all of the different possible combinations. In the meantime, to obtain a model that is well compressed and has a small model size, the number of filters should be preserved as less as possible and the performance should be preserved as much as possible. Formally formulate it as a multi-objective optimization problem in this work.

16.3.2 Multi-objective Modeling for CNN Compression

Assume that $W = \{W_1, W_2, \dots, W_L\}$ is the representation of the weights of each convolutional layer. On the dataset \mathcal{D} , the CNN cost function is defined as $C(\mathcal{D}; W)$, which can represent Softmax or MSE losses. The problem of compressing CNN can be formalized as the bi-objective optimization problem:

$$\left\{ \begin{array}{l} \min_{\mathcal{M}} C(\mathcal{D}; \mathcal{M} \circ W) \\ \min_{\mathcal{M}} \sum_{i=1}^L \|\mathcal{M}_i\|_1 \end{array} \right. \quad (16.8)$$

where \mathcal{M} has the same size as W and functions as a mask for the elimination of unnecessary weights in each layer. The ℓ_1 norm accumulating absolute values of \mathcal{M}_i (the mask in the i th layer) is $\|\cdot\|_1$, and element-wise multiplication is represented by \circ .

Because the convolutional layers consume the computation overhead, the efficiency of the compressed CNNs is not improved by the parameter-wise compression. Therefore, inference speed is constrained, and specialized hardware or sparse libraries are required to support the multiplications of matrix for sparse weights. For the purpose of avoiding the problem, a filter-wise approach is used for CNN compression, which allows for greater flexibility and prunes the convolutional filters in a direct way, which may be realized with the following Equation:

$$\mathcal{M}_i^{(n,:,:,:)} = 0 \text{ or } 1 \quad (16.9)$$

where the mask for the n th convolutional filter in the i th layer is represented by $\mathcal{M}_i^{(n,:,:,:)}$.

Numerous criteria for evaluating the importance of each filter based on their magnitudes, which can be used to prune filters, have been suggested in previous studies [4, 6]. One possible issue is that those criteria might not be robust in locating redundancy filters due to the fact that the value distribution of filters varies throughout layers, and some edge extractor filters that are very useful may have small values [28]. Unlike those methods based on criterion, a binary vector are used to indicate if the filters are active or not. This design does not make assumptions about the amount of the value of the filter, instead of evaluating the filter importance in relation to the performance loss of the pruned model. As a result, the problem of compressing CNN can be viewed as a search for a compact binary representation that is able to prune filters to the greatest extent possible while maintaining performance to the greatest extent possible.

For a well-trained CNN, the parameters W are generally frozen. As a result of optimizing the two objectives in Eq. (16.8), the value of loss function and the number of parameters should be minimized at the same time on a subset of data. However, there is a conflicting nature in that pruning parameters from W will result in a performance loss as represented by C . Accordingly, a group of non-dominated Pareto optimal solutions will exist, and the MCDM will be required to choose an optimal solution that achieves a satisfactory compromise between each objective. This is a difficult operation for DMs since it usually involves a lot of human labor to look through the pruning parameter space. In order to make the process of finding a decent tradeoff solution easier, a KGEA is designed.

16.3.3 KGEA

The number of filters in the DNN compression problem is relatively large for current networks, and performance is sensitive to filter changes, resulting in a large search space and making it hard to obtain a solution with a satisfactory tradeoff between each objective in a reasonable amount of time. KGEA is designed using three principles as a result of this consideration. To begin, the knee solution is determined and used to guide the search to the knee region. Furthermore, because the MMD approach focuses on estimating the range of the Pareto front, exploration of the boundary regions is emphasized. This means that identifying boundary solutions accurately is critical for obtaining the knee solution. At last, on the boundary and knee regions, a certain degree of diversity is maintained in order to prevent population aggregation in such places and therefore avoid premature convergence. KGEA is hoped to be successful in obtaining a knee solution by following the above design principles.

The boundary and knee solutions are shown in circles filled with black in Fig. 16.3. The one connected to the knee solution is known as the knee vector, and solutions that are nearer to the knee vector (measured in terms of angle) are preferred over those that are further away, guiding the search towards the knee area gradually. Solutions that are nearer to the boundary solution in the population, on the other hand, tend to be retained in environmental selection more likely. Accordingly, the strength of searching will concentrate on the boundary and knee regions, which means it will look for compressed CNNs that strike a satisfactory compromise between each objective, i.e., compressed models with the smallest model size or best performance.

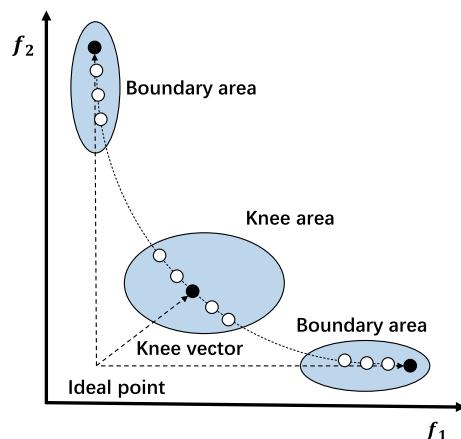


Fig. 16.3 An example to illustrate the KGEA design principles. The black filled circles represent boundary and knee solutions. The knee vector, in particular, is created by connecting the origin to the knee solution. In the meantime, the closeness of solutions to the boundary or knee solutions determines their association to the shadowed regions. Finding the minimum value of each objective can be used to create an ideal point

16.3.3.1 Main Framework

Algorithm 1 depicts the main KGEA process. It starts with the population \mathcal{P}_0 being initialized. Then, at each generation, the boundary and knee solutions are determined and kept. Particularly, by connecting the origin to those kept solutions in the objective space, a set of reference lines \mathcal{V}_t is created. By applying genetic operations to \mathcal{P}_t , the offspring \mathcal{Q}_t can be obtained. Each generation, the ideal vector \mathcal{Z}^{min} is updated. Following that, offspring \mathcal{Q}_t is incorporated into \mathcal{P}_t , and environmental selection is done on the combined population to generate next generation population.

Algorithm 1: Main Framework of KGEA

```

1  $\mathcal{P}_0 \leftarrow$  Population-Initialization;
2  $t \leftarrow 0$ ;
3 while stop criterion is not satisfied do
4    $[\mathcal{V}_t, \mathcal{Z}^{min}] \leftarrow$  Reference-Vector-Construction( $\mathcal{P}_t$ );
5    $\mathcal{Q}_t \leftarrow$  Variation( $\mathcal{P}_t$ );
6    $\mathcal{Z}^{min} \leftarrow \min([\mathcal{Z}^{min}, \mathcal{Q}_t])$ ;
7    $\mathcal{R}_t \leftarrow \mathcal{P}_t \cup \mathcal{Q}_t$ ;
8    $\mathcal{P}_{t+1} \leftarrow$  Environmental-Selection( $\mathcal{R}_t, \mathcal{V}_t, \mathcal{Z}^{min}$ );
9    $t \leftarrow t + 1$ ;
10 end
```

16.3.3.2 Reference Vector Construction

The boundary and knee solutions can be used to construct reference lines, as previously stated. Noting that the scaling problem is avoided by using normalization. In the normalized objective space, the reference vectors are formed, as shown in Algorithm 2, as a result, the set of reference vectors \mathcal{V}_t can be expanded by directly adding unit vectors along each axis. Meanwhile, by applying the Manhattan norm to the objective values of the normalized solutions, finding the current knee vector V_{knee} using the MMD approach is simple. The knee vector is combined with those unit vectors returned as reference vectors in the last step.

Algorithm 2: Reference-Vector-Construction

Input : Population \mathcal{P}_t
Output: Ideal Vector \mathcal{Z}^{min} and Reference Vectors \mathcal{V}_t

- 1 $\mathcal{V}_t \leftarrow$ Generating unit reference vectors along each axis;
- 2 $V_{knee}, \mathcal{Z}^{min} \leftarrow$ Finding-Knee-Vector-using-MMD(\mathcal{P}_t);
- 3 $\mathcal{V}_t = \mathcal{V}_t \cup V_{knee}$;

16.3.3.3 Environmental Selection

Non-dominated sorting, as illustrated in Algorithm 3, facilitates KGEA convergence by sorting solutions into different ranks like $(\mathcal{F}_1, \mathcal{F}_2, \dots)$. If the mating pool is not completely filled, those rank one solution will be chosen first to occupy the mating slots in the new population \mathcal{P}_{t+1} , followed by those with higher ranks. Once $|\mathcal{P}_{t+1}| + |\mathcal{F}_i| > N$, that is, the number of remaining mating slots is less than the number of solutions in the i th rank, to choose which solution is going to be kept, a selection mechanism is needed. A ranking strategy is designed according to the three design principles discussed above, which can emphasize the searching strength on the boundary and knee regions of the current population, as illustrated in Fig. 16.3.

Algorithm 3: Environmental-Selection

```

Input : Ideal Vector  $\mathcal{Z}^{min}$ , Reference Vectors  $\mathcal{V}_t$ , Population  $\mathcal{R}_t$ 
Output: A new population  $\mathcal{P}_{t+1}$ 
1  $\mathcal{P}_{t+1} \leftarrow \emptyset, i \leftarrow 1;$ 
2  $(\mathcal{F}_1, \mathcal{F}_2, \dots) \leftarrow$  Non-Dominated-Sorting( $\mathcal{R}_t$ );
3 while  $|\mathcal{P}_{t+1}| + |\mathcal{F}_i| \leq N$  do
4    $\mathcal{P}_{t+1} \leftarrow \mathcal{P}_{t+1} \cup \mathcal{F}_i;$ 
5    $i \leftarrow i + 1;$ 
6 end
7 The last front to be included  $\mathcal{F}_l = \mathcal{F}_i$ ;
8 if  $|\mathcal{P}_{t+1}| = N$  then
9    $\mid$  return  $\mathcal{P}_{t+1}$ ;
10 else
11    $\mid$  Choosing solutions from  $\mathcal{F}_l : K = N - |\mathcal{P}_{t+1}|$ ;
12    $\mid$   $Rank = \text{Ranking}(\mathcal{F}_l, \mathcal{V}_t, \mathcal{Z}^{min})$ ;
13    $\mid$  Choosing first  $K$  solutions from  $\mathcal{F}_l$  which is sorted by  $Rank$  ;
14 end

```

16.3.3.4 Ranking-Based Selection

The main idea of KGEA is to lead the search direction of the population using the knee solution, as well as to emphasize the search effort within the boundary regions. Algorithm 4 depicts the primary process of ranking-based selection. To begin, the nadir point \mathcal{Z}^{max} (the vector representing the values of the objective function that are considered to be the worst in the set of Pareto-optimal solutions) is used to normalize the corresponding population and ideal point \mathcal{Z}^{min} (the vector that represents the best objective function values in current population), then in the normalized objective space, calculation of angles can be done that exist between each solution in \mathcal{F}_l to the reference vectors \mathcal{V}_t . After that, sorting the angles between each solution and the i th reference vector yields ranks $Rank_i$. A larger rank value will be applied to the one with a smaller angle, and vice versa. Particularly, the solution that is nearest to

any of the reference vectors (measured in terms of angle) is determined and given a rank value of inf , preserving the found boundary and knee solutions in an explicit way. Accordingly, for a M -objective problem, there are totally $M + 1$ ranks. There will be three ranks in a bi-objective problem, one of which are related to the knee vector and two to the boundary vector. To get a combined result, all of those ranks are subjected to a max operation, which is calculated as $\max_{i=1:L}(\text{Rank}_i)$, where L is the number of reference vectors. The combined rank is hoped to emphasize any solution with a larger rank value which is nearer to each reference vector.

Algorithm 4: Ranking

Input : Ideal Vector \mathcal{Z}^{\min} , Reference Vector \mathcal{V}_t , and Population \mathcal{F}_l
Output: The rank of each solution

```

1  $\text{Rank}_{i=1,2,\dots} \leftarrow \emptyset;$ 
2 Computing nadir point  $\mathcal{Z}^{\max}$  of  $\mathcal{F}_l$ ;
3  $\mathcal{F}'_l \leftarrow \text{Normalize}(\mathcal{F}_l, \mathcal{Z}^{\min}, \mathcal{Z}^{\max})$  ;
4  $A \leftarrow \text{Computing angles between } \mathcal{F}'_l \text{ and } \mathcal{V}_t$ ;
5 Number of reference vectors  $L \leftarrow |\mathcal{V}_t|$  ;
6 for  $i = 1 : L$  do
7    $\text{Rank}_i \leftarrow \text{argsort}(A^i)$ ;
8    $\text{Rank}_i(0) = \text{inf}$ ;
9   if  $V_t^i \neq \text{knee vector}$  then
10    |  $\text{Rank}_i(\text{end}) = \text{inf}$ ;
11   end
12 end
13  $\text{cdist} = \text{CrowdingDistance}(\mathcal{F}_l)$ ;
14  $\text{Rank}_{L+1} = \text{argsort}(\text{cdist})$ ;
15 return  $\max_{i=1:L}(\text{Rank}_i) + \text{Rank}_{L+1}$ ;

```

A possible problem with the aforementioned selection strategy is the loss of diversity, and the solutions quickly aggregate together in boundary and knee regions, reducing the searching ability dramatically. A mechanism for keeping local diversity is introduced into the environmental selection process to alleviate this issue. The basic idea is to punish those aggregated solutions by using the crowding distance [10]. To combine this mechanism into the above ranking information, the summation operation is used, which is calculated as $\max_{i=1:L}(\text{Rank}_i) + \text{Rank}_{L+1}$, where Rank_{L+1} is the rank that represents the degree of crowding of each solution.

In particular, Fig. 16.4 shows an example of the ranking method. Assume there are ten possible solutions, eight of which will be chosen and passed down to the next generation population. The MMD approach is used to identify the knee vector. The ranking method determines which remaining solutions are preserved, while boundary and knee solutions are indicated in circles filled with black, which will undoubtedly be highlighted and kept to the next generation, as described with an example shown in Table 16.1.

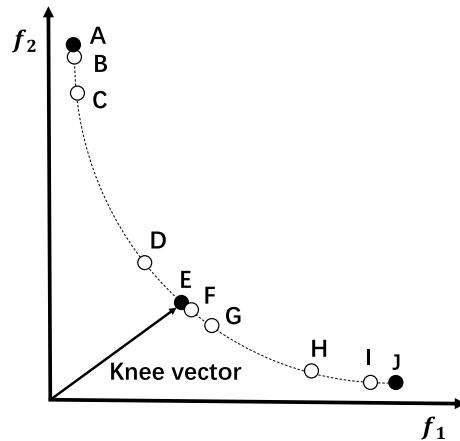


Fig. 16.4 An example to illustrate the ranking selection method. The black filled circles represent boundary and knee solutions. The knee vector, in particular, is formed by connecting the origin to the knee solution. In the meantime, the closeness of solutions to the boundary or knee solutions determines their association to the reference vectors. Finding the minimum value of each objective can be used to create an ideal point

Table 16.1 An example of the ranking method

Rank	A	B	C	D	E	F	G	H	I	J
$Rank_1(f_1)$	1 (Inf)	2	3	4	5	6	7	8	9	10 (Inf)
$Rank_2(f_2)$	10 (Inf)	9	8	7	6	5	4	3	2	1 (Inf)
$Rank_3$ (knee)	1	3	5	7	10 (Inf)	9	8	6	4	2
$\max_{i=\{1,2,3\}} Rank_i$	10 (Inf)	9	8	7	10 (Inf)	9	8	8	9	10 (Inf)
$Rank_4$ (crowding distance)	Inf	4	8	9	5	3	7	10	6	Inf
$\max_{i=\{1,2,3\}} Rank_i + Rank_4$	Inf	13	16	16	Inf	12	15	18	15	Inf
Sorted order						A>E>J>H>C>D>G>I>B>F				

The rank of the angles between solutions and reference vectors is calculated first as $Rank_1$ to $Rank_3$, as shown in Table 16.1 and Fig. 16.4, a solution will have a higher rank when the angle between it and the reference vector that is closest to it is small, and the reverse is also true. To explicitly maintain those solutions in the selection, the rank of the knee solution E and boundary solutions A, J are set to infinity. As a result, by applying $\max_{i=\{1,2,3\}}(Rank_i)$ to each of the ranks, solutions with large rank values will continue to be assigned the same value of rank, which is able to be employed as a criterion to choose solutions nearer to boundary and knee regions, and then lead the population search direction towards those regions. The $Rank_4$ shows the result of the mechanism for keeping local diversity, with F being the most crowded one and receiving the lowest value of rank. Therefore, despite being quite close to the knee vector, F ranked the last in the final result. As a result, solutions that are nearer to reference vectors but crowded in their neighborhood will

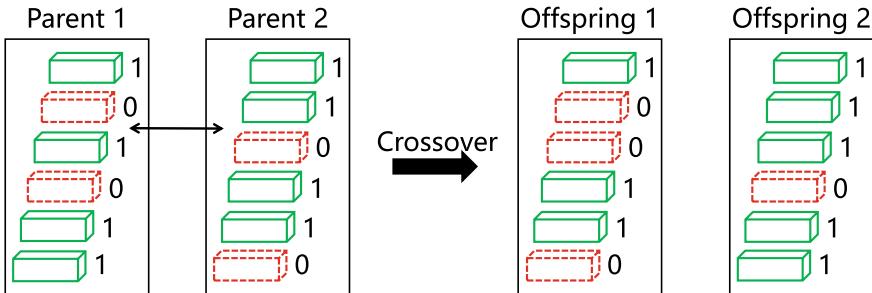


Fig. 16.5 Crossover operator. Each of the two parent solutions has two filters pruned, which are indicated by 0. The double arrow indicates the position for recombination. It is worth noting that the 1st and the 5th filter from the top have no chance to be pruned during this process

be degraded, while solutions that are further away from the reference vectors but not so crowded will have a higher probability of being chosen and kept for the next generation.

The following is a summary of the CNN compression process utilizing KGEA. To begin, initialize solutions in the population based on the layers number in a given CNN. After that, KGEA is used to iteratively optimize the objectives given in Eq. (16.8) until a specified criterion for stopping is satisfied. Finally, after the knee solution has been obtained, fine-tuning strategy can be employed in order to recover the performance of the network after compressed.

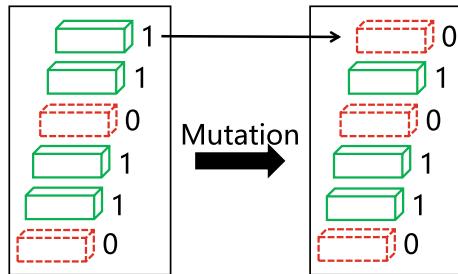
16.3.4 Encoding Scheme and Genetic Operators

KGEA uses binary encoding strategy, in which 1 and 0 are used to indicate whether a filter in a convolutional CNN is active or not. As a result, a set of genetic operators, which are detailed in this section, are used to modify individuals in the population in order to explore the search space.

16.3.4.1 Crossover

Figure 16.5 shows an example of the crossover operator. It selects a point for recombining the information of each pair of parent individuals at random. The binary code from these two parent individuals are then swapped to create two offspring individuals. One potential issue of this operator is that it only exploits existing knowledge in the individuals, thus some filters may not be pruned.

Fig. 16.6 Mutation operator. The individual shown on the left side is modified by mutation operator, and result in the individual shown on the right side, where the 1st filter from the top is pruned



16.3.4.2 Mutation

The mutation operator is also applied to the individuals in order to give all filters an opportunity to be pruned. Each bit of the binary code, in particular, can be flipped with a given probability. Every filter will have a chance to be pruned as a result of this. Figure 16.6 shows an example of the mutation operator.

16.3.5 Discussion

As can be observed, several fundamentals are shared by KGEA and preference-based MOEAs [30], which contains preference information during the course of evolution [31, 32]. Whereas, rather of working to promote convergence and variety throughout the entire Pareto front, KGEA concentrates on searching and selecting solutions in the boundary and knee regions. Furthermore, the MMD approach has been innovatively incorporated as the driving force behind KGEA, which is able to meaningfully and automatically choose satisfied trade-off solutions for the problems of compressing CNN. Using some existing MOEAs [10, 11] to obtain a set of Pareto optimal solutions, then applying some posterior decision making approaches to obtain the knee solution, on the other hand, is intuitive. Unluckily, majority of the search is spent on finding undesirable solutions, because convergence and diversity promotion strategies largely ignore the knee solution. Furthermore, because the redundancy in CNN is not known before being compressed, articulating preference information as prior knowledge is also impossible. Worse, CNN models are computationally expensive, which implicitly prohibits the use of a huge population. From this perspective, KGEA may still benefit from obtaining a knee solution with a small population, making it more suitable for the problems of compressing CNN.

The environmental selection, knee vector identification, and non-dominated sorting have $O(MN \log N)$, $O(N)$, and $O(MN^2)$ computational complexity, respectively, for an optimization problem with M objectives and a population with size of N . In comparison to most popular MOEAs, the entire time complexity of KGEA at each generation is $O(MN^2)$, which is really efficient in terms of calculation.

16.4 Experimental Studies

A series of experiments are designed in this section for evaluating the performance of KGEA in compressing CNNs. On the MNIST dataset with a fully convolutional LeNet and on the CIFAR-10 dataset with a large scale network VGG-19, the efficacy of KGEA in obtaining knee solutions is investigated. The results of the experiments, as well as a comparison to state-of-the-art methods, are presented and analyzed.

16.4.1 Experimental Settings

The goal of CNN compression is to find a set of filters that can be pruned. It is possible to accomplish this in one of two ways: either by beginning with no filters at all and gradually appending important ones, or by beginning with a complete set and gradually removing filters with less significance from the set. The KGEA in this study implements the latter by starting with all filters kept, then pruning filters using the designed genetic operators. Noting that, unlike existing greedy strategies [6, 21], any pruned filter is able to be recovered during the process of evolution. KGEA takes a well-trained model as input and creates a population of solutions, each of which indicates a pruned model. Mutation and crossover probabilities are set to 0.01 and 1, respectively. The MMD approach outputs a compressed model once the algorithm has terminated. For recovering its performance, parameters from the well-trained model are copied to initialize the network that is pruned, then it is re-trained on the training set for 10 epochs at 1/10 of its initial learning rate, and choose the model that achieves the highest accuracy on the validation set (1/10 of the training set) as the best model. The accuracy reported is based on the test set. The method is repeated 10 times since the random nature of KGEA, and the median result is chosen for comparison.

16.4.2 Experiments on Fully Convolutional LeNet

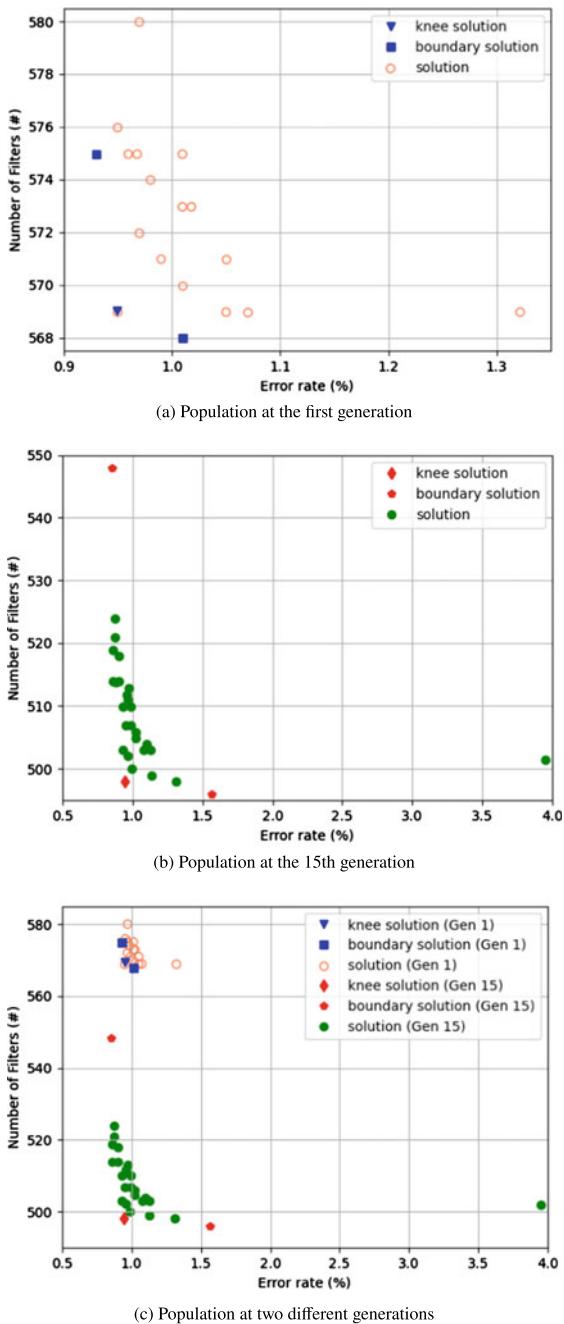
On the MNIST dataset, the method is first evaluated on a variant of LeNet. A fully convolutional LeNet is made up of four convolutional layers with sizes of $20 \times 1 \times 5 \times 5$, $50 \times 20 \times 5 \times 5$, $500 \times 50 \times 4 \times 4$ and $10 \times 500 \times 1 \times 1$, respectively. Because all layers, with the exception of the output layer, can be pruned, the search space for selecting a set of less significant filters is 2^{570} (there should always be 10 output classes) and obtaining an exact solution in an acceptable amount of time is unfeasible. Furthermore, the MNIST dataset is composed of a training set which contains 60,000 images and a test set which contains 10,000 images, which are hand-written digits in grey that correspond to ten different classes and have a dimension of 28×28 . The reported accuracy is obtained by evaluating it on the

test set. Although the test accuracy obtained on MNIST is almost saturated, it is nevertheless an extensively used dataset in the literature for investigating DNN compression problems [22, 33, 34] because the goal is not to push the state-of-the-art classification accuracy.

The training set of MNIST dataset is used to train the fully convolutional LeNet, which then obtains an accuracy of 99.03% on the test set after being trained on the training set. Based on computational cost consideration, the size of the KGEA population is set at 30, and each solution has a length that is identical to the number of convolutional filters contained in the CNN. To ensure algorithm convergence, the number of max iteration is set to 250 empirically. Based on previous studies [4, 8], 1,000 samples from the training set are randomly selected to evaluate the objectives in KGEA, which takes roughly 0.1 seconds in order to compute feedforward. Several different settings are compared and it is found that the current choice is capable of evaluating filter importance. The experiment takes about 1.3 hours to complete. Figure 16.7 shows a part of the evolution process. As seen in Fig. 16.7a, the solutions are initially distributed randomly in the objective space. The MMD approach is used to identify the knee solution, and the boundary solution is found by identifying the extreme points on the Pareto front. Due to the large number of parameters, the population indicates that none of the initial solutions are well compressed. As shown in Fig. 16.7b, much more filters have been pruned after 15 generations of evolution, while performance has remained stable. More crucially, the population search direction is directed to the knee regions, where the solutions can obtain a good balance of scale of parameters and performance. Solutions from both the 1st and 15th generations are shown in Fig. 16.7c to indicate how far the population has moved in the objective space. As can be observed, KGEA can push the population towards knee areas effectively while optimizing each of the objective simultaneously.

Figure 16.8 shows the population of the last generation. As can be observed, the knee solution obtains a satisfactory compromise between parameter scale and population performance. Experiments are conducted to compare KGEA with a variety of techniques for compressing the fully convolutional LeNet to demonstrate the effectiveness of KGEA for compressing CNNs, and Table 16.2 shows the results. The size of the parameters and accuracy in the baseline model are $4.31E + 05$ and 99.03%, respectively. Furthermore, the number of FLOPs is used to calculate the computational cost, assuming that the nonlinearity is calculated without cost and that the convolution is carried out using a sliding window. As can be observed, the knee solution has a parameter count of $8.05E + 04$ and an accuracy of 98.79%, implying that 64.32% of FLOPs are decreased and 81.34% of the parameters are removed from the baseline model. After being fine-tuned, the accuracy has been raised to 99.06%, which is higher than the baseline model. It is possible that the increased accuracy is due to the fact that small model often generalize better. In the meantime, the model after compression is trained from scratch and obtains a 99.02% accuracy. The performance is marginally lower compared with the baseline model, indicating that inheriting weights plays an important role in compression. Table 16.3 contains a detailed layer-wise analysis of the knee solution. More than half of the filters in the third convolutional layer are pruned, however, far fewer filters from the first two

Fig. 16.7 Populations from the evolution, where knee solution is obtained in each generation and utilized to lead the population search direction. **a** The first generation **b** The 15th generation and **c** Comparison of populations at the first and the 15th generations



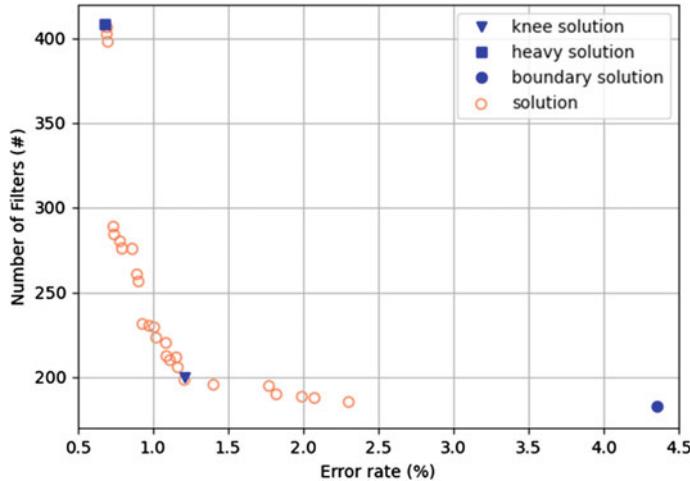


Fig. 16.8 The population obtained from KGEA when the algorithm terminated. Knee solution, the boundary solution with lowest error rate (heavy solution) and the other boundary solution are indicated by the triangle, square and circle, respectively

layers are pruned, implying that low layer filters are more significant than high layer filters.

Furthermore, for comparison, the weight sum (WS) [6], APoZ [4], and ThiNet [8] were chosen as competitive methods. First, setting the same pruning ratio as the knee solution (termed as *Pruning-ratio-constrained (Knee)* in Table 16.2), which is 81.34%. The ℓ_1 criterion is then used to prune the convolutional filters, yielding the WS. As can be observed, the accuracy of model which is compressed through the WS is 98.08%, which is significantly lower than the knee solution. The accuracy improves to 98.82% after fine-tuning, although it remains lower than that of the knee solution after fine-tuning. The lower performance could be due to the inability of the ℓ_1 criterion to effectively identify duplication in CNNs. On the other hand, every neuron in the sampled dataset is measured by APoZ in regard to the average percentage of zero activations, which is then utilized to indicate the importance of pruning parameters. Its performance, however, is inferior to both KGEA and WS. The weak performance of APoZ is most likely due to its limited ability to identify redundancy in low layers. ThiNet uses the sampled dataset for channel selection and formulates filter pruning as an optimization issue with the goal of minimizing the error generated during reconstruction in two successive layers. As can be observed, the performance is poorer than KGEA, perhaps because local information is the only thing considered by ThiNet. Take note that although a non-structured sparse model [3] is able to obtain far better results, it needs specialized software or hardware, and there is a restriction placed on the acceleration in practice. [8]. Furthermore, the computational cost of adding conditional mask checking or transforming weight

Table 16.2 On the MNIST dataset, multiple compression strategies for fully convolutional LeNet are compared. The accuracy of the baseline model, FLOPs, and number of parameters are given first. Then the knee solution is evaluated. “Accuracy-guaranteed (Knee)” indicates the accuracy is guaranteed to be identical to that of the knee solution, and the “Pruning-ratio-constrained (Knee)” indicates the pruning ratio equals that of the knee solution

Method	Model (Fully Convolutional LeNet)	Accuracy (%)	Parameters	Pruned (%)	FLOP	Pruned (%)
KGEA	Baseline model	99.03	4.31E+05	\	8.81E+05	\
	Knee solution	98.79	8.05E+04	81.34	3.14E+05	64.32
	Knee fine-tuned	99.06				
WS [6]	Knee scratch-train	99.02				
	Pruning-ratio-constrained (Knee)	98.08				
APoZ [4]	Pruning-ratio-constrained fine-tuned (Knee)	98.82				
	Pruning-ratio-constrained (Knee)	97.75				
ThiNet [8]	Pruning-ratio-constrained fine-tuned (Knee)	98.72				
	Pruning-ratio-constrained (Knee)	98.35				
WS [6]	Pruning-ratio-constrained fine-tuned (Knee)	98.48				
	Accuracy-guaranteed (Knee)	98.79	2.30E+05	46.68	6.01E+05	31.78
	Accuracy-guaranteed fine-tuned (Knee)	99.02				
APoZ [4]	Accuracy-guaranteed scratch-train (Knee)	98.81				
	Accuracy-guaranteed (Knee)	98.79	2.49E+05	42.13	6.13E+05	30.45
	Accuracy-guaranteed fine-tuned (Knee)	99.01				
ThiNet [8]	Accuracy-guaranteed scratch-train (Knee)	98.85				
	Accuracy-guaranteed (Knee)	98.79	2.09E+05	51.41	5.73E+05	34.99
	Accuracy-guaranteed fine-tuned (Knee)	99.02				
	Accuracy-guaranteed scratch-train (Knee)	98.96				

Table 16.3 Layer-wise analysis of the knee solution on compressing the fully convolutional LeNet

Network structure (Fully Convolutional LeNet)			Original model			Pruned model (Knee solution)			Pruned (%)	
Layer type	Filter size	Maps size	# Maps	# Parameters	FLOPs	# Maps	# Parameters	FLOPs	Parameters	FLOPs
Conv_1	5 × 5	12 × 12	20	5.20E+02	7.49E+04	16	4.16E+02	5.99E+04	20.00	20.00
Conv_2	5 × 5	4 × 4	50	2.51E+04	4.01E+05	29	1.16E+04	1.86E+05	53.58	53.58
Conv_3	4 × 4	1 × 1	500	4.01E+05	4.01E+05	144	6.70E+04	6.70E+04	83.28	83.28
Linear	\	1	10	5.01E+03	5.01E+03	10	1.45E+03	1.45E+03	71.06	71.06
Total				4.31E+05	8.81E+05		8.05E+04	3.14E+05	81.34	64.32

matrix to sparse representation may be considerably more expensive than the cost of original matrix multiplication with redundancy [4].

KGEA and WS prune both 2 and 3, however further pruning filters 1 and 4 will lead to inferior performance compared with pruning filters 5 and 6, as shown in Table 16.2. Although KGEA achieves better accuracy than the WS approach with the identical pruning ratio for parameter, the performance of compressing under the identical accuracy restriction still needs to be explored. The WS, APoZ, and ThiNet are used to prune as many filters from the baseline model as possible, achieving a 98.79% accuracy, which is identical to the knee solution (termed as *Accuracy-guaranteed (knee)* in Table 16.2). As a consequence, the performance of compression using WS on FLOPs and pruning parameters is 31.78% and 46.68%, respectively, which is worse than the knee solution produced by KGEA. Because of their data-driven nature, the ThiNet and APoZ are capable of pruning more parameters than WS. Noting that the best performance of boundary solution achieves a 99.33% accuracy, which is 0.3% better than the baseline model with no fine-tuning. The supplemental material contains more information on the boundary solution.

The best performance of boundary solution in the population is known as *heavy solution*, as illustrated in Table 16.4, since it has the least number of parameters reduced among all solutions in the population. As can be shown, fine-tuning or training has no effect on accuracy, implying that the performance improvement is primarily due to filter pruning. However, the reduction in the number of parameters and FLOPs is limited, implying that the model remains a storage-intensive and computationally expensive one. The performance of ThiNet [8], APoZ [4], and WS [6] are further investigated by using the same compression ratio as the *heavy solution* (termed as *Pruning-ratio-constrained* in Table 16.4). After pruning, WS, APoZ, and ThiNet acquired accuracies of 98.97%, 99.05%, and 99.07%, respectively, which are lower than the *heavy solution* obtained by KGEA. Fine-tuning improves them to 99.08%, 99.11%, and 99.15%, respectively, but they are still not as good as the *heavy solution*.

Due to the random nature of KGEA, the mean and deviation of its performance are also observed, which are displayed in Table 16.5. Because KGEA shares many fundamentals with it except the mechanism of knee guidance, NSGA-II [10] equipped with MMD [29] is chosen as a representative of conventional MOEAs to be compared. As can be seen, KGEA outperforms previous MOEAs on both knee and heavy solutions, demonstrating that the method is more effective than existing MOEAs.

16.4.3 Experiments on VGG-19

The performance of KGEA to compress VGG-19 [35] network using CIFAR-10 dataset is examined to show the efficacy of KGEA in compressing deep CNNs with a vast scale of parameters. VGG-19 is a large-scale network with 16 convolutional layers and 3 fully connected layers. For training and testing, the CIFAR-10 dataset contains 50,000 and 10,000 color images with the size of 32×32 in 10 classes,

Table 16.4 Comparisons of various methods to compress LeNet on MNIST dataset. “Heavy solution” represents the boundary solution achieves the lowest error rate, and “Pruning-ratio-constrained” represents the pruning ratio is identical to that in the heavy solution. Acc, PR and FT represent Accuracy, Pruning ratio and fine-tuned, respectively

Method	Model (Fully Conv LeNet)	Acc (%)	PR (%)
	Baseline model	99.03	\
KGEA	Heavy solution	99.33	38.11
	Heavy solution FT	99.33	
WS [6]	Pruning-ratio-constrained	98.97	
	Pruning-ratio-constrained FT	99.08	
APoZ [4]	Pruning-ratio-constrained	99.05	
	Pruning-ratio-constrained FT	99.11	
ThiNet [8]	Pruning-ratio-constrained	99.07	
	Pruning-ratio-constrained FT	99.15	

Table 16.5 Statistical results of KGEA for compressing LeNet on MNIST which consists of “Mean±deviation”

Methods	Model	Accuracy	PR(%)
KGEA	Knee solution	98.75±0.11	81.32±1.25
	Knee fine-tuned	99.07±0.05	
	Knee scratch-train	99.02±0.25	
NSGA-II+MMD	Knee solution	98.17±0.14	75.45±1.41
	Knee fine-tuned	99.03±0.10	
	Knee scratch-train	99.02±0.23	
KGEA	Heavy solution	99.31±0.15	38.10±0.43
	Heavy fine-tuned	99.33±0.02	
NSGA-II+MMD	Heavy solution	99.12±0.17	35.51±0.45
	Heavy fine-tuned	99.30±0.09	

Table 16.6 On the CIFAR dataset, multiple compression strategies for VGG-19 are compared. The accuracy of the baseline model, number of parameters, and FLOPs are given first. Then the knee solution is evaluated. The “Accuracy-guaranteed (Knee)” indicates the accuracy is guaranteed to be identical to that of the knee solution, and “Pruning-ratio-constrained (Knee)” indicates the pruning ratio is identical to that of the knee solution

Method	Model (VGG-19)	Accuracy (%)	Parameters	FLOP	Pruned (%)
KGEA	Baseline model	92.42	2.05E+07	\	3.90E+08
	Knee solution	91.35	4.34E+06	78.88	1.55E+08
	Knee fine-tuned	92.83			61.17
WS [6]	Knee scratch-train	91.67			
	Pruning-ratio-constrained (Knee)	91.15			
	Pruning-ratio-constrained (Knee) fine-tuned	91.88			
APoZ [4]	Pruning-ratio-constrained (Knee)	91.03			
	Pruning-ratio-constrained (Knee) fine-tuned	91.72			
	Pruning-ratio-constrained (Knee)	91.24			
ThiNet [8]	Pruning-ratio-constrained (Knee) fine-tuned	92.48			
	Accuracy-guaranteed (Knee)	91.35	5.53E+06	73.11	2.20E+08
	Accuracy-guaranteed (Knee) fine-tuned	92.40			44.93
APoZ [4]	Accuracy-guaranteed (Knee) scratch-train	91.38			
	Accuracy-guaranteed (Knee)	91.35	6.30E+06	69.36	2.39E+08
	Accuracy-guaranteed (Knee) fine-tuned	92.32			39.99
ThiNet [8]	Accuracy-guaranteed (Knee) scratch-train	92.11			
	Accuracy-guaranteed (Knee)	91.35	5.28E+06	74.31	2.17E+08
	Accuracy-guaranteed (Knee) fine-tuned	92.46			45.45
	Accuracy-guaranteed (Knee) scratch-train	92.40			

respectively. The accuracy is determined only by the test set. On the training set, the VGG-19 is first trained and obtain a 92.42% accuracy. After that, it is used as the baseline model for compression. In the VGG-19, the total amount of convolutional filters exceeds 5,000, resulting in a massive search space for finding a promising solution in an acceptable amount of time. To solve this problem, the convolutional filters are divided into some groups and KGEA is applied to each of them separately. The filter number in each group must be less than 1,000, according to a basic principle. Particularly, the first group contains the first four convolutional layers, followed by the 5th and 6th convolutional layers as the second group. For the remaining layers, the same settings are used. The population is set to 30 initially, and using the filter number in each group to determine the length of each solution. Furthermore, because the divided search space is smaller than the search space of the fully convolutional LeNet, 200 is set as the maximum iteration number. To speed up the experiment, sampling is used, where 1,000 samples are evenly selected from the training set, and it takes roughly 0.5 seconds to evaluate objective values one time. Because the network has a large number of parameters in the network are involved in the inference stage computation, the experiment takes roughly 5 hours to complete.

Table 16.6 shows the investigated performance of the knee solution and the comparison to that of other strategies. The knee solution, as can be observed, obtains a 61.17% FLOPs reduction and a 78.88% parameter reduction. In the meantime, because a huge number of parameters are pruned, it obtains slightly inferior performance compared to the baseline model, but this is able to be recovered to outperform the baseline model with fine-tuning (termed as *Knee fine-tuned* in Table 16.6). It means that a pruned model is capable of achieving higher accuracy than the baseline model. In terms of accuracy, the knee solution model which is trained from scratch (termed as *Knee scratch-train* in Table 16.6) performs somewhat worse than the baseline model, indicating that the performance is significantly affected by the inheritance of weight from the baseline model.

The efficiency of KGEA for VGG-19 compression is also compared to that of ThiNet, APoZ, and WS approaches. To prune the parameters, a similar setting in [6] is used and set the compression ratio so that it is identical to the knee solution (termed as *Pruning-ratio-constrained (Knee)* in Table 16.6). As can be observed, the model pruned by ThiNet, APoZ, and WS all perform worse than the model pruned by KGEA, implying that KGEA is capable of providing a more robust way for identifying redundant parameters and finding a solution with better accuracy than ThiNet, APoZ, and WS. In the meantime, the compression ratio of various methods under the same accuracy guarantee (termed as *Accuracy-guaranteed (Knee)* in Table 16.6) is investigated. The ThiNet, APoZ, and WS are used to compress the baseline model, and the compression ends when the accuracy is equal to that of the knee solution. The results show that the size of parameters in the compressed models is much higher than in the knee solution model, indicating that KGEA outperforms other competitive approaches in identifying redundant filters.

Because the KGEA can simultaneously optimize each objective, it is going to be quite interesting to look into the boundary solution that has the best performance found in the evolution, while being less concerned with the number of parameters

Table 16.7 On the CIFAR-10 dataset multiple compression strategies for VGG-19 are compared. “Heavy solution” represents the boundary solution achieves the lowest error rate, and “Pruning-ratio-constrained” indicates the pruning ratio is identical to that in the heavy solution. Acc, PR and FT represent Accuracy, Pruning ratio and fine-tuned, respectively

Method	Model (VGG-19)	Acc (%)	PR (%)
	Baseline model	92.42	\
KGEA	Heavy solution	93.28	57.86
	Heavy solution FT	93.28	
WS [6]	Pruning-ratio-constrained	91.22	
	Pruning-ratio-constrained FT	92.49	
APoZ [4]	Pruning-ratio-constrained	91.24	
	Pruning-ratio-constrained FT	92.52	
ThiNet [8]	Pruning-ratio-constrained	91.36	
	Pruning-ratio-constrained FT	92.65	

that have been lowered. The accuracy of the baseline model can be enhanced by removing a certain number of redundant filters, without any fine-tuning, similar to the case of compressing LeNet. The supplemental material contains more details on the boundary solution.

In conclusion, KGEA is capable of automatically finding a knee solution for CNNs that prunes a large number of parameters while retaining a high performance. Furthermore, when compared with the results achieved using ThiNet, APoZ, and WS, the filters kept by KGEA are more informative and can obtain a higher accuracy with the identical pruning ratio. Finally, KGEA is capable of finding several slightly compressed solutions which can obtain higher accuracy without fine-tuning, providing a direction for further investigation into the redundancy in CNNs.

Because the number of parameters is usually less than other solutions, the boundary solution in the population with the best performance is termed as *heavy solution*, which is illustrated in Table 16.7. As can be seen, the heavy solution can enhance accuracy without fine-tuning on the test dataset. Furthermore, fine-tuning does not improve accuracy any further, demonstrating that removing redundant parameters can improve the performance of the original model and the existence of redundancy in deep models. Meanwhile, the accuracy of the original model is significantly improved by WS, APoZ, and ThiNet, which can be attributed to the small pruning ratio, and it also illustrates their efficiency in identifying a small number of redundant parameters.

Table 16.8 Statistical results of KGEA for compressing VGG-19 on CIFAR-10 which consists of “Mean±deviation” (PR represents pruning ratio)

Method	Model	Accuracy	PR(%)
KGEA	Knee solution	91.37(±0.15)	78.72(±1.12)
	Knee fine-tuned	92.80(±0.09)	
	Knee scratch-train	91.65(±0.27)	
NSGA-II+MMD	Knee solution	90.58(±0.18)	73.43(±1.41)
	Knee fine-tuned	91.60(±0.12)	
	Knee scratch-train	91.04(±0.23)	
KGEA	Heavy solution	93.26(±0.17)	57.76(±0.24)
	Heavy fine-tuned	93.27(±0.03)	
NSGA-II+MMD	Heavy solution	93.10(±0.14)	53.80(±0.38)
	Heavy fine-tuned	93.18(±0.16)	

In addition, the mean and deviation of its performance are investigated and presented in Table 16.8. As can be seen, KGEA outperforms NSGA-II [10] equipped with MMD [29] on both knee and heavy solutions, demonstrating the effectiveness of the method.

Additionally, the knee solution obtained from the KGEA is analyzed layer-by-layer, as shown in Table 16.9. The results demonstrate that most of the parameters are reduced in the higher layers, indicating that lower-layer convolutional filters are more informative than higher-layer convolutional filters. This is consistent with the LeNet. The parameters in the last two fully connected layers, in particular, are not involved in the compression. There are several reasons why the fully connected layers should not be pruned. To begin, pruning them can result in the weight matrix [33] with irregular sparsity, which is troublesome for deployment on present deep learning frameworks. Then, the last two layers contribute very little to the overall computing cost of the CNN, and many existing CNN architectures lack fully connected levels altogether [36]. Furthermore, this strategy is based on previous studies and has been widely used for filter pruning [6, 8].

16.5 Chapter Summary

In this chapter, we introduced KGEA for DNN pruning, which is also a kind of DNN compression technique. Unlike the DeepPruningES algorithm introduced in the previous chapter, the KGEA algorithm concerns multiple conflicting objectives to find the Pareto front. In particular, the front consists of a set of balanced solutions in terms of both the parameter scale and the model performance, which could give DMs more choices based on their preferences. KGEA is designed based on the knee concept in the community of multi-objective optimization, which has the merit of

Table 16.9 Layer-wise analysis of the knee solution on compressing VGG-19

Network structure (VGG-19)		Original model			Pruned model (Knee)			Pruned (%)	
Layer type	Maps size	# Maps	# Parameters	FLOPs	# Maps	# Parameters	FLOPs	Parameters	FLOPs
Conv_1_1	32 × 32	64	1.73E+03	1.77E+06	52	1.40E+03	1.44E+06	18.75	18.75
Conv_1_2	32 × 32	64	3.69E+04	3.77E+07	37	1.73E+04	1.77E+07	53.03	53.03
Conv_2_1	16 × 16	128	7.37E+04	1.89E+07	89	2.96E+04	7.59E+06	59.80	59.80
Conv_2_2	16 × 16	128	1.47E+05	3.77E+07	82	6.57E+04	1.68E+07	55.46	55.46
Conv_3_1	8 × 8	256	2.95E+05	1.89E+07	216	1.59E+05	1.02E+07	45.95	45.95
Conv_3_2	8 × 8	256	5.90E+05	3.77E+07	213	4.14E+05	2.65E+07	29.80	29.80
Conv_3_3	8 × 8	256	5.90E+05	3.77E+07	207	3.97E+05	2.54E+07	32.72	32.72
Conv_3_4	8 × 8	256	5.90E+05	3.77E+07	202	3.76E+05	2.41E+07	36.20	36.20
Conv_4_1	4 × 4	512	1.18E+06	1.89E+07	180	3.27E+05	5.24E+06	72.26	72.26
Conv_4_2	4 × 4	512	2.36E+06	3.77E+07	175	2.84E+05	4.54E+06	87.98	87.98
Conv_4_3	4 × 4	512	2.36E+06	3.77E+07	180	2.84E+05	4.54E+06	87.98	87.98
Conv_4_4	4 × 4	512	2.36E+06	3.77E+07	198	3.21E+05	5.13E+06	86.40	86.40
Conv_5_1	2 × 2	512	2.36E+06	9.44E+06	184	3.28E+05	1.31E+06	86.10	86.10
Conv_5_2	2 × 2	512	2.36E+06	9.44E+06	226	3.74E+05	1.50E+06	84.14	84.14
Conv_5_3	2 × 2	512	2.36E+06	9.44E+06	174	3.54E+05	1.42E+06	85.00	85.00
Conv_5_4	2 × 2	512	2.36E+06	9.44E+06	164	2.57E+05	1.03E+06	89.11	89.11
Linear	\	512	2.62E+05	2.62E+05	512	8.40E+04	8.40E+04	67.97	67.97
Linear	\	512	2.62E+05	2.62E+05	512	2.62E+05	2.62E+05	0.00	0.00
Linear	\	10	5.12E+03	5.12E+03	10	5.12E+03	5.12E+03	0.00	0.00
Total			2.05E+07	3.99E+08		4.34E+06	1.55E+08	78.88	61.17

good effectiveness and efficiency compared to the peer competitors. The investigation of KGEA was conducted on the fully convolutional LetNet and also the VGG-19. The results justified the design motivation of KGEA.

At the beginning of this part, we introduced the E2EPP algorithm which could speed up the running of NAS algorithms through the learning process. On the other hand, if there are enough computational resources, i.e., GPUs, utilizing these computation resources in a parallel way is also an alternative. However, most existing parallel techniques are designed for CPUs. NAS, as an emerging topic, also requires designing such kind of parallel technique based on GPUs. In the next chapter, we will introduce such an algorithm designed for ENAS algorithms.

References

1. Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., & Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 1269–1277).
2. Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016). Xnor-net: classification using binary convolutional neural networks. In *Proceedings of European Conference on Computer Vision (ECCV)* (pp. 525–542). Springer.
3. Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

4. Hu, H., Peng, R., Tai, Y.-W., & Tang, C.-K. (2016). *Network trimming: A data-driven neuron pruning approach towards efficient deep architectures*. [arXiv:1607.03250](https://arxiv.org/abs/1607.03250).
5. Molchanov, P., Tyree, S., Karras, T., Aila, T., & Kautz, J. (2017). Pruning convolutional neural networks for resource efficient transfer learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
6. Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H. P. (2017). Pruning filters for efficient convnets. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
7. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324. ISSN 0018-9219. <https://doi.org/10.1109/5.726791>.
8. Luo, J.-H., Wu, J., & Lin, W. (2017). ThiNet: A filter level pruning method for deep neural network compression. In *2017 IEEE International Conference on Computer Vision (ICCV)* (pp. 5068–5076). IEEE. ISBN 978-1-5386-1032-9. <https://doi.org/10.1109/ICCV.2017.541>, <http://ieeexplore.ieee.org/document/8237803/>.
9. Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 1135–1143).
10. Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. A. M. T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2), 182–197.
11. Zhang, Q., & Li, H. (2007). MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6), 712–731.
12. Gong, Z., Chen, H., Yuan, B., & Yao, X. (2018). Multiobjective learning in the model space for time series classification. *IEEE Transactions on Cybernetics*. <https://doi.org/10.1109/TCYB.2018.2789422>.
13. LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 598–605).
14. Jin, J., Dundar, A., Culurciello, E. (2014). Flattened convolutional neural networks for feed-forward acceleration. [arXiv:1412.5474](https://arxiv.org/abs/1412.5474).
15. Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep learning with limited numerical precision. In *Proceedings of International Conference on Machine Learning (ICML)* (pp. 1737–1746).
16. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized neural networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 4107–4115).
17. Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the knowledge in a neural network. [arXiv:1503.02531](https://arxiv.org/abs/1503.02531).
18. Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., Bengio, Y. (2015). Fitnets: Hints for thin deep nets. In *Proceedings of the International Conference on Learning Representations (ICLR)*.
19. Cheng, Y., Wang, D., Zhou, P., & Zhang, T. (2018). Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1), 126–136.
20. Lin, J., Rao, Y., Lu, J., & Zhou, J. (2017). Runtime neural pruning. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)* (pp. 2178–2188).
21. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S. & Zhang, C. (2017b). Learning efficient convolutional networks through network slimming. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)* (pp. 2755–2763).
22. Li, G., Qian, C., Jiang, C., Lu, X., & Tang, K. (2018). Optimization based layer-wise magnitude-based pruning for dnn compression. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)* (pp. 2383–2389).
23. Lin, M., Chen, Q., & Yan, S. (2014). Network in network. In *Proceedings of the 2014 International Conference on Learning Representations*.

24. Zhang, X., Zhou, X., Lin, M., & Sun, J. (2017). Shufflenet: An extremely efficient convolutional neural network for mobile devices. [arXiv:1707.01083](https://arxiv.org/abs/1707.01083).
25. Deb, K., & Jain, H. (2014). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4), 577–601.
26. Gong, D., Sun, J., & Miao, Z. (2018). A set-based genetic algorithm for interval many-objective optimization problems. *IEEE Transactions on Evolutionary Computation*, 22(1), 47–60.
27. Gong, D., Liu, Y., & Yen, G. G. (2018a). A meta-objective approach for many-objective evolutionary optimization. *Evolutionary Computation*, 1–25.
28. Wang, Y., Xu, C., Qiu, J., Xu, C., & Tao, D. (2018e). Towards evolutionary compression. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 2476–2485). ACM. ISBN 978-1-4503-5552-0. <https://doi.org/10.1145/3219819.3219970>, <https://dl.acm.org/doi/10.1145/3219819.3219970>.
29. Chiu, W.-Y., Yen, G. G., & Juan, T.-K. (2016). Minimum manhattan distance approach to multiple criteria decision making in multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 20(6), 972–985. ISSN 1089-778X, 1089-778X, 1941-0026. <https://doi.org/10.1109/TEVC.2016.2564158>, <http://ieeexplore.ieee.org/document/7465803/>.
30. Yi, J., Bai, J., He, H., Peng, J., & Tang, D. (2018). Ar-moea: A novel preference-based dominance relation for evolutionary multi-objective optimization. *IEEE Transactions on Evolutionary Computation*. <https://doi.org/10.1109/TEVC.2018.2884133>
31. Gong, D., Sun, J., & Ji, X. (2013). Evolutionary algorithms with preference polyhedron for interval multi-objective optimization problems. *Information Sciences*, 233, 141–161.
32. Gong, D., Sun, F., Sun, J., & Sun, X. (2017). Set-based many-objective optimization guided by a preferred region. *Neurocomputing*, 228, 241–255.
33. Ding, X., Ding, G., Han, J., & Tang, S. (2018). Auto-balanced filter pruning for efficient convolutional neural networks. In *AAAI Conference on Artificial Intelligence*. <https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16450>.
34. Kung, H. T., McDanel, B., & Zhang, S. Q. (2019). Packing sparse convolutional neural networks for efficient systolic array implementations: column combining under joint optimization. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
35. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
36. He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).

Chapter 17

Distribution Training Framework for Architecture Design



17.1 Introduction

As discussed in Part I, for the time-consuming issue of the ENAS methods, there are two primary categories of available acceleration methods. First, various acceleration approaches for DNN evaluation are proposed, including weights sharing [1], low fidelity [2], and prediction model [3]. However, the reduction of training in these methods introduces bias in the estimation of DNN performance. Second, many well-known distributed parallelism techniques have been developed to accelerate large-scale DNN training, the most common ones include model parallelism and data parallelism. Specifically, for model parallelism, it mainly involves partitioning a DNN model into multiple parts (e.g., different layers in a DNN are assigned to different nodes), and then each node computes a specific part of the model during model training. Correspondingly, data parallelism refers to dividing a very large dataset into multiple smaller ones (generally distributed mostly equally) and then assigning them to different computational nodes for the purpose of simultaneous training on multiple different nodes. However, both techniques focus on training a single large-scale DNN and thus can seriously increase the communication cost among different nodes. Moreover, the distributed training of a DNN may cause model inconsistency problem, which is unsuitable for the training of ENAS. Furthermore, some mature distributed training platforms have been widely used. For example, the introduction of MPI into distributed TensorFlow proposed by Vishnu et al., and the Horovod distributed training platform developed and designed by Uber Inc. These distributed techniques are very advanced, but are difficult to be reproduced because of the budget constraints involved such as high performance computers and Infiniband networks [4].

In conclusion, the expensive time cost has become a fundamental problem limiting the development of ENAS, and the existing acceleration technology does not fit well with ENAS. Therefore, this chapter introduces a new distributed framework to reduce the consumption of ENAS training in terms of time cost. The framework consists

of servers as well as computational nodes together. To the best of our knowledge, this is the first work to design a distributed framework of ENAS for accelerated training. Specifically, in the framework, the computational nodes are responsible for training and evaluating the architecture (the most time-consuming phase in the whole ENAS), while the server is responsible for the rest of the work, including algorithm startup, other operations in the evolution algorithm, etc. In order to maximize the use of limited resources, all nodes are connected via socket communication and asynchronous working. The following are the specific contributions:

1. Build a distributed framework using the popular master-slave architecture and develop an asynchronous communication scheme from the ground up. To guarantee an asynchronous data exchange, the server maintains two processes for each computing node. The architecture can simply be modified to meet the computational needs of a specific algorithm.
2. To facilitate communication between populations in EAs, a new packet structure is constructed to be able to encapsulate the structural information of DNNs in the population. This framework can be employed by different ENAS algorithms by encapsulating multiple network representations into the packet structure.
3. To investigate the performance of the distributed framework, an algorithm called EA-Pelee is designed to perform image classification. The efficiency and scalability of the framework are demonstrated by multiple perspective analysis of the experimental results.

The following is the structure of this chapter. In Sect. 17.2, the related work of distributed deep learning is reviewed. The details of the distributed framework is illustrated in Sect. 17.3. Section 17.4 documents the experimental design and results analysis.

17.2 Distributed Deep Learning

Data parallelism is achieved by placing copy of the architecture in separate computing nodes and then computing a portion of the data in the training set in each node. After then, well-designed methods need to synchronize, aggregate, and disperse the parameters calculated by each worker. As demonstrated in Fig. 17.2, the parameter server collects all sub gradients (∇w) of each worker, calculates new parameters (w_{i+1}), and distributes them to the workers. In terms of parameter updates, the most common of the data parallelism is synchronous stochastic gradient descent [5, 6], which is very simple to implement but leads to its inefficiency in asynchronous scenarios, as all workers need to wait for the slowest one to finish before the next iteration can proceed. Additional network communication and synchronization expenses, in particular, may outweigh the benefits gained from the additional workers. Another improved technique, known as asynchronous SGD [7, 8], addresses this flaw by removing any explicit synchronization amongst workers. However, this method brought a new problem known as “gradient staleness”, which can cause the model to converge slowly or

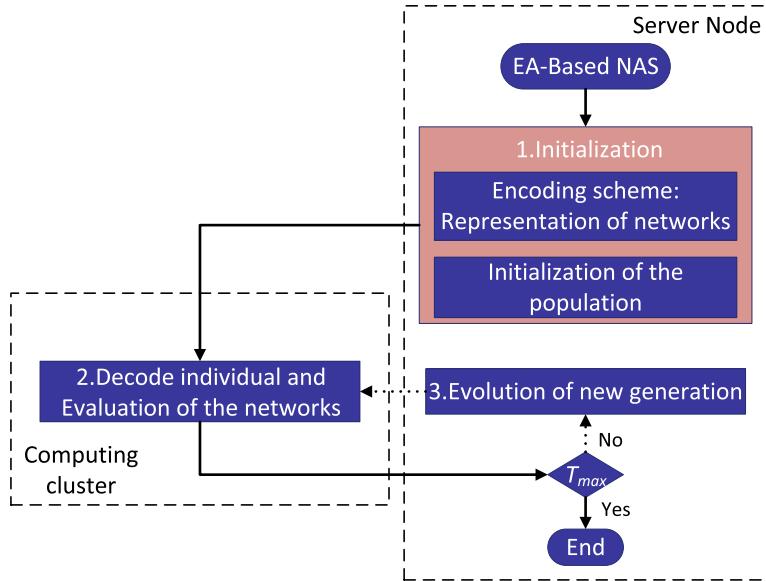


Fig. 17.1 The overall view of the ENAS algorithms. The time-consuming evaluation stage is delegated to a computing cluster in the distributed framework. In the meantime, the server node will handle the initialization and evolution of the ENAS. In Sect. 17.3, the details of the framework will be specified

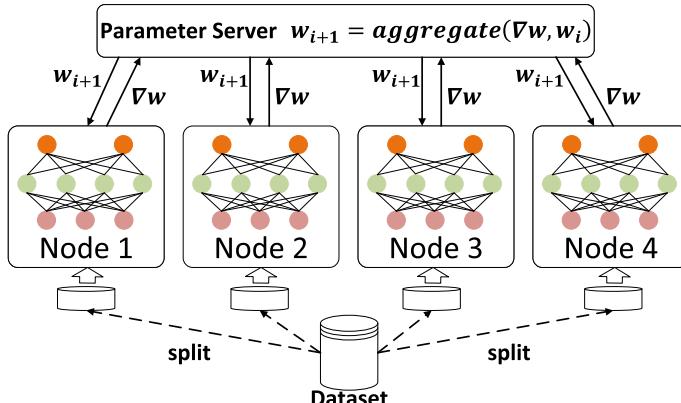


Fig. 17.2 The illustrative architecture of the data parallelism

even non-convergence. The structure of the model parallelism is shown in detail in Fig. 17.3, where each node holds a part of the DNN, while the same set of train data is replicated to all nodes separately. Since the networks are not in the same node, the relevant parameters need to be communicated between different nodes, which

Fig. 17.3 The illustrative architecture of the model parallelism

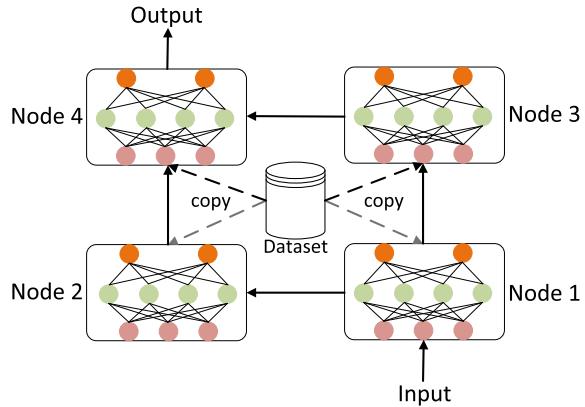


Table 17.1 The summary of existed acceleration methods for the ENAS

Category	Method	Focus
Model evaluation	Surrogate [3]	A single DNN
	Early stopping [11]	
	Weights sharing [1]	
	Low fidelity [2]	
Distributed training	Data Parallelism [12]	*A large-scale DNN
	Model Parallelism	
Distributed platform	Horovod [10], PyTorch .	A single DNN

makes the framework more difficult to implement and leads to the fact that model parallelism has not been widely promoted.

In particular, the distributed module [9], which includes data parallelism and model parallelism, is also provided in popular deep learning platforms such as PyTorch and TensorFlow. The Horovod [10] distributed library mentioned above, for example, enables distributed operations in TensorFlow by adding or modifying just a few lines of code. However, these methods involve multiple computing nodes, so the expensive communication overhead still does not guarantee that model training can be accelerated. Furthermore, a few classical NAS algorithms are also integrated in several AutoML toolkits, such as NNI and Paddleslim. However, instead of accelerating ENAS methods, their purpose is to provide interfaces for customers to easily deploy their NAS algorithms in different environments.

To conclude, Table 17.1 summarizes the aforementioned acceleration methods. Most of the acceleration strategies focus on DNN evaluation or improvement algorithms, so these strategies are generalized and can be used in any environment. However, there are few distributed frameworks specific to ENAS. Based on this, a distributed framework is designed that can effectively combine ENAS to solve the time-consuming problem. The framework is based on the master-slave model that

accelerates the ENAS optimization process by assigning individuals (DNN framework) to an extensible computing cluster, as illustrated in Fig. 17.1. On the other hand, the server node can be responsible for the initialization and evolutionary operations in the ENAS algorithm. Inspired by asynchronous SGD, all computational nodes work asynchronously as a way to maximize the utilization of computational resources. In addition, the framework does not fix which EA to use, so any ENAS can implement distributed training by calling the interface. In addition, the acceleration strategies mentioned above can still be used in the distributed framework.

17.3 The Distributed Framework

This section goes into detail regarding the motivation behind the distributed framework as well as its implementation.

17.3.1 Motivation

The NAS has been widely implemented using EAs with promising results. However, because the evaluation of each individual requires comprehensive validation on the target dataset and the total number of individuals is quite large, the issue of enormous computational resources and time consumption is unavoidable. Most of the acceleration methods that have emerged at this stage focus more on the evaluation of individual architecture rather than parallel processing of all architectures. In EA, the network architectures appear as populations (i.e., population contains information of multiple network architectures) and are therefore more suitable for a parallel technology, e.g., increasing a graphical processing unit to hundreds, which can then greatly speed up the ENAS computation.

In ENAS, the time-consuming evaluation of raw individuals can be characterized as a producer-consumer problem [13]. Two sorts of workers are comprised in the so-called producer-consumer problem. One is the task producer, and the other is the task consumer. The relationship between producer and consumer is interrelated and independent at the same time, and is generally decoupled using shared memory. For both, the producer only cares about task storage, while the consumer cares about task retrieval, and both operate in shared memory. In ENAS, the EA-related evolutionary process is modeled as the producer, while the individual fitness value evaluation is the consumer. Therefore, theoretically, the more consumers in the framework, the faster complete the tasks. As a result, increasing consumer capacity can effectively minimize the amount of time consumed.

Inspired by the producer-consumer model, a dual-mode distributed framework is design, i.e., with both types of nodes, to accelerate the optimization process of ENAS. The server nodes (i.e., producers) and computational nodes (i.e., consumers) in the framework are used for evolutionary operations and individual evaluation,

respectively. Sections 17.3.4 and 17.3.5 present the detailed implementations of the server and computing nodes, respectively.

17.3.2 Framework Overview

Figure 17.4 depicts the overall architecture of the framework. Specifically, the **server node** is in charge of population evolution and cluster management, whilst a group of computing nodes is in charge of individual evaluation. There are many GPU cards available in each **computing node** for DNN training. Since there are multiple nodes in the framework, there are no special requirements for the performance of **computing node** in the framework in order to facilitate the addition of additional nodes, and all **computing node** can run asynchronously. As a result, enhancing the cluster by adding computers with varying performance is easy. In addition, in order to facilitate the communication between **computing nodes** and **server node**, the communication data is packaged in the form of data structures and use the socket communication interface based on the TCP/IP protocol to realize this function, as shown in Sect. 17.3.3.

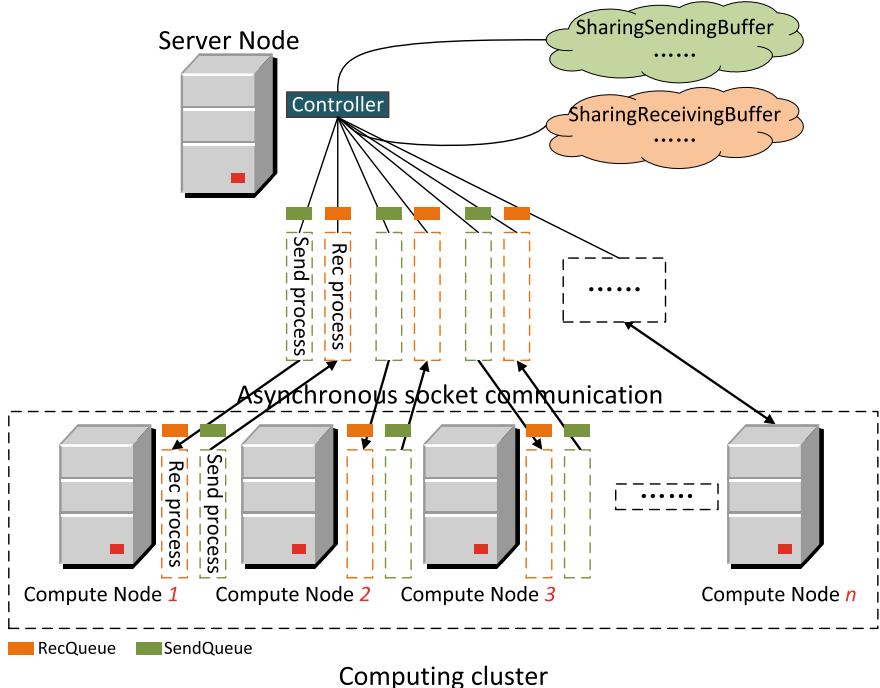


Fig. 17.4 The overall view of the distributed training architecture

The **server node** is equipped with a couple of sharing buffers, which are used to store the data that is going to be traded, as seen in Fig. 17.4. During each iteration, all individual architecture can be divided into two categories according to whether or not the fitness calculation has been performed (i.e., whether or not they have been trained), and the architectures with known fitness values are stored in the shared *SendingBuffer*, and vice versa in the shared *ReceivingBuffer*. The connection between the **computing node** and the **server nodes** is then established through socket communication. Once the connection is established, the server and computing nodes will launch a pair of processes for communication and message transmission that are corresponding to each other. As indicated in Fig. 17.4, a *RecQueue* will be generated to provide storage for the *RecProcess*, while a *SendQueue* will be provided for the *SendProcess*. The *RecProcess*, in particular, is in charge of receiving all data packets delivered by the other side and pushing them into the *RecQueue*. Correspondingly, the *sendProcess* gets a packet in the *SendQueue* and sends it to the other side. Note that both types of processes run independently when receiving and sending packets and do not depend on each other, and monitor if there are packets waiting to be processed. In addition, the control process in the **serve nodes** performs a number of tasks, including the post-processing of the accepted packets, generating packets of unevaluated architectures, performing the operations of the EA, etc.

17.3.3 Definition of the Data Packet

In ENAS, coding the network efficiently is the key to drive the EA to work. Most commonly, binary encoding strategy is used to represent the network architecture, such as in Genetic CNN [14], where 0 to indicate that there is no connectivity between network layers and 1 to indicate that the two are connected. The DAGs were utilized to represent the network in Cartesian GP (CGP) [15]. Furthermore, cellular encoding [16] encoded a family of NNs into a set of labeled trees using a simple graph grammar. To express the DNNS, some works have recently used an indirect encoding strategy [17, 18]. Elske et al., Lamarckian, for example, presented network morphism operators acting on the space of NNs that retain the function a network represents, obviating the need for retraining and lowering the necessary training time per network significantly.

Packets are very important in the distributed framework as communication information between different nodes. In addition, to make the framework effective for different EAs, a novel packet structure to encapsulate different chromosomes is designed. The data structure contains four members, namely *operation_type*, *ind_len* (individual length), *ind_set* (store individuals) and *termination_flag*. In this framework, the packets sent by the computing node are separated into two categories. For requesting data, the first type of packet is used (i.e., *operation_type* = 0), which indicates that the computing node is requesting data for the first time. The processed data is transmitted using the second type of packet (i.e., *operation_type* = 1), which indicates that the computing node has already evaluated all individuals in this type

packet. The second sort of packet, in particular, contains the request for new data as well as the feedback of the previous individuals. The reply packet from the service node exists in only two forms, one in which the packet is sent carrying several individuals to be evaluated during the ENAS optimization process, and the other in which a packet with a termination command (i.e., *termination_flag=True*) is sent when the framework needs to be terminated. Theoretically, the newly packets can contain the chromosomes of any NN architecture search algorithm.

17.3.4 Server Node

The controller of the distributed framework, the server node, establishes a main process to manage packets from computing nodes and population evolution. Furthermore, the main process also has the responsibility of managing the entire cluster, which includes initializing network connections and broadcasting the termination message. During the ENAS optimization, the main process of the server node needs to be in state at all times, because the computing node will send asynchronous requests at any time, and the service node needs to respond immediately to prevent the computing node from having idle periods. Algorithm 1 details the server node workflow.

At Step 1, all connection requests from the computing nodes are listened by the server node, and all established connections for the various computing nodes are saved in *conn_list*. The *SendingBuffer* and *ReceivingBuffer* sharing buffers will then be created. To assure the dependability of the communication channel for each computing node, the server node establishes a pair of processes and queues for each connection and runs all processes at Step 5. Notice that a computing node is associated with a connection. The server node, for example, will construct three pairs of *SendProcess* and *RecProcess* if there are three computing nodes. The *SendBuffer* is shared by all *SendProcess* in the server node, while the *ReceivingBuffer* is shared by all *RecProcess*. The distinction between a buffer and a queue is that a buffer keeps individual while a queue keeps packets. Following that, at Step 8, the population initialization is done. It is worth noting that the population generation differs depending on the encoding schemes and random seeds used by different ENAS algorithms. The population $P^t = \{p_1^t, p_2^t, \dots, p_i^t\}$ that waits to be assessed is then saved in *SendingBuffer*. At Step 10, when the framework has been initialized, the main process begins to process the received packets. When a packet is taken from the *RecQueue* by the server node, several operations are performed depending on the packet type. The destination of a packet is determined by the *operation_type* it carries. If the *operation_type* is 0, the packet is sent from the server node to the computing node and is accompanied by multiple individual messages that need to be evaluated. If *operation_type* is 1, it means that the current individual has been evaluated and stored in shared *ReceivingBuffer*, and then new individuals are prepared and resent by the packet to the computing node. After all computing nodes finish evaluating the population P^t , the server node performs an

Algorithm 1: Algorithm flow of the server node

```

1 Wait for all compute nodes to apply for socket connection : conn_list;
2 Create a couple of sharing buffers: SendingBuffer, ReceivingBuffer, both are set to
empty;
3 for each conn i ∈ conn_list do
4   Create a couple of queues: SendQueuei and RecQueuei. //i indicates the index of a
connection;
5   Create a couple of processes: SendProcesi, RecProcessi;
6 end
7 Generation number: t = 0. Initial population Pt : {p1t, p2t, ..., pnt};
8 Put the Pt into the sharing SendingBuffer;
9 for t = 0, Rec_size = 0, Tmax do
10  Process the packet from the RecQueue sent by computing nodes;
11  if operation_type = 1 then
12    Take out the evaluated individuals and store in the ReceivingBuffer;
13    Rec_size = Rec_size + ind_num;
14    if Rec_size == n then
15      Execute evolution operations to generate new population: Pt+1: selection,
mutation, crossover, then repeat step 8;
16      t = t + 1;
17      ReceivingBuffer.clear();
18      Rec_size = 0;
19    end
20  end
21  Create a packet with new individuals and push into SendQueue;
22 end
23 Notify all nodes to stop evaluation;
24 return the best individuals(i.e., the best performing DNNs);
25 End;

```

evolutionary operation to generate a new population P^{t+1} . The steps from 2 to 3 are repeated until the maximum number of iterations T_{max} is reached. Finally, when the optimization is complete, the server node broadcasts a termination command and finds the optimal individual architecture in the ENAS algorithm.

17.3.5 Computing Node

As shown in Fig. 17.4, the number of computing nodes is non-fixed and can be changed according to the actual situation, but it is obvious that the increase of the number of computing nodes will be more beneficial to the operation speed of the distributed framework. This is because after receiving the data packets sent from the server node, the computing nodes decode them into DNNs and train and validate them on the target dataset, which is the most time-consuming part of the whole optimization. On the target dataset, the DNN will then be trained and validated. The accuracy of the DNN will then be used to specify the fitness value of the individual. In

Algorithm 2: Algorithm flow of the computing node

```

1 Apply to server node for a socket connection by ip and port;
2 Create RecQueue and SendQueue to provide temp storage for RecProcess and
SendProcess;
3 Create a pair of processes: SendProces and RecProcess, both run independently;
4 while termination == False do
5   if Take a packet from Rec Queue then
6     Fetch the ind_set from the packet;
7     Decode individuals  $p_i^t$  and constructing DNNs;
8     Train and validate the models on the target dataset;
9     Assign the accuracy to the fitness value;
10    Create a packet containing the processed individuals(i.e., ind_set) with fitness
values, then push it into SendQueue.;
11   else
12     | Initiate task request to the server node //idle;
13   end
14 end
15 End;

```

Algorithm 2, the computing node workflow is detailed. In a nutshell, the initialization of the computing node is referred to as Steps from 1 to 3. It is worth noting that *RecProcess* and *SendProcess* are both running, indicating that if a packet comes or has to be transmitted, the processes will respond quickly. The packets received by the *RecProcess* are pushed into the *RecQueue*, whereas the *SendProcess* selects a packet from the *SendQueue* and delivers it to the server node. The evaluation of a NN is completed by steps from 5 to 10, and a packet containing the evaluated individuals is pushed into *SendQueue*. In the computing node, another process (i.e., *Send Process*) sends the packets in *SendQueue* to the server node, indicating that the corresponding computing node is idle and requesting the server node to send a new task, as shown in Step 12. If *RecQueue* is *NULL*, it means that all tasks have been completed. The computing node continues to work until the server node sends a termination notification (i.e., the *termination_flag* of the packet is **true**). Note that no synchronization is required during the whole execution of the ENAS method, so all computing nodes are fully utilized.

17.4 Experimental Studies

To verify the performance of the distributed framework, the corresponding environment in the lab is established, consisting of one server node and four computing nodes with the same configuration (same performance). The framework is then used to conduct several experiments. Hardware configuration is as follows: the operating system is Centos7, GPU model is GTX1080, memory size is 16G, and hard disk storage memory is 1T. The framework is developed from the ground up with

Python 3.6,¹ which may be used to deploy the ENAS building on various deep learning platforms like PyTorch,² and TensorFlow. In order to meet the realistic need to search for more accurate or lightweight image recognition network architecture is a trend, a NAS approach named evolutionary Pelee is designed, which adopts modules from PeleeNet [19] and can be efficiently deployed on mobile. Therefore, EA-Pelee will be used as an example of ENAS in this experiment. On this framework, several experiments with various scales are carried out. Specifically, the baseline for ENAS on a single node is the amount of time it takes to train. After this comparison and examination of training efficiency at various cluster scales, the framework is shown to be both effective and scalable. It is important to highlight that the framework is designed to increase the training efficiency of ENAS, not its accuracy.

This experiment uses the module in PeleeNet to design ENAS due to the powerful performance of the network itself, which is also widely used in mobile and can be further optimized. Therefore, EA-Pelee can search for more efficient DNN, such as DNNs with fewer number of parameters or higher accuracy, without losing performance. In Sect. 17.4.1, the EA-Pelee that also focuses on Image Classification, which will be covered in more detail. In the following experiments, the chosen benchmark dataset is CIFAR-10 [20]. The dataset contains 60,000 color images with a 32×32 resolution for 10 different classes. Each class, in particular, contains 6,000 images, including 5,000 training images and 1,000 test images. Most well-known ENAS, such as CARS [21], are performed on this dataset. The promising architectures found are then tested on larger datasets such as ImageNet [22]. In theory, the performance of the framework is unaffected by NAS methods or datasets. Only test the effectiveness of this framework on CIFAR-10 in a receivable amount of time and with constrained computational resources. The framework will be used to train the EA-Pelee at various scales. Our experiments are divided into two main parts to validate the performance of the distributed framework. First, in Sect. 17.4.2, the consumption time of one computing node is used as a benchmark to test the ability of the overall running speedup with clustering. In Sect. 17.4.5, the time consumption of the state-of-the-art methods is used as a benchmark to demonstrate the efficiency of our framework.

17.4.1 Evolutionary Pelee

The PeleeNet is a mobile version of the DenseNet [23] architecture for image classification, with strict memory and computational budget constraints. PeleeNet obtains a compelling result by following the DenseNet connectivity patterns and key design principles. PeleeNet, for example, has a higher accuracy than both the architecture of original DenseNet and MobileNet [24]. The PeleeNet, in particular, is made up of three key modules:

¹ <https://www.python.org/>.

² <https://pytorch.org/>.

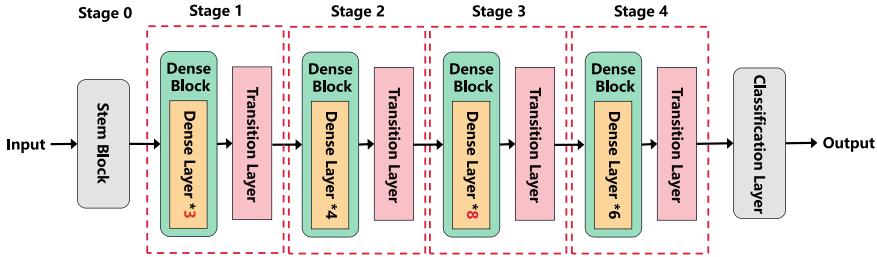


Fig. 17.5 The overall view of the Pelee architecture

1. **Dense Layer:** PeleeNet uses a 2-way dense layer to obtain varied scales of receptive fields. The 2-way dense layer uses a convolution kernel size of 3×3 , with the difference that one has only one convolutional layer and the other has a stack of two convolutional layers.
2. **Stem Block:** The stem block outperforms other more expensive strategies, such as increasing the number of channels in the initial convolutional layer or raising the growth rate, in terms of improving feature expression ability without adding too much computational expense.
3. **Transition Layer without Compression:** In transition layer, this module ensures that the number of output channels is always equal to the number of input channels.

Figure 17.5 depicts the whole architecture of the PeleeNet. The whole network is made up of a stem block and four feature extractor stages, each of which is made up of numerous dense layers and a transition layer. The number of output channels can be adjust via the transition layer. In PeleeNet, the number of output channels in transition layers equals to the number of input channels to avoid compression. In large model design, the multi-stage structure is widely employed. ShuffleNet [25], for example, employed a three-stage structure to reduce the size of the feature map at the start of each stage. It should be noted that the number of repetitions of the dense layers varies in the four stages, which sets manually to 3, 4, 8, and 6. This setting allows further optimization of this efficient lightweight network.

As mentioned before, GA [26] will be used as a search strategy and combine it with modules in PeleeNet to find a better network architecture (e.g., a lighter model or better performance). Specifically, a novel encoding strategy is designed for the convenient and fast representation of PeleeNet. There are multiple stages in PeleeNet, and using three integers to represent the dense way of the dense layer, the number of repetitions of the dense layer, and the rising growth rate between two dense layers, respectively. As a result, [**dense way**, **number of the dense layer**, **growth rate**] can be used to represent the dense block gene. The dense way has two values, 1 and 2, the growth rate has values of 8, 16 and 32, and the number of dense layers has values in the range [1, 15]. For example, obtaining a gene with these parameters [2, 3, 8], which indicates the presence of 3 dense layers in the corresponding stage, while each dense layer is 2-way and the growth rate between two dense layers is 8. To reduce the complexity of the network search, the nature of the four stages in PeleeNet is

maintained, so that a complete chromosome then consists of four genes (a total of 12 parameters). Since taking the image classification task as an example, the final classification accuracy is taken as the individual fitness value.

17.4.2 Speedup Analysis

The parameter settings of the following experiments are provided briefly in this section. Starting with a population of $N = 20$ individuals and set t_{max} to 20. For crossover and mutation operations, which are key means to generate better individuals in the EA, a higher probability of both indicates a higher probability of generating new individuals. Therefore, setting the crossover probability to $P_M = 0.58$ and the mutation probability $P_C = 0.5$. For the selection operation, the Russian roulette is used to select the next generation of individuals. The experiments are carried out on different scale clusters with the same settings. Adam optimizer [27] is employed to train each DNN model, and the loss function is set to the cross-entropy. The *batch-size* (number of samples chosen at one time) is set to 256, and the *epoch-size* (number of times to train the DNN on the target dataset) is set to 50. The *ind_len* in a packet has a range of $[0, \frac{pop_size}{n}]$, which will be discussed in Sect. 17.4.4. The cluster scale n is 4 and *pop_size* is 20 in the experimental configuration. As a result, the available value of *ind_len* is in the range $[0, 5]$, which is manually initialized to 3 and gradually declines to 1 as a computing node delivers a new task request. The *ind_len*, in particular, could be changed depending on the size of population and cluster scale. Equation 17.1 defines the formula of the speedup:

$$S_p = \frac{T_s}{T_n}. \quad (17.1)$$

Particularly, the speedup is represented by S_p . The training times on single node and multi-node are denoted by T_s and T_n , respectively, while the cluster scale is denoted by n . Furthermore, Fig. 17.6 shows the comparisons between the experimental speedup and theoretical speedup with various cluster scales.

The total evaluated individuals differ due to the randomness of the algorithm. However, as shown by the red line in Fig. 17.6, the training can be significantly accelerated using the distributed framework, and the speedup increases as the number of computational nodes increases. Although the gap between the theoretical (i.e., blue line in Fig. 17.6) speedup and experimental speedup becomes larger as the number of computational nodes increases, it is still consistent with the principle of linear speedup when there are fewer computational nodes. The primary reason is that as the cluster expands, the cost of communication and evolution increases, and the acceleration is going to continually decreases. Furthermore, the use of mutually exclusive resources (such as sharing queues) and the evolution of individuals in each generation also take time. According to the experimental results, the additional cost is acceptable in comparison to the considerable acceleration effect of the framework.

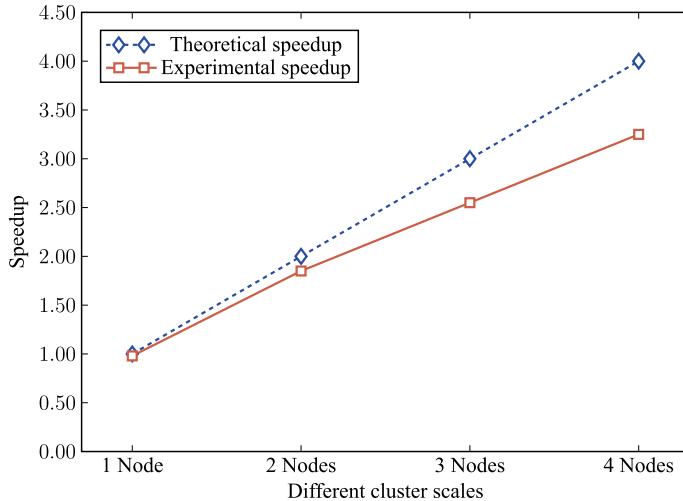


Fig. 17.6 The comparison of the experimental speedup and the theoretical speedup

17.4.3 Inconsistent Performance Node Analysis

As mentioned in Sect. 17.3.2 that computing nodes with different performance can exist in the framework, which can make the ENAS framework more applicable. In this section, to verify this, replacing one of the GPU cards (i.e., GTX1080, 12G of memory) from the previous experiments with a GPU card (i.e., GTX1060, 6G of memory) with a lower performance. The parameter settings are identical to those in Sect. 17.4.2. In Fig. 17.7, the experimental results of deploying the EA-Pelee algorithm to different clusters are shown. The horizontal coordinates in the figure indicate the different computing nodes, while the vertical coordinates indicate the number of individuals evaluated on the corresponding nodes. It can be seen that the number of individuals evaluated is proportional to the performance strength of the nodes, while nodes with similar performance are approximately the same in terms of the number of evaluations. The total utilized time of the cluster_2 is **26.36 h**, while it is **29.53 h** for the cluster_1, which is likewise less than the consumed time of two (i.e., **44.85 h**) or three nodes (i.e., **32.7 h**). It shows that the straggler (for example, Node1 in cluster_1) does not impair the overall performance of the cluster, and contributes to the distributed ENAS. For research groups with limited computing resources, the framework offers an alternative choice to the distributed ENAS.

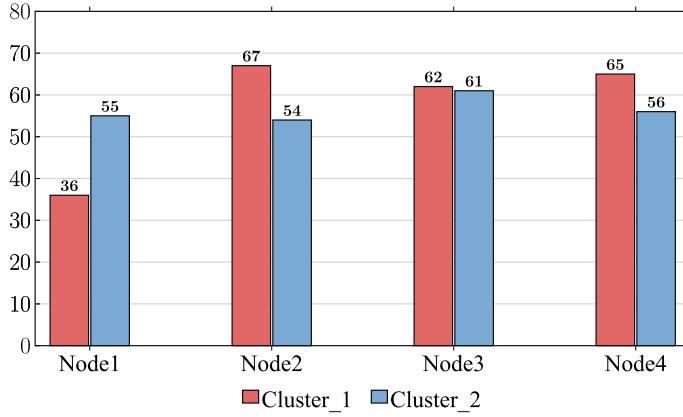


Fig. 17.7 The comparison of the number of evaluated individuals on different nodes. The **cluster_1** is equipped with three GTX1080 and one GTX1060 (i.e., Node1), while the **cluster_2** is equipped with four GTX1080

17.4.4 Communication Analysis

In most ENAS methods, the network representations of a DNN are always integers or some other sample data structure (e.g., binary, tree), as a result, the storage required for DNN encoding relatively small, implying that the communication cost for the exchanged packets is small. The length of a message distributed in our cluster can be calculated using Eq. 17.2:

$$L = l_p + l_{ind} * ind_len, \quad ind_len \in \left[0, \frac{pop_size}{n}\right] \quad (17.2)$$

In Eq. 17.2, n is the number of the computing nodes. The length of the data structure is denoted by l_p , the length of each individual is l_{ind} , and the number of individuals carried by a message at any given moment is ind_len , with a minimum value of 0 and a maximum value of $\lceil \frac{pop_size}{n} \rceil$. Because the l_p and l_{ind} are fixed, the overall length L of a message increases with ind_len . Even when the maximum value of ind_len is attained, the message size remains at the KB level. It's recommend that the ind_len not be set as a fixed number, and that it be set as a relatively big amount at the beginning, in order to reduce communication rounds and enhance cluster utilization. The ind_len progressively reduces to 1 as the number of individuals to be evaluated decreases. Furthermore, when the cluster adds a computing node, only two processes need to be formed, resulting in a total of $2n$ processes in the server with an affordable maintenance cost. In conclusion, when the cluster expands, the overall communication cost is low, making it flexible for the expansion of the entire cluster.

17.4.5 Efficiency Analysis

The searching results of the EA-Pelee are compared to several state-of-the-art methods in this section, and Table 17.2 summarizes the classification accuracy rates on CIFAR-10. The efficiency analysis of the framework is then presented. It is worth noting that the framework is centered on speeding up ENAS rather than improving the accuracy of an ENAS algorithm. The framework, in fact, will not improve the accuracy of the algorithm. Specifically, in the first column, the models are divided into three categories, the first one is the networks designed manually by experts and scholars with rich experience, including VGG, ResNet, Pelee, etc., while the second one is the algorithms obtained based on RL search, including MetaQNN and BlockQNN. Lastly, EA-based methods include Hierarchical [32], CGP, and AmoebaNet. The EA-Pelee-A and EA-Pelee-B are selected from the results that correspond to the EA-Pelee. Furthermore, the training time and resources are specified in the third and fourth columns, respectively. Particularly, GPU Days [35] are defined as $GPU\ Days = N * D$, where N indicates the number of GPUs and D indicates the actual number of days required for searching. Finally, in the fifth column, the size of the weight parameters of a DNN is presented.

As shown in Table 17.2, the accuracy of EA-Pelee is still competitive with the state-of-the-art methods obtained manually. Moreover, especially in terms of the parameter sizes of the models, it can be found that EA-Pelee-A and EA-Pelee-B are the best among all the compared algorithms. Furthermore, at the same level of accuracy, the framework takes significantly less time than other methods. Specifically, to achieve 93% accuracy, the MetaQNN requires 100 GPU Days, whereas the EA-Pelee-A only requires 5.3 GPU Days to achieve 94.01% accuracy. Despite the fact

Table 17.2 The performances comparison of different state-of-the-art algorithms and our EA-Pelee on CIFAR 10. It should be noted that the comparison of the accuracy and mode size (i.e., parameters) is to indicate that the EA-Pelee is a viable instance, whereas the comparison of the $GPU\ Days$ is to demonstrate the efficiency of the framework

Model	Accuracy (%)	GPU Days	Resources GPUs	Parameters (million)
VGG [28]	92.6	–	–	15.2
ResNet [29]	93.39	–	–	1.7
Pelee [19]	94.2	–	–	2.04
MetaQNN [30]	93.08	100	10	–
BlockQNN [31]	96.46	96	–	39.8
Hierarchical [32]	96.37	300	200	15.7
AmoebaNet [33]	96.7	3150	–	3.2
CGP [34]	94.02	30.4	–	2.64
EA-Pelee-A	94.01	5.3	4	0.94
EA-Pelee-B	93.6	5.3	4	0.85

that the Hierarchical method obtained higher accuracy, it has took 300 GPU Days to complete. Furthermore, when compared to the native Pelee, EA-Pele-A gains 50% model compression at the sacrifice of a little network performance, making it more suitable for running on mobile devices with limited storage. A valid example of the ENAS is the EA-Pele. In conclusion, as a case built on the framework, the EA-Pele could search for a competitive DNN in a reasonable amount of time and with resources that are affordable, demonstrating the efficiency of the framework.

According to the aforementioned experiments and analyses, the distributed framework may effectively minimize the training time of ENAS compared to a single node and is easy to expand due to the cheap additional cost.

17.5 Chapter Summary

In this chapter, we introduced a parallel framework elaborately designed for ENAS algorithms in a distributed environment having some GPUs. Specifically, the framework followed the traditional master-to-slave mechanism, but with additional concern on how to reasonably develop the asynchronous communication between different works having similar performance. In addition, to enhance the communication in ENAS algorithms, a structure is constructed to package the information of DNNs. Furthermore, a simple ENNAS algorithm named EA-Pele is designed to show the efficiency and the scalability of the designed distributed framework.

This chapter and also other chapters in this part mainly concern the recent advances in NAS algorithms (mostly for ENAS algorithms). these works include lowering the network complexity through a similar way of performing ENAS, and also the acceleration methods from the learning aspect as well as the parallel computation. In the future, more and more advanced work would appear to collectively promote the development of NAS/ENAS. In this part and also the previous two parts, we mainly documented the fundamentals, methods, and recent advances of ENAS. In the next part, we will conclude this book.

References

1. Pham, H., Guan, M., Zoph, B., Le, Q., & Dean, J. (2018). Efficient neural architecture search via parameters sharing. volume 80 of *Proceedings of Machine Learning Research* (pp. 4095–4104). PMLR.
2. Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2017). A downsampled variant of imagenet as an alternative to the cifar datasets. [arXiv:1707.08819](https://arxiv.org/abs/1707.08819).
3. Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., & Murphy, K. (2017a). Progressive neural architecture search. [arXiv:1712.00559](https://arxiv.org/abs/1712.00559).
4. Islam, N., Rahman, M., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., & Panda, D. K. (2012). High performance rdma-based design of hdf5 over infiniband. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

5. Povey, D., Zhang, X., & Khudanpur, S. (2014). Parallel training of dnns with natural gradient and parameter averaging. [arXiv:1410.7455](https://arxiv.org/abs/1410.7455).
6. Shi, S., Wang, Q., Zhao, K., Tang, Z., Wang, Y., Huang, X., & Chu, X. (2019). A distributed synchronous sgd algorithm with global top-k sparsification for low bandwidth networks. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (pp. 2238–2247).
7. Zhang, W., Gupta, S., Lian, X., & Liu, J. (2015). Staleness-aware async-sgd for distributed deep learning. [arXiv:1511.05950](https://arxiv.org/abs/1511.05950)
8. Lian, X., Zhang, W., Zhang, C., & Liu, J. (2018). Asynchronous decentralized parallel stochastic gradient descent. In J. Dy & A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning, Volume 80 of Proceedings of Machine Learning Research* (pp. 3043–3052). PMLR. <http://proceedings.mlr.press/v80/lian18a.html>.
9. Vishnu, A., Siegel, C., & Daily, J. (2016). Distributed tensorflow with mpi. [arXiv:1603.02339](https://arxiv.org/abs/1603.02339).
10. Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. [arXiv:1802.05799](https://arxiv.org/abs/1802.05799).
11. Mahsereci, M., Balles, L., Lassner, C., & Hennig, P. (2017). *Early stopping without a validation set*. [ArXiv:abs/1703.09580](https://arxiv.org/abs/1703.09580).
12. Dean, J., Corrado, G. S., Monga, R., Kai, C., & Ng, A. Y. (2012). Large scale distributed deep networks. *Advances in Neural Information Processing Systems*.
13. Jeffay, k. (1993). The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. *SAC '93: Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*. <https://doi.org/10.1145/162754.168703>.
14. Xie, L., & Yuille, A. L. (2017). Genetic CNN. In *IEEE International Conference on Computer Vision, ICCV 2017* (pp. 1388–1397). <https://doi.org/10.1109/ICCV.2017.154>, <https://doi.org/10.1109/ICCV.2017.154>.
15. Miller, J. F., & Thomson, P. (2000). Cartesian genetic programming. In *European Conference on Genetic Programming* (pp. 121–132). Springer.
16. Gruau, F. (1993). Cellular encoding as a graph grammar. In *IEE colloquium on grammatical inference: Theory, applications and alternatives* (pp. 17/1–1710).
17. Kim, M., & Rigazio, L. (2015). Deep clustered convolutional kernels. In *Feature Extraction: Modern Questions and Challenges* (pp. 160–172).
18. Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., Lanctot, M., & Wierstra, D. (2016). Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference* (pp. 109–116). ACM.
19. Wang, R. J., Li, X., & Ling, C. X. (2018d). Pelee: A real-time object detection system on mobile devices. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31* (pp. 1963–1972). Curran Associates, Inc. <http://papers.nips.cc/paper/7466-pelee-a-real-time-object-detection-system-on-mobile-devices.pdf>.
20. Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical Report, University of Toronto. <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>.
21. Yang, Z., Wang, Y., Chen, X., Shi, B., Xu, C., Xu, C., Tian, Q., & Xu, C. (2020). Cars: Continuous evolution for efficient neural architecture search. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. <https://doi.org/10.1109/cvpr42600.2020.00190>, <http://dx.doi.org/10.1109/cvpr42600.2020.00190>.
22. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3), 211–252. ISSN 0920-5691, 1573-1405. <https://doi.org/10.1007/s11263-015-0816-y>, <http://link.springer.com/10.1007/s11263-015-0816-y>.
23. Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 4700–4708).

24. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. [arXiv:1704.04861](https://arxiv.org/abs/1704.04861), <http://arxiv.org/abs/1704.04861>, arXiv: 1704.04861.
25. Zhang, X., Zhou, X., Lin, M. & Sun, J. (2017). Shufflenet: An extremely efficient convolutional neural network for mobile devices. [arXiv:1707.01083](https://arxiv.org/abs/1707.01083).
26. Mukhopadhyay, D. M., Balitanas, M. O., & Alisherov, F. A., Jeon, S., & Bhattacharyya, D. (2013). Genetic algorithm: A tutorial review. *International Journal of Grid and Distributed Computing*, 2.
27. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
28. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 32nd International Conference on Machine Learning*.
29. He, K., Zhang, X., Ren, S., Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778).
30. Baker, B., Gupta, O., Naik, N., & Raskar, R. (2016). Designing neural network architectures using reinforcement learning. [arXiv:1611.02167](https://arxiv.org/abs/1611.02167).
31. Zhong, Z., Yan, J., Wu, W., Shao, J., & Liu, C.-L. (2017). Practical block-wise neural network architecture generation. [arXiv:1708.05552](https://arxiv.org/abs/1708.05552).
32. Liu, H., Simonyan, K., Vinyals, O., Fernand C., & Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. [arXiv:1711.00436](https://arxiv.org/abs/1711.00436).
33. Real, E., Aggarwal, A., Huang, Y., & Le, Q. (2018). Regularized evolution for image classifier architecture search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33, 02. <https://doi.org/10.1609/aaai.v33i01.33014780>.
34. Suganuma, M., Shirakawa, S., & Nagao, T. (2018). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18* (pp. 5369–5373). <https://doi.org/10.24963/ijcai.2018/755>, <https://doi.org/10.24963/ijcai.2018/755>.
35. He, X., Zhao, K., & Chu, X. (2019). Automl: A survey of the state-of-the-art. [arXiv:1908.00709](https://arxiv.org/abs/1908.00709).

Book Conclusions

The purpose of this book is to offer the target readers with necessary ENAS details from the scratch. This objective has been addressed by giving fundamentals, representative methods, and recent advances. This book specifically covers the fundamentals of EC and DNNs, as well as some recent research concentrating on architectural design for unsupervised and supervised DNNs. The evolutionary designs in the unsupervised section are for DBN architectures, stacked AEs, stacked CAEs, and CVAEs. The supervised portion is primarily for CNNs, of which some are deliberately made to automatically design the optimal architectures for a general-purpose even though the experiments are verified on images, and others are specifically developed for specific tasks, such as HSI denoising.

Furthermore, we have contributed to this study by presenting recent advances that are not directly connected to the ENAS approaches but are closely related to them. Most of the encoding methods presented in the method part are traditional, we first introduce a graph-based encoding one to reveal the potential. In addition, considering the ENAS, which is prohibitively computationally costly, two ways for speeding up the architecture have been introduced: a well-designed acceleration mechanism and a distributed training framework, respectively. Furthermore, while large DNN networks cannot be directly conducted on mobile devices that provide artificial intelligence services, another two strategies for reducing model sizes while still achieving promising performance are explored.

The most notable feature of book is that it will offer systematic and comprehensive content for studying ENAS. Individuals who would like to discover more about this topic merely need to read this book. After reading this book, readers will understand standard approaches to developing ENAS methods for their tasks at hand, as well as unique approaches prompted by obstacles to advance research on this subject.