Evolving Deep Neural Networks

DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Technology
in
Computer Science

by

Manohar Mukku
[Roll No: CS-1703]

under the guidance of

Dr. Sushmita Mitra
Professor
Machine Intelligence Unit

Indian Statistical Institute
Kolkata-700108, India

July 2019

To my family, friends and my guide

CERTIFICATE

This is to certify that the dissertation entitled **“Evolving Deep Neural Networks”** submitted by **Manohar Mukku** to Indian Statistical Institute, Kolkata, in partial fulfillment for the award of the degree of **Master of Technology in Computer Science** is a bonafide record of work carried out by him under my supervision and guidance. The dissertation has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Prof. Sushmita Mitra
Machine Intelligence Unit,
Indian Statistical Institute,
Kolkata - 700108, India

Acknowledgements

I would like to show my highest gratitude to my advisor, *Prof. Sushmita Mitra*, Machine Intelligence Unit, Indian Statistical Institute, Kolkata, for her guidance and continuous support.

I would also like to convey my deepest thanks to *Subhashis Banerjee*, Senior Research fellow, Indian Statistical Institute, Kolkata, for his valuable suggestions and discussions.

Finally, I am very much thankful to my parents and my brother for their continuous encouragement and everlasting support.

Last but not the least, I would like to thank all of my friends for their help and support. Special thanks to *Love Varshney*, *Sangeet Jaiswal*, and *Sourav Aich* for their valuable suggestions from time-to-time. I thank all those, whom I have missed out from the above list.

Manohar Mukku
Indian Statistical Institute
Kolkata - 700108, India

Abstract

Human nervous system has evolved for over 600 million years and can accomplish a wide variety of tasks effortlessly - telling whether a visual scene contains animals or buildings feels trivial to us, for example. For Artificial Neural Networks (ANNs) to carry out activities like these, requires careful design of networks by experts over years of difficult research, and typically address one specific task, such as to find what's in a photograph, or to help diagnose a disease. Preferably for any given task, one would want an automated technique to generate the right architecture. One approach to generate these architectures is through the use of evolutionary techniques. In this work, we test three methods i.e. CGP technique, ECGP technique, and a new crossover technique of CGP, for generating CNN architectures and report the results. To our knowledge, this is the first attempt on using either ECGP or the new crossover technique of CGP for evolving CNN architectures. This study is still in progress.

Keywords: *Genetic Programming, convolutional neural network, designing neural network architectures, deep learning.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Human nervous system has evolved for over 600 million years and can accomplish a wide variety of tasks effortlessly - telling whether a visual scene contains animals or buildings feels trivial to us, for example. For Artificial Neural Networks (ANNs) to carry out activities like these requires careful design of networks by experts over years of difficult research, and typically address one specific task, such as to find what's in a photograph, or to help diagnose a disease.

Preferably for any given task, one would want an automated technique to generate the right architecture. One approach to generate these architectures is through the use of evolutionary techniques, on which my research primarily focuses on. Many groups are working on this subject, including OpenAI, Uber Labs, Sentient Labs, Google and Deep Mind.

One main issue related to using evolutionary techniques is computational power. The recent advent of Graphics Processing Units (GPUs) and the availability of huge computational power at low cost has helped the Deep Learning community solve many difficult problems. With the ever increasing computational power, the idea of using evolutionary techniques to solve many problems is starting to see a rise in recent years. One such problem is the automatic design of ANNs. Many groups started working on this subject with different evolutionary techniques. Some groups like Google have huge computational power and their proposed techniques have very high accuracies. For example, Google for AutoML [11] used thousands of GPUs for days to generate CNN architecture for a large dataset. We used a single GPU for months to generate comparable architectures for a smaller classification dataset.

In this work, we test three evolutionary algorithms for evolving CNN architectures based on CGP, ECGP evolutionary techniques and a new crossover technique on CGP. To our knowledge, no one has used ECGP or crossover on CGP for generating

Deep Neural Networks and this is the first attempt of trying these for generating Deep Neural Networks.

1.2 Our Contributions

The contributions of this thesis are summarized as follows.

- We have tested a new method for generating CNN architectures based on the standard Embedded Cartesian Genetic Programming evolutionary technique.
- We have tested an updated method for generating CNN architectures based on the standard Cartesian Genetic Programming evolutionary technique.
- We have tested a new method for generating CNN architectures based on the CGP technique with crossover.
- We have also provided the performance analysis of our schemes. We have compared our methods with the results of [12].

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we discuss the necessary background on the concepts of Cartesian Genetic Programming and Embedded Cartesian Genetic Programming used in our work. In Chapter 3, we discuss the related work done in the realm of using evolutionary techniques for deep learning. In Chapter 4, we discuss our first method of generating CNN architectures based on Cartesian Genetic Programming. In Chapter 5, we discuss the second method of generating CNN architectures based on Embedded Cartesian Programming. In Chapter 6, we discuss the third method of generating CNN architectures based on a new CGP technique which includes crossover. In Chapter 6, we summarize the work done and discuss the future directions of our work.

Chapter 2

Background

In this chapter, we discuss the concepts of Evolutionary Algorithms, Cartesian Genetic Programming (CGP), and Embedded CGP (ECGP) which are used in our work.

2.1 Evolutionary Algorithms

Evolutionary Algorithms (EAs) refer to the class of algorithms which use the concept of Darwinian natural selection for evolution. It broadly consists of four major categories of algorithms. They are:

1. Genetic Algorithms (GA)
2. Genetic Programming (GP)
3. Evolutionary Strategies (ES)
4. Evolutionary Programming (EP)

In natural selection, the organisms that are *well-adopted* to the environment reproduce more often than others. Similarly, in EAs, the individuals which produce better results for the problem statement (calculated using a fitness measure) reproduce more often than others.

Initially, an EA starts with a randomly generated set of solutions. Each solution is encoded as a string. Then operations inspired by biological evolution, such as selection, reproduction, crossover, and mutation are performed on these strings to generate new set of strings. This process continues until the desired solution is obtained.

This is the basic structure of any EA. Major differences between GA, GP, ES, or EP come from the operators they use, and the way they implement these operators. The basic algorithm of any EA is as follows:

1. Generate an initial population of solutions randomly
2. Evaluate the population of solutions for fitness

3. Perform *Selection* operation on the population based on fitness values

4. Apply reproduction operators like *Crossover* and *Mutation* on the selected population

5. Replace the original population with the new population

6. Repeat steps 2 through 5 until a termination criterion is satisfied

7. Return the best solution

In our work, we have used the concepts of Cartesian Genetic Programming and Embedded Cartesian Genetic Programming. Here is a brief background on these concepts.

2.2 Cartesian Genetic Programming

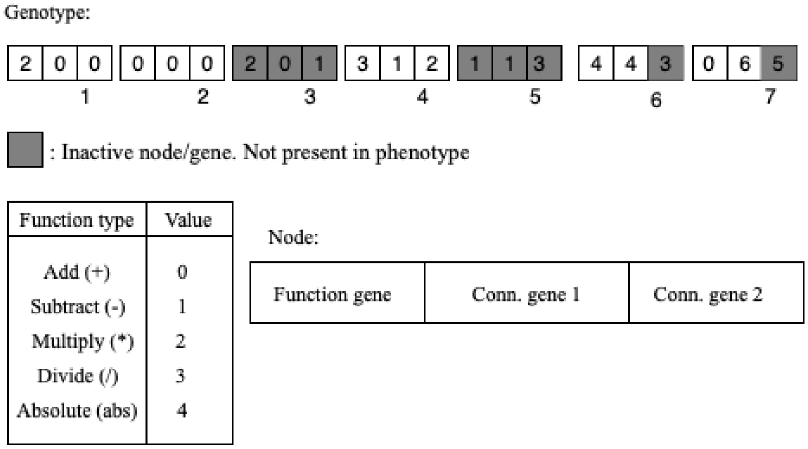
CGP is a type of Genetic Programming (GP) invented by Miller and Thomson [8], where computer programs are encoded as directed acyclic graphs. Any GP model consists of a Genotype and a Phenotype where a Genotype is a collection of nodes represented as a graph, and the Phenotype is the decoding of the Genotype as a computer program. The genetic operators used in CGP is only mutation. Crossover is not used in standard CGP.

2.2.1 Genotype - Encoding scheme

A Genotype consists of a two-dimensional grid of nodes with some user-defined number of rows and columns. Each node of the Genotype contains a function gene and one or more connection genes. The function gene represents the function/operation performed by the node, whereas the connection genes represent the node numbers from which it is getting its inputs from. All the function and connection genes are represented using integers.

For example, consider the genotype and function types in figure 2.1. It contains 6 intermediate nodes of 2 rows and 3 columns, and one input and one output nodes. Node number 6 has the function gene value of 4, which means it performs the absolute operation. The number of connection genes actually involved in the operation (called *active connection genes*) may vary depending on the type of the function gene. Even though, node number 6 has two connection genes, only one is active because, absolute is a unary operation. A node in column number c can get its inputs only from the nodes in column numbers $(c-1)$ to $(c-l)$. l is called the *levels-back parameter.*

Not all of the nodes in the genotype, participate in the formation of the phenotype. Only the nodes having a path to the output nodes are present in the phenotype.



Function type	Value
Add (+)	0
Subtract (-)	1
Multiply (*)	2
Divide (/)	3
Absolute (abs)	4

Figure 2.1: Example of a genotype in CGP. The first value in each node represents the operation of the node, and the remaining values represent the connections to the node

These nodes are called *active nodes*. Rest of the nodes are called *inactive nodes*. For example, node numbers 3 and 5 are inactive nodes, whereas node numbers 1, 2, 4 and 6 are active nodes because they have a path to the output node 7.

2.2.2 Phenotype - Decoding scheme

The Phenotype is constructed from the genotype of figure 2.1 by backtracking the connections from the output nodes till the input nodes. Figure 2.2 represents the phenotype constructed from the genotype in figure 2.1. As you can see, all the nodes in the genotype are not in the phenotype. Thus, even though the genotype is of fixed-length, the length and structure of the phenotype varies depending on the connections.

2.2.3 Mutation operation

The mutation operation in CGP is called *point mutation*. It randomly mutates the function and connection genes to new values according to some mutation probability. The mutation can change both the active and inactive nodes/genes. If only the inactive nodes/genes are changed in the Genotype after mutation, then the Phenotype is unchanged as there is no change in active nodes/genes. The offsprings are generated by applying the point mutation operation on the parent.

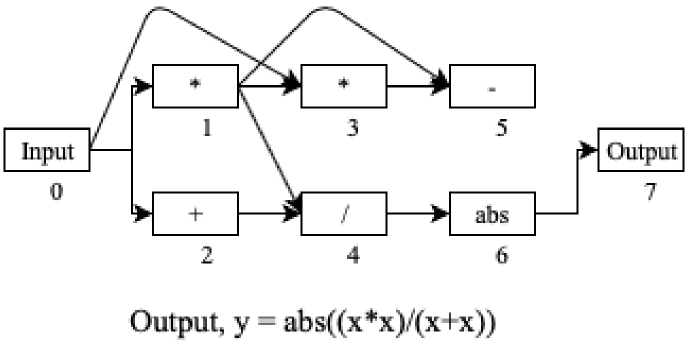


Figure 2.2: Example of a phenotype in CGP. Nodes 3 & 5 are inactive nodes, while the rest are active

2.3 Embedded Cartesian Genetic Programming

ECGP is invented by Walker and Miller [13] as an extension of CGP incorporating ideas from a technique known as Module Acquisition [1]. It is similar to CGP, except that in ECGP along with primitive function types, there will be modules. Modules are nothing but a group of successive primitive function nodes constructed from the ECGP genotype. The genetic operators used in ECGP are *compress*, *expand* and *point mutation*. The workings of the genotype and the phenotype in ECGP is similar to that as CGP with the extra addition of the presence of modules. The reader may refer to [14] for an in-depth discussion of ECGP.

2.3.1 Modules

A module node is a list of primitive function type nodes of the genotype combined to be a single node. A module node contains a module header, the list of primitive nodes and, the list of its outputs. The module header contains the module id, number of module inputs, number of module nodes, and the number of module outputs. The compress operator selects a list of successive primitive function nodes randomly and combines them to be a single module, and adds it to a list of module nodes. Module node obtained by the compress operator is called a type-I node. During point mutation, the function gene of a node can change to any module node randomly selected from the list of module nodes. The module node obtained in this way is called a type-II node. The nodes after the module node are updated accordingly. Figure 2.3 explains the formation of a module node by the compress operator.

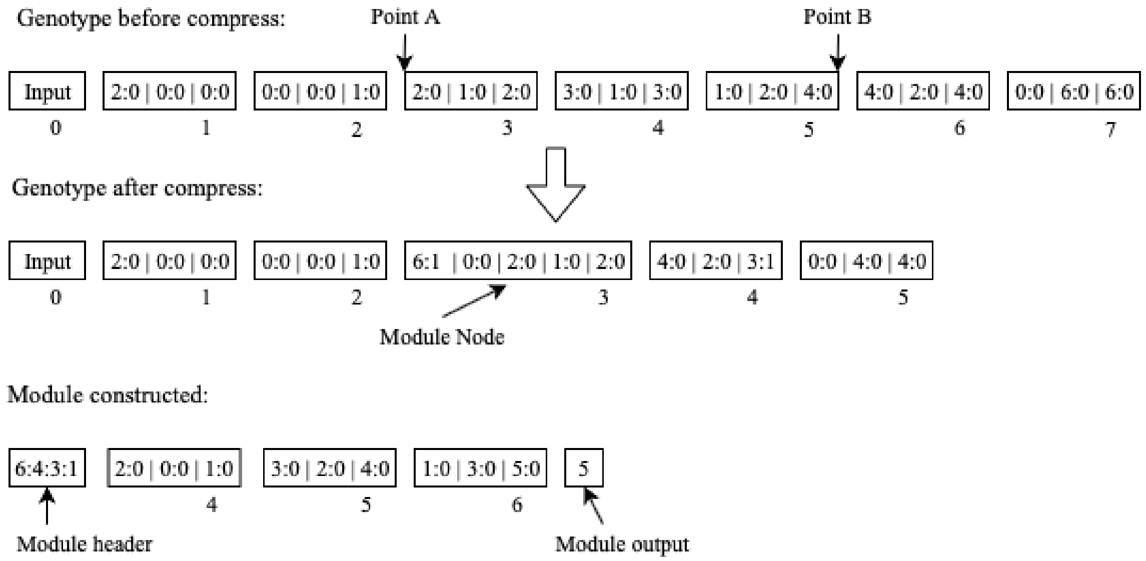


Figure 2.3: Example of a genotype in ECGP, and compress operation. The nodes between points A & B are compressed to a single module node. The genes within and after the module node are updated accordingly

2.3.2 Genotype - Encoding scheme

The Genotype encoding scheme for ECGP is similar to that as of CGP. It is a list of nodes which when decoded represents a directed acyclic graph. Similar to that of CGP, each node consists of a function gene and one or more connection genes. Different from CGP, an ECGP function gene is represented by two integers, where the first integer represents either the module number or the primitive operation performed by the node, and the second integer represents the type of the node. Table 2.1 explains the different values taken by the second integer.

Type of node	Value	Explanation
Primitive function type	0	Non-module node
Type-I module	1	Obtained by compress
Type-II module	2	Obtained by mutation

Table 2.1: Function node types

Also different from CGP, a connection gene in ECGP is represented by two integers. The first integer represents the node number from which it is getting its input. The second integer represents that, if it is getting its input from a module node, then from which node within the module it is getting its input, or else the value is 0, if it is getting its input from a primitive node. For example in figure 2.3, the second connection gene of node number 4 in genotype after compress, is getting its input from module's (node 3's) second node (represented with 1 as indexing starts from 0). Thus it is represented as 3:1. The module node in the genotype may have more than one connection genes. All the connection genes of a module represent the inputs coming to the module and are written inside the module node of the genotype, after the function gene. For example, in figure 2.3, node number 3 which is a module node has 4 external inputs. All the inputs of the module node are represented within the node after the function gene 6:1.

2.3.3 Compress operation

The compress operation randomly selects two points of the genotype (points A & B in figure 2.3), according to some compress probability, such that none of the nodes between the two points is a module node. It then replaces the selected nodes with a single module node. The module constructed is added to the global list of modules and is marked as a type-I node. The nodes within and after the module node are updated accordingly. Figure 2.3 illustrates the compress operation.

2.3.4 Expand operation

The expand operation randomly selects a type-I module node in the genotype, according to some expand probability, and replaces it with the primitive nodes of the module. The nodes after the module node are updated accordingly. The expand operation is the reverse of the process in figure 2.3.

2.3.5 Mutation operation

The point mutation operation in ECGP randomly mutates the function and connection genes of the nodes to new values, according to some mutation probability. The function gene can be mutated to any of the primitive or module nodes. If the function gene is mutated to a module node, it is labelled as a type-II node.

2.3.6 Phenotype - Decoding scheme

The Phenotype construction process is similar to that as CGP. The only difference is the presence of the module nodes. The construction of a phenotype proceeds as follows:

1. First, a duplicate of the genotype is created and all the module nodes in the duplicate are expanded.

2. Then the phenotype is formed from the expanded duplicate genotype.
3. After the phenotype construction, the duplicate genotype is destroyed.

Figure 2.4 represents the corresponding phenotype for the genotype in figure 2.3. Node numbers 1, 2 & 6 are active nodes, whereas node numbers 3, 4, & 5 are inactive nodes as they do not have a path to the output node 7.

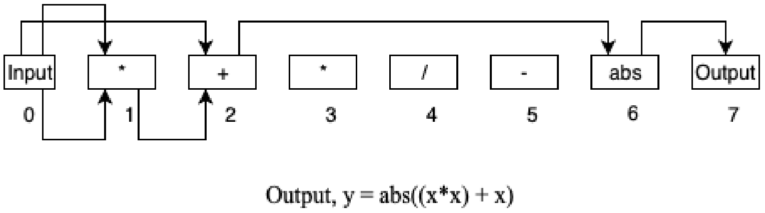


Figure 2.4: Example of a phenotype in ECGP. Nodes 3, 4, & 5 are inactive nodes, while the rest are active

Chapter 3

Related Work

Much of the work on evolving deep neural networks has started only recently. We will discuss some of the works done in this realm.

[11] evolved a population of trained models. A computer chose two individuals at random from the population. It then compared their fitnesses and the worst of the two is killed and removed from the population. A copy of the best of the two is created. The copy is mutated, evaluated, and is added back to the population. This comprises a single evolutionary step. During each mutation operation, the computer chooses a mutation at random from a set of predetermined mutations, like altering learning rate, altering stride, adding skip connections, altering number of channels, adding convolutions etc. For children, weights of the parent are inherited whenever possible. To save time, many parallel evolutionary steps are performed. The whole process is automatic and no human intervention is ever needed at any step of the process. They achieved a test accuracy of $94.1\,\% \pm 0.4\,\%$ for the best model based on validation accuracy, evolved on CIFAR10 dataset.

[19] designed a new search space called *"NASNet Search Space"* which allows searching for an architectural building block (or cell) on a small dataset like CIFAR10 and then transferring it to a larger dataset like ImageNet by stacking copies of this cell each with different parameters. They call it *"NASNet architecture"*. The basic building blocks of the *"NASNet architecture"* are two types of cells called Normal and Reduction cells. The Normal cell is a convolutional cell that retains the feature maps' dimensions, whereas the Reduction cell reduces the height and width of the feature map by a factor of two. The algorithm proceeds as follows. Initially, a fixed architecture consisting of Normal and Reduction cells is manually pre-built for a smaller dataset. Then the cells are generated by a controller RNN using Neural Architecture Search (NAS) framework proposed by [17]. The generated cells are stacked following the pre-built architecture. This architecture is trained and evaluated, and the controller RNN is updated using the evaluation accuracy. The updated controller RNN becomes better at generating better cells for the next iteration. A cell generated in this way, on

a small dataset like CIFAR10 is then used for a larger dataset like ImageNet with its own manually pre-built architecture. They achieved state-of-the-art accuracy of 82.7 % top-1 and 96.2 % top-5 on ImageNet, using the cells generated on CIFAR10 dataset.

[10] evolved an image classifier (*AmoebaNet-A*) for ImageNet dataset, that for the first time surpassed human hand-designed architectures. They used a modified tournament selection evolutionary-algorithm called aging evolution which introduced age property to favour younger genotypes and the search process followed *"NASNet Search Space"* proposed by [19]. They achieved a new state-of-the-art accuracy of 83.9 % top-1 and 96.6 % top-5 on ImageNet dataset.

[12] who used Cartesian Genetic Programming approach for designing Convolutional Neural Network architectures. They started with a single parent in the population which is randomly generated. Then two copies of the parent are created. Each copy is then mutated and evaluated for fitness. The architecture having the maximum fitness among the three is progressed to the next generation whereas the remaining two are killed and removed from the population. This process is repeated for a certain number of generations until an architecture with better validation accuracy is evolved. They achieved an accuracy of 93.25 % after 500 generations evolved on CIFAR10 dataset.

[18] have used recurrent networks to generate descriptions of models of neural networks. This recurrent network along with reinforcement learning is trained to maximize the validation accuracy of the generated architectures. They used the CIFAR10 dataset for generating the network architectures. Their methods rivals the state-of-the-art human designed architectures with accuracies of 96.35 % on CIFAR10. On Penn TreeBank dataset, their method beats the widely-used LSTM cell, and other state-of-the-art methods.

[2] introduced a new crossover technique to perform in the standard CGP. They changed the standard CGP genotype representation from integer-based to floating-point values. The crossover technique works on these floating-point values. They have tested their method for generating architectures for simple regression problems and found that incorporating crossover to CGP greatly speeds-up the convergence process. They have experimented with various crossover probability values and found that using variable crossover probabilities with respect to generations by decreasing linearly, achieved better convergence.

Chapter 4

CGP for evolving CNNs

4.1 Introduction

CGP is a type of GP which represents nodes as a directed acyclic graph. Refer section 2.2 for a background of CGP. Initially, CGP has been used for generating electric circuits. Presently, CGP is being used in many diverse applications. In this chapter, we propose our method for generating CNN architectures using the standard CGP evolutionary technique,.

4.2 Method

Figure 4.1 illustrates a single generation of the method. Initially we start with a randomly initialized genotype as parent. Then, a user-defined number of offsprings are generated using *point mutation* of CGP. Phenotypes (CNNs) for both the parent and offsprings are then constructed from their respective genotypes. Each of them are then trained on a training dataset for a fixed number of epochs. The fitnesses of each of the trained CNNs are then evaluated on a test dataset. The elite individual among the parent and offsprings according to their fitness values is then chosen as parent for the next generation. This process is repeated for some generations until a stopping criterion is reached. The architecture of the parent in the last generation is the final required architecture.

To evaluate fitnesses, we used Matthews Correlation Coefficient (MCC) metric [7] because it considers both the true and false positives and negatives and provides a balanced measure even if the classes are imbalanced. MCC metric returns a value between -1 and $+1$. A $+1$ indicates a perfect prediction, 0 indicates no better than random prediction, and -1 indicates a complete disagreement between observation and prediction. MCC is calculated using the formula in equation (4.1), where *TP* indicates *true positives*, *TN* indicates *true negatives*, *FP* indicates *false positives*, and

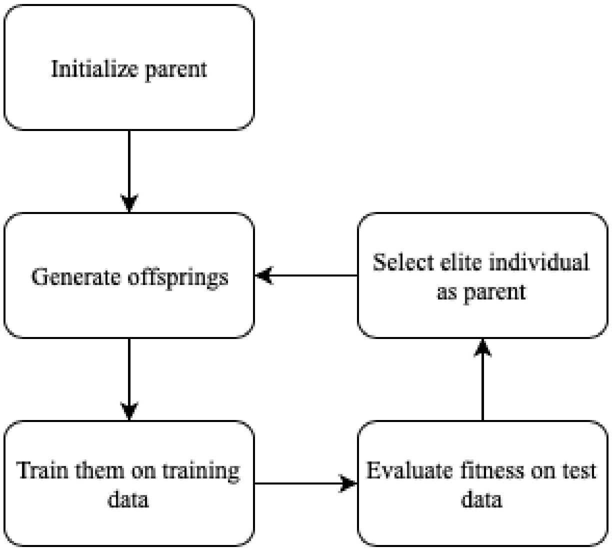

Initialize parent
Generate offsprings
Select elite individual as parent
Train them on training data
Evaluate fitness on test data

Figure 4.1: Overview of our method for generating CNN architectures using standard CGP

FN indicates *false negatives.*

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.1}$$

In this section, we describe in detail the representation of the network and the evolutionary algorithm used in our proposed method.

4.2.1 Representation of the Network

To represent the CNN architecture, we use the CGP encoding scheme (refer section 2.2), which represents the program as a directed acyclic graph of computational nodes. In general, the directed acyclic graph of the CGP is represented as a two-dimensional grid having some user-defined N_r rows and N_c columns. However, various experiments and analysis later done by [16] on CGP concluded that taking the number of rows equal to one found to be more effective in producing better results for CGP in many tasks. Thus in our experiments, we have set N_r to be equal to one. Our CGP representation is a one-dimensional grid having one row and N_c columns with a total of N_c intermediate nodes. The number of input nodes, N_i and the number of output nodes, N_o depend on the task.

The genotype of the CGP has $N_c + N_o$ nodes where each node consists of integers of fixed length representing the function and connection genes. The integer corresponding to the function gene of the node represents the type of function performed by the node, whereas the integer corresponding to the connection gene of the node specifies the node number from which it is getting its input. A node in c-th column should get its inputs only from within $(c-l)$ to $(c-1)$ nodes, where l is the levels-back parameter. Figure 4.2 gives an example of a genotype with four columns and one output node, and its corresponding network, the CNN architecture called the phenotype. Whereas the number of nodes in the genotype are fixed, the number of nodes in the phenotype varies because not all nodes have a path to the output nodes. The nodes which do not participate in the phenotype are called inactive nodes. Node numbers 2 & 4 are inactive in the genotype in the left of figure 4.2 because they do not have a path to the output node 5.

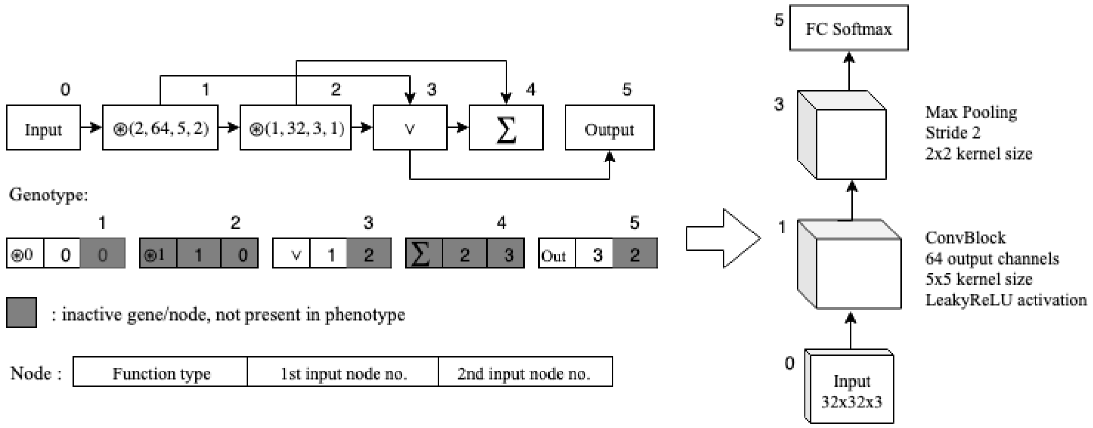


Figure 4.2: Example of a genotype(left) and a phenotype(right) in CGP-CNN

The function genes for nodes of CGP are chosen to be highly functional modules of CNN. In general, CNNs mostly use convolution and pooling followed by batch normalization and nonlinear activations. Rather than using each of them separately as a function gene for a node as done in [11], we combined them to construct highly functional modules. We prepared five types of functions called ConvBlock, average pooling, max pooling, summation, and concatenation. These operate on the three-dimensional tensors called feature maps defined by the dimensions of rows, columns and channels.

The ConvBlock consists of dilated convolution [15] with different dilation factors and stride of 1, followed by batch normalization [4] and either leaky rectified linear units (LeakyReLU) [6] or rectified linear units (ReLU) [9] activation. Several ConvBlocks are prepared with different values for dilation factor $\in \{1, 2, 4\}$, different number of output channels $\in \{32, 64, 128\}$, different kernel sizes $\in \{3 \times 3, 5 \times 5\}$, and different activations $\in \{ReLU, LeakyReLU\}$ and are added to the function set of CGP. In ConvBlock, before the convolution operation, we pad the input feature maps with zeroes to preserve the row and column sizes of the output. Thus, the $P \times Q \times C$ input feature maps are converted into $P \times Q \times C'$ output feature maps where P, Q,

C, and C' are respectively the number of rows, columns, input channels, and output channels.

The average and max pooling respectively do an average and max operation on the feature maps. A kernel size of 2 × 2 and a stride of 2 are used for pooling. After pooling, the $P \times Q \times C$ input feature maps are converted into $P' \times Q' \times C$ output feature maps, where $P' = \left\lfloor \frac{P}{2} \right\rfloor$ and $Q' = \left\lfloor \frac{Q}{2} \right\rfloor$. We have added a constraint that any pooling node should come only after a ConvBlock node.

The summation function does channel by channel element-wise addition of two feature maps. If the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. If the input feature maps have different number of channels, the smaller one is padded with zeroes in the channels dimension to make them have the same number of channels. Thus after summation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions $\min(P_1, P_2) \times \min(Q_1, Q_2) \times \max(C_1, C_2)$.

The concatenation function does concatenation of two feature maps in the channel dimension. Similar to summation, if the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. Thus after concatenation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions $\min(P_1, P_2) \times \min(Q_1, Q_2) \times (C_1 + C_2)$.

The output node is a fully connected softmax layer of the specified number of classes. All the nodes of the previous layer are fully connected to the outputs. Table 4.1 specifies all the node functions used.

Node function	Representation	Values
ConvBlock	$\circledast(d, c, k, a)$	dilation factor, $d \in \{1, 2, 4\}$ # of channels, $c \in \{32, 64, 128\}$ kernel size, $k \in \{3 \times 3, 5 \times 5\}$ activation, $a \in \{ReLU, LeakyReLU\}$
Max pooling	$\vee$	-
Average pooling	μ	-
Summation	$\sum$	-
Concatenation	$\|\|$	-

Table 4.1: Node functions used for the CGP method of generating CNNs

4.2.2 Evolutionary Algorithm

Algorithm 1 summarizes the evolutionary algorithm used for the method of generating CNN architectures using the standard CGP technique. The genetic operator used in CGP is point mutation. We perform point mutation on the genes of the genotype according to some mutation probability, with a constraint that the point mutation mutates at least one active node. We call this *forced mutation*. Forced mutation is used for generating offsprings different from their parent. Also, we perform *neutral mutation*, where in only the inactive nodes/genes change. We have used the $(1 + \lambda)$ evolutionary strategy with λ equal to 2 in our experiments.

Algorithm 1 Evolutionary Algorithm for generating CNNs based on CGP

```
Randomly initialize parent genotype, $p_0$
generation ← 1
while NOT termination criteria do
    Generate $\lambda$ offsprings, $p_1, p_2, ..., p_\lambda$ by applying forced mutation on parent, $p_0$
    Convert genotypes, $p_0, p_1, ..., p_\lambda$ to phenotypes(CNNs), $c_0, c_1, ..., c_\lambda$
    Train CNNs, $c_0, c_1, ..., c_\lambda$ on training data
    Calculate fitnesses (MCC), $f_0, f_1, ..., f_\lambda$ of trained CNNs, $c_0, c_1, ..., c_\lambda$, on validation data
    Perform neutral mutation on parent, $p_0$
    m ← $argmax(f_0, f_1, ..., f_\lambda)$
    Set $p_0 \leftarrow p_m$ as parent for next generation
    generation ← generation + 1
end while
Return $p_0$ as the best architecture
```

4.3 Experiments and Results

4.3.1 Dataset

We test our method on an image classification dataset called CIFAR10. This dataset was chosen because, to reach high accuracies it requires large networks, which present a computational challenge. CIFAR10 dataset is being used as the benchmark dataset for many similar problems. For example, [11], [18], [19], [12], [10] and many more have used CIFAR10 to evolve their architectures.

The CIFAR10 dataset has 50,000 train images and 10,000 test images. The size of each image is 32 × 32, the number of input channels is 3, and the number of classes is 10. The train images is split into train and validation datasets of ratio 9:1 i.e., 45,000 train images and 5,000 validation images.

4.3.2 Experimental Setting

We used stochastic gradient descent (SGD) to train the CNN architectures on the training dataset with a mini-batch size of 128. Weights are initialized randomly. Adam optimizer [5] is used with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rate is kept constant at 0.01. Each CNN is trained for 50 epochs on the training dataset. After the training is complete, the MCC metric evaluated on the validation dataset is used as the fitness of the individual.

The whole train images are normalized per-pixel along all the 3 channels with mean and standard deviation values of the dataset. Data augmentation method mentioned in [3] is used by padding 4 pixels on all the sides of the image followed by taking a 32 × 32 random crop, and then by taking a random horizontal flip on the cropped image.

Parameter	Value
Mutation probability	0.05
No. of rows (N_r)	1
No. of columns (N_c)	30
Levels-back (l)	10

Table 4.2: Parameters used for CGP method of generating CNNs

The parameters used for CGP method are displayed in table 4.2. To generate deep architectures we used the number of columns to be 30. A total of 40 node functions are used which are obtained with all the combinations of functions in table 4.1.

We have trained our method for 50 generations and recorded its performance. The best CNN architecture obtained after the whole process is re-trained on the whole training dataset of 50,000 images. The trained model is used to classify the 10,000 test images of CIFAR10 and the test accuracy is calculated.

We have implemented our method using Pytorch (version 1.1) and Python3 (version 3.6.5) and the experiments were run on a machine with Tesla P6 GPU having 16GB GPU RAM.

4.3.3 Results

Figure 4.3 displays the test accuracies for each generation compared with the method implemented by [12]. We will call their method as Suganuma-CNN from now onwards. The blue line represents the accuracies for Suganuma-CNN method, whereas the orange line represents the accuracies for our method. As we can see, Suganuma-CNN method outperformed our method. It was converging faster.

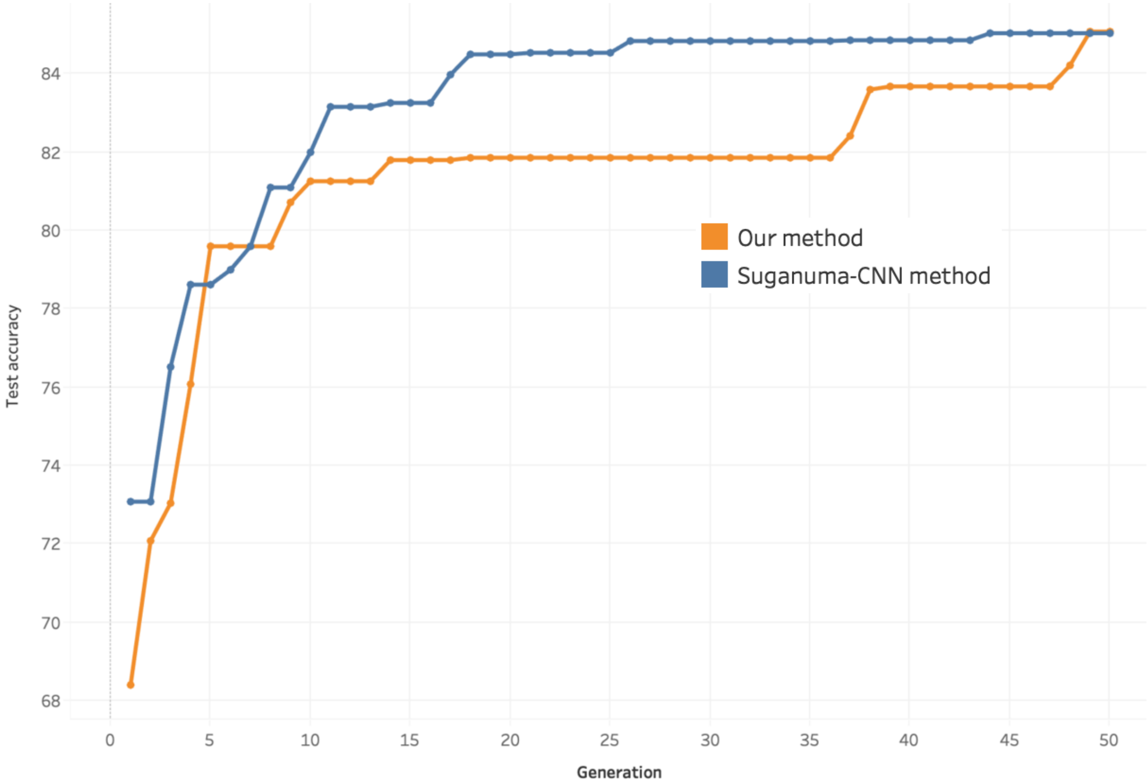


Figure 4.3: Test accuracies for 50 generations for standard CGP technique

Table 5.4 displays the test accuracies for both the methods after 50 generations averaged over 3 runs.

Method	Test Accuracy
Suganuma-CNN	86.08 % ± 0.43 %
StandardCGP-CNN	85.32 % ± 0.51 %

Table 4.3: Classification performance of CGP-CNN

Figure 4.4 displays the initial and final architectures generated by our method. In figure 4.4, CB_c_f_d_a stands for ConvBlock with c channels, f × f filter size, dilation factor d and activation a. Also MP_f_s stands for max pooling with f × f filter size and stride s. AP_f_s stands for average pooling with f × f filter size and stride s. Full stands for fully connected Softmax layer.

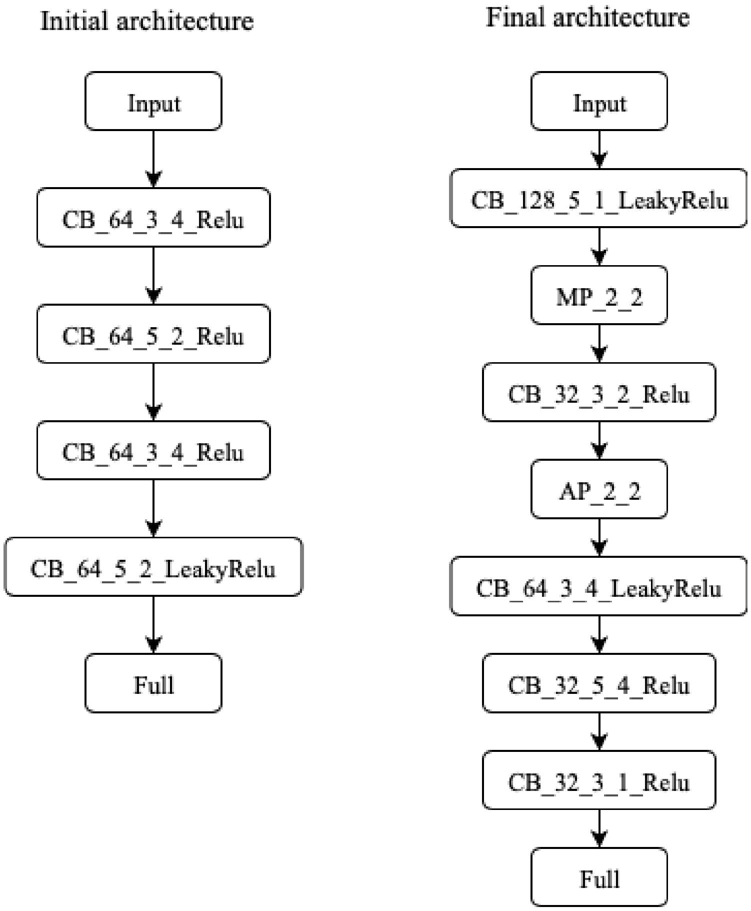


Figure 4.4: Initial and final architectures generated for standard CGP technique

4.4 Conclusion

We have modified the Suganuma's CNN method by having the number of rows to be 1, constraint that max pooling to come only after a ConvBlock, dilated convolution operation, and adding LeakyReLU activation. But the accuracies obtained were not better than Suganuma-CNN method. Also in our experiments, we have noticed that having dilated convolution slowed down the running of the program. Also we have noticed that because of the presence of a large number of ConvBlock functions in the function set, the remaining pooling, sum, concat functions were not getting selected much during mutation operation.

The code can be downloaded from https://github.com/manoharmukku/cnn-cgp-pytorch.

Chapter 5

ECGP for evolving CNNs

5.1 Introduction

ECGP is an extended version of CGP with the additional presence of modules, which are a reusable list of nodes. Modules help in saving the best substructures of the genotype protecting them from modifications by mutation. Thus the best substructures obtained until a particular generation are saved as modules. Refer section 2.3 for a background on ECGP. To our knowledge, no one has used ECGP for evolving Deep Neural Networks. This is the first attempt of trying to use ECGP for evolving Convolutional Neural Networks for classification.

5.2 Method

Figure 5.1 illustrates a single generation of the method. Initially we start with a randomly initialized genotype as parent. Then, a user-defined number of offsprings are generated by using the genetic operators of ECGP i.e. by using *compress*, *expand* and *point mutation* operators. Phenotypes (CNNs) for both the parent and offsprings are then constructed from their respective genotypes. Each of them are then trained on a training dataset for a fixed number of epochs. The fitnesses of each of the trained CNNs are then evaluated on a test dataset. The elite individual among the parent and offsprings according to their fitness values is then chosen as parent for the next generation. This process is repeated for some generations until a stopping criterion is reached. The architecture of the parent in the last generation is the final required architecture.

To evaluate fitnesses, we used Matthews Correlation Coefficient (MCC) metric [7] because it considers both the true and false positives and negatives and provides a balanced measure even if the classes are imbalanced. MCC measure returns a value between -1 and $+1$. A $+1$ indicates a perfect prediction, 0 indicates no better than

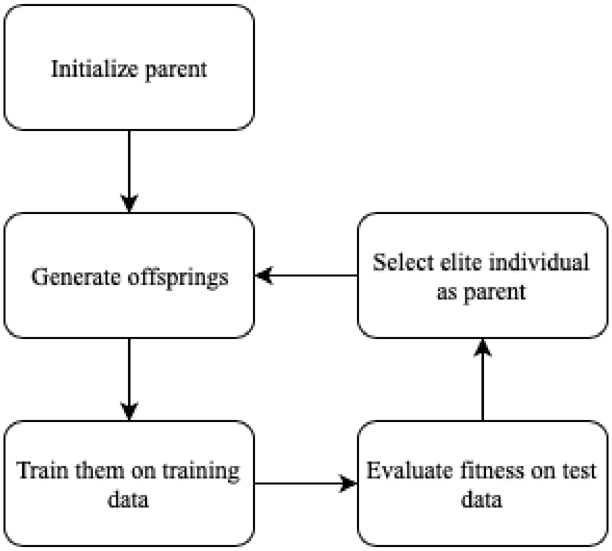

Initialize parent
Generate offsprings
Select elite individual as parent
Train them on training data
Evaluate fitness on test data

Figure 5.1: Overview of our method for ECGP

random prediction, and −1 indicates a complete disagreement between observation and prediction. MCC is calculated using the formula in equation (6.6), where *TP* indicates *true positives*, *TN* indicates *true negatives*, *FP* indicates *false positives*, and *FN* indicates *false negatives.*

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.1}$$

In this section, we describe in detail the representation of the network and the evolutionary algorithm used in our proposed method.

5.2.1 Representation of the Network

To represent the CNN architecture, we use the ECGP encoding scheme (refer section 2.3), which represents the program as a directed acyclic graph of computational nodes. In general, the directed acyclic graph of the ECGP is represented as a two-dimensional grid having some user-defined N_r rows and N_c columns. However, various experiments and analysis later done by [14] on ECGP concluded that taking the number of rows equal to one found to be more effective in producing better results for ECGP in many tasks. Thus in our experiments, we have set N_r to be equal to one. Our ECGP representation is a one-dimensional grid having one row and N_c columns with a total of N_c intermediate nodes. The number of input nodes, N_i and the number of output nodes, N_o depend on the task.

The genotype of the ECGP has $N_c + N_o$ nodes where each node consists of integers of fixed length representing the function and connection genes where each gene is represented by a pair of integers. For function gene, the first integer in the pair represents either the primitive function type or the module id of some module. The second integer represents the type of the function gene. Table 5.1 summarizes the various values taken by the second integer of the function gene.

Table 5.1: Function node types of ECGP

Type of node	Value	Explanation
Primitive function type	0	Non-module node
Type-I module	1	Obtained by compress
Type-II module	2	Obtained by mutation

The connection gene is also represented by a pair of integers. The first integer in the pair represents the node number from which it is getting its input. The second integer represents that, if it is getting its input from a module node, then from which node within the module it is getting its input, or else value is 0, if it is getting its input from a primitive node.

A node in c-th column should get its inputs from $(c-l)$ to $(c-1)$ nodes, where l is the levels-back parameter. Figure 5.2 gives an example of a genotype with four columns and one output node, and its corresponding network, the CNN architecture called the phenotype. Whereas the number of nodes in the genotype are fixed, the number of nodes in the phenotype varies because not all nodes have a path to the output nodes. The nodes which do not participate in the phenotype are called inactive nodes. Node number 3 is an inactive node in the genotype of figure 5.2 because it does not have a path to the output node.

The function genes for nodes of CGP are chosen to be highly functional modules of CNN. In general, CNNs mostly use convolution and pooling followed by batch normalization and nonlinear activations. Rather than using each of them separately as a function gene for a node as done in [11], we combined them to construct highly functional modules. We prepared five types of functions called ConvBlock, average pooling, max pooling, summation, and concatenation. These operate on the three-dimensional tensors called feature maps defined by the dimensions of rows, columns and channels.

The ConvBlock consists of a convolution operation with a stride of 1, followed by batch normalization [4] and either leaky rectified linear units (LeakyReLU) [6] or rectified linear units (ReLU) [9] activation. We have not used dilated convolution, as the run-time of the program drastically increased with dilation. Several ConvBlocks are prepared with different number of output channels $\in \{32, 64, 128\}$, different kernel sizes $\in \{3 \times 3, 5 \times 5\}$, and different activations $\in \{ReLU, LeakyReLU\}$ and are added

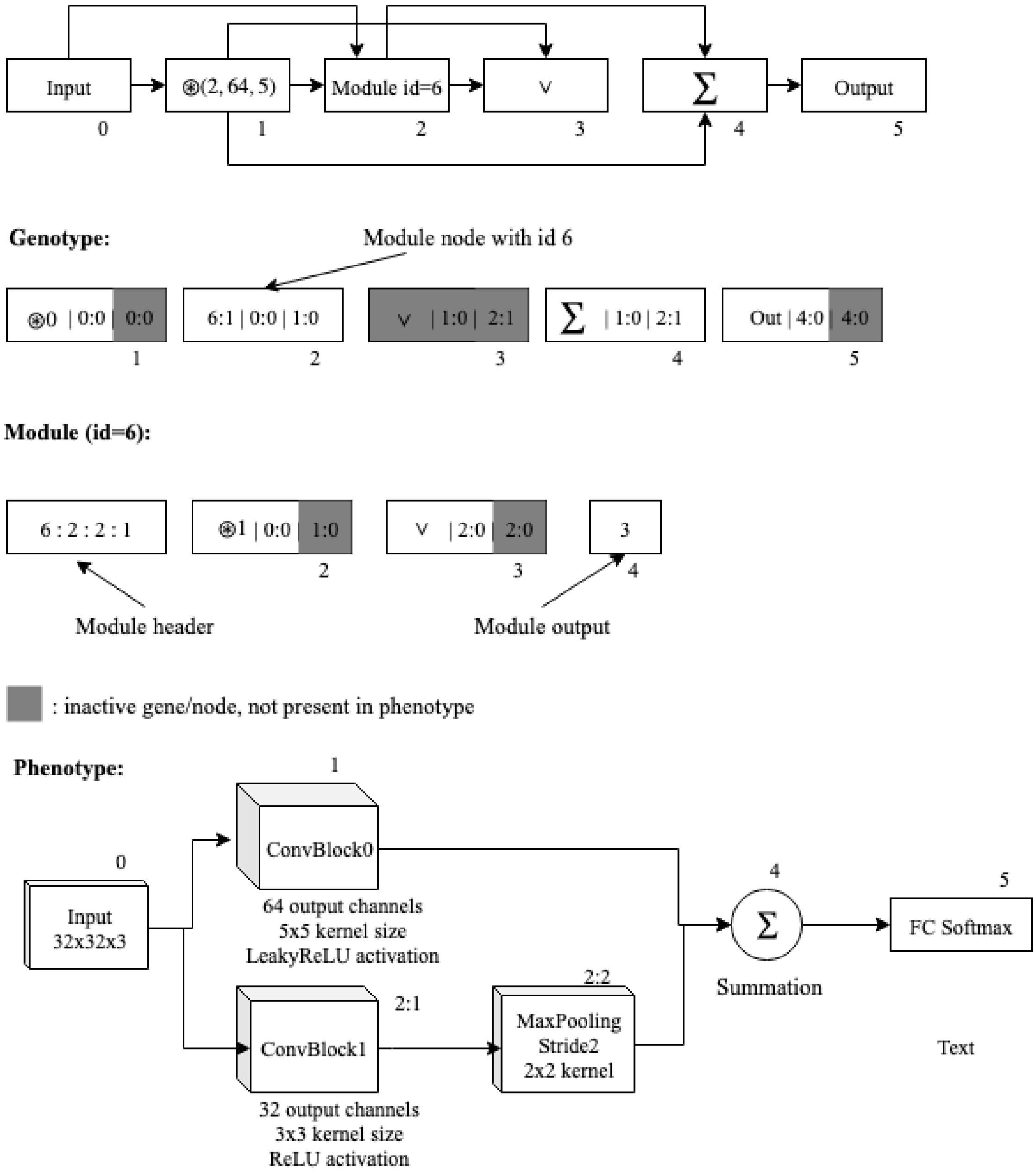


Figure 5.2: Example of a genotype(top) and phenotype(bottom) in ECGP-CNN

to the function set of CGP. In ConvBlock, before the convolution operation, we pad the input feature maps with zeroes to preserve the row and column sizes of the output. Thus, the $P \times Q \times C$ input feature maps are converted into $P \times Q \times C'$ output feature maps where P, Q, C, and C' are respectively the number of rows, columns, input channels, and output channels.

The average and max pooling respectively do an average and max operation on the feature maps. A kernel size of 2 × 2 and a stride of 2 are used for pooling. After pooling, the $P \times Q \times C$ input feature maps are converted into $P' \times Q' \times C$ output feature maps, where $P' = \left\lfloor \frac{P}{2} \right\rfloor$ and $Q' = \left\lfloor \frac{Q}{2} \right\rfloor$. We have added a constraint that any pooling node should come only after a ConvBlock node.

The summation function does channel by channel element-wise addition of two feature maps. If the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. If the input feature maps have different number of channels, the smaller one is padded with zeroes in the channels dimension to make them have the same number of channels. Thus after summation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions $\min(P_1, P_2) \times \min(Q_1, Q_2) \times \max(C_1, C_2)$.

The concatenation function does concatenation of two feature maps in the channel dimension. Similar to summation, if the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. Thus after concatenation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions $\min(P_1, P_2) \times \min(Q_1, Q_2) \times (C_1 + C_2)$.

The output node is a fully connected softmax layer of the specified number of classes. All the nodes of the previous layer are fully connected to the outputs. Table 5.2 specifies all the node functions used.

Node function	Representation	Values
ConvBlock	$\circledast(c, k, a)$	# of channels, $c \in \{32, 64, 128\}$ kernel size, $k \in \{3 * 3, 5 * 5\}$ activation, $a \in \{ReLU, LeakyReLU\}$
Max pooling	$\vee$	-
Average pooling	μ	-
Summation	$\sum$	-
Concatenation	$\|\|$	-

Table 5.2: Node functions used for the ECGP method of generating CNNs

5.2.2 Evolutionary Algorithm

Algorithm 2 summarizes the evolutionary algorithm used for generating CNN architectures based on the ECGP technique. The genetic operators used in ECGP are *compress*, *expand* and *point mutation* according to some probabilities. We perform point mutation on the genes of the genotype according to some mutation probability, with a constraint that the point mutation mutates at least one active node. We call this *forced mutation*. Forced mutation is used for generating offsprings different from their parent. Also, we perform *neutral mutation*, where mutation is done only on the inactive nodes/genes. Similar to the case of CGP, forced mutation is used in ECGP too. We have used the $(1 + \lambda)$ evolutionary strategy with λ equal to 2 in our experiments.

Algorithm 2 Evolutionary Algorithm for CNN generation based on ECGP

```
Randomly initialize parent genotype, $p_0$
generation ← 1
while NOT termination criteria do
    Generate $\lambda$ offsprings, $p_1, p_2, ..., p_\lambda$ by applying compress, expand, and forced mutation operations on parent, $p_0$
    Convert genotypes, $p_0, p_1, ..., p_\lambda$ to phenotypes(CNNs), $c_0, c_1, ..., c_\lambda$
    Train CNNs, $c_0, c_1, ..., c_\lambda$ on training data
    Calculate fitnesses (MCC), $f_0, f_1, ..., f_\lambda$ of trained CNNs, $c_0, c_1, ..., c_\lambda$, on validation data
    Perform neutral mutation on parent, $p_0$
    m ← $argmax(f_0, f_1, ..., f_\lambda)$
    Set $p_0 \leftarrow p_m$ as parent for next generation
    generation ← generation + 1
end while
Return $p_0$ as the best architecture
```

5.3 Experiments and Results

5.3.1 Dataset

We test our method on an image classification dataset called CIFAR10. This dataset was chosen because, to reach high accuracies it requires large networks, which present a computational challenge. CIFAR10 dataset is being used as the benchmark dataset for many similar problems. For example, [11], [18], [19], [12], [10] and many more have used CIFAR10 to evolve their architectures.

The CIFAR10 dataset has 50,000 train images and 10,000 test images. The size of each image is 32 x 32, the number of input channels is 3, and the number of classes is 10. The train images is split into train and validation datasets of ratio 9:1 i.e., 45,000 train images and 5,000 validation images.

5.3.2 Experimental Setting

We used stochastic gradient descent (SGD) to train the CNN architectures on the training dataset with a mini-batch size of 128. Weights are initialized randomly. Adam optimizer [5] is used with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rate is kept constant at 0.01. Each CNN is trained for 50 epochs on the training dataset. After the training is complete, the MCC metric evaluated on the validation dataset is used as the fitness of the individual.

The whole train images are normalized per-pixel along all the 3 channels with mean and standard deviation values of the dataset. Data augmentation method mentioned in [3] is used by padding 4 pixels on all the sides of the image followed by taking a 32 x 32 random crop, and then by taking a random horizontal flip on the cropped image.

Parameter	Value
No. of rows (N_r)	1
No. of columns (N_c)	30
Levels-back (l)	10
Mutation probability	0.03
Compress probability	0.1
Expand probability	0.2
Min. module size	2
Max. module size	5
Module list initial state	Empty

Table 5.3: Parameters used for ECGP method of generating CNNs

Table 5.3 displays the parameters used for generating CNN using ECGP technique. To generate deep architectures we used the number of columns to be 30. A total of 40 node functions are used obtained with all the combinations of functions from table 5.2.

We have trained all our method for 50 generations and recorded its performance. The best CNN architecture obtained after the whole process is re-trained on the whole training dataset of 50,000 images. The trained model is used to classify the 10,000 test images of CIFAR10 and the test accuracy is calculated.

We have implemented our method using Pytorch (version 1.1) and Python3 (version 3.6.5) and the experiments were run on a machine with Tesla P6 GPU having 16GB GPU RAM.

5.3.3 Results

Figure 5.3 displays the test accuracies for each generation compared with the Suganuma-CNN method. The blue line represents the accuracies for Suganuma-CNN method, whereas the orange line represents the accuracies for our method. As we can see, there is not much improvement. They both converge in the similar way.

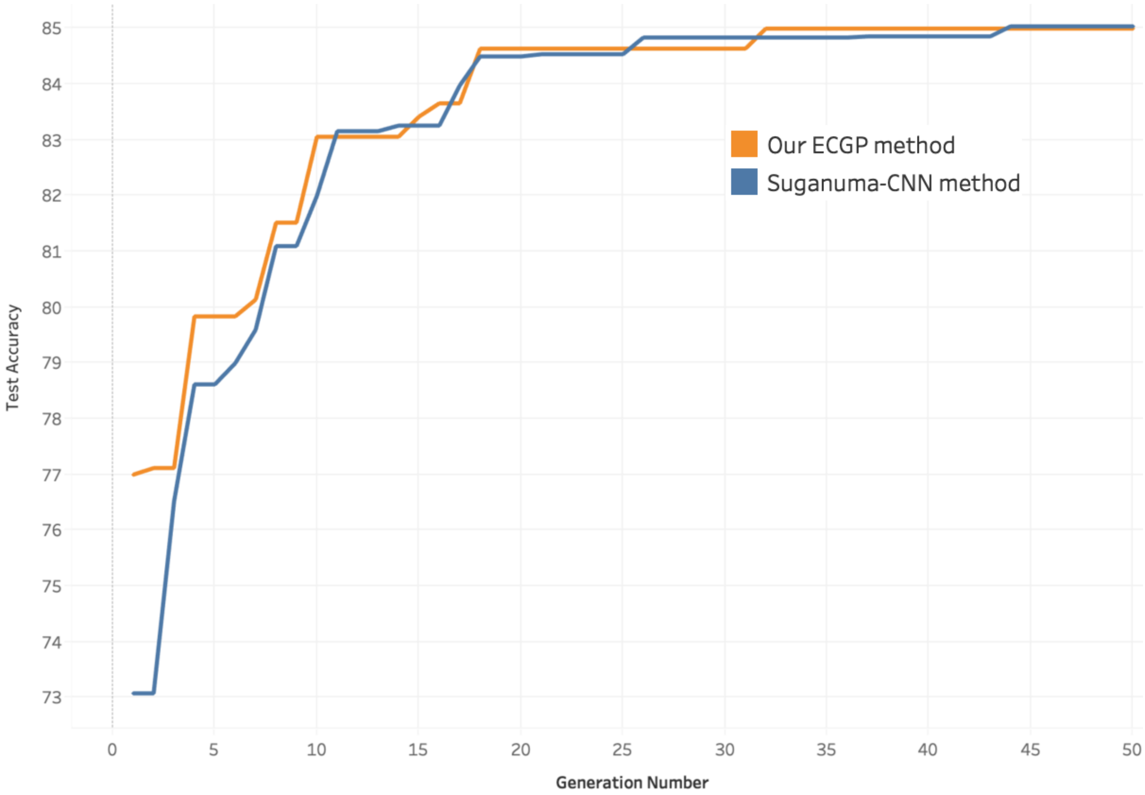


Figure 5.3: Test accuracies for 50 generations for ECGP technique

Table 5.4 displays the test accuracies for both the methods after 50 generations averaged over 3 runs.

Method	Test Accuracy
Suganuma-CNN	86.08 % ± 0.43 %
ECGP-CNN	85.21 % ± 0.31 %

Table 5.4: Classification performance of ECGP-CNN

Figure 5.4 displays the initial and final architectures generated by our method. In figure 5.4, CB_c_f_a stands for ConvBlock with c channels, f × f filter size, and activation a. Also AP_f_s stands for average pooling with f × f filter size and stride s. Full stands for fully connected Softmax layer.

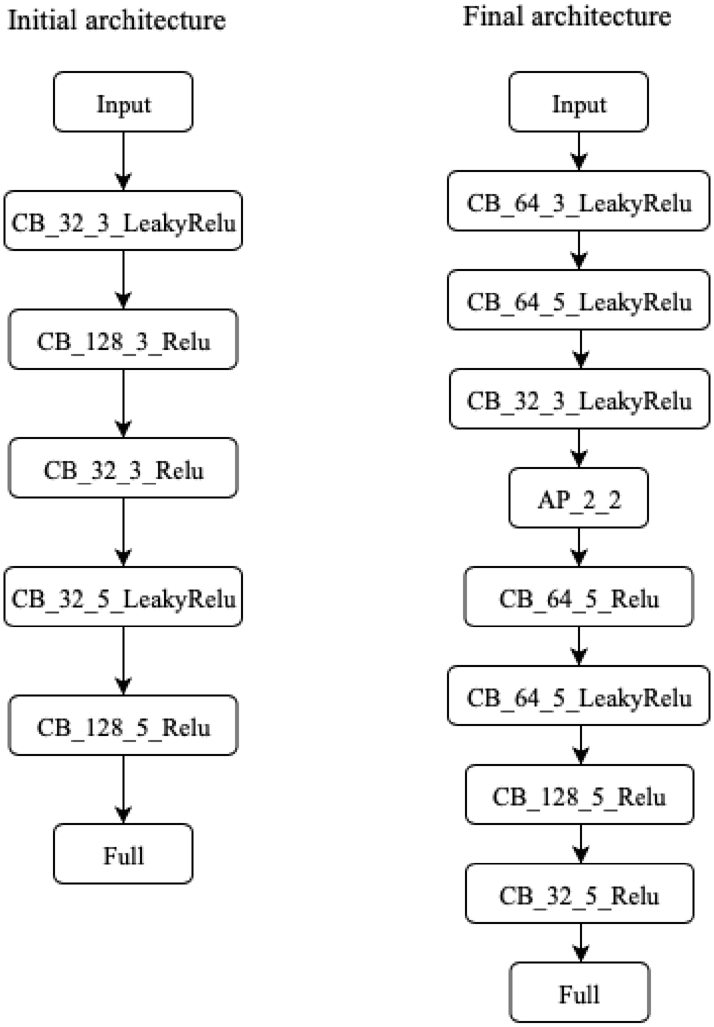


Figure 5.4: Initial and final architectures generated for ECGP technique

5.4 Conclusion

Removal of the dilated convolution and the addition of modules in ECGP has shown some improvement over the CGP-CNN method. But the improvement is negligible, and the performance is similar to that of the Suganuma-CNN method.

The code can be downloaded from https://github.com/manoharmukku/cnn-ecgp-pytorch

Chapter 6

New crossover technique for CGP-CNN

6.1 Introduction

The standard CGP technique introduced by Miller and Thomson has been shown to perform better than standard GP but it uses only mutation operation to generate offsprings, it does not use crossover. In contrast to Genetic Algorithms(GA) and Genetic Programming(GP), the mutation operation is considered to be of only secondary importance than crossover operation. Although the mutation operation is important in GA, the crossover operation contributes a great deal to its performance.

[2] introduced a new crossover technique incorporating it in CGP for simple regression problems. They have shown by experiments that including the crossover operation to CGP, makes the CGP to converge much faster.

In this chapter, we try this new crossover technique on CGP for generating CNN architectures for CIFAR10 dataset. For incorporating crossover to standard CGP, the standard CGP must be changed from integer representation to floating-point representation. We will discuss the new representation of CGP, the crossover operation, the algorithm used, and the results obtained on CIFAR10 dataset.

6.2 CGP with floating-point representation

Generally, a standard CGP genotype consists of a list of integers (as described in section 2.2.1). But to incorporate the new crossover technique in CGP, the representation itself of the standard CGP genotype needs to be modified to use floating-point values. Each floating-point number corresponds to a single gene in the genotype (as in the case with the standard CGP representation) and its value lies in the range [0,1].

Each node in the genotype is still represented as a collection of genes and the purpose of each gene remains same as in standard CGP; the first gene of the node encodes the function/operation performed by the node, and the remaining genes encode the connections to other nodes or inputs. Figure 6.1 provides an example of the new genotype representation, and the decoding process of the floating-point genotype to the standard integer-based genotype.

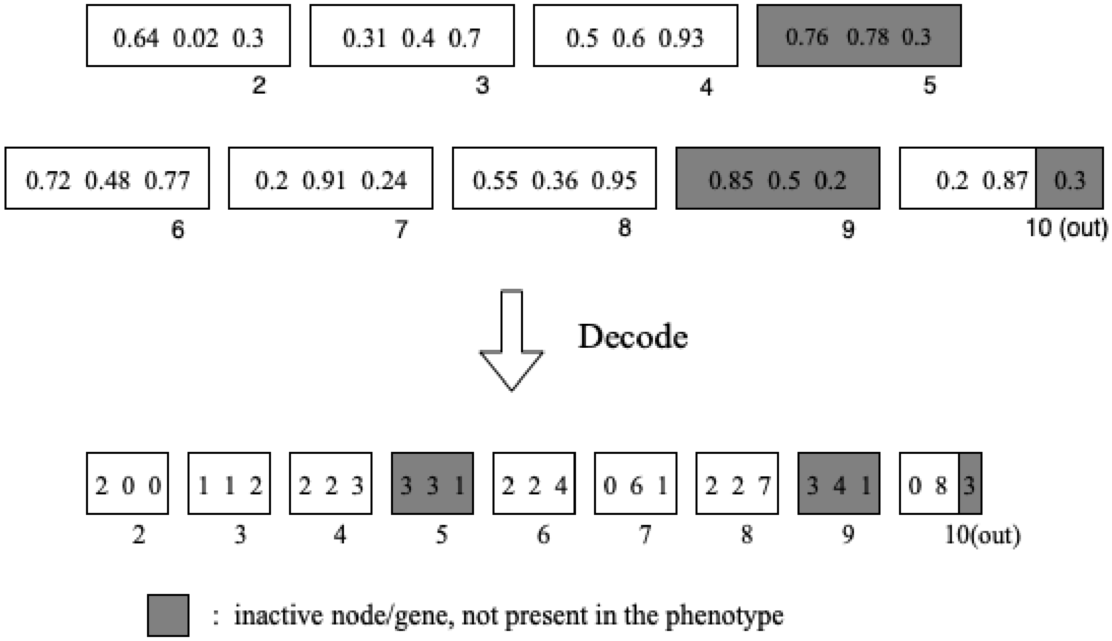


Figure 6.1: The decoding process between the float-point and integer-based genotypes. The function genes are decoded using equation 6.1 whilst the connection genes are decoded using equation 6.2

Equation 6.1 specifies the decoding process of the function gene from the floating-point to the integer-based, and equation 6.2 specifies the decoding of the connection gene from floating-point to integer-based.

$$decoded\_func = floor(gene_i * func_{total}) \tag{6.1}$$

$$decoded\_conn = floor(gene_i * nodenumber_j) \tag{6.2}$$

In equations 6.1 and 6.2, i is defined as $0 \leq i < gene_{total}$, where $gene_{total}$ is the total number of genes in the genotype, $func_{total}$ is the total number of primitive functions in the function set, $nodenumber_j$ is the node number, where j is defined as $0 \leq j < node_{total}$, where $node_{total}$ is the total number of nodes in the genotype including the terminal nodes.

This decoding process is a many-to-one mapping from many floating-point values to a single integer value. Equations 6.3 and 6.4 respectively specify the range of values mapped to a single function gene or connection gene.

$$func_k \in \left[\frac{func_k}{func_{total}}, \frac{func_k + 1}{func_{total}}\right] \tag{6.3}$$

$$input_j \in \left[\frac{nodenumber_j}{node_{total}}, \frac{nodenumber_j + 1}{node_{total}}\right] \tag{6.4}$$

In equations 6.3 and 6.4, $func_k$ is the k^{th} primitive function type in the function set, $func_{total}$ is the total number of primitive functions in the function set, $input_j$ is the node's input connection to the j^{th} node.

6.3 The new crossover operation

Crossover is performed similar to the one in a floating point Genetic Algorithm. Two parents p_1 and p_2 are chosen and crossover is performed according to equation 6.5 to generate two new offsprings o_1 and o_2. A uniformly distributed random number, r_i is generated for each offspring, o_i, where $0 \leq i < 2$ and $0 \leq r_i < 1$.

$$o_i = (1 - r_i) * p_1 + r_i * p_2 \tag{6.5}$$

The crossover operation is performed according to some crossover probability, and is performed on each individual gene of the floating-point genotype.

6.4 The mutation operation

The mutation operation is the same as the one performed in standard CGP, with the only difference that it mutates the genes to uniformly generated random floating-point values in the range [0, 1]. Similar to the one we used in CGP and ECGP technique, we perform forced mutation, i.e. mutate until at least one active node changes. Also neutral mutation is performed on parents, if no better offsprings are generated after a generation.

Without the crossover operation, the new floating-point representation does not change the behaviour of the standard CGP at all. It is same as the standard CGP technique with only mutation.

6.5 Fitness measure

To evaluate fitnesses, we used Matthews Correlation Coefficient (MCC) metric [7] because it considers both the true and false positives and negatives and provides a balanced measure even if the classes are imbalanced. MCC measure returns a value between -1 and $+1$. A $+1$ indicates a perfect prediction, 0 indicates no better than random prediction, and -1 indicates a complete disagreement between observation and prediction. MCC is calculated using the formula in equation (6.6), where TP indicates *true positives*, TN indicates *true negatives*, FP indicates *false positives*, and FN indicates *false negatives*.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{6.6}$$

6.6 Node functions

The function genes for nodes of CGP are chosen to be highly functional modules of CNN. In general, CNNs mostly use convolution and pooling followed by batch normalization and nonlinear activations. Rather than using each of them separately as a function gene for a node as done in [11], we combined them to construct highly functional modules. We prepared five types of functions called ConvBlock, average pooling, max pooling, summation, and concatenation. These operate on the three-dimensional tensors called feature maps defined by the dimensions of rows, columns and channels.

The ConvBlock consists of convolution with stride of 1, followed by batch normalization [4] and rectified linear units (ReLU) [9] activation. Dilated convolution is not used here too. Several ConvBlocks are prepared with different number of output channels $\in \{32, 64, 128\}$, different kernel sizes $\in \{3\times3, 5\times5\}$, and different activations $\in \{ReLU, LeakyReLU\}$ and are added to the function set of CGP. In ConvBlock, before the convolution operation, we pad the input feature maps with zeroes to preserve the row and column sizes of the output. Thus, the $P \times Q \times C$ input feature maps are converted into $P \times Q \times C'$ output feature maps where P, Q, C, and C' are respectively the number of rows, columns, input channels, and output channels.

The average and max pooling respectively do an average and max operation on the feature maps. A kernel size of 2×2 and a stride of 2 are used for pooling. After pooling, the $P \times Q \times C$ input feature maps are converted into $P' \times Q' \times C$ output feature maps, where $P' = \lfloor \frac{P}{2} \rfloor$ and $Q' = \lfloor \frac{Q}{2} \rfloor$.

The summation function does channel by channel element-wise addition of two feature maps. If the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. If the input feature maps have different number of channels, the smaller one is padded with zeroes in the channels dimension to make them have the same number of channels. Thus after summation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions min(P_1, P_2) × min(Q_1, Q_2) × max(C_1, C_2).

The concatenation function does concatenation of two feature maps in the channel dimension. Similar to summation, if the input feature maps have different numbers of rows or columns, the larger one is down-sampled by max pooling to become the same size as the smaller one. Thus after concatenation, two input feature maps of dimensions $P_1 \times Q_1 \times C_1$ and $P_2 \times Q_2 \times C_2$ are converted into an output feature map of dimensions min(P_1, P_2) × min(Q_1, Q_2) × ($C_1 + C_2$).

The output node is a fully connected softmax layer of the specified number of classes. All the nodes of the previous layer are fully connected to the outputs. Table 6.1 specifies all the node functions used.

Node function	Representation	Values
ConvBlock	$\circledast(c, k, a)$	# of channels, $c \in \{32, 64, 128\}$ kernel size, $k \in \{3 \times 3, 5 \times 5\}$ activation, $a \in \{ReLU\}$
Max pooling	$\vee$	-
Average pooling	μ	-
Summation	$\sum$	-
Concatenation	$\|\|$	-

Table 6.1: Node functions used for the CGP method with crossover, for generating CNNs

6.7 Evolutionary Algorithm

Algorithm 3 specifies the evolutionary algorithm used for evolving CNN architectures based on the new crossover technique of CGP. Figure 6.2 illustrates a single generation of the method.

6.8 Experimental Setting

We test our method on an image classification dataset called CIFAR10. This dataset was chosen because, to reach high accuracies it requires large networks, which present a computational challenge. CIFAR10 dataset is being used as the benchmark dataset

Algorithm 3 Evolutionary Algorithm for the new crossover technique on CGP-CNN

```
Randomly initialize two parent genotypes, $p_1$ and $p_2$
Convert $p_1$ and $p_2$ to phenotypes(CNN), $c_1$ and $c_2$
Train $c_1$ and $c_2$ on training data
Calculate fitnesses(MCC), $f_1$ and $f_2$ of trained CNNs, $c_1$ and $c_2$, on validation data
generation $\leftarrow$ 1
while NOT termination criteria do
    Perform crossover on $p_1$ and $p_2$ to generate two offsprings $p_3$ and $p_4$
    Perform forced mutation on $p_3$ and $p_4$
    Convert $p_3$ and $p_4$ to phenotypes(CNNs), $c_3$ and $c_4$
    Train $c_3$ and $c_4$ on training data
    Calculate fitnesses(MCC), $f_3$ and $f_4$ of trained CNNs, $c_3$ and $c_4$, on validation data
    $i_1, i_2 \leftarrow find\_args\_of\_max\_two\_fitnesses(f_1, f_2, f_3, f_4)$
    Set $p_1 \leftarrow p_{i_1}$ and $p_2 \leftarrow p_{i_2}$ as parents for next generation
    Perform neutral mutation on $p_1$ and $p_2$
    generation $\leftarrow$ generation + 1
end while
Return best of $p_1$, $p_2$ (based on fitness) as the final architecture
```

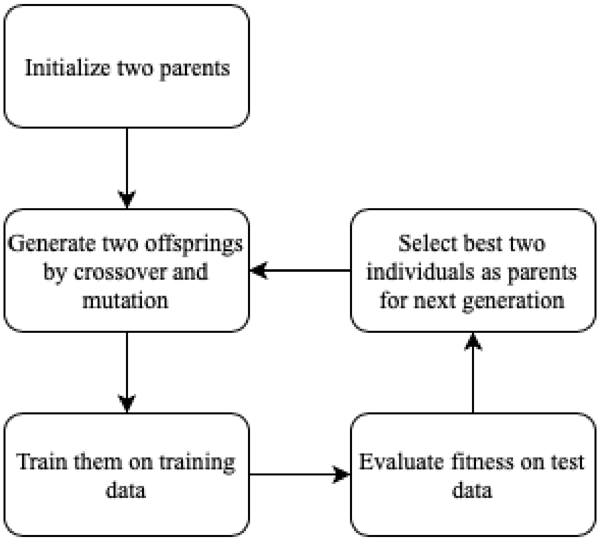

Initialize two parents
Generate two offsprings by crossover and mutation
Select best two individuals as parents for next generation
Train them on training data
Evaluate fitness on test data

Figure 6.2: Overview of a single generation of the new CGP method with crossover

for many similar problems. For example, [11], [18], [19], [12], [10] and many more have used CIFAR10 to evolve their architectures.

The CIFAR10 dataset has 50,000 train images and 10,000 test images. The size of each image is 32 × 32, the number of input channels is 3, and the number of classes is 10. The train images is split into train and validation datasets of ratio 9:1 i.e., 45,000 train images and 5,000 validation images.

We used stochastic gradient descent (SGD) to train the CNN architectures on the training dataset with a mini-batch size of 128. Weights are initialized randomly. Adam optimizer [5] is used with parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rate is kept constant at 0.01. Each CNN is trained for 50 epochs on the training dataset. After the training is complete, the MCC metric evaluated on the validation dataset is used as the fitness of the individual.

The whole train images of CIFAR10 are normalized per-pixel along all the 3 channels with mean and standard deviation values of the dataset. Data augmentation method mentioned in [3] is used by padding 4 pixels on all the sides of the image followed by taking a 32 × 32 random crop, and then by taking a random horizontal flip on the cropped image.

The CNN is evolved for 20 generations and the accuracies are reported.

Parameter	Value
No. of rows (N_r)	5
No. of columns (N_c)	30
Crossover probability	0.75
Mutation probability	0.02

Table 6.2: Parameters used for CGP with crossover method of generating CNNs

Table 6.2 displays the parameters used for generating CNN using CGP with crossover technique. To generate deep architectures we used the number of columns to be 30 and the number of rows to be 5. A total of 10 node functions are used obtained with all the combinations of different values of the functions in table 6.1.

6.9 Results

Figure 6.3 displays the test accuracies for each generation compared with the Suganuma-CNN method for 15 generations. The blue line represents the accuracies for Suganuma-CNN method, whereas the orange line represents the accuracies for our method. As we can see, there is much improvement. CGP-crossover method started to converge fast initially but the convergence slowed down with generations.

Table 6.3 displays the test accuracies for both the methods after 15 generations averaged over 3 runs.

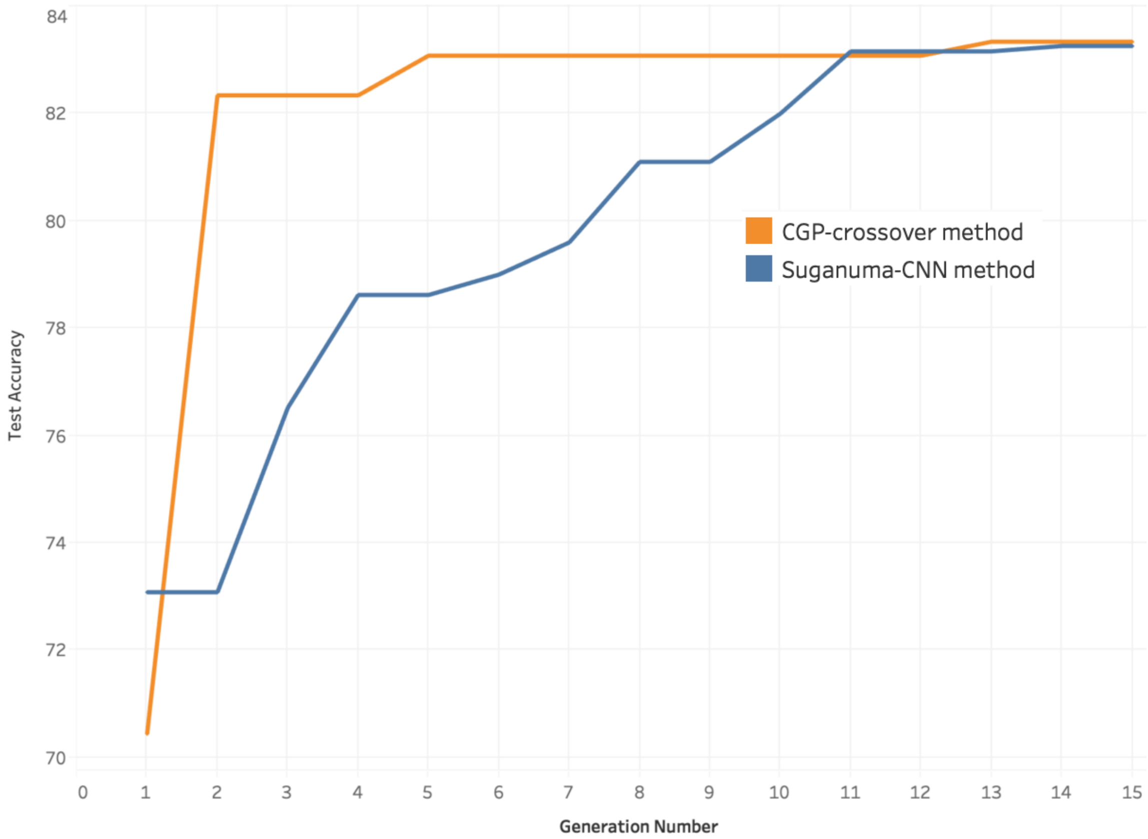


Figure 6.3: Test accuracies for 15 generations of CGP with crossover technique

Method	Test Accuracy
Suganuma-CNN	$83.34\,\% \pm 0.21\,\%$
CGP-Crossover-CNN	$83.26\,\% \pm 0.15\,\%$

Table 6.3: Classification performance of CGP-Crossover-CNN

Figure 6.4 displays the initial and final architectures generated by our method. In figure 6.4, CB_c_f_a stands for ConvBlock with c channels, f × f filter size, and activation a. Also MP_f_s stands for max pooling with f × f filter size and stride s. AP_f_s stands for average pooling with f × f filter size and stride s. Full stands for fully connected Softmax layer.

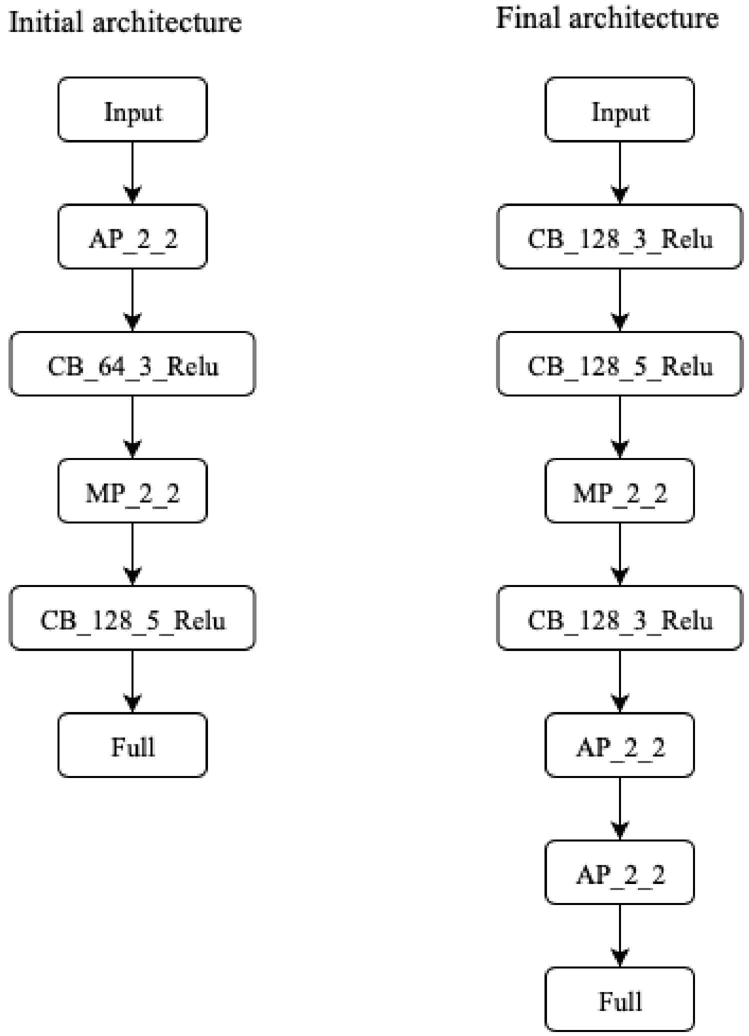


Figure 6.4: Initial and final architectures generated for CGP-Crossover technique in 15 generations

6.10 Conclusion

The CGP technique with crossover has been tested for only 15 generations. In these 15 generations, the CGP-crossover technique started to converge quickly in the beginning but later the convergence slowed down. More testing needs to be done with more number of generations, different values for number of rows, different crossover probabilities, varying crossover probability with generations, etc.

The code can be downloaded from https://github.com/manoharmukku/cnn-cgp-crossover-pytorch

Chapter 7

Conclusions and Future work

We have tested three methods based on CGP, ECGP and CGP-with-crossover by modifying the standard algorithms to use with CNN generation. In the updated CGP method we have tried dilated convolution, number of rows as 1, LeakyRelu activation, a constraint that max pooling should appear only after ConvBlock. In the ECGP method, we have tried the presence of modules and varying parameters. In CGP-Crossover, we have incorporated crossover operation to the standard CGP and tested it for CNN generation. We have reported the results obtained. This work is still in progress and needs to be tested on various other parameters.

Future work

1. We have seen that for large architectures, the network is getting overfitted, resulting in a low test accuracy, thus losing the propogation of a better solution to next generation. We need to handle overfitting by using techniques like Dropout, etc.

2. The presence of a large number of ConvBlocks (40 in CGP-CNN) and a small number of other functions (4 in CGP-CNN) results in a very high probability of ConvBlocks getting selected after mutation. The other functions are getting selected very rarely, and are equivalent to not getting used at all. This issue needs to be solved, say by assigning probabilities for selecting a particular function gene during mutation.

3. Different crossover probabilities, and varying crossover probabilities by decreasing linearly with the generation number, need to be tested for the method of CGP-with-crossover.

4. Various row, column numbers, levels-back parameter, mutation probabilities need to be tried.

Bibliography

[1] Peter J Angeline and Jordan Pollack. Evolutionary module acquisition. In *Proceedings of the second annual conference on evolutionary programming*, pages 154–163. Citeseer, 1993.

[2] Janet Clegg, James Alfred Walker, and Julian Frances Miller. A new crossover technique for Cartesian genetic programming. (May 2014):1580, 2007.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[5] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[6] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[7] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405, 1975.

[8] Julian Francis Miller and Simon L Harding. Cartesian genetic programming. In *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, pages 2701–2726. ACM, 2008.

[9] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[10] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized Evolution for Image Classifier Architecture Search. pages 52–54, 2018.

[11] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-Scale Evolution of Image Classifiers. 2017.

[12] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. *IJCAI International Joint Conference on Artificial Intelligence*, 2018-July:5369–5373, 2018.

[13] James Alfred Walker and Julian Francis Miller. Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems. page 911, 2006.

[14] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in Cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, 2008.

[15] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[16] Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In *European Conference on Genetic Programming*, pages 204–217. Springer, 2001.

[17] Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. pages 1–16, 2016.

[18] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[19] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. 2017.