Evolution of Deep Convolutional Neural Networks Using Cartesian Genetic Programming

Masanori Suganuma suganuma@vision.is.tohoku.ac.jp
RIKEN Center for AIP, Tokyo, Japan
Tohoku University, Miyagi, Japan

Masayuki Kobayashi kobayashi-masayuki-xc@ynu.jp
Shinichi Shirakawa shirakawa-shinichi-bg@ynu.ac.jp
Tomoharu Nagao nagao@ynu.ac.jp
Yokohama National University, Kanagawa, Japan

https://doi.org/10.1162/evco_a_00253

Abstract

The convolutional neural network (CNN), one of the deep learning models, has demonstrated outstanding performance in a variety of computer vision tasks. However, as the network architectures become deeper and more complex, designing CNN architectures requires more expert knowledge and trial and error. In this article, we attempt to automatically construct high-performing CNN architectures for a given task. Our method uses Cartesian genetic programming (CGP) to encode the CNN architectures, adopting highly functional modules such as a convolutional block and tensor concatenation, as the node functions in CGP. The CNN structure and connectivity, represented by the CGP, are optimized to maximize accuracy using the evolutionary algorithm. We also introduce simple techniques to accelerate the architecture search: rich initialization and early network training termination. We evaluated our method on the CIFAR-10 and CIFAR-100 datasets, achieving competitive performance with state-of-the-art models. Remarkably, our method can find competitive architectures with a reasonable computational cost compared to other automatic design methods that require considerably more computational time and machine resources.

Keywords

Genetic programming, convolutional neural network, deep learning.

1 Introduction

Deep learning, a machine learning approach using deep neural networks, is becoming popular for solving artificial intelligence tasks. Deep neural networks (DNNs) have been successful in various tasks such as image recognition (LeCun et al., 1998; Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012), and reinforcement learning tasks (Mnih et al., 2013, 2015). Particularly, convolutional neural networks (CNNs) (LeCun et al., 1998) have demonstrated outstanding performance on image recognition tasks in the last few years and have been applied to a variety of computer vision applications (Vinyals et al., 2015; Zhang et al., 2016). A commonly used CNN architecture consists of a series of convolution and pooling layers followed by fully connected layers. Several recent studies have demonstrated significant progress by developing

Manuscript received: 14 October 2018; revised: 15 February 2019; accepted: 22 February 2019.

CNN architectures. Powerful CNN models (e.g., GoogLeNet (Szegedy et al., 2015), ResNet (He et al., 2016), and DenseNet (Huang et al., 2017)) have continued to show considerable improvement over the years, achieving state-of-the-art results on various benchmark datasets.

Despite their success, designing CNN architectures is still a difficult task because many design parameters exist, such as the depth of a network, the type and parameters of each layer, and the connectivity of the layers. As the successful networks have become deeper and more complex, they require a greater number of structural hyperparameters to be tuned to achieve the best performance on a specific dataset. Therefore, considerable trial and error or expert knowledge is required to construct suitable architectures for a target task. Considering this situation, automatic design methods for CNN architectures are highly beneficial.

Neural network architecture design can be viewed as a model selection problem in machine learning. The straight forward approach is to deal with the architecture design as a hyperparameter optimization problem. Namely, the hyperparameters regarding network structure such as the number of layers and neurons are optimized using techniques based on Bayesian optimization or evolutionary computation (Snoek et al., 2012; Loshchilov and Hutter, 2016) to improve the performance of the validation dataset.

Evolutionary computation has been traditionally applied to optimize both the network topology and the connection weights (Schaffer et al., 1992; Stanley and Miikkulainen, 2002). There are two types of encoding schemes for network representations: direct and indirect encoding (Yao, 1999). Direct encoding represents the number and connectivity of neurons directly as the genotype, whereas indirect encoding represents a generation rule for network architectures. Although almost all traditional approaches optimize the number and connectivity of low-level neurons, modern neural network architectures for deep learning have many units and various types of units to be optimized. Optimizing so many structural parameters in a reasonable amount of computational time may be difficult for traditional evolutionary neural network methods. A promising approach is the use of highly functional modules as a minimum unit to reduce the search space of deep architectures.

In this article, we attempt to design CNN architectures based on genetic programming (GP). We use the Cartesian genetic programming (CGP) (Miller and Thomson, 2000; Harding, 2008; Miller and Smith, 2006) encoding scheme, which is a direct encoding scheme, to represent the CNN structure and connectivity. As we aim to search the CNN architectures, the phenotype of GP should be the network structure. Therefore, we adopt a graph-based GP rather than a tree-based GP. The advantage of the CGP-based method is its simplicity and flexibility; it can represent variable-length network structures including skip connections and encode the network by a fixed length string. To reduce the search space, we also adopt relatively highly functional modules operating the three-dimensional tensor (e.g., convolutional block and tensor concatenation) as the node function in CGP. For instance, one of our node functions consists of a convolution layer, batch normalization, and a rectified linear unit (ReLU), not only a single function or a low-level neuron. To evaluate the architecture represented by the CGP, we train the network using a training dataset by a usual stochastic gradient descent method, and then the performance on another training dataset is assigned as the fitness of the architecture. We call the former dataset the *model training dataset* and the latter dataset the *architecture evaluation dataset*. Based on this fitness function, an evolutionary algorithm optimizes the CNN architectures. We evaluate our method on the CIFAR-10 and CIFAR-100 datasets (Krizhevsky and Hinton, 2009). The experimental results show

that our method can find the competitive CNN architectures with a reasonable computational cost compared with state-of-the-art models.

This article expands on the work in Suganuma et al. (2017). In this article, we additionally propose simple speed-up techniques for an architecture search and add the result for the CIFAR-100 dataset.

The rest of this article is organized as follows. The next section presents related work on the neural network architecture design as well as their limitations. In Section 3, we describe our genetic programming approach to designing CNN architectures. We test the performance of our method in Section 4. Finally, in Section 5, we describe our conclusion and future work.

2 Related Work

This section provides a review of related works on automatic neural network architecture design. We roughly divide the previous studies into two categories: optimization of learning and structure parameters and optimization of neural network architectures. The former indicates the traditional hyperparameter optimization approach, whereas the latter aims to search the flexible architectures, which is more relevant to this article.

2.1 Optimization of Learning and Model Parameters

The hyperparameters in machine learning methods, such as learning rate, regularization coefficients, and the number of neurons in neural networks, should be tuned to improve predictive performance. In general, we cannot obtain gradients of such hyperparameters. The naive methods for hyperparameter optimization are the grid search and the random search (Bergstra and Bengio, 2012). A more sophisticated approach is to use the sequential model-based global optimization methods such as Bayesian optimization (Snoek et al., 2012). Bayesian optimization is a global optimization method of black-box and noisy objective functions, and it maintains a surrogate model learned by using previously evaluated solutions. A Gaussian process is usually adopted as the surrogate model, which can easily handle the uncertainty and noise of the objective function.

Snoek et al. (2012) optimized nine hyperparameters in CNN, such as the number of epochs and the learning rate, and showed that an automatically tuned network outperforms networks tuned by a human expert. Snoek et al. (2015) succeeded in improving the scalability of hyperparameter search by using a deep neural network instead of the Gaussian process to reduce the computational cost for surrogate model building, and they optimized the learning hyperparameters of the fixed CNN architecture. Bergstra et al. (2011) proposed the tree-structured Parzen estimator (TPE) and showed better results than manual search and random search. They also proposed a meta-modeling approach (Bergstra et al., 2013) based on the TPE for supporting automatic hyperparameter optimization. Hutter et al. (2011) proposed an algorithm called sequential model-based algorithm configuration (SMAC) for general algorithm configuration. SMAC adopts a random forest as the surrogate model instead of a Gaussian process. Several studies optimized the hyperparameters of DNNs using SMAC-based methods (e.g., Domhan et al., 2015).

The hyperparameter optimization approach often tunes the learning parameters (e.g., learning rate, mini-batch size, and regularization parameters) and the predefined structural parameters (e.g., the numbers of layers and neurons, and the type of activation functions). In general, this approach requires predefined architectures, which means it is hard to design flexible architectures from scratch.

2.2 Optimization of Neural Network Architectures

2.2.1 Reinforcement Learning Approach

Interesting works, the automatic design of deep neural network architecture using reinforcement learning, have been attempted recently (Zoph and Le, 2017; Baker et al., 2017). In Zoph and Le (2017), a recurrent neural network (RNN) was used to generate neural network architectures, and the RNN was trained with reinforcement learning to maximize the expected accuracy on a learning task. This method uses distributed training and asynchronous parameter updates with 800 graphics processing units (GPUs) to accelerate the reinforcement learning process. Baker et al. (2017) proposed a metamodeling approach based on reinforcement learning to produce CNN architectures. A Q-learning agent explores and exploits a space of model architectures with an ϵ-greedy strategy and experience replay. Additionally, this method optimizes the number of layers, which is a useful feature because it allows a system to design suitable architectures for a target dataset.

We can see that these methods adopt the indirect encoding scheme for network representation from the evolutionary computation perspective because they train the generative rules for network architectures. These methods have succeeded in constructing competitive CNN architectures for image classification tasks. Unlike these methods, our approach uses direct encoding based on Cartesian genetic programming to design the CNN architectures as described in Section 3. In addition, we introduce relatively highly functional modules such as convolutional block and tensor concatenation to efficiently find better CNN architectures.

2.2.2 Evolutionary Computation Approach

Evolutionary algorithms have been applied in optimizing neural network architectures so far (Schaffer et al., 1992; Stanley and Miikkulainen, 2002). The methods for evolutionary neural networks optimize the connection weights and/or network structure of low-level neurons by the evolutionary algorithm. Morse and Stanley (2016) compared the evolutionary algorithm, stochastic gradient descent, and RMSProp on the optimization of the weights in neural networks. The evolutionary algorithm showed competitive performance, but the number of neural network weights used in the experiment was 1,500, which is quite a small number compared to the modern deep neural networks used for computer vision tasks. Sun et al. (2018) proposed a neural network training method for unsupervised DNNs such as the auto-encoder and the restricted Boltzmann machine, which combines the genetic algorithm and stochastic gradient descent. The method, however, was not applied to CNNs. Verbancsics and Harguess (2013, 2015) optimized the weights of artificial neural networks and CNNs by using the hypercube-based neuroevolution of augmenting topologies (HyperNEAT) (Stanley et al., 2009). However, to the best of our knowledge, the methods with HyperNEAT have not achieved competitive performance compared with the state-of-the-art methods. These methods seem to be difficult to scale for recent deep neural networks having a large number of neurons and connections.

To deal with large-scale architectures, an approach combining back-propagation and evolution is promising. In this approach, neural network architectures are designed by the evolutionary algorithm, and the network weights are optimized by a stochastic gradient descent method through back-propagation. Compared to the hyperparameter optimization approach, this approach can design more flexible architectures from scratch by using the evolutionary algorithm.

Real et al. (2017) optimized large-scale neural networks by using the evolutionary algorithm and achieved better performance than modern CNNs in image classification tasks. In this method, they represent the CNN architecture as a graph structure and optimize it by the evolutionary algorithm. The connection weights of the reproduced architecture are optimized by stochastic gradient descent as well as the usual neural network training, and the accuracy for the architecture evaluation dataset is assigned as the fitness. The individuals are initialized by the small networks and become larger through evolution. However, this method was run on 250 computers and required approximately 10 days for optimization.

Miikkulainen et al. (2017) proposed a method called CoDeepNEAT, which is an extended version of NEAT, and showed good performance in image classification and image captioning tasks. This method designs the network architectures using blueprints and modules. The blueprint chromosome is a graph where each node has a pointer to a particular module species. Each module chromosome is a graph that represents a small DNN. Specifically, each node in the blueprint is replaced with a module selected from a particular species to which that node points. During the evaluation phase, the modules and blueprints are combined to generate assembled networks, and the networks are evaluated.

Xie and Yuille (2017) designed CNN architectures using the genetic algorithm with a binary string representation. They proposed a method for encoding a network structure, where the connectivity of each layer is defined by a binary string representation. The type of each layer, the number of channels, and the size of a receptive field are not evolved in this method.

These studies focus on designing large and flexible network architectures. In general, these methods require considerable computational cost to optimize the neural network architectures because they need to train the connection weights to calculate the fitness of the architectures. In this work, we propose using Cartesian genetic programming (CGP) to represent the deep neural network architectures and to use highly functional modules as the node functions to reduce the search space. In addition, we introduce simple techniques to reduce the computational cost of the architecture search.

3 Designing CNN Architectures Using Cartesian Genetic Programming

In our method, we directly encode the CNN architectures based on CGP (Miller and Thomson, 2000; Miller and Smith, 2006; Harding, 2008) and use highly functional modules as node functions. The CNN architecture defined by CGP is trained by a stochastic gradient descent using a model training dataset and assigns the fitness value based on the accuracies for another training dataset (i.e., the architecture evaluation dataset). Then, the architecture is optimized to maximize the accuracy on the architecture evaluation dataset by using the evolutionary algorithm. Figure 1 illustrates an overview of the proposed method. In this section, we describe the network representation and the evolutionary algorithm used in the proposed method. Additionally, we explain the simple speed-up techniques of the architecture search.

3.1 Representation of CNN Architectures

For the CNN architecture representation, we use the CGP encoding scheme which represents an architecture of CNNs as directed acyclic graphs with a two-dimensional grid. CGP was proposed as a general form of genetic programming in Miller and

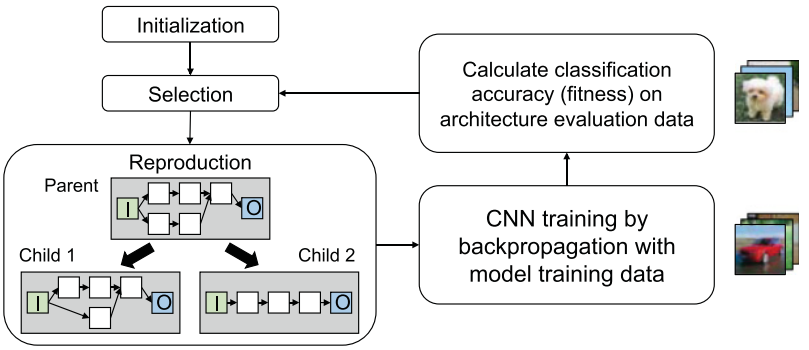


Figure 1: Overview of the proposed method. The method represents the CNN architectures based on CGP. The CNN architecture is trained on a learning task and assigned a fitness based on the accuracies of the trained model for the architecture evaluation dataset. The evolutionary algorithm searches the better architectures.

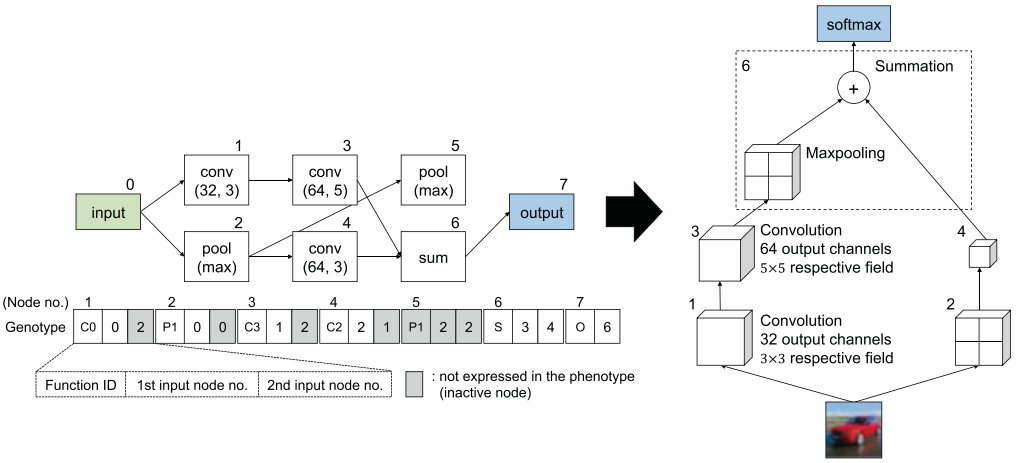


Figure 2: Example of a genotype and a phenotype. The genotype (left) defines the CNN architecture (right). Node No. 5 on the left is inactive and does not appeared in the path from the inputs to the outputs. The summation node applies max pooling to downsample the first input into the same size as the second input.

Thomson (2000), and it is called *Cartesian* because CGP represents a program using a two-dimensional grid of nodes. The graph corresponding to a phenotype is encoded to a string called a genotype and optimized by the evolutionary algorithm.

Let us assume that the grid has N_r rows by N_c columns; then the number of intermediate nodes is $N_r \times N_c$, and the numbers of inputs and outputs depend on the task. The genotype consists of a string of integers with a fixed length, and each gene determines the function type of the node and the connection between nodes. The c-th column's node is only allowed to be connected from $(c-1)$ to $(c-l)$-th column's nodes, where l is called a level-back parameter. Figure 2 shows an example of the genotype, the phenotype, and the corresponding CNN architecture. As seen in Figure 2, the CGP encoding

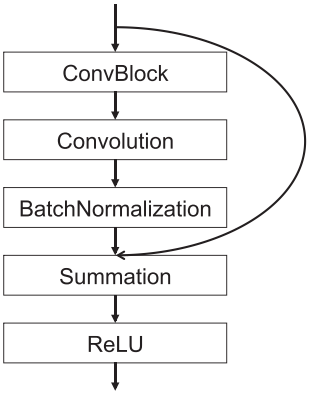


Figure 3: The ResBlock architecture.

scheme has a possibility that not all of the nodes are connected to the output nodes (e.g., node No. 5 in Figure 2). We call these nodes inactive nodes. Whereas the genotype in CGP is a fixed-length representation, the number of nodes in the phenotypic network varies because of the inactive nodes, which is a desirable feature because the number of layers can be determined by the evolutionary algorithm.

Referring to the modern CNN architectures, we select the highly functional modules as the node function. The frequently used processing in the CNN is convolution and pooling; the convolution processing uses local connectivity and spatially shares the learnable weights, and the pooling is nonlinear downsampling. We prepare the six types of node functions called ConvBlock, ResBlock, max pooling, average pooling, concatenation, and summation. These nodes operate on the three-dimensional (3-D) tensor (also known as the feature map) defined by the dimensions of the row, column, and channel.

The ConvBlock consists of a convolutional layer with the stride of one followed by the batch normalization (Ioffe and Szegedy, 2015) and the rectified linear unit (ReLU) (Nair and Hinton, 2010). To maintain the size of the input, we pad the input with zero values around the border before the convolutional operation. Therefore, the ConvBlock takes the $M \times N \times C$ tensor as input and produces the $M \times N \times C'$ tensor, where M, N, C, and C' are the numbers of rows, columns, input channels, and output channels, respectively. We prepare several ConvBlocks with different output channels and receptive field size (kernel size) in the function set of CGP.

As shown in Figure 3, the ResBlock is composed of the ConvBlock, the batch normalization, the ReLU, and the tensor summation. The ResBlock is a building block of the modern succeeded CNN architectures, for example, He et al. (2016), Zagoruyko and Komodakis (2016), and Kupyn et al. (2018). Following this recent trend of human architecture design, we decided to use ResBlock as the building block in our method. The ResBlock performs identity mapping by the shortcut connection as described in He et al. (2016). The row and column sizes of the input are preserved in the same way as the ConvBlock after convolution. As shown in Figure 3, the output feature maps of the ResBlock are calculated by the ReLU activation and the summation with the input. The ResBlock takes the $M \times N \times C$ tensor as an input and produces the $M \times N \times C'$ tensor. We prepare several ResBlocks with different output channels and receptive field size (kernel size) in the function set of CGP.

Table 1: The node functions and abbreviated symbols used in the experiments.

Node type	Symbol	Variation
ConvBlock	CB (C', k)	$C' \in \{32, 64, 128\}$ $k \in \{3 \times 3, 5 \times 5\}$
ResBlock	RB (C', k)	$C' \in \{32, 64, 128\}$ $k \in \{3 \times 3, 5 \times 5\}$
Max pooling	MP	—
Average pooling	AP	—
Concatenation	Concat	—
Summation	Sum	—

C': Number of output channels
k: Receptive field size (kernel size)

The max and average poolings perform a max and average operation, respectively, over the local neighbors of feature maps. We use the pooling with the 2×2 receptive field size and the stride of two. The pooling layer takes the $M \times N \times C$ tensor and produces the $M' \times N' \times C$ tensor, where $M' = \lfloor M/2 \rfloor$ and $N' = \lfloor N/2 \rfloor$.

The concatenation function takes two feature maps and concatenates them in the channel dimension. When concatenating the feature maps with different numbers of rows and columns, we downsample the larger feature map by max pooling to make them the same sizes as the input. Let us assume that we have two inputs of size $M_1 \times N_1 \times C_1$ and $M_2 \times N_2 \times C_2$, then the size of the output feature maps is $\min(M_1, M_2) \times \min(N_1, N_2) \times (C_1 + C_2)$.

The summation performs the element-wise summation of two feature maps, channel by channel. Similar to the concatenation, when summing the two feature maps with the different numbers of rows and columns, we downsample the larger feature map by max pooling. In addition, if the inputs have different numbers of channels, we expand channels of the feature maps with a smaller channel size by filling with zero. Let us assume that we have two inputs of size $M_1 \times N_1 \times C_1$ and $M_2 \times N_2 \times C_2$, then the sizes of the output feature maps are $\min(M_1, M_2) \times \min(N_1, N_2) \times \max(C_1, C_2)$. In Figure 2, the summation node applies the max pooling to downsample the first input into the same size as the second input. By using the summation and concatenation operations, our method can express the shortcut connection or branch layers, such as those used in GoogLeNet (Szegedy et al., 2015) and residual network (ResNet) (He et al., 2016).

The output node represents the softmax function to produce a distribution over the target classes. The outputs fully connect to all elements of the input. The node functions used in the experiments are displayed in Table 1.

3.2 Evolutionary Algorithm

Following the standard CGP, we use a point mutation as the genetic operator. The function and the connection of each node randomly change to valid values according to the mutation rate. The fitness evaluation of the CNN architecture involves the CNN training and requires approximately 0.5 to 1 hours in our setting. Therefore, we need to efficiently evaluate some candidate solutions in parallel at each generation. To efficiently use the computational resource, we repeatedly apply the mutation operator while an active node does not change, and obtain the candidate solutions to be evaluated. We call this mutation the forced mutation. Moreover, to maintain a neutral drift, which is

Algorithm 1 The evolutionary algorithm used in the proposed method. A modified $(1 + \lambda)$ evolution strategy is used to search better architectures

```
Generate an initial individual as parent P.
Train a CNN represented by P and assign the fitness based on the accuracies
  for the architecture evaluation dataset.
repeat
    Generate the set of λ offspring C by applying the forced mutation to P.
    Train the λ CNNs represented by C in parallel and assign the fitness based
      on the accuracies for the architecture evaluation dataset.
    if The best fitness in the offspring C is worse than the parent P then
        Apply the neutral mutation to the parent P.
    end
    Select an elite individual from the set of P and C.
    Replace P with the elite individual.
until stopping criteria is satisfied;
```

effective for the CGP evolution (Miller and Smith, 2006; Miller and Thomson, 2000), we modify a parent by the neutral mutation if the fitnesses of the offspring do not improve. The neutral mutation operates on only the genes of inactive nodes without modification of the phenotype. We use the modified $(1 + \lambda)$ evolution strategy (with $\lambda = 2$ in our experiments) using the above artifice. The procedure of our evolutionary algorithm is listed in Algorithm 1.

The $(1 + \lambda)$ evolution strategy, the default evolutionary algorithm in CGP, is an algorithm with fewer strategy parameters: the mutation rate and the offspring size, meaning that we do not need to expend considerable effort to tune such strategy parameters. This is a reason we use the $(1 + \lambda)$ evolution strategy in our method.

3.3 Speed-Up Techniques

The proposed CNN architecture optimization is time-consuming because it requires training the candidate CNN architectures in the usual way to assign the fitness value. We introduce two speed-up techniques for the architecture search: the rich initialization and the early termination of training. In the rich initialization, we initialize the individual by ResNet (He et al., 2016) or DenseNet (Huang et al., 2017) such as structure and start the evolution with a good architecture. The early termination technique stops the neural network training runs that are unlikely to reach better accuracy.

3.3.1 Rich Initialization

Using a good and hand-designed initial network architecture helps the efficient architecture search more than using a randomly initialized network architecture. By initializing the individual using a sophisticated architecture, we expect to find a better architecture in an early generation. In the experiment, we consider two well-known CNN architectures, the residual network (ResNet) (He et al., 2016) and the densely connected convolutional network (DenseNet) (Huang et al., 2017) as the initial individuals. Because we use the direct encoding of network architectures based on CGP, it is easy to edit the genotype string to represent the architectures we want. Specifically, to leverage the original node functions, the CGP-CNN starts with the modified ResNet and DenseNet architectures. The architectures of the modified ResNet and DenseNet are displayed in Table 2. The ResNet and DenseNet architectures can be represented by

Table 2: The modified ResNet and DenseNet architectures for the rich initialization. The symbols of the node functions are defined in Table 1.

(a) The modified ResNet.	
Layers	Node functions
ResBlock (1)	RB (32, 3) × 3
Pooling Layer	MP
ResBlock (2)	RB (64, 3) × 3
Pooling Layer	MP
ResBlock (3)	(128, 3) × 3
Pooling Layer	MP
Classification Layer	Fully-connected

(b) The modified DenseNet.	
Layers	Node functions
Convolution	CB (32, 3)
Dense Block (1)	[CB (32, 3), Concat] × 2
Transition Layer (1)	CB (32, 3), AP
Dense Block (2)	[CB (32, 3), Concat] × 4
Transition Layer (2)	CB (64, 3), AP
Dense Block (3)	[CB (32, 3), Concat] × 4
Classification Layer	AP, Fully-connected

using ResBlock and ConvBlock, respectively. Although these architectures are slightly different from those in He et al. (2016) and Huang et al. (2017), considering the pooling layers in ResNet and the transition layers in the DenseNet, they are more promising initial individuals than the randomly initialized ones. We denote this rich initialization technique as RichInit.

3.3.2 Early Termination of Network Training

In the training procedure for the CNNs used in Suganuma et al. (2017), CNNs are trained for a fixed number of epochs. However, one can stop the training early to save computational costs if the architecture seems to have no chance of reaching good performance. Here, we introduce a simple early termination technique based on a reference curve.

The reference curve is constructed by using the previous accuracy curves of network training and is used to decide whether the current architecture is promising or not. Let $\boldsymbol{f}^t = (f_1^t \ldots, f_{N_{\text{epoch}}}^t)$ denote the reference curve at the t-th generation, where N_{epoch} indicates the maximum number of epochs, and $\boldsymbol{f}^1$ is initialized with the initial parent's accuracy curve for the architecture evaluation dataset. We terminate the CNN training if the accuracies for the architecture evaluation dataset of the current architecture are worse than the values of the reference curve by N consecutive times. If the training is terminated early, the fitness of the architecture is assigned zero. Figure 4 illustrates the concept of the early termination.

When the best fitness among the λ offspring exceeds the parent's fitness, the values of the reference curve $\boldsymbol{f}^t$ are updated by taking the average of the reference curve and

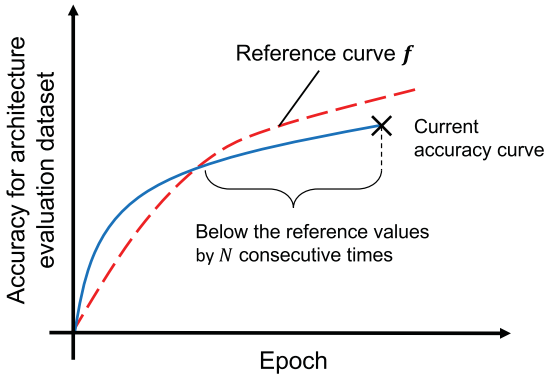


Figure 4: Concept image of our early termination criterion. When an architecture records low accuracies for the architecture evaluation dataset in several consecutive epochs, we terminate the architecture training.

Algorithm 2 The procedure for network training with the early termination

Input : N_{epoch}: The maximum number of epochs for weight training
$\boldsymbol{f}^t = (f_1^t, \ldots, f_{N_{\text{epoch}}}^t)$: Reference curve at the t-th generation
N: Parameter to determine early termination

Initialize counters as $c = 0$, $i = 1$
repeat
  Optimize the weights by a stochastic gradient descent using the model training dataset for an epoch. Then, calculate the current network's accuracy f_i^c for the architecture evaluation dataset.
  if $f_i^c < f_i^t$ **then**
    $c = c + 1$
  else
    $c = 0$
  end
  $i = i + 1$
until $i \leq N_{\text{epoch}}$ *and* $c < N$;

the accuracy curve of the best offspring: $\boldsymbol{f}^{t+1} = \frac{1}{2}(\boldsymbol{f}^t + \boldsymbol{f}^c)$, where $\boldsymbol{f}^c$ is the accuracy curve of the best offspring, and the values consist of the accuracies for the architecture evaluation dataset at each epoch. The procedures for early termination and the reference curve update are listed in Algorithms 2 and 3, respectively. We expect that the evaluation process of the architectures to speed up without performance deterioration by using early termination.

4 Experiments and Results

4.1 Dataset

We apply our method to the CIFAR-10 and CIFAR-100 datasets consisting of 60,000 color images (32 × 32 pixels) in 10 and 100 classes, respectively. Each dataset is split into a training set of 50,000 images and a test set of 10,000 images. We randomly

Algorithm 3 The procedure for reference curve update

Input : $\boldsymbol{f}^t = (f_1^t, \ldots, f_{N_{\text{epoch}}}^t)$: Reference curve at the t-th generation
F_{p}: Parent's fitness, F_{c}: Best fitness value among offspring
$\boldsymbol{f}^c$: Accuracy curve of the best offspring

if $F_{\text{c}} > F_{\text{p}}$ **then**
  Update the reference curve as:
  $\boldsymbol{f}^{t+1} = \frac{1}{2}(\boldsymbol{f}^t + \boldsymbol{f}^c)$
else
  $\boldsymbol{f}^{t+1} = \boldsymbol{f}^t$
end

sample 45,000 examples from the training set to train the CNN, and the remaining 5,000 examples are used for architecture evaluation (i.e., fitness evaluation of CGP).

On the CIFAR-10 dataset, we also consider a small-data scenario. The small-data scenario is a situation where we only have a small number of data for training. We use only one-tenth of the dataset in the experiment. In general, the performance of the CNN architectures highly depends on the dataset, and it is difficult to manually design an appropriate network architecture for a new dataset. The hand-designed CNN architectures seem to be tuned for the benchmark datasets such as CIFAR-10. The purpose of the small-data scenario is to simulate the new dataset. In the small-data scenario, we randomly sample 4,500 images for the model training and 500 images for architecture evaluation.

4.2 Experimental Setting

To assign the fitness value to the candidate CNN architecture, we train the CNN by stochastic gradient descent (SGD) with a mini-batch size of 128. The softmax cross-entropy loss is used as the loss function. We initialize the weights by the method described in He et al. (2015) and use the Adam optimizer (Kingma and Ba, 2015) with an initial learning rate $\alpha = 0.01$, and momentum $\beta_1 = 0.9$, $\beta_2 = 0.999$. We train each CNN for 50 epochs and use the maximum accuracy of the last ten epochs as the fitness value. We reduce the learning rate by a factor of 10 at the 30th epoch.

We preprocess the data with pixel-mean subtraction. To prevent overfitting, we use a weight decay with coefficient 1.0×10^{-4}. We also use data augmentation based on He et al. (2016): padding 4 pixels on each size and randomly crop a 32×32 patch from the padded image or its horizontally flipped image.

The parameter setting for CGP is shown in Table 3. We use a relatively large number of columns to generate deep architectures. The number of active nodes in the individual of CGP is restricted. Therefore, we apply the mutation operator until the CNN architecture that satisfies the restriction of the number of active nodes is generated. The offspring size λ is two; that is the same number of GPUs in our experimental machines. We test two node function sets called ConvSet and ResSet for our method. The ConvSet contains ConvBlock, max pooling, average pooling, summation, and concatenation in Table 1, and the ResSet contains ResBlock, Max pooling, Average pooling, Summation, and Concatenation. The difference between these two function sets is whether the set contains ConvBlock or ResBlock. The numbers of generations are 500 for ConvSet, 300 for ResSet, and 500 for the small-data scenario.

The best CNN architecture from the CGP process is retrained using all 50,000 images in the training set, and then we compute the test accuracy. We optimize the weights

Table 3: Parameter setting for the CGP.

Parameters	Values
Mutation rate	0.05
# Offspring (λ)	2
# Rows (N_r)	5
# Columns (N_c)	30
Minimum number of active nodes	10
Maximum number of active nodes	50
Levels-back (l)	10

of the obtained architecture for 500 epochs with a different training procedure; we use SGD with a momentum of 0.9, a mini-batch size of 128 and a weight decay of 5.0×10^{-4}. Following the learning rate schedule in He et al. (2016), we start with a learning rate of 0.01 and set it to 0.1 at the 5th epoch, and reduce it by a factor of 10 at the 250th and 370th epochs. We report the test accuracy at the 500th epoch as the final performance.

We implement the proposed method using the Chainer framework (Tokui et al., 2015) (version 1.16.0) and run it on a machine with two NVIDIA GeForce GTX 1080 or two GTX 1080 Ti GPUs. We use a GTX 1080 and 1080 Ti for the experiments on the CIFAR-10 and 100 datasets, respectively. Due to the memory limitation, the candidate CNNs occasionally take up the GPU memory, and the network training process fails due to an out-of-memory error. In that case, we assign a zero fitness to the candidate architecture.

4.3 Result on the CIFAR-10 and 100 Datasets

We run the proposed method ten times on each dataset and report the classification errors. Here, the result of the proposed method without speed-up techniques (rich initialization and early termination described in Subsection 3.3) is discussed. We compare the classification performance with the state-of-the-art CNN models, including the hand-designed CNNs and automatically designed CNNs by the architecture search methods, on the CIFAR-10 and 100 datasets. A summary of the classification performances is provided in Tables 4 and 5. We refer to the architectures constructed by the proposed method as CGP-CNN. For instance, CGP-CNN (ConvSet) means the proposed method with the ConvSet node function set. The models, Maxout, Network in Network, VGG, ResNet, FractalNet, and Wide ResNet, are the hand-designed CNN architectures, whereas MetaQNN, Neural Architecture Search, Large-Scale Evolution, Genetic CNN, and CoDeepNEAT are the models obtained by the architecture search methods. The hand-designed CNN architectures mean that the CNN architectures are designed by human experts. The values of other models, except for VGG and ResNet on CIFAR-100, are referenced from the literature. We implemented the VGG net and ResNet for CIFAR-100 because they were not applied to these datasets in Simonyan and Zisserman (2015) and He et al. (2016). The architecture of the VGG is identical with configuration D in Simonyan and Zisserman (2015). We denote this model as VGG in this article. In Tables 4 and 5, the numbers of learnable weight parameters in the models are also listed. In CGP-CNN, the numbers of learnable weight parameters of the best architecture are reported.

Table 4: Comparison of the error rates (%), the numbers of learnable weight parameters, and the search costs on the CIFAR-10 dataset. We run the proposed method ten times for each dataset and report the classification errors in the format of "best (mean ± std)." We refer to the architectures constructed by the proposed method as CGP-CNN. In CGP-CNN, the numbers of learnable weight parameters of the best architecture are reported. The values of other models are referenced from the literature.

Model	# params	Test error	GPU days
Maxout (Goodfellow et al., 2013)	—	9.38	—
Network in Network (Lin et al., 2014)	—	8.81	—
VGG (Simonyan and Zisserman, 2015)	15.2M	7.94	—
ResNet (He et al., 2016)	1.7M	6.61	—
FractalNet (Larsson et al., 2017)	38.6M	5.22	—
Wide ResNet (Zagoruyko and Komodakis, 2016)	36.5M	4.00	—
CoDeepNEAT (Miikkulainen et al., 2017)	—	7.30	—
Genetic CNN (Xie and Yuille, 2017)	—	7.10	17
MetaQNN (Baker et al., 2017)	3.7M	6.92	80–100
Large-Scale Evolution (Real et al., 2017)	5.4M	5.40	2750
Neural Architecture Search (Zoph and Le, 2017)	37.4M	3.65	16800–22400
CGP-CNN (ConvSet)	1.50M	5.92 (6.48 ± 0.48)	31
CGP-CNN (ResSet)	2.01M	5.01 (6.10 ± 0.89)	30

Table 5: Comparison of the error rates (%) and the numbers of learnable weight parameters on the CIFAR-100 dataset. We run the proposed method ten times for each dataset and report the classification errors in the format of "best (mean ± std)." We refer to the architectures constructed by the proposed method as CGP-CNN. In CGP-CNN, the numbers of learnable weight parameters of the best architecture are reported. The values of other models except for VGG and ResNet are referenced from the literature.

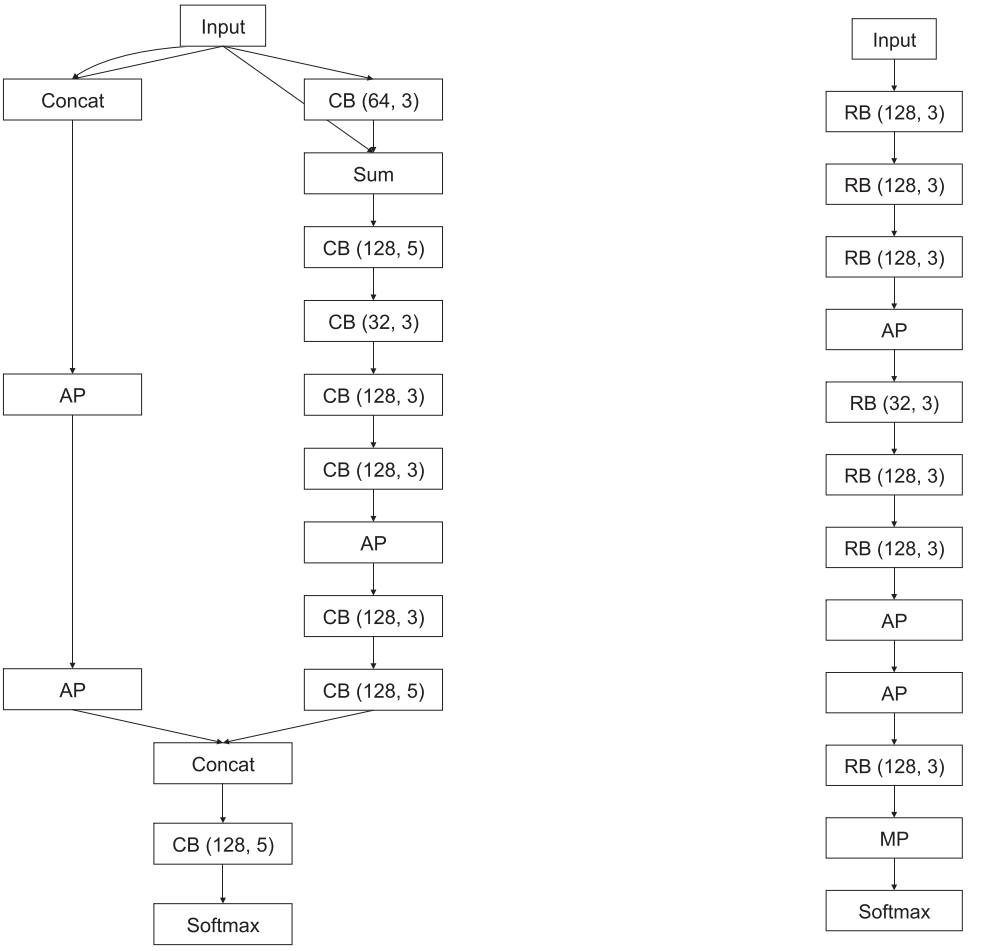
Model	# params	Test error
Maxout (Goodfellow et al., 2013)	—	38.57
Network in Network (Lin et al., 2014)	—	35.68
VGG (Simonyan and Zisserman, 2015)	15.2M	33.45
ResNet (He et al., 2016)	1.7M	32.40
FractalNet (Larsson et al., 2017)	38.6M	23.30
Wide ResNet (Zagoruyko and Komodakis, 2016)	36.5M	19.25
CoDeepNEAT (Miikkulainen et al., 2017)	—	—
Neural Architecture Search (Zoph and Le, 2017)	37.4M	—
Genetic CNN (Xie and Yuille, 2017)	—	29.03
MetaQNN (Baker et al., 2017)	3.7M	27.14
Large-Scale Evolution (Real et al., 2017)	40.4M	23.0
CGP-CNN (ConvSet)	2.01M	26.7 (28.1 ± 0.83)
CGP-CNN (ResSet)	4.60M	25.1 (26.8 ± 1.21)

On the CIFAR-10 dataset, CGP-CNNs outperform most of the hand-designed models and have a good balance between the classification errors and the number of parameters. CGP-CNN (ResSet) shows better performance compared to CGP-CNN (ConvSet). Compared with other architecture search methods, CGP-CNN (ConvSet and ResSet) outperforms MetaQNN (Baker et al., 2017), Genetic CNN (Xie and Yuille, 2017), and CoDeepNEAT (Miikkulainen et al., 2017), and the best architecture of CGP-CNN (ResSet) outperforms Large-Scale Evolution (Real et al., 2017). The Neural Architecture Search (Zoph and Le, 2017) achieved the best error rate, but this method used 800 GPUs and required considerable computational costs to search the best architecture. Table 4 also shows the number of GPU days (the computational time multiplied by the number of GPUs used in the experiments) for the architecture search. As seen in this table, our method can find a good architecture with a reasonable computational cost. We guess that our method could reduce the search space and find better architectures in early iteration by using the highly functional modules. The CIFAR-100 dataset is a very challenging task because there are many classes. CGP-CNN finds the competitive network architectures in a reasonable computational time. Even though our model is not at the same level as the state-of-the-art architectures, our model has a good balance between the classification errors and the number of parameters.

The error rates of the architecture search methods (not only our method) do not reach the Wide ResNet which is a human-designed architecture. However, these human-designed architectures are developed with the expenditure of tremendous human effort. An advantage of architecture search methods is that they can automatically find a good architecture for a new dataset. Another advantage of CGP-CNN is that the numbers of weight parameters in the discovered architectures are fewer than those in the human-designed architectures, which is beneficial when we want to implement CNN on a mobile device. Note that we did not introduce any criteria for the architecture complexity in the fitness function. It might be possible to find more compact architectures by introducing the penalty term into the fitness function, which is important future work.

Figure 5 illustrates the examples of the CNN architectures obtained by the proposed method, CGP-CNN (ConvSet and ResSet). As seen in Figure 5, we observe the complex architectures that are hard to design by hand. Specifically, CGP-CNN (ConvSet) uses the summation and concatenation nodes leading a wide network and allowing the formation of skip-connections. Therefore, the CGP-CNN (ConvSet) architecture is wider than that of CGP-CNN (ResSet). Additionally, we also observe that CGP-CNN (ResSet) has a similar structure to ResNet (He et al., 2016). ResNet consists of a series of two types of modules: the module with several convolutions and shortcut connections without downsampling, and downsampling convolution with a stride of 2. Although our method cannot downsample in the ConvBlock and the ResBlock, we see that CGP-CNN (ResSet) uses pooling layer as an alternative to the downsampling convolution. We can say that our method can also find an architecture similar to one designed by human experts.

In addition, we conducted the Wilcoxon rank sum test to statistically compare ResSet and ConvSet. We set the significance level to 5% (i.e., 5.0×10^{-2}). On CIFAR-10 and CIFAR-100, the p-values for ResSet and ConvSet were 3.19×10^{-2} and 1.15×10^{-2}, respectively, indicating that the architectures using ResSet outperform those using ConvSet. We cannot conduct a statistical comparison test between CGP-CNN and other architecture search methods because we cannot obtain the detailed experimental results of other methods.



(a) CGP-CNN (ConvSet) (b) CGP-CNN (ResSet)

Figure 5: The CNN architectures obtained by the proposed method on the CIFAR-10 dataset.

4.4 The Effect of the Rich Initialization

To investigate the effect of the rich initialization described in Subsection 3.3.1, we compare the CGP-CNN (ConvSet) and CGP-CNN (ResSet) models using rich initialization to those models without the rich initialization. Since the DenseNet and ResNet-like architectures can be constructed by using our ConvSet and ResSet, respectively, we initialize the CGP-CNN (ConvSet) with the DenseNet and initialize the CGP-CNN (ResNet) with the ResNet. A summary of the error rates and computational times is provided in Table 6. We observe that rich initialization contributes to improving the performance on both the CIFAR-10 and CIFAR-100 datasets. Additionally, the performances of the ResNet-like initialization show better performance than those of the DensNet-like initialization.

We additionally conducted the Wilcoxon rank sum tests for two independent samples to analyze the effects of rich initialization. Based on a statistical test with the

Table 6: Comparison of the error rates (%), the computational times, and the numbers of learnable weight parameters, on the CIFAR-10 and 100 datasets. CGP-CNNs with/without rich initialization (RichInit) are compared. We run each method ten times and report the classification errors in the format of "best (mean ± std)." The computational times are the mean over the ten runs. Note that we used different GPUs; GTX 1080 and 1080 Ti were used for CIFAR-10 and 100, respectively. The numbers of the learnable weight parameters of the best architecture are reported.

Model	# params	Time (days)	Error rate (CIFAR-10)	Error rate (CIFAR-100)
CGP-CNN (ConvSet)	1.50M	15.6	5.92 (6.48 ± 0.48)	—
with RichInit (DenseNet)	2.01M	14.5	5.01 (5.80 ± 0.52)	—
CGP-CNN (ConvSet)	2.04M	13.0	—	26.7 (28.1 ± 0.83)
with RichInit (DenseNet)	2.95M	15.7	—	24.6 (26.4 ± 1.32)
CGP-CNN (ResSet)	3.52M	14.7	5.01 (6.10 ± 0.89)	—
with RichInit (ResNet)	2.72M	12.4	4.90 (5.60 ± 0.52)	—
CGP-CNN (ResSet)	3.43M	10.9	—	25.1 (26.8 ± 1.21)
with RichInit (ResNet)	4.34M	11.7	—	23.8 (25.8 ± 1.22)

significance level of 5%, the p-values for ResSet with/without rich initialization on CIFAR-10 and 100 were 5.5×10^{-3} and 2.2×10^{-2}, respectively, and the values for ConvSet with/without the rich initialization on CIFAR-10 and 100 were 1.4×10^{-2} and 1.9×10^{-2}, respectively. The p-values for both ResSet and ConvSet were less than 5.0×10^{-2}, and thus, rich initialization can significantly improve the performance over that of the original models. We, however, notice that the rich initialization of the architecture has a possibility of becoming stuck in a local optimum solution. Therefore, we may need to take into account the diversity of the population when rich initialization is used.

4.5 The Effect of the Early Termination

We conducted an experiment introducing early termination of network training on the CIFAR-10 dataset to check the effect. The parameter for determining the termination, N, is varied as $N = 3, 5$, and 10. The parameter setting is the same as that described in Subsection 4.2. Table 7 shows the error rates and the computational times of the proposed method with/without early termination. The average numbers of epochs for network training are also listed in Table 7. We also report the case when we use both speed-up techniques, rich initialization and early termination, in ResSet.

From Table 7, early termination reduces the optimization time without significant performance deterioration and the optimization time decreases as N decreases. The average number of epochs is half that of the number in the original method when $N = 10$ and decreases approximately linearly as N decreases. The computational time also decreased as the average number of epochs decreases. In the CGP-CNN (ResSet) with $N = 3$, the computational time becomes 2.39 days (i.e., 16.3% of that of the original CGP-CNN (ResSet)). This is reasonable for execution by most users. Additionally, we observe that the combination of early termination and rich initialization is more effective considering both computational time and performance; that is, the performances are better than those of the architectures without rich initialization.

Table 7: Comparison of the error rates (%), the computational times, and the average numbers of epochs for network training, on the CIFAR-10 dataset with the different parameter N. We run each method ten times and report the errors in the format of "best (mean ± std)." The values of the computational time and the average number of epochs are the mean over the ten runs.

Method	Error rate	Time (days)	Average # epochs
CGP-CNN (ConvSet)	5.92 (6.48 ± 0.48)	15.6	50.0
with early termination ($N = 3$)	5.61 (6.52 ± 0.59)	3.32	8.97
with early termination ($N = 5$)	5.81 (6.34 ± 0.39)	6.73	18.8
with early termination ($N = 10$)	6.40 (6.93 ± 0.50)	11.4	30.6
CGP-CNN (ResSet)	5.01 (6.10 ± 0.89)	14.7	50.0
with early termination ($N = 3$)	5.30 (6.22 ± 0.46)	2.39	7.96
with early termination ($N = 5$)	5.42 (6.34 ± 0.61)	4.93	15.1
with early termination ($N = 10$)	5.27 (6.12 ± 0.50)	6.05	23.9
CGP-CNN (ResSet) with RichInit	4.90 (5.60 ± 0.52)	11.7	50.0
with early termination ($N = 3$)	5.06 (5.83 ± 0.47)	1.29	6.88
with early termination ($N = 5$)	5.07 (5.76 ± 0.57)	2.86	12.4
with early termination ($N = 10$)	4.90 (5.54 ± 0.60)	6.36	22.2

In addition, we conducted Kruskal-Wallis rank sum tests for four independent samples—the CGP-CNN with early terminations of $N = 3$, 5, and 10, and the one without early termination—to analyze the effect of the early termination technique. For ConvSet, ResSet and ResSet with RichInit, the p-values were 1.28×10^{-1}, 4.77×10^{-1} and 4.86×10^{-1}, respectively. The p-values for all cases were large ($> 5.0 \times 10^{-2}$), indicating that each model achieved similar classification performance. As a result, our early termination technique can reduce the computational time without performance deterioration.

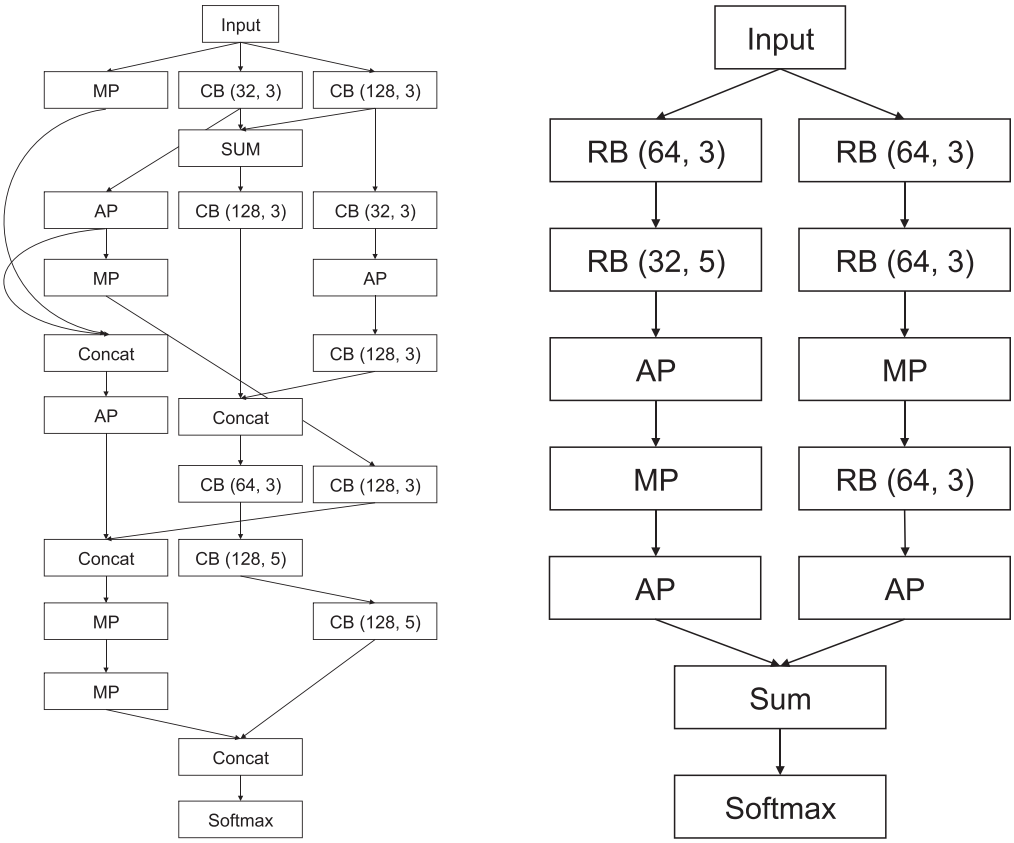
4.6 Result on the Small-Data Scenario

In the small-data scenario, we compare our method with VGG and ResNet. We trained the VGG and ResNet models with the same settings used in the retraining process of the proposed method. Table 8 shows the comparison of error rates in the small-data scenario. We observe that our methods, CGP-CNN (ConvSet) and CGP-CNN (ResSet), can find better architectures than VGG and ResNet.

VGG and ResNet are designed and tuned for a relatively large amount of data and have millions of parameters to train. Therefore, the number of samples in the small-data scenario seems to be too small to prevent overfitting. Meanwhile, our method can automatically tune the architecture depending on the dataset and achieve better performance even on the small datasets. The numbers of learnable weight parameters are relatively small, suggesting that the proposed method finds the compact architectures to prevent overfitting. Figure 6 illustrates the architectures constructed by using the proposed method. From this figure, our method found relatively wide structure compared with the architectures that appeared in Figure 5. The computational time of the proposed method in the small-data scenario is a few days using a NVIDIA GeForce GTX 1080 Ti.

Table 8: Comparison of the error rates on the CIFAR-10 dataset (small-data scenario). We run the experiment ten times and report the classification errors in the format of "best (mean ± std)." The numbers of learnable weight parameters of the best architecture are also reported.

Model	Error rate	# params
VGG (Simonyan and Zisserman, 2015)	23.05 (24.07 ± 0.43)	15.2M
ResNet (He et al., 2016)	24.01 (24.83 ± 0.51)	1.70M
CGP-CNN (ConvSet)	19.78 (22.14 ± 1.80)	1.94M
CGP-CNN (ResSet)	19.33 (20.52 ± 1.36)	0.92M



(a) CGP-CNN (ConvSet) (b) CGP-CNN (ResSet)

Figure 6: The CNN architectures obtained by the proposed method on the small-data scenario.

We additionally conducted a Wilcoxon rank sum test for comparison of our models and other models. Based on the statistical test with the significance level of 5%, the p-value between ResSet and VGG (Simonyan and Zisserman, 2015) was 7.25×10^{-4}, and

the p-value between ResSet and ResNet (He et al., 2016) was 2.1×10^{-5}. The p-value for ConvSet compared to VGG and ResNet were 1.05×10^{-1} and 1.1×10^{-2}, respectively.

5 Conclusion

In this article, we proposed a CGP-based approach for designing deep CNN architectures and verified its potential. The proposed method generates the CNN architectures based on the CGP encoding scheme with highly functional modules and uses the modified evolutionary algorithm to efficiently find good architectures. Moreover, we introduced simple speed-up techniques, rich initialization and early termination of network training, to reduce the computational time. We constructed CNN architectures for the image classification task with the CIFAR-10 and CIFAR-100 datasets and considered two different data size settings. The experimental results showed that the proposed method could find competitive CNN architectures compared with the state-of-the-art models. Regarding the speed-up techniques, rich initialization can improve the discovered architecture performance, and early termination has succeeded to reduce the computational time without significant performance deterioration. By combining early termination with rich initialization, the computational time can be reduced, and the performance improved. In the small-data scenario, the proposed method can also find better and compact architectures compared with the existing hand-designed architectures.

The bottleneck of the architecture search of the DNN is the computational cost. Another possible speed-up technique is that we start with a small data size and increase the training data for the neural networks as the generation progresses. Moreover, to simplify and compact the CNN architectures, we may introduce regularization techniques to the architecture search process. Or, we may be able to manually simplify the obtained CNN architectures by removing redundant or less effective layers.

Another possible future topic would be to apply other evolutionary algorithms such as the standard genetic algorithm used in Real et al. (2017) to our proposed method. While we employed a simple evolution strategy in this article, we did not discuss how other evolutionary algorithms affect the performance on the architecture search. Thus, we would like to investigate this point in the future.

References

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2017). Designing neural network architectures using reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations*. Retrieved from arXiv:1611.02167.

Bergstra, J., and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305.

Bergstra, J., Yamins, D., and Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 115–123.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, 24, pp. 2546–2554.

Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pp. 3460–3468.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning*, pp. 1319–1327.

Harding, S. (2008). Evolution of image filters on graphics processor units using Cartesian genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 1921–1928.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778.

Hinton, G., Deng, L., Yu, D., Dahl, G., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4700–4708.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, pp. 507–523.

Ioffe, S., and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pp. 448–456.

Kingma, D. P., and Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*. Retrieved from arXiv:1412.6980.

Krizhevsky, A., and Hinton, G. E. (2009). *Learning multiple layers of features from tiny images*. Technical Report.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 25, pp. 1097–1105.

Kupyn, O., Budzan, V., Mykhailych, M., Mishkin, D., and Matas, J. (2018). DeblurGAN: Blind motion deblurring using conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8183–8192.

Larsson, G., Maire, M., and Shakhnarovich, G. (2017). FractalNet: Ultra-deep neural networks without residuals. In *Proceedings of the 5th International Conference on Learning Representations*. Retrieved from arXiv:1605.07648.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lin, M., Chen, Q., and Yan, S. (2014). Network in network. In *Proceedings of the 2nd International Conference on Learning Representations*. Retrieved from arXiv:1312.4400.

Loshchilov, I., and Hutter, F. (2016). CMA-ES for hyperparameter optimization of deep neural networks. In *Proceedings of the 4th International Conference on Learning Representations Workshop*. Retrieved from arXiv:1604.07269.

Miikkulainen, R., Liang, J. Z., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Hodjat, B. (2017). Evolving deep neural networks. Retrieved from arXiv:1703.00548.

Miller, J. F., and Smith, S. L. (2006). Redundancy and computational efficiency in Cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174.

Miller, J. F., and Thomson, P. (2000). Cartesian genetic programming. In *Proceedings of the European Conference on Genetic Programming*, pp. 121–132.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. In *Proceedings of Workshop on Deep Learning in Neural Information Processing Systems*. Retrieved from arXiv:1312.5602v1.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Morse, G., and Stanley, K. O. (2016). Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO)*, pp. 477–484.

Nair, V., and Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 807–814.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 2902–2911.

Schaffer, J. D., Whitley, D., and Eshelman, L. J. (1992). Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Proceedings of International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 1–37.

Simonyan, K., and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations*. Retrieved from arXiv:1409.1556v6.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 25, pp. 2951–2959.

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M. M. A., Prabhat, P., and Adams, R. P. (2015). Scalable Bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on Machine Learning*, pp. 2171–2180.

Stanley, K. O., D'Ambrosio, D. B., and Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2):185–212.

Stanley, K. O., and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Suganuma, M., Shirakawa, S., and Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017 (GECCO)*, pp. 497–504.

Sun, Y., Yen, G. G., and Yi, Z. (2018). Evolving unsupervised deep neural networks for learning meaningful representations. *IEEE Transactions on Evolutionary Computation*. Retrieved from doi:10.1109/TEVC.2018.2808699.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: A next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-Ninth Annual Conference on Neural Information Processing Systems*.

Verbancsics, P., and Harguess, J. (2013). Generative neuroevolution for deep learning. Retrieved from arXiv:1312.5355.

Verbancsics, P., and Harguess, J. (2015). Image classification using generative neuro evolution for deep learning. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision*, pp. 488–493.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3156–3164.

Xie, L., and Yuille, A. (2017). Genetic CNN. In *Proceedings of the International Conference on Computer Vision*, pp. 1388–1397.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Zagoruyko, S., and Komodakis, N. (2016). Wide residual networks. In *Proceedings of the British Machine Vision Conference*, pp. 87.1–87.12.

Zhang, R., Isola, P., and Efros, A. A. (2016). Colorful image colorization. In *Proceedings of the 14th European Conference on Computer Vision*, pp. 649–666.

Zoph, B., and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *Proceedings of the 5th International Conference on Learning Representations*. Retrieved from arXiv:1611.01578.