VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Genetic Programming for Automatic Design of Convolutional Neural Network Architectures

William Irwin-Harris

Supervisors: Bing Xue, Yanan Sun, Mengjie Zhang

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Convolutional neural networks (CNNs) have achieved great success at image classification, with many real-world applications in computer vision. Yet developing new CNN architectures is a challenging task, often employing specific human expertise. As a result, a number of automatic architecture design approaches have been proposed. However, previous evolutionary computation–based methods typically apply restrictions to the possible architectures that can be evolved. This project employs a graph-based genetic programming (GP) approach, to allow evolution of CNNs of unbounded depth and with completely flexible graph structure. This work demonstrates the potential of the graph-based method, termed GraphGP-CNN: the results show that the method can obtain accuracy comparable to, or better than, the peer competitors. These include state-of-the-art handcrafted CNNs and CNNs from other automatic design methods. This project demonstrates that a completely automatic evolutionary approach, that makes minimal assumptions and allows an arbitrary graph structure, can achieve good results while using reasonable computational resources.

Acknowledgements

I would like to thank my supervisors Bing Xue, Yanan Sun and Mengjie Zhang for their tireless support, expert feedback and for the many engaging conversations. Their countless hours of assistance, often within tight timeframes, have been critical for this project, and their encouragement and enthusiasm throughout have inspired me to continue my interest in this area. Thank you to my family and friends for all their support.

Contents

Figures

Chapter 1

Introduction

Computer vision is a broad, interdisciplinary field with wide-ranging applications, including in autonomous vehicles [1], medicine [2] and technologies such as augmented reality [3]. One of the most fundamental computer vision tasks is image classification, in which a computational system must determine a class label for an image. Generally, this involves supervised learning from human-labelled data. Today, the dominant paradigm for this task is based on deep learning using convolutional neural networks (CNNs).

1.1 Motivations

Deep learning [4] has achieved great success in image classification [5], with convolutional neural networks (CNNs) becoming the dominant choice of model. Advances in deep learning for image recognition have been driven in large part by the development of new CNN architectures: following the introduction of CNNs with LeNet5 [6], other successful architectures have included AlexNet [5], VGG [7], GoogLeNet [8], ResNet [9] and DenseNet [10].

Yet designing CNNs is still a challenging problem: it requires not only expertise from machine learning researchers familiar with CNNs, but also knowledge of the specific problem domain in order to achieve the best results [11]. Typically, substantial trial-and-error experimentation is also required. Hence designing new CNNs for a specific application is often time-intensive and costly, due to the need for human expertise. Moreover, even if a decision is made to use an already-developed (e.g. published) CNN architecture for a real-world problem, this may not produce the best results, as the architecture has not been designed specifically for the target application and its corresponding data. In particular, for a small amount of training data, a relatively shallow CNN may give the best results. Conversely, if there is a large amount of training data available, the best results may be obtained by a very deep CNN.

Because of these challenges associated with manually selecting a CNN architecture, automatic techniques for deriving new architectures have been a topic of recent work. Previous work in this area has generally employed either of two key architecture search strategies. One of these is reinforcement learning, which is the approach taken by methods such as neural architecture search [12], MetaQNN [13] and Block-QNN [14]. These methods have achieved good results, but have generally required extensive computational resources. The other key strategy employed is based on evolutionary deep learning, in which evolutionary computation methods are employed to derive new deep structures. Work in this category includes the Genetic CNN method [15], the large-scale evolution method [16], the hierarchical representations method [17], the Cartesian genetic programming–based method [18], the CNN-GA method [19] and the GP-CNAS method [20].

Evolutionary architecture search algorithms have achieved good results on benchmark data sets, including CIFAR-10 and CIFAR-100 [21], demonstrating the potential of this approach. However, several limitations remain; these can be attributed to the choice of encoding strategy, which is a key component in evolutionary deep learning. Both the Genetic CNN [15] and CGP-CNN methods [18] limit the final depth of the evolved CNN, such that a maximum depth must be specified by the user before beginning the evolutionary process. For CGP-CNN [18], this is due to the use of the standard Cartesian genetic programming encoding of a grid with a fixed number of rows and columns ($N_r \times N_c$). Another limitation of CGP-CNN [18] is that a crossover operator is not used; the algorithm evolves architectures using point mutation only. In the case of Genetic CNN [15], the maximum CNN depth is also fixed before the algorithm is run, due to the encoding strategy of a fixed-length binary string.

More broadly, these limitations can be described as showing the algorithm to be *semi-automatic* rather than fully automatic; that is, in the previous two examples, user input is still required to specify the maximum depth for the encoding. A similar requirement also exists for the hierarchical representation method [17], in which the user must predetermine the number of levels in the hierarchical representation, and also the number of nodes in the graph at each level.

Although the CNN-GA [19] approach is fully automatic, as it uses an encoding of a linked list of building blocks that may be grown to arbitrary depth during the evolution, the connectional structure of the architectures it generates is restricted. Each building block in the linked list encoding is either a pooling layer or a 'skip layer', where a skip layer includes two convolutional layers and a skip connection [19]; this design is modelled on ResNet [9]. This search space achieves good results, but means that CNN-GA [19] cannot generate CNN architectures of arbitrary graph structure. Notable architectures that use a complex graph structure that differs from ResNet [9] include DenseNet [10] and the Block-QNN-S cell designed by the Block-QNN reinforcement learning approach [14].

1.2 Goals

The goal of this project is to apply graph-based genetic programming (GP) to the task of automatically deriving a CNN architecture for a given data set. The specific objectives of this project are to:

1. Develop an encoding strategy for evolutionary deep learning that addresses the limitations of previous work. This encoding method should apply minimal restrictions to the search space, to allow CNNs of arbitrary graph structure and unbounded depth to be represented. This may enable search algorithms to derive novel structures, such as new applications of skip connections.

2. Develop a novel graph GP–based architecture search algorithm using this encoding, including population initialisation, fitness evaluation, selection and genetic operators (i.e., crossover and mutation). In particular, crossover and mutation must be able to be applied to individuals of arbitrary graph structure.

3. Analyse the discovered architectures in comparison with hand-crafted architectures and other automatically designed architectures. This work will occur after the evaluation step where the proposed algorithm is applied to benchmark datasets.

4. Explore opportunities for reducing the computational resources required by the algorithm (i.e., the number of GPU days used during search), to ensure that the computa-

tion time required is reasonable in comparison with peer competitor algorithms. This is a secondary objective of the project.

1.3 Major Contributions

In this work, a new genetic programming approach has been developed for evolving CNN architectures with a flexible graph structure, and which is fully automatic in that the user does not need to limit the maximum depth of the architectures. The key contributions of this work are as follows:

1. An encoding strategy for evolutionary deep learning is proposed, in which each CNN architecture is represented as a directed acyclic graph. Compared to other representations designed for evolutionary search, this encoding applies significantly fewer constraints to the search space. This may enable search algorithms to derive novel structures, such as new applications of skip connections.

2. A method for randomly generating architectures according to the proposed encoding is introduced, such that the method can be used either as the population initialisation component of an evolutionary algorithm, or on its own in a random search. The proposed initialisation method is designed to apply minimal restrictions to the search space, while avoiding generation of architectures that are invalid or evidently suboptimal.

3. The effectiveness of this encoding strategy and the associated initialisation method is demonstrated using a simple random search, using only 10% of the final training data during the search process. This shows that even with a large search space, limited training data during search, and a simplistic search algorithm, the representation is able to achieve promising accuracy.

4. A genetic programming-based architecture search approach, including crossover and mutation operators specific to the graph representation, is proposed using this encoding. This method achieves results that are highly competitive with both state-of-the-art handcrafted architectures and with other automatic search approaches, while using reasonable computation time. Overall, this work demonstrates the potential of a fully-automatic architecture search approach, using a large search space.

Part of this work has been accepted for publication by the 2019 IEEE Congress on Evolutionary Computation [22].

1.4 Report Organisation

The remaining chapters of this report are organised as follows. Chapter 2 provides an overview of key concepts that form the background for this work, including computer vision, deep learning with CNNs, genetic programming, and related work for CNN architecture search. Chapter 3 introduces the proposed algorithm, including the encoding method, initialisation method, genetic operators and the evolutionary process. Chapter 4 provides results and discussion for experiments with the proposed algorithm applied to the CIFAR-10 and CIFAR-100 data sets. Chapter 5 focuses on evaluation of the results from the proposed method, including analysis of the evolved CNNs. Chapter 6 concludes the report.

Chapter 2

Background

This chapter provides an overview of important topics that form the background and the foundations for this work. This includes a summary of relevant ideas from the field of computer vision, an overview of convolutional neural networks, genetic programming (including graph-based GP), and related work for using EC to evolve deep structures.

2.1 Computer Vision

Computer vision is a broad field aimed at mimicking humans' ability to derive information from their visual environment; this encompasses many tasks, such as image classification, object detection and segmentation. Image classification is the process of assigning a class label to an image, which is a supervised process based on a training set. Object detection is the task of determining the location and size (e.g. via a bounding box) of all objects in an image, and also identifying the type of each object.

Traditional methods for computer vision have often relied on manually extracting useful image features, and then employing these features as inputs to classification algorithms. In recent years, deep learning has become the predominant technique, allowing features to be automatically learned without human input. This offers the potential for deep learning algorithms to be applied to different classification tasks or data sets, without the need for manual "feature engineering" (tailored to the specific image types) that was required by traditional methods. However, because the neural network architecture still needs to be designed, deep learning in itself does not offer a completely automatic learning approach.

This work focuses on image classification, due to its fundamental importance in computer vision: classification is a key task in itself, but methods developed for image classification can also have applications in other computer vision tasks. For example, a CNN architecture such as VGG [7], which was originally designed for classification, can be effectively used for featurisation for object detection in methods such as Faster-RCNN [23].

By combining the automatic feature extraction ability of deep learning, with architecture learning via graph-based genetic programming, this work aims to develop a novel approach for fully automatic image classification.

2.2 Convolutional Neural Networks

This section provides an overview of the key components of CNNs. Subsection 2.2.1 details the standard components of CNN, and then Subsection 2.2.2 highlights the important role of skip connections in modern CNN architectures, which were a particular inspiration for this work.

2.2.1 Standard CNN Layers

The two key components of CNNs are convolutional layers and pooling layers. As this project focuses on image recognition, we consider the case of 2D convolution kernels. For the 2D situation, the output of each layer can be described as a $C \times M \times N$ array, where C is the number of channels, and M and N are the width and height of a 2D array. Each such 2D array is often referred to as a feature map. The input of one layer can be taken directly from the output of a previous layer, or constructed by an operation such as summation or concatenation; these two operations are discussed in the next section. In addition, a layer may be connected directly to the image pixels, in which case the number of input channels is generally three (for the red, green and blue components).

A convolutional layer produces a stack of output feature maps by applying convolution kernels, which are most often square matrices of odd dimensions; typical sizes are 3×3, 5×5 and 7×7. A convolution kernel is applied by performing element-wise multiplication of the kernel values with the corresponding array elements from the input (either the input feature map or image), and then summing these products to produce the output of the kernel. During network training, each convolutional layer learns the values of its convolution kernels, where the number of kernels is determined by the dimensionality of the input data and the number of output feature maps. The number of output feature maps is a parameter of the layer.

Pooling layers are applied similarly to convolutional layers, but rather than using convolution kernels, an operator is applied to aggregate input data over a spatial area; typically this is either the arithmetic mean (average pooling) or maximum (max pooling). Most often, pooling layers are used for spatial dimensionality reduction; the number of channels is unchanged.

CNNs may optionally include one or more fully-connected (dense) layers at the end of the CNN structure, immediately prior to the output layer. These layers follow the standard structure of a multilayer feed-forward neural network, where each node in the layer is connected to every node in the previous layer. However, the proposed encoding method avoids the use of fully-connected layers. This is consistent with the approach taken in related works [18], [19]; fully-connected layers are prone to overfitting due to their dense connections [24].

Hence the task of encoding a CNN architecture can be described as that of specifying the number of layers, their connectional structure, and the type and parameters of each layer.

2.2.2 Skip Connections

A key technique in recent CNN architectures has been *skip connections* (also known as shortcut connections). The most straightforward CNN architecture consists of a linear sequence of layers, where each layer takes its input directly from the previous layer. However, with the inclusion of skip connections, a layer's input may be specified as the summation of the immediate previous layer's output and also the output of a layer earlier in the sequence. The effectiveness of skip connections has been highlighted by their successful application in ResNet [9]. In ResNet, the structure is based on a repeating sequence of 'residual blocks' (each of which contains two convolutional layers and a skip connection) [9]. The benefits of skip connections in alleviating the vanishing gradient issue (which is common in deep neural networks) has been discussed in [25], and their effectiveness in deep learning has also been experimentally shown in [26].

The concept of skip connections can be generalised by considering CNNs as directed acyclic graphs of potentially arbitrary structure. In this case, we consider that the input to a convolutional or pooling layer may be the output of any other layer, or the summation or

concatenation of the outputs of other layers, with the requirement that no cycles are formed. The summation operator performs the element-wise addition of two $C \times M \times N$ arrays; i.e., the two multidimensional arrays must have identical spatial dimensions and numbers of channels. The concatenation operator joins the channels of a $C_1 \times M \times N$ and a $C_2 \times M \times N$ array to produce an output array of size $(C_1 + C_2) \times M \times N$. As with summation, the spatial dimensions of the two arrays must be equal for concatenation.

2.3 Evolutionary Computation

Evolutionary computation (EC) [27] refers to a family of algorithms inspired by biological evolution. Key EC techniques include particle swarm optimisation [28], [29], and evolutionary algorithms (EAs) [30]. EAs are a class of population-based metaheuristic optimisation methods; two of the most well-known sub-categories of EAs are genetic algorithms [31] and genetic programming [32]. Typically, genetic algorithms are based on a binary string representation of each individual in the population, where in general some decoding method is applied to each representation (genotype) to produce a phenotype for which a fitness can be determined.

In contrast, genetic programming (GP) typically represents a program structure as a tree. This work focuses on a GP-based approach, because a CNN can be considered as a computational structure that transforms an image into a class label, in the same way that tree-based GP is used to represent program structures. In addition, the binary string representation used in GAs is generally fixed-length, whereas GP is typically able to represent individuals of variable complexity, which is important for the goals of this project. However, one key way in which this work differs from standard genetic programming is that a tree data structure is not sufficient to represent a general CNN architecture; instead, the appropriate representation is a directed acyclic graph.

This section is organised as follows. Subsection 2.3.1 provides an overview of standard genetic programming, using a tree-based representation. Subsection 2.3.2 considers graph-based GP and its relevance to CNN architecture search, including a brief review of important previous work for graph-based EAs, including GP.

2.3.1 Tree-Based Genetic Programming

Genetic programming (GP) [32] is a type of evolutionary algorithm that is typically based on a tree-based representation. The key components of genetic programming are population initialisation, in which a starting population of individuals (trees) are randomly generated; fitness evaluation, where each individual is assigned a fitness value by means of a fitness function; and creation of a new generation (population) using genetic operators. Figure 2.1 shows the key components in the evolutionary process for GP.

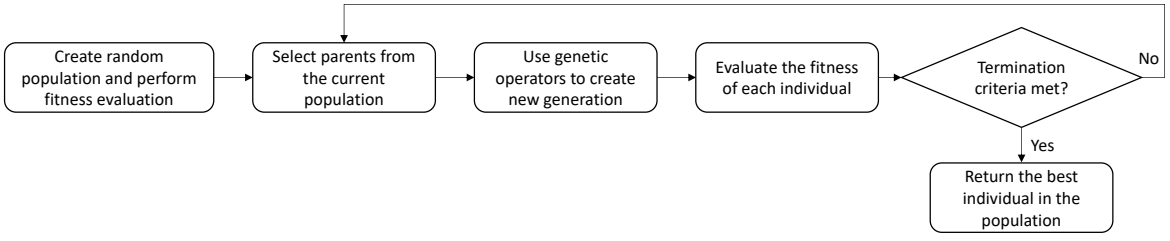


Figure 2.1: The general evolutionary process for genetic programming.

The specific choices of nodes in the tree are referred to as the terminal set and the function set. The terminal set corresponds to the node types that can be used as leaf nodes, and the function set specifies the available types of internal nodes in the tree. Thus, applying genetic programming to a specific problem requires specifying a number of parameters and choices. These include the population size, the termination criteria, the initialisation method for the starting population, specific choices for genetic operators, the terminal and function sets, and the fitness function.

Often, the termination criterion may be simply a maximum number of generations; one alternative approach is to specify a target fitness value. Examples of fitness functions include the mean squared error between the output of the program and a training set (for regression problems), or the accuracy on a training set for classification problems.

Figure 2.2 shows an example of a typical GP individual represented as a tree, where the program represents an arithmetic calculation with the variables X and Y and the constant 2. In this example, the terminal set may be the variables X and Y and the set of integers, and the function set may be the standard binary arithmetic operators.

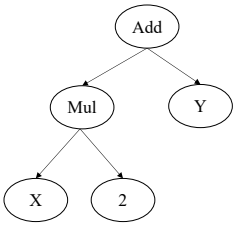


Figure 2.2: Example of a GP individual.

Typical population initialisation methods include the *full* method, the *grow* method, and the *ramped half-and-half* approach. With a *full* population initialisation approach, internal nodes are added to each tree until a specified depth is reached, and then leaf nodes are added for each internal node. With the *grow* method, each node may be randomly selected as either a leaf or an internal node; the branch stops growing once a leaf node is chosen. With the ramped half-and-half method, half of the individuals in the population are produced by the full method and half are produced by the grow method.

Two of the most common selection methods are roulette wheel selection and tournament selection. In roulette wheel selection, the probability of choosing an individual as a parent is proportional to its fitness, so that fitter individuals are more likely to be selected, but there is still a non-zero probability of choosing the individuals with low fitness. This helps to maintain the diversity of the population, and prevent premature convergence.

In tournament selection, a fixed number of individuals are randomly selected (each with uniform probability of selection) and the best individual of these is chosen as a parent. Similarly to roulette wheel selection, this also gives some probability of selecting the relatively worse individuals. One benefit of tournament selection is that by choosing the tournament size (i.e., the number of individuals selected), it is possible to control the amount of selection pressure in the algorithm.

The two main types of genetic operators used to produce a new generation are mutation and crossover. Mutation performs a random modification to a single individual, and is used to maintain the diversity of the population, to avoid the algorithm converging early to a local optimum. Crossover generally acts on two individuals, and is used to combine their genetic material to produce two *offspring* individuals. In addition, *elitism* is often employed, in which some small number of individuals with the highest fitness values are copied unchanged to the next generation. This ensures that the maximum fitness of the child popula-

tion cannot become worse than that of the parent population.

In genetic programming, it is important that the requirement of *closure* is satisfied; specifically, any function in the function set must be able to accept as input the output of any other function. Theoretically, the terminal and function sets should also satisfy the requirement of *sufficiency*; that is, there should exist some combination of the terminals and functions that is able to solve the problem. However, in practice, for problems such as classification, it can be difficult to determine a value that constitutes an acceptable level of accuracy.

2.3.2 Graph-Based Genetic Programming

This section briefly considers previous work on graph-based GP and GA and their relevance to evolving CNN architectures, with a particular focus on the encoding method. Choosing the encoding strategy is a key challenge with graph-based GP. This differs from tree-based GP which, unlike genetic algorithms, has the advantage of typically not requiring an encoding or decoding step. In many cases with tree-based GP, the tree representation directly corresponds to the evolved program.

In contrast, many previous works for graph-based GP have required an encoding strategy, in a way that is more similar to GAs; often a fixed-size two-dimensional grid is employed as the representation. The encoding strategy is generally chosen to aid in designing genetic operators. In particular, developing a crossover operator for graph-based GP can be especially complex.

For example, the Parallel Distributed Genetic Programming (PDGP) method by Poli [33] was an early work that uses the parallel computational structure of neural networks as inspiration for developing a graph-based GP method (however, the original work does not apply the method to the task of evolving neural networks). In particular, Poli notes the difficulty of designing crossover operators using a direct graph encoding. Instead, each node is assigned to a location in a 2D grid, and connections are only allowed upwards in the grid. Crossover can be performed by randomly choosing a node in one parent, thus defining a subgraph that can be substituted into the other parent grid, to create the offspring.

A well-known graph-based GP approach is Cartesian genetic programming (CGP) [34], which is also based on a fixed-size 2D grid (but represented as an integer string). Applying CGP to a specific problem requires specifying the size of the grid (i.e., the number of rows and columns), and also a *levels-back* parameter [34]. The *levels-back* parameter determines the maximum distance allowed (in terms of the number of columns) between a node and its input nodes [34]. For CNN architecture evolution, this parameter would act to limit the number of nodes that can be spanned (i.e., skipped) by a skip connection or concatenation. However, it is possible to set *levels-back* to the number of columns to avoid this limitation. Cartesian genetic programming has been applied for evolving CNN structures in the CGP-CNN method [18], showing promising results. Generally, CGP operates using mutation only, without any crossover operator [34].

As the goal of this project is to allow evolving CNN architectures without limiting their depth or other structural properties, a fixed-size encoding approach is not suitable. A direct graph encoding has been considered by other works, although this makes design of crossover operators more complex. For example, although not focusing on GP, a graph-based encoding for GAs has been applied to the context of computational chemistry for evolving molecules in works by Globus et al. [35] and Brown et al. [36]. One general strategy for graph crossover considered by both these works is based on splitting each parent graph into two subgraphs and then exchanging subgraphs between parents. The key challenge in this method is determining how to perform the subgraph split, and how to reconnect the exchanged subgraphs if the splitting process has resulted in a loss of information [35], [36].

In addition, Brown et al. [36] discuss an alternative method in which subgraphs are randomly selected (but not removed) from the two parent graphs, and combined to form an offspring; this is repeated twice to produce two different offspring. However, these works are more complex than is needed for evolving CNN architectures: molecular design requires the ability to produce cyclic structures, whereas CNNs are acyclic.

2.4 Related Work

This section considers related work for CNN architecture search, including manually-designed architectures, as well as other EC methods for automatically evolving CNN architectures and other deep structures.

2.4.1 Key Handcrafted CNN Architectures

To develop a new method for automatic CNN architecture search, it is informative to consider the key principles underpinning the success of recent handcrafted CNN architectures. As earlier discussed, one of the most well-known of these is ResNet [9]. Figure 2.3 depicts the basic structure of a *residual block*, which is the fundamental unit of ResNet.

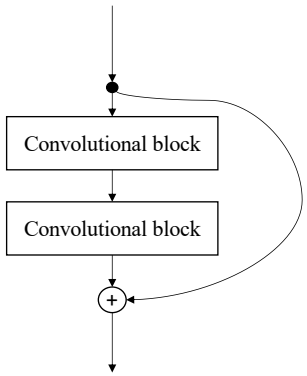


Figure 2.3: The general structure of a residual block as used in ResNet [9].

In the original proposal for ResNet [9], the residual block consists of two convolutional blocks (each block contains convolution and batch normalisation layers). These are spanned by a skip connection, so that the input feature maps to the first convolutional layer are summed with the output of the second convolutional layer, and this sum is provided as the input to the next residual block. The motivation for the ResNet design is that it has been observed that adding more convolutional layers to a model can increase training error. This is despite the fact that the deeper model should be expected to provide performance at least as good as the shallower model, if the additional layers were set as identity mappings [9]. Because the worsened performance occurs on the training set, the issue is not related to overfitting. Thus, the goal of ResNet is to aid the optimisation algorithm in training deep networks, based on the hypothesis that re-formulating the problem as learning a residual mapping can improve the effectiveness of the optimiser [9]. In practice, ResNet has been empirically shown to achieve good performance [9].

Given this basic design, there are several key choices that must be made: selecting the ordering of components such as convolution layers, ReLU activation, and batch normalisation; determining the filter (kernel) dimensions; and choosing the depth and width of the network. The original ResNet [9] implementation applies batch normalisation after each convolution layer, and then applies ReLU after the first convolution layer and also after

the summation in each residual block. However, in [37] it has been shown empirically that improved performance can be obtained by using a pre-activation design with batch normalisation applied first, then ReLU, then the convolution layer. ResNet consists almost entirely of 3×3 kernels, other than in the first layer, which may use a 7×7 kernel [9]. As originally proposed, ResNet uses a scheme based on doubling the number of feature maps in each convolution layer whenever the spatial dimensionality is halved, with the first layer having 64 feature maps. Given these specifications, ResNet maximises its accuracy on the CIFAR-10 data set [21] when the network has 110 layers [9].

A modification to this approach is the Wide Residual Networks [38] concept, in which it is shown that superior accuracy can be obtained using a *less* deep, but wider network. In particular, the WRN-40-2 architecture has a comparable number of parameters to ResNet-110, but achieves better performance, by using a network that is twice as wide (i.e., has twice as many feature maps per layer compared to ResNet-110) but with only 40 layers [38]. For Wide Residual Networks, the best accuracy overall is achieved by WRN-28-10 using only 28 layers, but with 10 times as many feature maps per layer as the original ResNet. Hence, this suggests that optimising performance requires finding the best balance between the depth and width of the network (while not exceeding constraints such as the desired complexity or memory usage).

Another extension to ResNet is ResNeXt, which generalises ResNet to include the new dimension of *cardinality* [39]. Figure 2.4 depicts the building block used in ResNeXt.

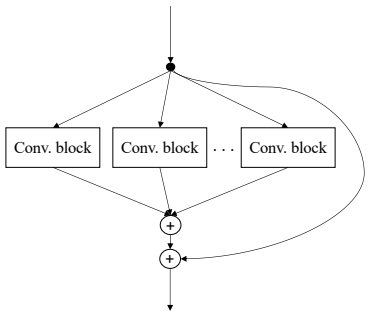


Figure 2.4: The general structure of a building block used in ResNeXt [39].

In Figure 2.4, each of the convolutional blocks is generally defined as a bottleneck block as described in the original ResNet; that is, a sequence of a 1×1, then 3×3 and finally a 1×1 convolution [9], [39]. The key concept with ResNeXt is to allow multiple branches of identical topology within the building block, where the output of each branch is aggregated via summation and the number of branches per building block is termed the cardinality [39]. It has been shown that increasing cardinality can improve accuracy more effectively than increasing depth or width, for the same CNN complexity [39].

Another important modern CNN architecture is DenseNet [10], which is based on the dense block, as depicted in Figure 2.5. In DenseNet, the input to each convolutional layer is defined as the concatenation of the outputs of every preceding layer in the dense block. An architecture consists of a sequence of dense blocks, separated by pooling layers. DenseNet differs signicantly from ResNet [9] in that it is *not* based on skip connections; instead, learning performance is improved by promoting feature reuse [10]. Because of the large number of input feature maps for each convolutional layer, DenseNet typically uses a low number of output feature maps (referred to as the growth factor). It has been shown that DenseNet can achieve superior performance to ResNet at similar complexity [10].

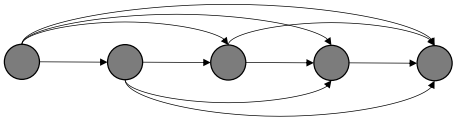


Figure 2.5: The structure of a dense block as used in DenseNet [10].

2.4.2 EC Methods for Evolving CNNs for Image Classification

This section reviews several notable examples of previous work for automatic CNN architecture design with EC.

Genetic CNN [15]. Based on a standard GA, Genetic CNN represents each architecture as a fixed-length binary string. The string represents a sequence of stages where, in the decoded CNN, each stage is separated by a pooling layer. Each stage is a sequence of convolutional layers. Stages always have at least one convolutional layer; it is not possible to represent two consecutive pooling layers. As well as fixing the length of the binary string, the number of stages and the maximum number of layers in each stage must be fixed by the user. Equivalently, the user must specify the position of each pooling layer. This method uses a densely-represented graph encoding where one bit is stored for every possible connection in each stage. A one bit represents a connection and a zero bit represents no connection; the length of the string increases quadratically with the maximum network depth. Specifically, these connections are skip connections: they are summed to produce the input for each layer. This encoding can represent ResNet [9] and VGGNet [7], but *not* architectures such as DenseNet [10] that use concatenation. Mutation is performed by changing each bit with a small probability. For crossover, the fundamental unit is the stage: each stage (in the equivalent position) may be swapped at random between the two parent architectures.

CGP-CNN [18]. This approach is based on Cartesian genetic programming (CGP) [34]. Each architecture is represented as a 2D fixed-size grid of nodes, encoded as an integer string. The available nodes in the graph structure are tensor summation, concatenation and a convolutional building block node. The building block can be chosen as ConvBlock, which includes convolution, batch normalisation and ReLU; or as ResBlock, which is equivalent to the basic unit of ResNet [9]. Separate results are reported using each of the two building blocks; the ResBlock is shown to achieve higher accuracy. The evolutionary process employs the $(1 + \lambda)$ evolutionary strategy, and the standard CGP [34] approach of using point mutation only, without a crossover operator. The number of columns is set to 30, and *levels-back* is set to 10, which limits the distance that skip connections can span. The available convolution kernel sizes are 3×3 and 5×5; this differs from other works such as Genetic CNN [15] that mostly use 3×3 kernels.

CNN-GA [19]. A fully automatic method, CNN-GA does not limit the depth of the evolved networks, so that human input is not required to specify a maximum. This method encodes each architecture as a linked list of nodes. Each node can be either a pooling layer or a building block, termed a *skip layer*. The structure of a skip layer is based on the residual block used in ResNet [9]. CNN-GA does not enable skip connections outside of the skip layers; concatenation is also not used in the evolved architectures. In contrast with Genetic CNN [19], the total number or location of the pooling layers do not need to be fixed by the user for CNN-GA. In the evolutionary process, for each generation, a set of offspring is created by using binary tournament selection to choose parents, and applying mutation and crossover operators. The next generation is produced by environmental selection; i.e., binary tournament selection is performed from the offspring and the parent generation. The mutation operator is a point mutation that can add, modify or delete a random node. The crossover is based on one-point crossover: each parent architecture is split in half at random,

and the two halves are exchanged between the parents.

Large-Scale Evolution [16]. This approach is considerably more general than most others, exploring a large search space: the architectures have unbounded depth, and moreover most other parameters are not fixed. In particular, the learning rate is evolved as well as the architectures; in addition, parameters such as the number of feature maps per convolution layer can be evolved to arbitrary values. The algorithm starts with architectures of only a single layer and no convolutions, so that the structure is completely evolved. Weight inheritance is employed between parents and children to improve efficiency. The evolution is based on a mutation operator only. However, the graph structure is not fully general; the algorithm considers a 'convolutional backbone' that is used when adding convolution operators, and is prioritised when performing edge mutations. It is noted that experiments with recombination (crossover) were performed, but the results are stated as being worse than mutation only, and not reported in the original work. This method requires extensive computational resources (about 2,750 GPU days).

Hierarchical Representations [17]. This work extends the modular design pattern used in architectures such as ResNet [9] to propose a hierarchical approach. An architecture is restricted to be composed of connected units (referred to as *motifs*), each of which is itself composed of smaller motifs. Each of the motifs is a directed acyclic graph. The lowest level of the hierarchy consists of the basic operations (convolution, etc.). The representation does not directly encode skip connections with summation, but instead these can be evolved as the equivalent combination of concatenation followed by a 1×1 convolution. The algorithm uses mutation only; this is based on randomly changing one of the operations in a graph. The user is required to specify the number of levels in the hierarchy, the number of motifs at each level, and the number of nodes in each graph (motif).

Regularized Evolution (AmoebaNet) [40]. This method uses mutation to evolve small cells of fixed size that are manually stacked to produce the final architecture. Specifically, a *normal* cell and a *reduction* cell are evolved separately; the reduction cell reduces the spatial dimensionality of the input. The key contribution of the work is an 'aging evolution' method. A single population is maintained, with children added to the existing population. The oldest CNNs are always removed first (rather than those of lowest fitness). This is believed to improve accuracy by preferentially retaining CNNs that maintain good performance after being re-trained or after a slight mutation.

HGAPSO [41]. This is a hybrid two-level optimisation method using PSO to search for the overall structure of the CNN, and GA to optimise the skip connections. Specifically, the architecture consists of a sequence of blocks; GA is used to select the number of blocks, and the number of layers in each block. The skip connections are encoded with a binary string, similar to Genetic CNN [15] and optimised via GA. This method can be considered fully automatic as it requires no human input. However, the evolved CNN is fixed to use a linear structure similar to the 'convolutional backbone' in Large-Scale Evolution [16]. PSO has also been applied for multi-objective CNN evolution in [42].

Table 2.1 summarises these methods. In some cases, whether a method should be classed as GA or GP was unclear. In this case, the method was categorised as GA if the encoding is fixed-size, and GP if variable complexity individuals were permitted.

2.4.3 GP for Evolving Deep Structures for Image Classification

GP has also been used directly for image classification, without using CNNs. In this case, the GP tree itself can constitute a deep structure that performs image classification. This section provides a brief review of other works applying GP for classifying images.

Table 2.1: Overview of previous EC methods for CNN architecture search.

Method	EC Type	Crossover Operator	Category
Genetic CNN [15]	GA	Yes	Semi-automatic
CGP-CNN [18]	GP	No	Semi-automatic
Hierarchical Representations [17]	GA	No	Semi-automatic
Regularized Evolution [40]	GA	No	Semi-automatic
CNN-GA [19]	GA	Yes	Fully-automatic
Large-Scale Evolution [16]	GP	No	Fully-automatic
HGAPSO [41]	GA & PSO	Yes	Fully-automatic

Three-Tier GP [43]. This approach uses a GP tree comprising three distinct components (tiers). In the lowest tier, filtering functions accept matrices, either from the raw image pixels or from the output of other filtering functions; the result is a new matrix. The middle tier consists of aggregation functions that accept a matrix as input and output a real number. The uppermost tier consists of classifying functions that take as input two or more real numbers to produce another real number. The output of the root node determines the class label.

Two-Tier GP [44]. This demonstrates improved performance relative to Three-Tier GP [43], while being conceptually simpler in that it only has two tiers: aggregation and classification (i.e., the filtering tier is removed). The simplified structure was achieved mainly by improving the flexibility of the aggregation functions.

ConvGP [45]. Inspired by CNNs, this approach integrates the convolution operation into a GP tree, using a three tier structure (convolution, aggregation and classification).

Classification using High-Level Features [46–49]. A variety of GP methods have been proposed for image classification using high-level feature descriptors. These include GP-HoG [46], GP-RDP [47], MLGP [49] and GP-GLF [48].

One-Shot GP and Compound-GP [50]. GP has also been applied to the problem of image classification in the case of a very small training set. Here, GP for feature extraction enables effective generalisation from limited training data, for binary image classification.

Evolving Image Descriptors [51–54]. GP has been applied to the task of automatically generating image descriptors, which can be used to produce a feature vector for an image.

Texture Classification [55]. This is a GP method designed specifically for the task of texture classification; i.e., classifying images on the basis of textural patterns (this differs from, for example, classification based on specific objects represented in the image).

2.5 Chapter Summary

It can be seen that using EC methods to develop new CNN architectures is a highly active research area. Yet most works have focused on aiding human researchers, rather than developing a completely automatic method. In many cases, the required human input is relatively straightforward: e.g., CGP-CNN [18] requires only choosing a maximum depth. However, for other methods, detailed understanding of the algorithm is required to choose the parameters: for example, to apply the Hierarchical Representations [17] method to a new data set, the user must understand the hierarchical graph representation to specify the number of levels, the size of each motif, and the number of motifs per level. In addition, most methods do not use crossover, even though this is a core component of standard EAs. Only a small number of methods such as Large-Scale Evolution [16], CNN-GA [19] and HGAPSO [41] require no human input, but each of these has limitations. Hence, development of completely automatic methods for architecture evolution is still an emerging area of research [19].

Chapter 3

The Proposed Method

This chapter describes the CNN encoding strategy, and introduces the proposed graph-based GP method (termed GraphGP-CNN) for automatic evolution of CNN architectures.

3.1 Chapter Goals

The overall goal was to develop a graph-based GP algorithm, and its associated encoding strategy, that addresses the limitations of existing methods by allowing evolution of arbitrary graph structures. The key contributions of this chapter are the new encoding strategy, its initialisation method and new genetic operators that allow crossover and mutation of arbitrary graph structures.

3.2 Encoding Strategy

As discussed earlier, the components of a CNN can be considered as a directed acyclic graph. This section details the proposed graph-based encoding. A general graph can be described as an ordered pair $G = (V, E)$ with vertices V and edges E. For a directed acyclic graph, each edge is associated with a direction, and there is a restriction that there is no path from a node v to itself. For this CNN architecture encoding, there is a further restriction that there is always *precisely one* node r such that there are no edges ending at r (but there may be edges beginning at r). In the following, this is termed the root node, which represents the output of the CNN.

In this encoding, each node in the graph corresponds to either a building block, an operator, or a reference to the input data. We assign each node a type, which is one of SUM, CONCAT, CONV, MAX, AVERAGE and INPUT. In addition, CONV nodes also have a $feature\_maps$ attribute, which specifies the integer number of channels in the output of the convolutional layer. In this encoding, a convention is chosen that an edge from node n_1 to node n_2 represents n_1 taking its input data from n_2. This is *opposite* to the direction of data flow in the network. That is, edges are directed from the network's output layer towards the input data (shown in the left-hand side of Fig. 3.1). This convention is arbitrary, but is convenient for the description of algorithms acting on this structure.

Each node type has a fixed *arity*, which specifies the required number of edges beginning at that node. Note that a node may have any number of edges ending at that node. Nodes of type INPUT are leaf nodes and so have arity 0; SUM and CONCAT nodes are defined to have arity 2; and all other nodes have arity 1. Summation and concatenation of arbitrary numbers of arrays can still be represented by chaining SUM and CONCAT nodes.

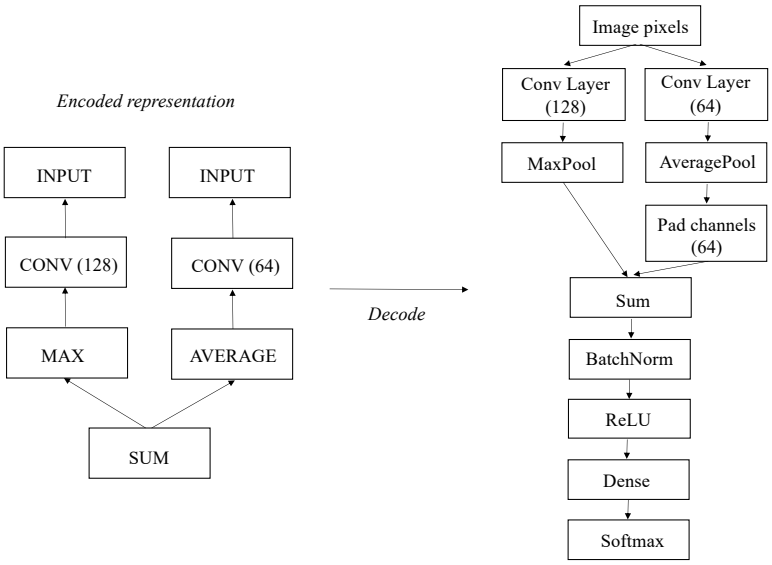


Figure 3.1: An example of an encoded representation and its corresponding decoded CNN architecture.

3.3 Definitions

This section provides mathematical formalisms relating to graph-based CNN structures that are used throughout this work.

The predicate $edge(m, n)$ is defined as true if and only if there exists a directed edge from node m to node n.

We define the predicate $path(m, n)$ as indicating whether a directed path exists between two nodes m and n. That is, $path(m, n)$ is true if and only if there exists a sequence of nodes $v_1, \ldots, v_k$ with $v_1 = m$ and $v_k = n$, such that $edge(v_i, v_{i+1})$ holds for all i where $1 \leq i < k$. In particular, $path(m, n)$ is true if $m = n$.

By analogy with tree-based GP representations, the **children** of a node n are defined as the set of nodes that can be reached by a single directed edge beginning at n. That is,

$$children(n) = \{m : edge(n, m)\}. \tag{3.1}$$

For a given node, the **pooling index** is the maximum number of pooling nodes (including potentially the node itself) along any path from that node to an INPUT node. Formally, the pooling index is defined recursively as

$$pooling\text{-}idx(n) = \max_{m \in children(n)} pooling\text{-}idx(m) + \begin{cases} 1 & \text{if } type(n) \in \{\text{AVERAGE}, \text{MAX}\} \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Similarly, the **depth** of a node is defined as the maximum number of edges along any path from that node to an INPUT node. Formally,

$$depth(n) = \max_{m \in children(n)} [1 + depth(m)] \tag{3.3}$$

where the value of this maximum is zero if the node n has no children.

3.4 Graph Structure and Decoding Method

This section provides further details about the graph structure, and describe the decoding method used to transform this representation (genotype) into the final CNN architecture (phenotype). Fig. 3.1 gives an example of the encoded representation, and shows its corresponding decoded CNN architecture.

In the following, the available node types are detailed, with justifications for these design choices. A CONV node represents a convolutional building block in the final CNN, which consists of a batch normalisation [56], a ReLU activation function [57] and a convolutional layer with a stride of 1, in that order. This 'pre-activation' ordering of batch normalisation and ReLU is inspired by DenseNet [10], and this ordering has also been shown in [37] to achieve superior performance in ResNet, compared to the original ResNet ordering [9]. In addition, the use of batch normalisation and ReLU components is also consistent with other architecture search methods such as CGP-CNN [18] and CNN-GA [19]. An exception is for the first CONV nodes in the architecture (i.e., those that connect directly to the input image pixels). These are decoded to a convolutional layer only, *without* any batch normalisation or ReLU layers.

The 'same' padding mode and the 1×1 stride are used, so that the image dimensions do not change after applying convolution. The number of feature maps is configurable and is a parameter of the CONV node. In contrast, in this project a fixed filter (kernel) size, which is always 3×3, is used. This choice is made because 3×3 is a very commonly used filter size in manually designed CNNs such as ResNet [9] and DenseNet [10]; in addition, it has been suggested that filters of other sizes can be replaced by stacks of 3×3 filters without losing representational ability [58]. Thus, using a fixed filter size acts to reduce the search space.

The MAX and AVERAGE node types represent max and average pooling layers, respectively. These use a kernel size of 2 and stride of 2 for both max and average pooling layers, so that these layers always halve the spatial dimensions of the input data. Greater reductions in dimension can then be achieved by stacking these pooling layers. The INPUT node type is used to reference the current image (or minibatch of images) from the training or test data to which the CNN is being applied. Conceptually, only one INPUT node is required; however, it can be convenient for the algorithm implementation to allow multiple INPUT nodes, all of which are considered equivalent.

The SUM and CONCAT nodes represent the summation and concatenation operations, respectively. During the decoding step, if the dimensions of the two inputs to either SUM or CONCAT nodes are not equal, a max pooling layer is added to downsample the larger feature map, such that the dimensions of the two feature maps become equal. This is based on the procedure used in [18] and [19]. When decoding a SUM node, if the number of channels of the two inputs differ, padding channels containing all zeros are added to the input with the smaller number of channels, so that the number of channels is equal (see Fig. 3.1). This approach is based on a method discussed in the ResNet paper [9], which also suggests 1×1 convolutions as an alternative for adjusting channel dimensionality. For this decoding method, zero-padded channels are used as this technique results in fewer parameters compared to 1×1 convolutions.

To construct the final CNN architecture from the encoded representation, each node in the graph is transformed into its corresponding CNN layers, as detailed above. Finally, several components are added at the end of the CNN:

1. A batch normalisation layer, followed by a ReLU activation layer.
2. A fully-connected layer with a number of nodes equal to the number of image classes.
3. A softmax activation function.

3.5 Initialisation Algorithm

This section presents an algorithm for random generation of a CNN architecture represented by the proposed encoding strategy, with the overall framework given in Algorithm 1.

Algorithm 1 Framework of the Initialisation Method

Output: The root node r of the randomly generated graph.

1: $d \leftarrow$ Generate a random integer depth greater than zero
2: $Q \leftarrow$ A new FIFO queue
3: $S \leftarrow$ A new, empty multiset
4: $level \leftarrow 0$
5: $nonleaf \leftarrow$ False
6: Generate a random non-null, non-INPUT node r as the root, and add the pair $(r, 0)$ to Q
7: **while** Q is not empty **do**
8: $(n, \ell) \leftarrow$ Dequeue a node and level from Q
9: $a \leftarrow$ Get the arity of n
10: **if** $\ell \neq level$ **then**
11: $nonleaf \leftarrow$ False
12: $level \leftarrow \ell$
13: **end if**
14: $i \leftarrow 0$
15: **while** $i < a$ **do**
16: **if** $\ell + 1 = d$ **then**
17: Add an edge from n to a new INPUT node
18: **else**
19: $R \leftarrow$ Compute a set of disallowed node types
20: $c \leftarrow$ Generate a random node restricted by R
21: **if** $c = null$ **then**
22: $S \leftarrow S \cup \{n\}$
23: **else**
24: Add an edge from n to c
25: **if** $type(c) \neq$ INPUT **then**
26: Enqueue the pair $(c, \ell + 1)$ to Q
27: $nonleaf \leftarrow$ True
28: **end if**
29: **end if**
30: **end if**
31: $i \leftarrow i + 1$
32: **end while**
33: **end while**
34: Add missing edges to all nodes in S, with root r
35: **if** $pooling\text{-}idx(r) > \log_2(\min(W_{\text{image}}, H_{\text{image}}))$ **then**
36: Repeat from line 1
37: **end if**
38: **return** r

First, an integer depth d (line 1) is randomly chosen, which is referred to as the *target depth* within this initialisation algorithm. This allows for CNNs of a variety of depths when generating a population, which may increase the likelihood of obtaining an architecture with an appropriate depth for the target data set. The algorithm then uses a standard first-

in, first-out (FIFO) queue to construct the graph in a breadth-first order.

Each item in the queue contains both a node and also the level ℓ of the node; the level is equal to the minimum number of edges from the node to the root. The level information is used to avoid generating nodes that exceed the target depth. Nodes with a level $\ell = d - 1$ are constrained to have only INPUT nodes as children; nodes of type INPUT are leaf nodes, which prevents the graph from growing further (lines 16-17). Otherwise, for $\ell < d - 1$, the algorithm randomly generate new child nodes (lines 19-29).

The set R (line 19) is used to determine node types that are disallowed, in order to avoid generating invalid or suboptimal architectures (detailed in Algorithm 2).

We note that a simple breadth-first algorithm, in which new child nodes are generated for each current node, is only capable of producing trees. To enable generation of directed acyclic graphs, the procedure that generates random nodes can return NULL (lines 21-22). The NULL value is used to indicate that rather than adding an edge to a new node, instead an edge should be added to reference an existing node, producing a directed acyclic graph. Because at this stage in the algorithm, the graph has not been fully constructed, this process is deferred by adding the node to a multiset S for later processing (line 22). Hence the overall algorithm can be described as first constructing a tree, and then using the multiset S to add further edges to produce a directed acyclic graph. Otherwise, if a non-NULL randomly generated node is returned, we add the new edge, and also enqueue the new node.

After the main section of the algorithm, further processing is performed to add the 'missing' edges to the tree to produce a graph (line 34); this is detailed in Algorithm 3. In addition, postprocessing is required to validate the number of pooling nodes. Because each pooling node (either MAX or AVERAGE) reduces the spatial dimensions by $1/2$, there is a maximum number of pooling nodes before the image dimensions (width W_{image} and height H_{image}) become 1×1, at which point further pooling is invalid. In the implementation, if there are too many pooling nodes, the main section of the algorithm is repeated until a valid architecture is randomly generated.

Algorithm 2 Determine Disallowed Node Types

Input: A node n with level ℓ; a depth d; the boolean $nonleaf$.
Output: A set R of disallowed node types.

1: $R \leftarrow \emptyset$
2: $b \leftarrow$ True if an edge exists from n to an INPUT node; else False
3: **if** b or not $nonleaf$ **then**
4: $\quad R \leftarrow R \cup \{\text{REFERENCE}\}$
5: **end if**
6: **if** $\ell + 2 = d$ or $type(n) \in \{\text{SUM}, \text{CONCAT}\}$ **then**
7: $\quad R \leftarrow R \cup \{\text{BINARY}\}$
8: **end if**
9: **return** R

3.5.1 Initialisation Algorithm Details

This subsection details the components that are used within the overall framework of the initialisation algorithm previously discussed. Algorithm 2 presents the process for determining the set R of restricted node types. The inputs to Algorithm 2 are the current node n (to which the new nodes will be added) and its level ℓ, and the target depth d for the graph. The boolean value $nonleaf$ is computed in Algorithm 1 and stores whether a node of any type other than INPUT has been added at the current level.

The algorithm first initialises R as an empty set (line 1). If there is already an edge from n to an INPUT node (condition 1), or if no non-leaf node has been added at the current level (condition 2), then REFERENCE is added as a restricted node type (lines 2-5). Adding REFERENCE to the set ensures that it is not possible to generate an INPUT node (see Algorithm 4).

The justification for the first condition is that the nodes with arity $a > 1$ are SUM and CONCAT and it would be redundant to sum or concatenate the input image data with itself. The second condition is used to ensure that the depth of the directed acyclic graph is able to continue growing: if a non-leaf node has not yet been added at this level, adding a INPUT (i.e., leaf) node could cause the generation of the graph to terminate before reaching the target depth.

The second main step in Algorithm 2 determines whether nodes for binary operators (i.e., SUM and CONCAT) are permitted (lines 6-8). The first condition specifies that binary operators are only allowed if the 'parent' node has level $\ell < d - 2$, where d is the target depth for the graph. This condition is added because if the node has a level $\ell = d - 2$, a new SUM or CONCAT node would only be able to take its input from the image input pixels. If this occurred, redundant summation or concatenation of the image pixels would be produced. The second condition disallows edges from one of the two binary nodes to another binary node. This condition is included to prevent the number of branches from rapidly increasing (as this can lead to overly complex and suboptimal CNNs). The CNN encoding itself is still general and edges between binary operators can be generated via crossover or mutation.

Algorithm 3 Add Missing Edges

Input: The multiset S containing each node p with a missing edge; the root node r of the directed acyclic graph.

```
1:  for p in S do
2:      k ← 0
3:      t ← NULL
4:      valid ← False
5:      while k is less than some maximum iteration count do
6:          t ← A random node in the graph with root r
7:          if type(t) ≠ INPUT and not path(t, p) and not edge(p, t) then
8:              valid ← True
9:              break
10:         end if
11:         k ← k + 1
12:     end while
13:     if valid then
14:         Add an edge from p to t
15:     else
16:         Add an edge from p to a new INPUT node
17:     end if
18: end for
```

Algorithm 3 details the process used to add new edges based on the multiset S of 'missing' edges. That is, the algorithm adds edges specified in S so that the tree becomes a directed acyclic graph.

For each node p in the multiset, the algorithm selects a random node n in the graph (line 6). As the node n may not be a valid choice (discussed in the following), the algorithm loops repeatedly until a valid node n is found, up to some maximum number of iterations. If this maximum is exceeded, an edge from p to an INPUT node is added. There are two reasons

why adding an edge from node p to n may not be valid: one is that a cycle could be formed, or an edge from p to n may already exist. In addition, an INPUT node is not considered a valid choice for n (unless the maximum number of iterations is exceeded without identifying a valid n); this is because the goal of this method is to create additional connections between nodes internal to the graph (similar to the concept of skip connections, as discussed earlier).

Algorithm 4 details the method used to randomly generate a node and its parameters; this method is based on that used in the CNN-GA paper [19], but the addition of the set R, which specifies node types that are disallowed. The set R is used to construct a set P of permitted node types (lines 1-2), from which a node type is randomly chosen with a uniform distribution (line 3).

Algorithm 4 Generate Random Node

Input: A set R of restricted node types.
Output: A randomly generated node n.

1: $U \leftarrow \{\text{BINARY}, \text{CONV}, \text{POOL}, \text{REFERENCE}\}$
2: $P \leftarrow U - R$
3: $t \leftarrow$ A random node type in P
4: $n \leftarrow$ A new node
5: **if** $t =$ BINARY **then**
6: $\quad q \leftarrow$ Uniformly generate a number between 0 and 1
7: $\quad$ **if** $q < 0.5$ **then**
8: $\quad\quad n.type \leftarrow$ SUM
9: $\quad$ **else**
10: $\quad\quad n.type \leftarrow$ CONCAT
11: $\quad$ **end if**
12: **else if** $t =$ CONV **then**
13: $\quad n.type \leftarrow$ CONV
14: $\quad n.feature\_maps =$ Random choice in $\{64, 128, 256\}$
15: **else if** $t =$ POOL **then**
16: $\quad q \leftarrow$ Uniformly generate a number between 0 and 1
17: $\quad$ **if** $q < 0.5$ **then**
18: $\quad\quad n.type \leftarrow$ MAX
19: $\quad$ **else**
20: $\quad\quad n.type \leftarrow$ AVERAGE
21: $\quad$ **end if**
22: **else if** $t =$ REFERENCE **then**
23: $\quad n =$ NULL
24: **end if**
25: **return** n

Note that the set U (line 1) contains different values than simply the available node types in the graph representation. That is, rather than including MAX and AVERAGE pooling nodes in U, we simply use a POOL type; similarly, there is a BINARY type, rather than SUM and CONCAT types. The reason for this is that each type in P is assigned equal probability, and so this enables choosing uniformly between a convolutional node, a pooling node, or a binary operation.

If, for example, MAX and AVERAGE pooling types were included separately in P, there would be a higher probability of generating a pooling node than a convolutional node. It is preferable to avoid this as pooling layers reduce the dimensionality and so can lose information if used excessively. Similarly, if there was a higher probability of a binary operation

(SUM or CONCAT) than a convolutional node, this could result in excessively complex CNN architectures.

If the randomly selected node type is BINARY, a random number q is generated (with uniform distribution) between 0 and 1. If it is less than 0.5, a new SUM node is returned; otherwise, a CONCAT node is produced. If the selected node type is CONV, a new CONV node is generated, and the number of feature maps is chosen randomly. Specifically, the number of feature maps is chosen uniformly from $\{64, 128, 256\}$, which are the values used by the CNN-GA approach [19]. In addition, this fixed set of options for the number of feature maps contributes to reducing the search space (and hence improving the efficiency of search). If the randomly selected node type is POOL, a MAX pooling node is generated with probability 0.5; otherwise, an AVERAGE pooling node is used.

The special REFERENCE type is used to implement the case where no new node is generated, but instead an edge to a existing node in the graph should be added (or an INPUT node, if there is no available existing node that would not cause a cycle). When the type is REFERENCE, Algorithm 4 simply returns NULL. This value is then used as a flag by the initialisation method (Algorithm 1), which updates the multiset S of 'missing' edges to be processed, as was earlier discussed.

3.6 Graph-Based Genetic Programming

This section presents a graph-based genetic programming method for evolving CNN architectures, and is organised as follows. Subsection 3.6.1 describes the overall framework of the algorithm, which is mostly standard. Subsection 3.6.2 introduces the new contributions; i.e., mutation and crossover operators operating on the graph-based CNN encoding.

3.6.1 Framework of the Graph-Based GP Method

The overall framework for the evolutionary process is based closely on that given in [19]; this is detailed in Algorithm 5. Figure 3.2 also depicts this process.

Algorithm 5 Framework of the GP Method

1: $P_0 \leftarrow$ Initialise a fixed-size population, using Algorithm 1 to create each individual
2: Compute the fitness of each individual in P_0
3: $t \leftarrow 0$
4: **while** $t <$ *the maximum generation index* **do**
5: $\quad Q_t \leftarrow$ Generate offspring from P_t
6: $\quad P_{t+1} \leftarrow$ Environmental selection from $P_t \cup Q_t$
7: $\quad$ Compute the fitness of each individual in P_{t+1}
8: $\quad t \leftarrow t + 1$
9: **end while**

Note that at each step, to produce the next generation environmental selection is performed from both the current population P_t and the offspring population Q_t, based on [19]. In our experiments, a small population size is chosen (due to the expensive CNN training process), so the use of this environmental selection step is believed to reduce the chance of good CNN architectures being discarded.

Fitness evaluation is performed by training each architecture asynchronously on an available GPU, up to a maximum number of epochs. The accuracy is calculated after each epoch on a held-out fitness evaluation set. The fitness of each individual is set as the maximum accuracy on the fitness evaluation set during training; this is the approach taken by [19].

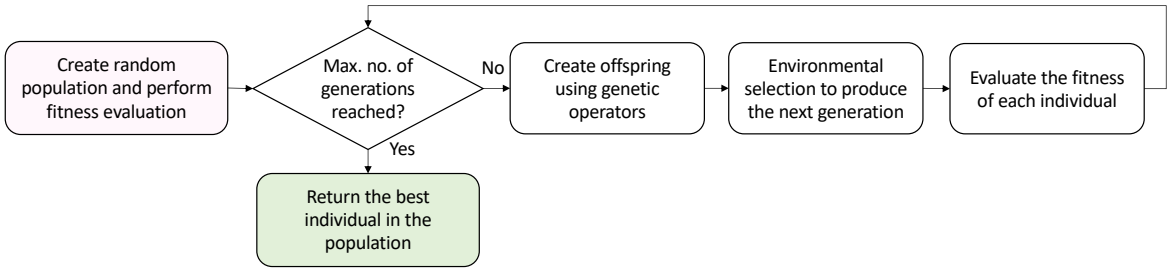


Figure 3.2: Framework of the evolutionary process.

Algorithm 6 details the method used to produce the new population P_{t+1} given the parent population P_t and offspring Q_t. This method is the same as that given in [19], but restated here for clarity. Notably, binary tournament selection is employed; this works by choosing two different individuals at random and selecting the one with the higher fitness. This is believed to provide a balance between selection pressure as well as maintaining the diversity of the population. Top-1 elitism is also employed to ensure that the best CNN architecture is not discarded (lines 9-12).

Algorithm 6 Environmental Selection

Input: The current population P_t with offspring Q_t.
Output: The new population P_{t+1}.

1: $P_{t+1} \leftarrow \emptyset$
2: **while** $|P_{t+1}| < |P_t|$ **do**
3: $p \leftarrow$ Binary tournament selection from $P_t \cup Q_t$
4: **while** $p \in P_{t+1}$ **do**
5: Repeat line 3
6: **end while**
7: $P_{t+1} \leftarrow P_{t+1} \cup \{p\}$
8: **end while**
9: $p_{\text{best}} \leftarrow$ Get the individual with the best fitness in $P_t \cup Q_t$
10: **if** $p_{\text{best}} \notin P_{t+1}$ **then**
11: Replace the individual with the worst fitness in P_{t+1} with p_{best}
12: **end if**
13: **return** P_{t+1}

3.6.2 Offspring Generation and Genetic Operators

This subsection describes the offspring generation method, including the proposed mutation and crossover operators for the graph-based CNN architecture representation.

Algorithm 7 shows the overall process for generating the offspring set Q_t given the parent population P_t. The method employs binary tournament selection (in the same way as the environmental selection process) to choose two different individuals p_1 and p_2. This method was chosen as the CNN-GA paper [19] successfully applies this approach for offspring generation.

The offspring generation method presented in Algorithm 7 differs from [19] in that each offspring architecture is produced by either crossover or mutation, but *not* both. The algorithm also ensures that architectures with excess pooling nodes are not produced: the genetic operator is repeatedly applied until a valid architecture is generated.

Algorithm 8 presents a crossover operator for the graph-based CNN encoding.

Algorithm 7 Offspring Generation

Input: The current population P_t

1: $Q_t \leftarrow \emptyset$
2: $M_{\text{pooling}} \leftarrow \log_2(\min(W_{\text{image}}, H_{\text{image}}))$
3: **while** $|Q_t| < |P_t|$ **do**
4: $p_1 \leftarrow$ Binary tournament selection from P_t
5: $p_2 \leftarrow$ Binary tournament selection from P_t
6: **while** $p_1 = p_2$ **do**
7: Repeat line 4
8: **end while**
9: $r \leftarrow$ Uniformly generate a random number in $[0, 1]$
10: **if** $r <$ *the crossover probability* **then**
11: $o_1, o_2 \leftarrow$ Perform crossover of p_1 and p_2 via Algorithm 8
12: **if** *pooling-idx*$(o_1) > M_{\text{pooling}}$ or *pooling-idx*$(o_2) > M_{\text{pooling}}$ **then**
13: Repeat line 11
14: **end if**
15: **else**
16: $o_1 \leftarrow$ Perform mutation of p_1 via Algorithm 10
17: **if** *pooling-idx*$(o_1) > M_{\text{pooling}}$ **then**
18: Repeat line 16
19: **end if**
20: $o_2 \leftarrow$ Perform mutation of p_2 via Algorithm 10
21: **if** *pooling-idx*$(o_2) > M_{\text{pooling}}$ **then**
22: Repeat line 20
23: **end if**
24: **end if**
25: $Q_t \leftarrow Q_t \cup \{o_1, o_2\}$
26: **end while**
27: **return** Q_t

Algorithm 8 Crossover Operator

Input: Two graphs defined by root nodes r_1 and r_2; the graphs are modified in-place.

1: $n_1 \leftarrow$ Get a random node in the graph defined by r_1
2: $n_2 \leftarrow$ Get a random node in the graph defined by r_2
3: **while** either n_1 or n_2 are INPUT nodes, or either has an edge to an INPUT node **do**
4: Repeat lines 1 and 2
5: **end while**
6: $m_1 \leftarrow$ Get the node along a random outgoing edge from n_1
7: $m_2 \leftarrow$ Get the node along a random outgoing edge from n_2
8: $P_1 \leftarrow$ Disconnect-Subgraph(r_1, n_1, m_1)
9: $P_2 \leftarrow$ Disconnect-Subgraph(r_2, n_2, m_2)
10: Add edge $n_1 \rightarrow m_2$
11: Add edge $n_2 \rightarrow m_1$
12: Use Algorithm 3 to add edges for all nodes in P_1 to nodes in the subgraph defined by m_2
13: Use Algorithm 3 to add edges for all nodes in P_2 to nodes in the subgraph defined by m_1

Algorithm 9 Disconnect-Subgraph

Input: Nodes v_{root}, v_{parent} and v_{subgraph}
Output: A set P of nodes for which edges were removed.

1: $S \leftarrow \{(n, p) : \textit{edge}(p, n)$ and $\textit{path}(v_{\text{subgraph}}, n)$ and not $\textit{path}(v_{\text{subgraph}}, p)\}$
2: **for** $(n, p) \in S$ **do**
3: Remove the edge from p to n
4: **end for**
5: $P \leftarrow \{p : p \neq v_{\text{parent}}$ and $(\exists n)[(n, p) \in S]\}$
6: **return** P

This crossover algorithm is inspired by the single-point crossover in traditional genetic algorithms, and also by the crossover operator used by CNN-GA [19]. The method also follows the general structure discussed by Brown et al. in [36]; they describe the concept of a multipoint crossover in which edges are removed to split a graph into two disjoint components. The two subgraphs can then be swapped between the parents, and reconnected using some combination of random connections and information obtained when disconnecting the subgraphs [36]. The algorithm proposed here follows this general format, but includes specific choices tailored to the case of an acyclic computational graph. Algorithm 9 details the proposed method for how a parent graph is split into two components.

For each parent graph, the crossover algorithm selects a random parent node and a child node. The child node defines a subgraph of all vertices and edges reachable by directed paths from that child node. This subgraph is referred to as the *lower* subgraph; the nodes and edges that are *not* in the lower subgraph are referred to as the *upper* subgraph. These two subgraphs are then disconnected by removing all edges from the upper to the lower subgraph; this process is performed for both original graphs. Next, the subgraphs are swapped between the two parent graphs (lines 10-11). Finally, for each of the parent graphs, the nodes in the upper subgraph that had edges removed are reconnected by adding random edges to nodes in the lower subgraph.

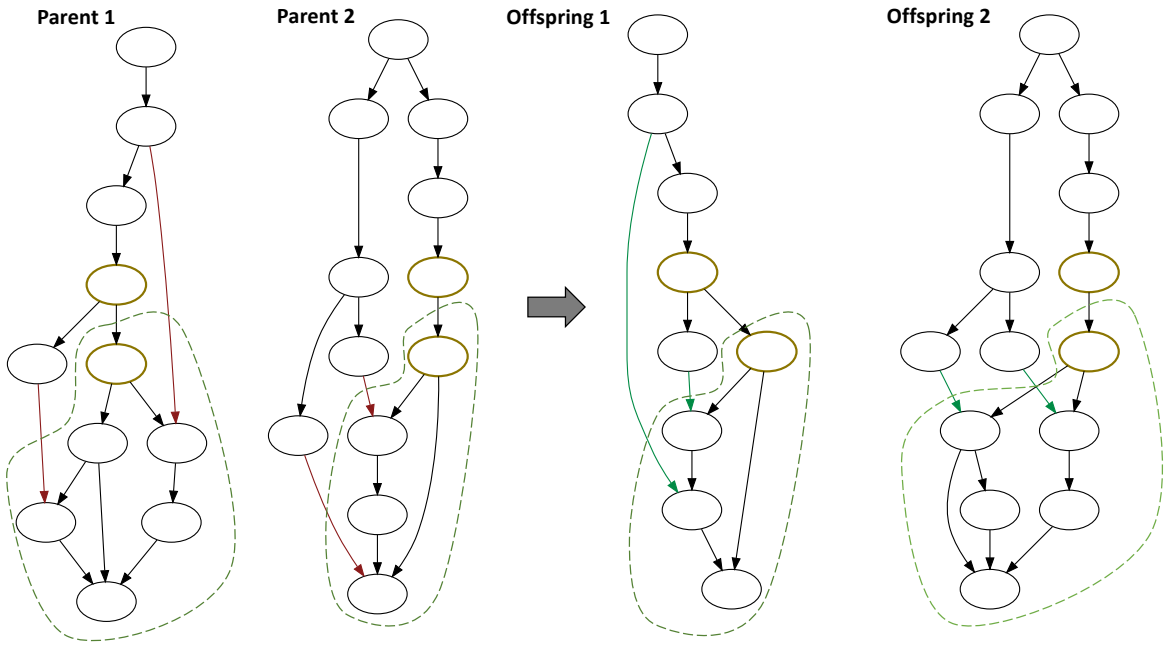


Figure 3.3: Example of the crossover operator applied to two parent graphs.

Figure 3.3 gives an example of this crossover method being applied for two specific parent graphs. The specific parameters for each node (i.e., the node type or number of feature maps) are not shown, as they are ignored by the crossover method. The regions defined by dotted lines indicate the subgraph defined by the child node (i.e., the *lower* subgraph) for each parent and offspring graph. For each graph, the randomly selected parent and child node are highlighted (shown with a gold colour, and a thickened outline).

In the two parent graphs on the left of the figure, the edges highlighted in red are the edges that cross from the upper to the lower subgraph. These can be seen as the edges that cross the dotted line regions. Note that the edge from the parent to the child node is *not* highlighted, as this edge is treated separately by the algorithm. These edges must be deleted in order to disconnect the subgraphs during crossover. In the offspring graphs on the right of the figure, the edges highlighted in green show the new edges that have been randomly added to reconnect nodes that had edges removed.

Algorithm 10 presents a mutation operator for the graph-based CNN encoding.

Algorithm 10 Mutation Operator

Input: A graph $G = (V, E)$ representing a CNN architecture; this is modified in-place.

1: $n_1 \leftarrow$ Select a random non-INPUT node in G
2: $n_2 \leftarrow$ Select the node at a random outgoing edge from n_1
3: $r \leftarrow$ Uniformly generate a random number in $[0, 1]$
4: **if** $r < 0.2$ **then**
5: $n_3 \leftarrow$ Generate a CONV node with a random number of layers in $\{64, 128, 256\}$
6: Remove edge $n_1 \rightarrow n_2$; add edges $n_1 \rightarrow n_3$ and $n_3 \rightarrow n_2$
7: **else if** $r < 0.4$ **then**
8: **while** n_1 has an edge to an INPUT node **do**
9: Repeat lines 1 and 2
10: **end while**
11: $n_3 \leftarrow$ Randomly generate either a SUM or CONCAT node
12: Remove edge $n_1 \rightarrow n_2$; add edges $n_1 \rightarrow n_3$ and $n_3 \rightarrow n_2$
13: Add an edge from n_3 to a random node in G that would not produce a cycle
14: **else if** $r < 0.6$ **then**
15: $n_3 \leftarrow$ Randomly generate either a MAX or AVERAGE pooling node
16: Remove edge $n_1 \rightarrow n_2$; add edges $n_1 \rightarrow n_3$ and $n_3 \rightarrow n_2$
17: **else if** $r < 0.8$ **then**
18: $T \leftarrow \{n_3 \in V$: there exists an edge $n_3 \rightarrow n_1$ in $E\}$
19: **for** each n_3 in T **do**
20: $n_2 \leftarrow$ Select the node at a random outgoing edge from n_1
21: Remove edge $n_3 \rightarrow n_1$; add edge $n_3 \rightarrow n_2$
22: **end for**
23: **else**
24: **if** $type(n_1) \in \{$SUM, CONCAT$\}$ **then**
25: Swap the type of n_1 between SUM and CONCAT
26: **else if** $type(n_1) \in \{$MAX, AVERAGE$\}$ **then**
27: Swap the type of n_1 between MAX and AVERAGE
28: **else**
29: Set the number of feature maps in n_1 to a different, random value in $\{64, 128, 256\}$
30: **end if**
31: **end if**

This algorithm is based on intuitive operations for modifying a CNN architecture, which is similar to the approach taken by the large-scale evolution [16] and CNN-GA [19] methods. The approach presented here differs in that it applies to a very general graph structure. Specifically, the operations include adding a random CONV node; adding a pooling node; adding a SUM or CONCAT node; deleting a random layer; or randomly changing the parameters of a layer. To avoid biasing the evolved structures, all operations are assigned equal probabilities.

Chapter 4

Experiments and Results

4.1 Design of Experiments

The proposed method was tested on the CIFAR-10 and CIFAR-100 benchmark data sets [21]. CIFAR-10 consists of 10 classes and a total of 60,000 32×32 pixel images, of which 50,000 form the training set and 10,000 form the test set. CIFAR-100 is similar to CIFAR-10, but it has 100 classes rather than 10, still with 60,000 images. Thus CIFAR-100 is more challenging than CIFAR-10 in that it not only has more classes, but there are also fewer training images per class relative to CIFAR-10. That is, CIFAR-10 has 5000 training images per class, whereas CIFAR-100 only has 500 per class.

Two main experiments were performed: a random search, in order to provide an indication of the effectiveness of the encoding strategy itself; and an experiment using the graph-based GP method. The random search was tested only on CIFAR-10; whereas the GP method was tested on both CIFAR-10 and CIFAR-100, producing two architectures, with one evolved specifically for each data set.

4.1.1 Random Search Experiment

To demonstrate the effectiveness of the proposed encoding, its performance was tested using a simple random search in which 200 CNN architectures were randomly generated using the proposed initialisation method.

During the architecture search process, only 10% of the full 50,000 training images is used, in order to improve the efficiency of the search. Specifically, during the architecture search, 4,500 images are used as the training set and 500 as the validation set. These subsets of the full 50,000 training images are selected using a stratified sampling process to maintain an even distribution of classes across the training and validation sets.

Each CNN is trained using the 4,500 images, and after each epoch, the performance is tested on the validation set. For each architecture, the the maximum validation accuracy achieved is recorded. After training all architectures on the reduced data set, the three architectures with the best accuracy are selected and re-trained using 45,000 training images and 5000 validation images. The reason for selecting three architectures (rather than the single best architecture) is to improve the likelihood of choosing an architecture that generalises from the reduced training set to the full training set. The epoch that gives the best validation accuracy is recorded for each architecture. Finally, each architecture is re-trained another time, up to the maximum epoch previously determined, using the full training set (50,000 images) *without* a validation set; the accuracy of the re-trained CNN is then calculated on the test set.

This experiment used a slightly different (i.e., earlier) version of the method presented in Chapter 3. Specifically, the two differences were:

1. For the random search, Algorithm 4 was able to randomly generate an INPUT node. In contrast, in the final version of the algorithm presented in Chapter 3, input nodes are not directly generated, but are generated in Algorithm 3 if there is no valid target node that would not cause a cycle. This modification was made in the final version of the algorithm based on the belief that adding a skip connection (or concatenation node) from one of the later (or 'deeper') layers in the CNN back to the input data may not be useful.

2. The convolutional building blocks for the random search used an ordering with the convolutional layer first, then batch normalisation, then ReLU activation; i.e., a post-activation rather than pre-activation ordering. This was based on CNN-GA [19]. Because of this, the final step detailed in Chapter 3 where batch normalisation and ReLU layers were added to the end of the CNN was *not* included (this would be redundant). In the GP experiment, the pre-activation ordering was used as it was found empirically to achieve better performance.

4.1.2 Genetic Programming Experiment

The graph-based GP experiment used the full CIFAR-10/100 training set (50,000 images), with 45,000 images as the training set and 5,000 as the fitness evaluation set. The population size and the number of generations were both 20. A crossover rate of 0.6 and a relatively high mutation rate of 0.4 were used, with the mutation rate selected in order to increase diversity to compensate for the small population size. The final test accuracy is obtained by re-training the CNN with the best fitness using the full 50,000 training images, without a fitness evaluation set, and then calculating the accuracy on the test set.

To reduce the time taken for the experiment, an asynchronous training approach was employed, using a pool of GPUs of heterogeneous models (specifically, a combination of GTX 1080 Ti and RTX 2080 GPUs). To evaluate the fitness of the offspring produced for each generation, each architecture was assigned to the first available GPU in the pool, without regard to the specific model, and trained asynchronously. If all GPUs were in use (either for training other CNN architectures in the experiment, or by other users of the GPU resources) then the system would wait until a GPU became available. Specifically, this was implemented using a centralised queue of architectures to be trained. The main evolutionary process would add all offspring to this queue at each generation. A process was created on each server with GPU hardware to periodically poll this queue, and train any architecture on one of the available GPUs on that server (either two or three GPUs per server).

The specific training approach is relevant in that one of the aims of this project (although only a secondary goal) was to develop a method with relatively good computational efficiency. Because of the asynchronous training method, a distinction must be made between the wall-clock time occupied by the process, and the total number of GPU days, across all GPUs, employed by the algorithm. In addition, another factor that must be accounted for is that, towards the end of evaluating the fitness values for a given offspring set, some or most of the GPUs may be idle. For example, it is often the case that a few architectures may take much longer to train than others, so that these GPUs remain in use, while the other GPUs become idle.

As a result, there are conceptually three distinct measures of computation time:

1. The wall-clock time between the start and end of the evolutionary process.
2. The total number of GPU days during which GPUs were occupied by training CNNs.
3. The total number of GPU days during which GPUs were held by the algorithm, even though some GPUs may be idle during this time. This can be calculated as the wall-clock time multiplied by the number of GPUs used.

Hence, to make meaningful comparisons between works for architecture search requires knowing the GPU models that were used, as well as the timing metric that was reported.

4.1.3 Parameter Settings

For determining the performance of each CNN architecture, stochastic gradient descent (SGD) [59] with momentum was used to train each CNN, with a minibatch size of 128. The softmax cross-entropy loss was used as the loss function. In cases where an error occurred while training a CNN, such as the program exceeding available GPU memory, or a NaN (not-a-number) loss value being obtained, the CNN architecture was simply discarded. Different settings were used for the random search and GP experiments; these are detailed in the following.

For the random search, the training routine and learning rate schedule were based on the settings used in [20]. That is, SGD with a momentum of 0.9 and an initial learning rate of 0.1 was used to train up to 200 epochs. The learning rate was decayed by a factor of 0.1 after the 60th, 120th and 160th epochs. In the CNN population intialisation algorithm, the minimum depth was set to 6 and the maximum depth was 12.

For the GP experiment, SGD with a momentum of 0.9 was used to train each architecture for 50 epochs during the evolutionary process, with cosine annealing of the learning rate starting from 0.024 down to zero over the 50 epochs. This initial learning rate and the use of cosine annealing were based on the NASNet approach [60], and the value of 50 epochs was based on CGP-CNN [18]. After completing the 20 generations, the best architecture was selected and re-trained (with randomly initialised weights) for 350 epochs with cosine annealing, using the same learning rate, and with the full training set. The value of 350 epochs was chosen based on its use in CNN-GA [19]. Using a reduced number of epochs was chosen as a computational efficiency measure; a number of other works use a similar approach [18], [60], [40]. In the GP experiment, an initialisation method was used in which, of the 20 CNNs in the initial population, 7 were randomly generated with depths between 6 and 8; another 7 were generated with depths between 9 and 12; and 6 were generated with depths between 12 and 14. This approach was believed to improve the diversity of the initial population.

In both experiments, to reduce overfitting, weight decay was applied with the coefficient 0.5×10^{-3}, which is the setting used in [20]. In addition, a data augmentation technique was employed based on the method used in [18] and [19]: a random horizontal flip is performed, and a random 32×32 crop is chosen after padding the image by 4 pixels on each side. For both training and testing, the image pixel values were normalised using the mean and standard deviation calculated over all training set images (computed separately for each RGB channel). For CIFAR-10, the mean μ_{10} and standard deviation σ_{10} were computed as

$$\mu_{10} = \begin{bmatrix} 0.4914, & 0.4822, & 0.4465 \end{bmatrix} \quad \text{and} \quad \sigma_{10} = \begin{bmatrix} 0.2470, & 0.2435, & 0.2616 \end{bmatrix}.$$

For CIFAR-100, the mean μ_{100} and standard deviation σ_{100} were computed as

$$\mu_{100} = \begin{bmatrix} 0.5071, & 0.4865, & 0.4409 \end{bmatrix} \quad \text{and} \quad \sigma_{100} = \begin{bmatrix} 0.2673, & 0.2564, & 0.2762 \end{bmatrix}.$$

Table 4.1: Results achieved by the proposed method in comparison with peer competitors.

	CIFAR-10 Accuracy (%, Test)	**CIFAR-100 Accuracy (%, Test)**	**Parameters**	**GPU Days**
ResNet (depth=110) [9]	93.57	74.84	1.7 M	-
ResNet (depth=1,202) [9]	92.07	72.18	10.2 M	-
VGG [7]	93.34	71.95	20.04 M	-
DenseNet ($k = 24$) [10]	94.17	76.58	27.2 M	-
Hierarchical Evolution [17]	96.37	-	-	300
Block-QNN-S [14]	95.62	79.35	6.1 M	90
Genetic CNN [15]	92.90	70.97	-	17
Large-Scale Evolution [16]	94.60	-	5.4 M	2,750
Large-Scale Evolution [16]	-	77.0	40.4 M	2,750
CGP-CNN (ResBlock) [18]	94.02	-	1.68 M	27
CGP-CNN (ConvBlock) [18]	93.25	-	1.52 M	-
CNN-GA [19]	95.22	-	2.9 M	35
CNN-GA [19]	-	77.97	4.1 M	40
Random search (trial #1, best accuracy)	92.45	-	1.45 M	1.33
Random search (trial #1, fewest parameters)	90.98	-	0.83 M	1.33
Random search (trial #2, best accuracy)	92.57	-	1.14 M	1.33
Random search (trial #2, fewest parameters)	91.44	-	0.75 M	1.33
GraphGP-CNN	**95.62, 95.33**	-	9.31 M	40 (approx.)
GraphGP-CNN	-	**75.89**	7.14 M	48 (approx.)

4.2 Results and Discussions

Table 4.1 shows the accuracy obtained on the CIFAR-10 and CIFAR-100 test sets (each 10,000 images) by the random search and by the GP method (GraphGP-CNN); these are compared with 10 state-of-the-art peer competitors, including both hand-crafted CNNs and CNNs produced by automatic design algorithms.

All accuracies for the random search and the GraphGP-CNN method were obtained on the test set after re-training the discovered architecture using the full 50,000 training images. For each model, the classification accuracy, the number of trainable parameters (in millions) and the GPU days required to determine the architecture are shown; this is a similar table format to that used in [19]. For the hand-crafted architectures, the 'GPU days' column is marked '-'. For the CGP-CNN approach, two variants are shown (ConvBlock and ResBlock). For GraphGP-CNN on CIFAR-10, the CNN parameters (weights) were trained twice, using the same architecture; commonly, re-training the weights for the same architecture can give different results due to random variation. Hence, two accuracies are shown in the table.

Two separate trials of the random search are included. For each trial, the results from the CNN with the best accuracy and from the CNN with the fewest parameters are shown. For the GPU days required, the architecture search itself takes about 10 hours using two GTX 1080 Ti GPUs; the re-training of the final three architectures requires approximately 12 hours with one GPU, such that the process takes approximately 1.33 GPU days overall.

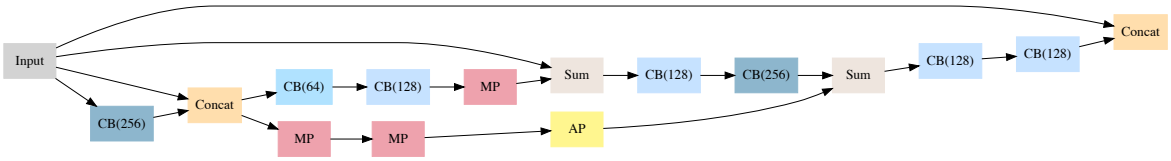


Figure 4.1: The best CNN architecture generated by the random search (trial #2), with an accuracy of 92.57% after re-training on the full 50,000 training images.

Hence although the classification accuracies achieved by the random search are generally worse than those of the hand-crafted CNNs and other automatic design methods, this approach required significantly less computational resources, and provided promising performance despite the reduced data set during architecture search.

Furthermore, the random search was able to produce models with different complexities (numbers of parameters); for example, one generated CNN achieves 91.44% accuracy with only 750k parameters, after re-training using 50,000 images.

Figure 4.1 shows the best architecture from trial #2 of the random search. In the figure, the 'CB' nodes represent convolutional building blocks, and indicate the number of feature maps, and the 'MP' and 'AP' nodes denote max and average pooling, respectively.

4.2.1 GraphGP-CNN Results

The architecture from the GraphGP-CNN method achieved better test accuracy on CIFAR-10 than ResNet [9] and DenseNet ($k = 24$) [10], and was also better than the architecture produced by Genetic CNN [15], CGP-CNN [18] and Large-Scale Evolution [16]. Fig. 4.2 and Fig. 4.3 show graph representations for the best architecture generated by the GraphGP-CNN search for CIFAR-10 and CIFAR-100, respectively.

The values specified for the GPU days for the GraphGP-CNN algorithm are approximate only. For the CIFAR-10 experiment, a maximum of 12 GPUs, including three of model GTX 1080 Ti and nine of model RTX 2080, were employed for about 3.4 days. For the CIFAR-100 experiment, a maximum of 14 GPUs, including five of model GTX 1080 Ti and nine of model RTX 2080, were employed for about 4.0 days.

However, as earlier discussed, not all GPUs were used at all times; for example, if one CNN architecture required much longer to train than others, the other GPUs would be idle. The values reported are the total GPU days where GPUs were held by the algorithm, calculated as the wall-clock time multiplied by the number of GPUs used. This was chosen as the metric to include as it gives a conservative description of the running time of the algorithm. The actual number of GPU days where the GPUs were occupied by running processes is likely to be lower. It can be seen that the time taken by the GraphGP-CNN algorithm is similar to that taken by CNN-GA [19], even though the search space explored by GraphGP-CNN is more complex; this indicates that GraphGP-CNN is relatively efficient with computational resources.

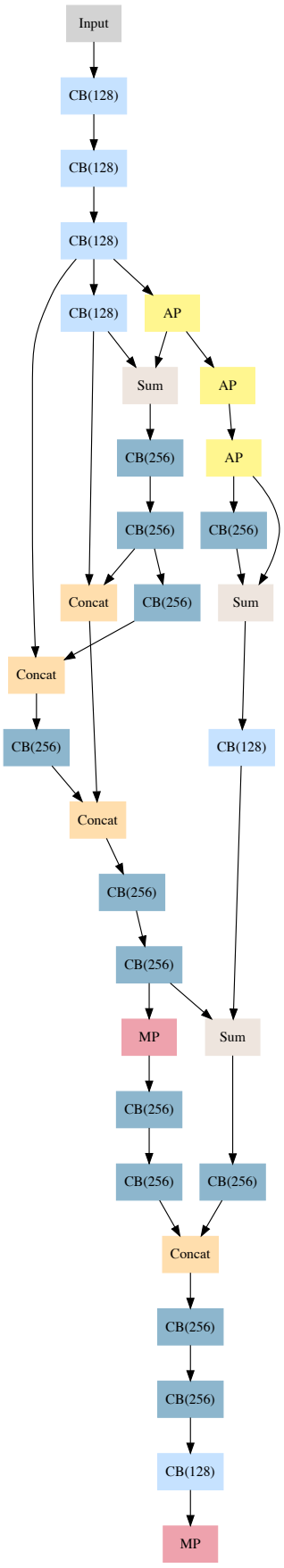


Figure 4.2: Evolved CIFAR-10 architecture.

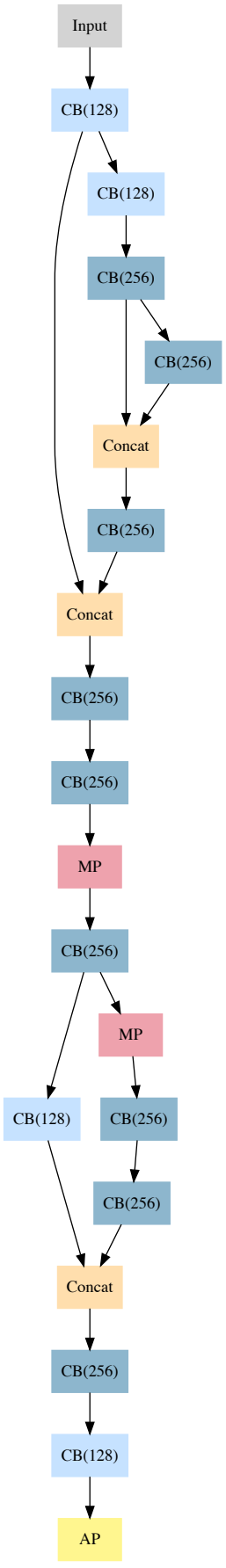


Figure 4.3: Evolved CIFAR-100 architecture.

Chapter 5

Further Evaluation and Analysis

5.1 Comparison of CIFAR-10 and CIFAR-100 Architectures

In this section, the architectures evolved for CIFAR-10 and CIFAR-100 are examined in order to assess the differences that have resulted from the evolutionary process for the two data sets. The original experiments were performed using the same initial random state, including the randomly generated initial population of architectures. Hence, the differences in the resulting architectures can be expected to reflect differences in the search space for the two data sets.

Notably, the CIFAR-100 architecture evolved by GraphGP-CNN has fewer parameters (7.14 M), compared to the CIFAR-10 architecture (9.31 M). This is not typical, as other architecture search algorithms generally produce architectures with more parameters for CIFAR-100; for example, see the results for Large-Scale Evolution [16] and CGP-CNN in Table 4.1. An architecture with more parameters for CIFAR-100 would normally be expected due to its increased complexity relative to CIFAR-10.

In addition, the architecture evolved for CIFAR-100 seems relatively worse compared to that evolved for CIFAR-10. That is, the CIFAR-10 architecture achieves better accuracy than most of the competitors in Table 4.1. The CIFAR-100 architecture is still better than some manually designed architectures such as ResNet-110 [9], but has worse accuracy than a number of other architectures.

To evaluate this further, an experiment was performed in which the architecture evolved for CIFAR-10 was trained on the CIFAR-100 data set. There was no change to the architecture other than the final fully-connected classifier layer. The results of this experiment are shown in Table 5.1; the accuracy of the architecture evolved for CIFAR-100 is also restated for comparison.

Table 5.1: Comparison of the architectures evolved for CIFAR-10 and CIFAR-100.

Data Set	Architecture	Training Epochs	Parameters	Test Accuracy (%)
CIFAR-100	Evolved (CIFAR-100)	350	7.14 M	75.89
CIFAR-100	Evolved (CIFAR-10)	350	9.31 M	77.41
CIFAR-100	Evolved (CIFAR-100)	50	7.14 M	73.76
CIFAR-100	Evolved (CIFAR-10)	50	9.31 M	73.43

As shown in the table, for 350 epochs of training, the architecture evolved for CIFAR-10 achieves significantly better performance on the CIFAR-100 data set than the architecture evolved for CIFAR-100. However, for 50 epochs of training, as used during the architecture search process, the architecture evolved specifically for CIFAR-100 achieves better test accu-

racy (73.76% compared to 73.43%). This suggests that an architecture that achieves the best performance with 50 epochs of training may not be the best architecture with 350 epochs of training. In particular, since the technique of using a reduced number of epochs during the evolutionary search was employed for both CIFAR-10 and CIFAR-100, but worse results were obtained for CIFAR-100, it may be the case that using a reduced number of epochs is less effective for CIFAR-100. For example, this may be due to the increased search space complexity of CIFAR-100. Hence, these results indicate that accuracy may be able to be improved by using a full number of training epochs (350) during the evolutionary search for CIFAR-100.

5.2 Ablation Studies for the Evolved Architectures

This section analyses the evolved architectures by deconstructing or simplifying them, in particular by removing concatenation and skip connections (i.e., summation nodes). The goal is to determine whether the specific graph-based structure evolved is of benefit relative to simpler structures. In addition, this enables determining the amount of impact on the classification accuracy of the specific connectional structure, relative to other factors such as the depth of the CNN.

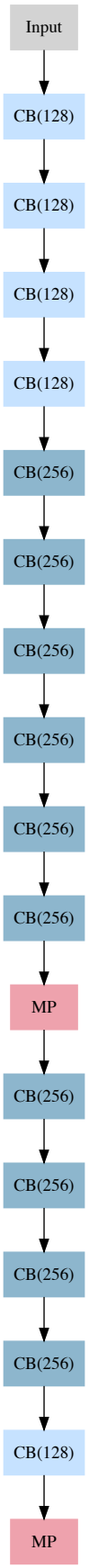
Figures 5.1 and 5.2 show modified versions of the architectures evolved for CIFAR-10 and CIFAR-100, respectively, where all summation and concatenation nodes were removed to produce "linear" structures. Specifically, this was achieved by starting from the final layer, and performing a graph traversal where each time a SUM or CONCAT node was encountered, it was replaced by one of its two child nodes. The particular child node was selected as the one that has the greater depth, where the depth is the maximum number of edges from that node to an INPUT node (as defined in Section 3.3). Selecting the child node with the greater depth results in a linear structure that most closely matches the original structure (in contrast, if the child node with the lower depth was chosen, this would have the effect of reducing the total depth of the architecture).

Table 5.2: Accuracies for both the evolved CNNs and the manually simplified CNNs.

Data Set	Architecture	Training Epochs	Parameters	Test Accuracy (%)
CIFAR-10	Evolved (original)	350	9.31 M	95.33, 95.62
CIFAR-10	Modified (linear)	350	7.55 M	95.55
CIFAR-10	Evolved (original)	50	9.31 M	93.71
CIFAR-10	Modified (linear)	50	7.55 M	93.42
CIFAR-100	Evolved (original)	350	7.14 M	75.89
CIFAR-100	Modified (linear)	350	5.67 M	75.76
CIFAR-100	Evolved (original)	50	7.14 M	73.76
CIFAR-100	Modified (linear)	50	5.67 M	73.69

Table 5.2 shows the classification accuracies obtained by the architectures evolved automatically by the proposed method on CIFAR-10 and CIFAR-100, in comparison with the accuracies obtained after simplifying the architectures to remove all summation and concatenation (producing linear structures).

For each architecture in Table 5.2, the test accuracy is included for two cases: with the CNN trained for either 50 epochs or 350 epochs (50 epochs is the value that was used during the architecture search for fitness evaluation). For the CIFAR-10 architecture, two accuracy values are shown for the 350 epoch case as re-training this architecture twice gave two results (due to factors such as random initialisation, or possibly non-deterministic GPU



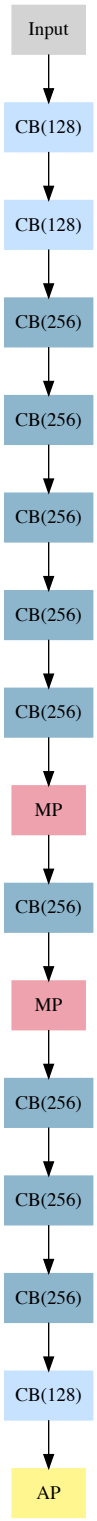


Figure 5.1: CIFAR-10 simplified architecture. Figure 5.2: CIFAR-100 simplified architecture.

computations during training).

Notably, for both CIFAR-10 and CIFAR-100, the difference in the test accuracy after removing the summation and concatenation is small. This suggests that most of the accuracy of these architectures is due to their overall structure (i.e., the depth of the CNN, and the specific arrangement of convolutional blocks and pooling layers), rather than the skip connections. For CIFAR-100, the original, evolved architectures have better performance than the simplified architectures for both the 350 epoch and 50 epoch situations.

This indicates that the proposed GP algorithm has been able to effectively evolve useful instances of skip connections and concatenation for CIFAR-100. In addition, because a relatively small modification to the evolved architecture leads to a reduction in the accuracy, this suggests that the evolved architecture may be close to a local optimum. However, this architecture may still be quite distant from the global optimum.

When the evolved (not manually modified) CIFAR-10 architecture was trained for 350 epochs (with two trials), one trial gave better accuracy (95.62%) than the simplified architecture (95.55%), and another trial gave worse accuracy (95.33%). Thus, it is not clear from this data as to whether the evolved summation and concatenation nodes improve accuracy in the 350 epoch case. To determine this more conclusively, re-training both the original and simplified architecture more times would be required in order to assess how much of the discrepancies in the accuracy are due to the CNN structure or to random variation. It was not feasible to perform more repeated trainings in this work due to the large amount of computation time required to train a CNN for the full 350 epochs. However, these results do suggest that the difference between the original and simplified architectures must be minimal in the 350 epoch situation. For the case of 50 epochs, this experiment showed that the original evolved CIFAR-10 architecture was able to achieve slightly better accuracy than the simplified architecture (73.76% vs 73.69%).

Hence, this analysis has shown that the summation and concatenation operations generated by the graph-based GP algorithm are consistent with improved test accuracy relative to a linear architecture with no summation/concatenation when trained for 50 epochs (the value used during the architecture search). The CIFAR-100 architecture also achieved better performance with summation/concatenation operations when using a higher number of epochs (350). For the CIFAR-10 architecture with 350 epochs, because one trial achieved worse accuracy with summation and concatenation than without these operations, this suggests that a graph structure that improves accuracy at a small number of epochs may not be as good when trained for a larger number of epochs. Hence, this indicates that the accuracy achieved by this algorithm could potentially be improved by training CNNs for the full number of epochs during the architecture search. However, this would require a much longer computation time.

5.3 Analysis of Image Misclassifications

This section examines images that are incorrectly classified by the CNNs evolved using the proposed algorithm. The goal is to provide insight into whether the predictions for misclassified images are evidently wrong (i.e., the error would be obvious to a human observer), or occur for more ambiguous cases of the images. Images in which the true class is difficult to discern even for a human are relatively common in both CIFAR-10 and CIFAR-100, because the images are of small size (32×32 pixels), and were produced by downsampling larger source images. This process can lead to loss of information for some images, so that the actual content of the image is not clear.

This form of analysis may be useful in certain real-world situations, where the specific

types of images that are misclassified may matter more than a simple accuracy rate; for example, obvious misclassifications are likely to be more of an issue than misclassifying ambiguous images in a practical scenario. In addition, if an (incorrectly) predicted class label is semantically similar to the actual class label, that will typically be preferable to a semantically distant category (e.g., misclassifying a truck as a car is generally better than misclassifying a truck as a dog).

The test sets for CIFAR-10 and CIFAR-100 both comprise 10,000 images. As a result, the number of misclassified images is in the hundreds or thousands, and so it is not feasible within the time and length constraints of this work to examine each misclassified image. Instead, a random sample of 20 images was taken for both CIFAR-10 and CIFAR-100 from the set of all misclassified images for each data set.

5.3.1 CIFAR-10 Misclassifications

Figure 5.3 shows a random sample of 20 images taken from the set of all CIFAR-10 test set images misclassified by the architecture evolved for CIFAR-10. The original images are 32×32 pixels and were upsampled for inclusion in the figure, leading to the visibly low resolution (i.e., blurriness). At an accuracy of 95.33%, there are 467 misclassified test images for CIFAR-10. Thus, a sample of 20 images is 4.28% of the total number of misclassified images.

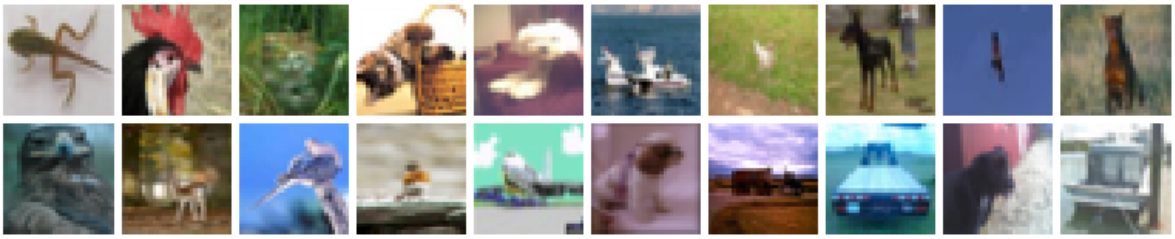


Figure 5.3: A random sample of 20 misclassified images from the CIFAR-10 test set (using the CNN evolved by the proposed algorithm).

Table 5.3 shows the predicted and true class labels for this sample of misclassified images (these correspond to the two rows of images in the figure, with values from left to right). In addition, the table provides a subjective evaluation of the classification difficulty; images in which the contents seem obvious to a human are labelled "clear", whereas images that seem more ambiguous are labeled "amb." Some images for which the contents are neither obvious nor wholly ambiguous are noted as "med." (for medium classification difficulty).

In the following, analysis is provided for selected examples from these 20 images. The top left image of a frog is misclassified as a plane; this is a fairly obvious error, but there are some plausible reasons for this mistake (e.g., for the CNN, the frog's back legs might generate similar feature maps to airplane wings). The image of a chicken (second from the top-left) could be considered the most clear mistake out of these images. Furthermore, it is difficult to determine why it was classified as a dog. It may be the case that there are not many images of only the head of a chicken/bird in the training set (rather than the full body), whereas there may be more partial images of dogs. Similarly, the bottom left image is again obviously a bird to a human observer, but is classified as a cat.

A number of these images are genuinely ambiguous. For example, the third image from the top left (a cat) is quite difficult to discern due to the low resolution. The CNN classifies this as a frog, which may be because of the green colour in the background (likely common for frogs in outdoor environments). Many of the images of cats in CIFAR-10 are in domestic

Table 5.3: Predicted and true class labels for the random sample of 20 misclassified images from the CIFAR-10 test set.

Row 1 Actual	frog	bird	cat	cat	dog	plane	bird	dog	bird	dog
Row 1 Pred.	plane	dog	frog	bird	cat	ship	deer	horse	plane	deer
Row 1 Eval.	clear	clear	amb.	amb.	clear	amb.	amb.	med.	clear	med.
Row 2 Actual	bird	deer	bird	bird	plane	dog	truck	truck	dog	ship
Row 2 Pred.	cat	horse	plane	ship	ship	cat	plane	car	bird	truck
Row 1 Eval.	clear	med.	amb.	amb.	clear	clear	amb.	clear	amb.	clear

(indoor) settings, so a green-coloured background may be less common for them. The sixth image from the top left is labelled in the data set as a plane, but the CNN classifies it as a ship. The background is clearly a body of water, and the image appears to contain a vessel on the surface of the water, so for a human observer this could easily be interpreted as a ship. Another interesting ambiguous case is the fourth image from the bottom left. With a true class label of "bird", this image likely shows a small bird perched on a branch against the background of the sky. Yet the dark colour of this branch can also be interpreted as water, and so this image is also easily viewed as a distant boat on the surface of the sea (the CNN classifies this as a "ship").

Overall, based on this subjective analysis, 9 (45%) of these images are obvious misclassifications, 8 (40%) are highly ambiguous images, and 3 (15%) are of medium difficulty.

5.3.2 CIFAR-100 Misclassifications

Figure 5.4 shows a random sample of 20 images taken from the set of all CIFAR-100 test set images misclassified by the architecture evolved for CIFAR-100. At an accuracy of 75.89%, there are 2,111 misclassified test images for CIFAR-10. Thus, a sample of 20 images is 0.947% of the total number of misclassified images.

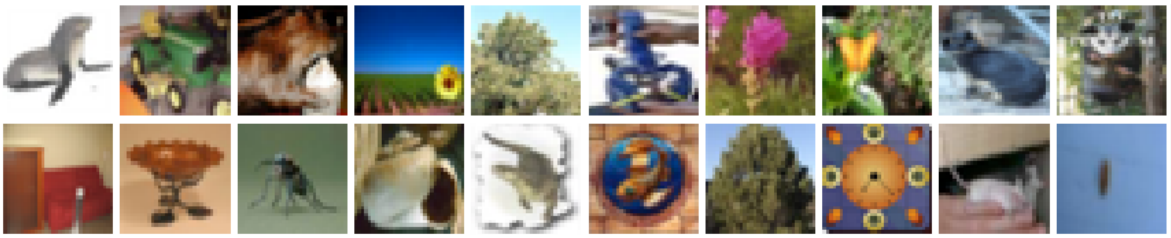


Figure 5.4: A random sample of 20 misclassified images from the CIFAR-100 test set (using the CNN evolved by the proposed algorithm).

Similarly to the CIFAR-10 example, Table 5.4 shows the predicted and true class labels for this sample of misclassified images from CIFAR-100. A subjective evaluation of the classification difficulty is also given.

In contrast to the CIFAR-10 case, there are a larger number of images of "medium" classification difficulty compared to CIFAR-10, based on the subjective analysis. This reflects the more complex nature of CIFAR-100 compared to CIFAR-10. For example, this sample of images shows that the CNN can misclassify oak trees as pine trees, or vice versa. For a human

Table 5.4: Predicted and true class labels for the random sample of 20 misclassified images from the CIFAR-100 test set.

Row 1 Actual	seal	lawn mower	beaver	sun flower	pine	lobster	tulip	butter-fly	rabbit	possum
Row 1 Pred.	kanga-roo	tractor	fox	plain	oak	man	orchid	tulip	dolphin	chimp
Row 1 Eval.	clear	amb.	med.	med.	med.	med.	clear	med.	clear	clear
Row 2 Actual	couch	bowl	beetle	snail	otter	bowl	pine	clock	mouse	cater-pillar
Row 2 Pred.	ward-robe	cup	bee	bowl	mouse	snake	oak	snake	rabbit	rocket
Row 2 Eval.	med.	clear	med.	clear	amb.	amb.	med.	clear	clear	amb.

observer, it is possible to differentiate the two trees, but doing so requires specific knowledge and is still difficult given the low resolution of the images. Thus, these cases were noted as "med." Another example is the image at the bottom-left; this has the label "couch", but the CNN classifies it was "wardrobe". The image actually appears to contain multiple items of furniture, and so the CNN may have been classifying the image as a wardrobe based on the item on the left of the image. There may not have been enough training images similar to this for the CNN to be able to determine that the couch is the most relevant object in the scene.

There are still several ambiguous images. For example, the second image from the top left (a lawn mower) is classified by the CNN as a tractor, and this classification seems reasonable as the low resolution and the background provides minimal contextual cues as to the scale of the object. A very ambiguous image is at the bottom-right: the class label is noted in the data set as "caterpillar". Even for a human, this is hard to discern; in fact, the CNN's classification of "rocket" seems quite plausible, as the blue background could be interpreted as the sky. The sixth image from the bottom left has the class label "bowl", but is predicted by the CNN as a "snake". It appears that this a bowl with an image of a fish, but the fact that the object is a bowl is difficult to determine even for a human observer. Evidently, the classification by the CNN is based only on the object in the centre of the image. Similar to the earlier example with the couch, it appears that images with multiple components may be likely to result in incorrect predictions.

The most obvious error is the classification of a clock (the eighth image from the bottom-left) as a snake. It is very difficult to determine a potential reason for this classification. Another example is the seal at the top-left, which is classified as a kangaroo; however, this error is more understandable in that both categories are animals. Overall, based on this subjective analysis, 8 images (40%) are obviously misclassified; another 8 images (40%) are of "medium" difficulty, where the predicted label is reasonable given the content of the image, but does not match the true class; and 4 images (20%) are ambiguous.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this project, a novel graph-based genetic programming method for evolving CNN architectures has been developed based on a directed acyclic graph encoding strategy. In contrast to encoding methods employed by many previous evolutionary computation–based architecture search approaches, this proposed representation imposes fewer restrictions, thus allowing more architectures to be represented. The key conclusions of this project are as follows:

1. It appears that this encoding strategy has good potential: a random search using only 10% of the CIFAR-10 training data during the architecture search still achieved promising results after the three best discovered architectures were re-trained on the full 50,000 training images. In particular, the method showed good capability for generating models with a variety of complexities. Due to the reduced training set used during the random search, the random search also required much less computation time compared to other methods.

2. As expected, the proposed GraphGP-CNN algorithm improved upon the random search, achieving a better accuracy than ResNet-110 [9] and DenseNet ($k = 24$) [10]. Moreover, the GraphGP-CNN algorithm was able to obtain a better accuracy on CIFAR-10 than a number of previous architecture search methods, specifically Genetic CNN [15], CGP-CNN [18], CNN-GA [19] and Large-Scale Evolution [16]. For the architecture evolved for CIFAR-100, the results obtained were also competitive with other works. The GraphGP-CNN method was able to achieve these results despite using a large search space where the depth of the network was unbounded, and arbitrary graph structures were allowed. That is, the user of GraphGP-CNN does not need to manually specify a maximum depth or otherwise limit the evolved architectures.

3. As a computational efficiency measure, this work used a reduced number of training epochs (50) during the architecture search, relative to the final number of training epochs (350). This project empirically demonstrated that applying this efficiency measure for graph-based GP for architecture search can achieve good results, especially with CIFAR-10. The experiments in the evaluation component of this work indicated that this reduced number of training epochs may be less effective for CIFAR-100. In particular, these experiments have shown that using the reduced training epochs may slightly distort the accuracy comparisons between two architectures, although overall the method is effective.

6.2 Future Work

This project has explored the potential for using GP to evolve CNN architectures, using a fully automatic approach that does not limit the structure or depth of the evolved architectures. Furthermore, in contrast to other works that often use mutation only, this work emphasises the potential of employing recombination (crossover). Many interesting research directions remain for future work:

1. It could be informative to apply visualisation or interpretation methods to the evolved networks; for example, to assess any learned filters that are suboptimal, and then to incorporate this knowledge to improve the algorithm.

2. A promising option would be to explore other measures for improving computational efficiency that do not rely on reducing the number of training epochs. For example, several works have effectively employed weight sharing between parent and child architectures; for example, [16] and [61]. Alternatively, if more computational resources were available, the algorithm could be run using the full number of epochs during the architecture search, which would be expected to increase accuracy.

3. Another interesting direction could be to develop an algorithm that allows a balance between a semi-automatic and fully-automatic approach. This would enable users to apply the method to new data sets without requiring any specific knowledge, or to input any domain-specific knowledge into the algorithm if available.

4. An approach analogous to transfer learning could be developed. For example, the GP algorithm could be provided with a population of architectures already evolved by the algorithm for one data set (e.g. CIFAR-10). Further generations could then be computed, using a different data set (e.g., CIFAR-100) to optimise the architectures for that data set, while making use of the learned information for the old data set. Alternatively, the algorithm could be provided with a handcrafted CNN architecture, which it could then use as the starting point for further optimisation.

Bibliography

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *ArXiv e-prints*, p. arXiv:1604.07316, Apr. 2016.

[2] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning," *IEEE Transactions on Medical Imaging*, vol. 35, pp. 1285–1298, May 2016.

[3] Y. Nakajima and H. Saito, "Robust camera pose estimation by viewpoint classification using deep learning," *Computational Visual Media*, vol. 3, pp. 189–198, Jun 2017.

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

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems*, pp. 1097–1105, 2012.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov 1998.

[7] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *Proceedings of the 32nd International Conference on Machine Learning*, (Lille, France), 2015.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016.

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

[11] Y. Sun, B. Xue, and M. Zhang, "Evolving deep convolutional neural networks for image classification," *ArXiv e-prints*, p. arXiv:1710.10741, Oct. 2017.

[12] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," in *Proceedings of the 2017 International Conference on Learning Representations*, (Toulon, France), 2016.

[13] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing Neural Network Architectures using Reinforcement Learning," in *Proceedings of the 2017 International Conference on Learning Representations*, (Toulon, France), 2016.

[14] Z. Zhong, J. Yan, and C.-L. Liu, "Practical network blocks design with q-learning," in *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*, (Louisiana, USA), 2018.

[15] L. Xie and A. L. Yuille, "Genetic CNN," in *Proceedings of 2017 IEEE International Conference on Computer Vision*, (Venice, Italy), pp. 1388–1397, 2017.

[16] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-Scale Evolution of Image Classifiers," in *Proceedings of Machine Learning Research*, (Sydney, Australia), pp. 2902–2911, 2017.

[17] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," in *Proceedings of 2018 Machine Learning Research*, (Stockholm, Sweden), 2018.

[18] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, (Berlin, Germany), pp. 497–504, ACM, 2017.

[19] Y. Sun, B. Xue, M. Zhang, and G. G. Yen, "Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification," *ArXiv e-prints*, p. arXiv:1808.03818, Aug. 2018.

[20] Y. Zhu, Y. Yao, Z. Wu, Y. Chen, G. Li, H. Hu, and Y. Xu, "GP-CNAS: Convolutional Neural Network Architecture Search with Genetic Programming," *arXiv e-prints*, p. arXiv:1812.07611, Nov. 2018.

[21] A. Krizhevsky and G. E. Hinton, "Learning multiple layers of features from tiny images," 2009. *Online:* `https://www.cs.toronto.edu/~kriz/cifar.html`.

[22] W. Irwin-Harris, Y. Sun, B. Xue, and M. Zhang, "A graph-based encoding for evolutionary convolutional neural network architecture design," in *Proceedings of 2019 IEEE Congress on Evolutionary Computation (CEC 2019)*, June 2019.

[23] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, (Cambridge, MA, USA), pp. 91–99, MIT Press, 2015.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997.

[26] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training Very Deep Networks," in *Advances in Neural Information Processing Systems*, (Montreal, Canada), 2015.

[27] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway, NJ, USA: IEEE Press, 1995.

[28] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, Nov 1995.

[29] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation*, pp. 69–73, May 1998.

[30] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. New York, NY, USA: Oxford University Press, Inc., 1996.

[31] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[32] J. R. Koza, "Genetic programming as a means for programming computers by natural selection," *Statistics and Computing*, vol. 4, pp. 87–112, Jun 1994.

[33] R. Poli, "New ideas in optimization," ch. Parallel Distributed Genetic Programming, pp. 403–432, Maidenhead, UK, England: McGraw-Hill Ltd., UK, 1999.

[34] J. F. Miller, *Cartesian Genetic Programming*, pp. 17–34. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[35] A. Globus, J. Lawton, and T. Wipke, "Automatic molecular design using evolutionary techniques," *Nanotechnology*, vol. 10, pp. 290–299, August 1999.

[36] N. Brown, B. McKay, F. Gilardoni, and J. Gasteiger, "A graph-based genetic algorithm and its application to the multiobjective evolution of median molecules," *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 3, pp. 1079–1087, 2004. PMID: 15154776.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," *arXiv e-prints*, p. arXiv:1603.05027, Mar 2016.

[38] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proceedings of the British Machine Vision Conference (BMVC)* (E. R. H. Richard C. Wilson and W. A. P. Smith, eds.), pp. 87.1–87.12, BMVA Press, September 2016.

[39] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5987–5995, July 2017.

[40] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," *arXiv e-prints*, p. arXiv:1802.01548, Feb 2018.

[41] B. Wang, Y. Sun, B. Xue, and M. Zhang, "A Hybrid GA-PSO Method for Evolving Architecture and Short Connections of Deep Convolutional Neural Networks," *arXiv e-prints*, p. arXiv:1903.03893, Mar 2019.

[42] B. Wang, Y. Sun, B. Xue, and M. Zhang, "Evolving Deep Neural Networks by Multi-objective Particle Swarm Optimization for Image Classification," *arXiv e-prints*, p. arXiv:1904.09035, Mar 2019.

[43] D. Atkins, K. Neshatian, and M. Zhang, "A domain independent genetic programming approach to automatic feature extraction for image classification," in *2011 IEEE Congress of Evolutionary Computation (CEC)*, pp. 238–245, June 2011.

[44] H. Al-Sahaf, A. Song, K. Neshatian, and M. Zhang, "Two-tier genetic programming: towards raw pixel-based image classification," *Expert Systems with Applications*, vol. 39, no. 16, pp. 12291 – 12301, 2012.

[45] B. Evans, H. Al-Sahaf, B. Xue, and M. Zhang, "Evolutionary deep learning: A genetic programming approach to image classification," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–6, July 2018.

[46] A. Lensen, H. Al-Sahaf, M. Zhang, and B. Xue, "Genetic programming for region detection, feature extraction, feature construction and classification in image data," in *Genetic Programming* (M. I. Heywood, J. McDermott, M. Castelli, E. Costa, and K. Sim, eds.), (Cham), pp. 51–67, Springer International Publishing, 2016.

[47] Y. Bi, M. Zhang, and B. Xue, "An automatic region detection and processing approach in genetic programming for binary image classification," in *2017 International Conference on Image and Vision Computing New Zealand (IVCNZ)*, pp. 1–6, Dec 2017.

[48] Y. Bi, M. Zhang, and B. Xue, "Genetic programming for automatic global and local feature extraction to image classification," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, July 2018.

[49] Y. Bi, B. Xue, and M. Zhang, "An automatic feature extraction approach to image classification using genetic programming," in *Applications of Evolutionary Computation* (K. Sim and P. Kaufmann, eds.), (Cham), pp. 421–438, Springer International Publishing, 2018.

[50] H. Al-Sahaf, M. Zhang, and M. Johnston, "Binary image classification: A genetic programming approach to the problem of limited training instances," *Evol. Comput.*, vol. 24, pp. 143–182, Mar. 2016.

[51] H. Al-Sahaf, M. Zhang, A. Al-Sahaf, and M. Johnston, "Keypoints detection and feature extraction: A dynamic genetic programming approach for evolving rotation-invariant texture image descriptors," *IEEE Transactions on Evolutionary Computation*, vol. 21, pp. 825–844, Dec 2017.

[52] H. Al-Sahaf, A. Al-Sahaf, B. Xue, M. Johnston, and M. Zhang, "Automatically evolving rotation-invariant texture image descriptors by genetic programming," *IEEE Transactions on Evolutionary Computation*, vol. 21, pp. 83–101, Feb 2017.

[53] H. Al-Sahaf, B. Xue, and M. Zhang, "A multitree genetic programming representation for automatically evolving texture image descriptors," in *Simulated Evolution and Learning* (Y. Shi, K. C. Tan, M. Zhang, K. Tang, X. Li, Q. Zhang, Y. Tan, M. Middendorf, and Y. Jin, eds.), (Cham), pp. 499–511, Springer International Publishing, 2017.

[54] H. Al-Sahaf, B. Xue, and M. Zhang, "Evolving texture image descriptors using a multitree genetic programming representation," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, (New York, NY, USA), pp. 219–220, ACM, 2017.

[55] H. Al-Sahaf, M. Zhang, M. Johnston, and B. Verma, "Image descriptor: A genetic programming approach to multiclass texture classification," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2460–2467, May 2015.

[56] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, pp. 448–456, 2015.

[57] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 2011.

[58] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of CVPR*, pp. 2818–2826, 2016.

[59] L. Bottou, "Stochastic gradient descent tricks," in *Neural Networks: Tricks of the Trade: Second Edition*, pp. 421–436, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[60] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," *arXiv e-prints*, p. arXiv:1707.07012, Jul 2017.

[61] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proceedings of the 35th International Conference on Machine Learning* (J. Dy and A. Krause, eds.), vol. 80, (Stockholm, Sweden), pp. 4095–4104, PMLR, 10–15 Jul 2018.