# A Genetic Programming Approach to Designing Convolutional Neural Network Architectures

1st Autor
Institution line
Street Adress
City, Country Post Code
emailadress

2nd Autor
Institution line
Street Adress
City, Country Post Code
emailadress

3rd Autor
Institution line
Street Adress
City, Country Post Code
emailadress

## ABSTRACT

The convolutional neural network (CNN), which is one of the deep learning models, has seen much success in a variety of computer vision tasks. However, designing CNN architectures still requires expert knowledge and a lot of trial and error. In this paper, we attempt to automatically construct CNN architectures for an image classification task based on Cartesian genetic programming (CGP). In our method, we adopt highly functional modules, such as convolutional blocks and tensor concatenation, as the node functions in CGP. The CNN structure and connectivity represented by the CGP encoding method are optimized to maximize the validation accuracy. To evaluate the proposed method, we constructed a CNN architecture for the image classification task with the CIFAR-10 dataset. The experimental result shows that the proposed method can be used to automatically find the competitive CNN architecture compared with state-of-the-art models.

## CCS CONCEPTS

•**Computing methodologies → Heuristic function construction; Neural networks;** *Computer vision problems;*

## KEYWORDS

genetic programming, convolutional neural network, designing neural network architectures, deep learning

**ACM Reference format:**
1st Autor, 2nd Autor, and 3rd Autor. 2017. A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference 2017, Berlin, Germany, July 15–19, 2017 (GECCO '17),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Deep learning, which uses deep neural networks as a model, has shown good performance on many challenging artificial intelligence and machine learning tasks, such as image recognition [17, 18], speech recognition [11], and reinforcement learning tasks [23, 24]. In particular, convolutional neural networks (CNNs) [18] have seen huge success in image recognition tasks in the past few years and are applied to various computer vision applications [37, 38]. A commonly used CNN architecture consists mostly of several convolutions, pooling, and fully connected layers. Several recent studies focus on developing a novel CNN architecture that achieves higher classification accuracy, e.g., GoogleNet [33], ResNet [10], and DensNet [12]. 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. State-of-the-art CNN architectures have become deep and complex, which suggests that a significant number of design parameters should be tuned to realize the best performance for a specific dataset. Therefore, trial-and-error or expert knowledge is required when users construct suitable architectures for their target datasets. In light of this situation, automatic design methods for CNN architectures are highly beneficial.

Neural network architecture design can be viewed as the model selection problem in machine learning. The straight-forward approach is to deal with architecture design as a hyperparameter optimization problem, optimizing hyperparameters, such as the number of layers and neurons, using black-box optimization techniques [20, 28].

Evolutionary computation has been traditionally applied to designing neural network architectures [26, 32]. There are two types of encoding schemes for network representation: direct and indirect coding. Direct coding represents the number and connectivity of neurons directly as the genotype, whereas indirect coding 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, e.g., convolution, pooling, and normalization. Optimizing so many parameters in a reasonable amount of computational time may be difficult. Therefore, the use of highly functional modules as a minimum unit is promising.

In this paper, we attempt to design CNN architectures based on genetic programming. We use the Cartesian genetic programming (CGP) [8, 21, 22] encoding scheme, one of the direct encoding schemes, to represent the CNN structure and connectivity. The advantage of this representation is its flexibility; it can represent variable-length network structures and skip connections. Moreover, we adopt relatively highly functional modules, such as convolutional blocks and tensor concatenation, as the node functions in CGP to reduce the search space. To evaluate the architecture represented by the CGP, we train the network using a training dataset in an ordinary way. Then, the performance of another validation dataset is assigned as the fitness of the architecture.

Based on this fitness evaluation, an evolutionary algorithm optimizes the CNN architectures. To check the performance of the proposed method, we conducted an experiment involving constructing a CNN architecture for the image classification task with the CIFAR-10 dataset [16]. The experimental result shows that the proposed method can be used to automatically find the competitive CNN architecture compared with state-of-the-art models.

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

## 2 RELATED WORK

This section briefly reviews the related work on automatic neural network architecture design: hyperparameter optimization, evolutionary neural networks, and the reinforcement learning approach.

### 2.1 Hyperparameter Optimization

We can consider neural network architecture design as the model selection or hyperparameter optimization problem from a machine learning perspective. There are many hyperparameter tuning methods for the machine learning algorithm, such as grid search, gradient search [2], random search [3], and Bayesian optimization-based methods [13, 28]. Naturally, evolutionary algorithms have also been applied to hyperparameter optimization problems [20]. In the machine learning community, Bayesian optimization is often used and has shown good performance in several datasets. 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 [28], which can easily handle the uncertainty and noise of the objective function. Bergstra et al. [5] have proposed the tree-structured Parzen estimator (TPE) and shown better results than manual search and random search. They have also proposed a meta-modeling approach [4] based on the TPE for supporting automatic hyperparameter optimization. Snoek et al. [29] used a deep neural network instead of the Gaussian process to reduce the computational cost for the surrogate model building and succeeded in improving the scalability.

The hyperparameter optimization approach often tunes predefined hyperparameters, such as the numbers of layers and neurons, and the type of activation functions. Although this method has seen success, it is hard to design more flexible architectures from scratch.

### 2.2 Evolutionary Neural Networks

Evolutionary algorithms have been used to optimize neural network architectures so far [26, 32]. Traditional approaches are not suitable for designing deep neural network architectures because they usually optimize the number and connectivity of low-level neurons.

Recently, Fernando et al. [6] proposed differentiable pattern-producing networks (DPPNs) for optimizing the weights of a denoising autoencoder. The DPPN is a differentiable version of the compositional pattern-producing networks (CPPNs) [30]. This paper focuses on the effectiveness of indirect coding for weight optimization. That is, the general structure of the network should be predefined.

Verbancsics et al. [35, 36] have optimized the weights of artificial neural networks and CNN by using the hypercube-based neuroevolution of augmenting topologies (HyperNEAT) [31]. However, to the best of our knowledge, the methods with HyperNEAT have not achieved competitive performance compared with the state-of-the-art methods. Also, these methods require an architecture that has been predefined by human experts. Thus, it is hard to design neural network architectures from scratch.

### 2.3 Reinforcement Learning Approach

Interesting approaches, including the automatic designing of the deep neural network architecture using reinforcement learning, were attempted recently [1, 39]. These studies showed that the reinforcement learning-based methods constructed the competitive CNN architectures for image classification tasks. In [39], 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 graphic processing units (GPUs) to accelerate the reinforcement learning process. Baker et al. [1] have proposed a meta-modeling approach based on reinforcement learning to produce CNN architectures. A Q-learning agent explores and exploits a space of model architectures with an $\epsilon-$greedy strategy and experience replay.

These approaches adopt the indirect coding scheme for the network representation, which optimizes generative rules for network architectures such as the RNN. Unlike these approaches, our approach uses direct coding based on Cartesian genetic programming to design the CNN architectures. In addition, we introduce relatively highly functional modules, such as convolutional blocks and tensor concatenations, to find better CNN architectures efficiently.

## 3 CNN ARCHITECTURE DESIGN USING CARTESIAN GENETIC PROGRAMMING

Our method directly encodes the CNN architectures based on CGP [8, 21, 22] and uses the highly functional modules as the node functions. The CNN architecture defined by CGP is trained using a training dataset, and the validation accuracy is assigned as the fitness of the architecture. Then, the architecture is optimized to maximize the validation accuracy by the evolutionary algorithm. Figure 1 illustrates an overview of our method.

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

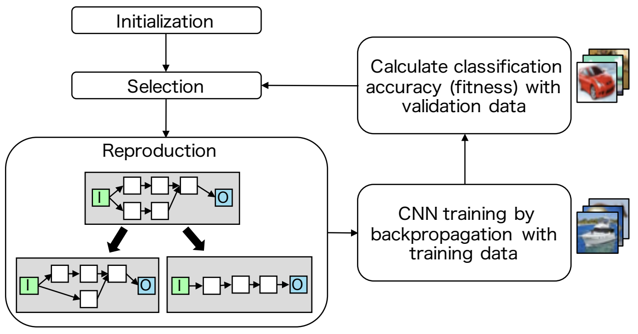

Initialization
Selection
Reproduction
I
O
Calculate classification accuracy (fitness) with validation data
CNN training by backpropagation with training data

**Figure 1: Overview of our method. Our method represents CNN architectures based on Cartesian genetic programming. The CNN architecture is trained on a learning task and assigned the validation accuracy of the trained model as the fitness. The evolutionary algorithm searches the better architectures.**

## 3.1 Representation of CNN Architectures

We use the CGP encoding scheme, representing the program as directed acyclic graphs with a two-dimensional grid defined on computational nodes, for the CNN architecture representation. 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 integers with fixed lengths, and each gene has information regarding the type and connections of the node. The $c$-th column's nodes should be connected from the $(c - l)$ to $(c - 1)$-th column's nodes, where $l$ is called the levels-back parameter. Figure 2 provides an example of the genotype, the corresponding network, and the CNN architecture in the case of two rows by three columns. Whereas the genotype in CGP is a fixed-length representation, the number of nodes in the phenotypic network varies because not all of the nodes are connected to the output nodes. Node No. 5 on the left side of Fig. 2 is an inactive node.

Referring to the modern CNN architectures, we select the highly functional modules as the node function. The frequently used processings in the CNN are convolution and pooling; convolution processing uses a local connectivity and spatially shares the learnable weights, and pooling is nonlinear down-sampling. We prepare the six types of node functions called ConvBlock, ResBlock, max pooling, average pooling, concatenation, and summation. These nodes operate the three-dimensional (3-D) tensor defined by the dimensions of the row, column, and channel. Also, we call this 3-D tensor feature maps, where a feature map indicates a matrix of the row and column as an image plane.

The ConvBlock consists of standard convolution processing with a stride of 1 followed by batch normalization [14] and rectified linear units (ReLU) [25]. In the ConvBlock, we pad the outside of input feature maps with zero values before the convolution operation so as to maintain the row and column sizes of the output. As a result, the $M \times N \times C$ input feature maps are transformed into $M \times N \times C'$ output ones, where $M$, $N$, $C$, and $C'$ are the numbers of rows, columns, input channels, and output channels, respectively. We prepare several ConvBlocks with the different output channels and the receptive field size (kernel size) in the function set of CGP.

The ResBlock is composed of a convolution processing, batch normalization, ReLU, and tensor summation. A ResBlock architecture is shown in Fig. 3. The ResBlock performs identity mapping by shortcut connections as described in [10]. The row and column sizes of the input are preserved in the same way as ConvBlock after convolution. The output feature maps of the ResBlock are calculated by the ReLU activation and the summation with the input feature maps and the processed feature maps as shown in Fig. 3. In the ResBlock, the $M \times N \times C$ input feature maps are transformed into the $M \times N \times C'$ output ones. We also prepare several ResBlocks with the different output channels and the receptive field size in the function set of CGP.

The max and average poolings perform a max and average operation, respectively, over the local neighbors of the feature maps. We use the pooling with the $2 \times 2$ receptive field size and the stride size 2. In the pooling operation, the $M \times N \times C$ input feature maps are transformed into the $M' \times N' \times C$ output ones, where $M' = \lfloor M/2 \rfloor$ and $N' = \lfloor N/2 \rfloor$.

The concatenation function concatenates two feature maps in the channel dimension. If the input feature maps to be concatenated have different numbers of rows or columns, we down-sample the larger feature maps by max pooling so that they become the same sizes of the inputs. In the concatenation operation, the sizes of the output feature maps are $\min(M_1, M_2) \times \min(N_1, N_2) \times (C_1 + C_2)$, where as the sizes of the inputs are $M_1 \times N_1 \times C_1$ and $M_2 \times N_2 \times C_2$.

The summation performs the element-wise addition of two feature maps, channel by channel. In the same way as the concatenation, if the input feature maps to be added have different numbers of rows or columns, we down-sample the larger feature maps by max pooling. In addition, if the inputs have different numbers of channels, we pad the smaller feature maps with zeros for increasing channels. In the summation operation, the size of the output feature maps are $\min(M_1, M_2) \times \min(N_1, N_2) \times \max(C_1, C_2)$, where the sizes of the inputs are $M_1 \times N_1 \times C_1$ and $M_2 \times N_2 \times C_2$. In Fig. 2, the summation node performs max pooling to the first input so as to get the same input tensor sizes. Adding these summation and concatenation operations allows our method to represent shortcut connections or branching layers, such as those used in GoogleNet [33] and Residual Net [10] without ResBlock.

The output node represents the softmax function with the number of 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

We use a point mutation as the genetic operator in the same way as the standard CGP. The type and connections of each node randomly change to valid values according to a mutation rate. The standard CGP mutation has the possibility of affecting only the inactive node. In that case, the phenotype (representing the CNN architecture) does not change by the mutation and does not require a fitness evaluation again.

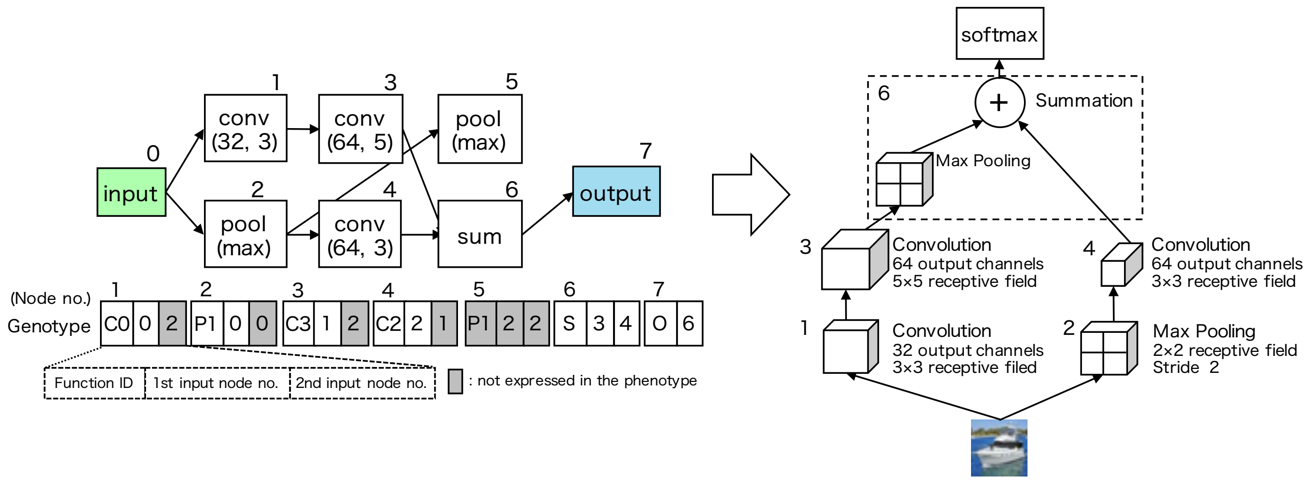


**Figure 2: Example of a genotype and a phenotype. The genotype (left) defines the CNN architecture (right). In this case, node No. 5 on the left side is an inactive node. The summation node performs max pooling to the first input so as to get the same input tensor sizes.**

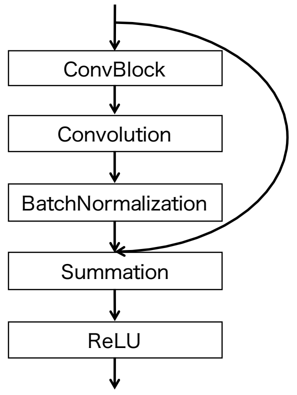


**Figure 3: ResBlock architecture.**

The fitness evaluation of the CNN architectures is so expensive because it requires the training of CNN. To efficiently use the computational resource, we want to evaluate some candidate solutions in parallel at each generation. Therefore, we apply the mutation operator until at least one active node changes for reproducing the candidate solution. We call this mutation a forced mutation. Moreover, to maintain a neutral drift, which is effective for CGP evolution [21, 22], we modify a parent by the neutral mutation if the fitnesses of the offsprings do not improve. Here, the neutral mutation changes only the genes of the inactive nodes without the modification of the phenotype.

We use the modified $(1 + \lambda)$ evolutionary strategy (with $\lambda = 2$ in our experiments) in the above artifice. The procedure of our modified algorithm is as follows:

(1) Generate an initial individual at random as parent $P$, and train the CNN represented by $P$ followed by assigning the validation accuracy as the fitness.

**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 | – |
| Summation | Sum | – |
| Concatenation | Concat | – |

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

(2) Generate a set of $\lambda$ offsprings $C$ by applying the forced mutation to $P$.
(3) Train the $\lambda$ CNNs represented by offsprings $C$ in parallel, and assign the validation accuracies as the fitness.
(4) Apply the neutral mutation to parent $P$.
(5) Select an elite individual from the set of $P$ and $C$, and then replace $P$ with the elite individual.
(6) Return to step 2 until a stopping criterion is satisfied.

## 4 EXPERIMENTS AND RESULTS

### 4.1 Dataset

We test our method on the image classification task using the CIFAR-10 dataset in which the number of classes is 10. The numbers of training and test images are 50,000 and 10,000, respectively, and the size of images is $32 \times 32$.

We consider two experimental scenarios: the default scenario and the small-data scenario. The default scenario uses the default

**Table 2: Parameter setting for the CGP**

| Parameters | Values |
|---|---|
| Mutation rate | 0.05 |
| # Rows ($N_r$) | 5 |
| # Columns ($N_c$) | 30 |
| Levels-back ($l$) | 10 |

numbers of the training images, whereas the small-data scenario assumes that we use only 5, 000 images as the learning data.

In the default scenario, we randomly sample 45, 000 images from the training set to train the CNN, and we use the remaining 5, 000 images for the validation set of the CGP fitness evaluation. In the small-data scenario, we randomly sample 4, 500 images for the training and 500 images for the validation.

## 4.2 Experimental Setting

To assign the fitness to the candidate CNN architectures, 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 He's method [9] and use the Adam optimizer [15] with an initial learning rate of 0.01. We train each CNN for 50 epochs and reduce the learning rate by a factor of 10 at 30th epoch.

We preprocess the data with the per-pixel mean subtraction. To prevent overfitting, we use a weight decay with the coefficient $1.0 \times 10^{-4}$ and data augmentation. We use the data augmentation method based on [10]: padding 4 pixels on each side followed by choosing a random $32 \times 32$ crop from the padded image, and random horizontal flips on the cropped $32 \times 32$ image.

The parameter setting for CGP is shown in Table 2. We use the relatively larger number of columns than the number of rows to generate deep architectures that are likely. The offspring size of $\lambda$ is set to 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 we adopt ConvBlock or ResBlock. The numbers of generations are 500 for ConvSet, 300 for ResSet in the default scenario, and 1, 500 in the small-data scenario, respectively.

After the CGP process, we re-train the best CNN architecture using each training image (50, 000 for the default scenario and 5, 000 for the small-data scenario), and we calculate the classification accuracy for the 10, 000 test images to evaluate the constructed CNN architectures.

In this re-training phase, we optimize the weights 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 $5.0 \times 10^{-4}$. We start a learning rate of 0.01 and set it to 0.1 at 5th epoch, then we reduce it to 0.01 and 0.001 at 250th and 375th epochs, respectively. This learning rate schedule is based on the reference [10].

We have implemented our methods using the Chainer [34] (version 1.16.0) framework and run the experiments on the machines with 3.2GHz CPU, 32GB RAM, and two NVIDIA GeForce GTX 1080 (or two GTX 1070) GPUs. It is possible that the large architectures generated by the CGP process cannot run in the environment due to the GPU memory limitation. In that case, we assign a zero fitness to the candidate solution.

**Table 3: Comparison of error rates on the CIFAR-10 dataset (default scenario). The values of Maxout, Network in Network, ResNet, MetaQNN, and Neural Architecture Search are referred from the reference papers.**

| Model | Error rate | # params ($\times 10^6$) |
|---|---|---|
| Maxout [7] | 9.38 | – |
| Network in Network [19] | 8.81 | – |
| VGG [27] [1] | 7.94 | 15.2 |
| ResNet [10] | 6.61 | 1.7 |
| MetaQNN [1] [2] | 9.09 | 3.7 |
| Neural Architecture Search [39] | 3.84 | 32.0 |
| CGP-CNN (ConvSet) | 6.75 | 1.52 |
| CGP-CNN (ResSet) | 5.98 | 1.68 |

## 4.3 Result of the Default Scenario

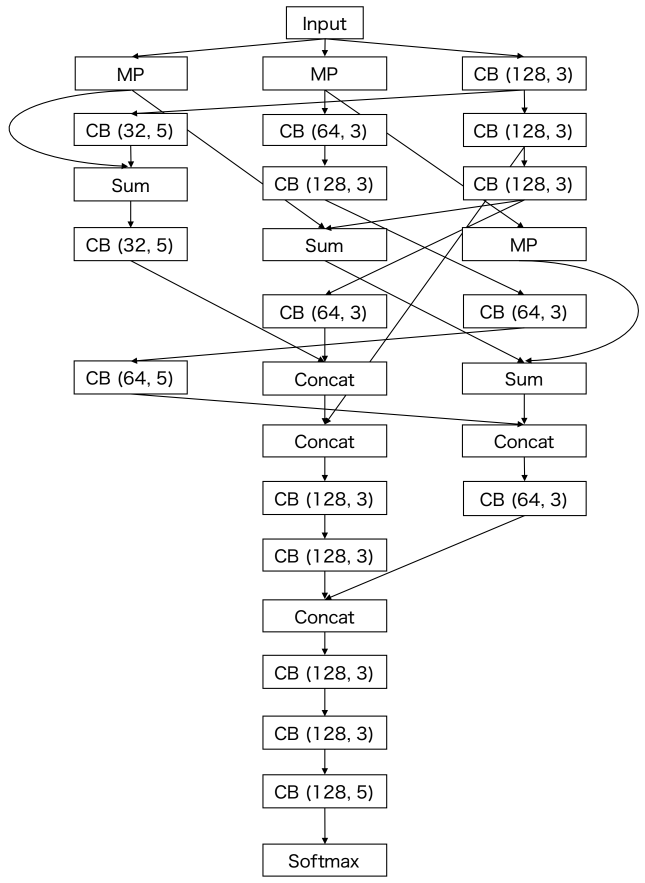
We compare the classification performance of our method with the state-of-the-art methods and summarize the classification error rates in Table 3. We refer to the architectures constructed by the proposed method as CGP-CNN. For instance, CGP-CNN (ConvSet) means the proposed method with the node function set of ConvSet. The models, Maxout, Network in Network, VGG, and ResNet, are hand-crafted CNN architectures, whereas the MetaQNN and Neural Architecture Search are the models constructed by the reinforcement learning-based method. In Table 3, the numbers of learnable weight parameters in the models are also listed.

As can be seen in Table 3, the error rates of our methods are competitive with the state-of-the-art methods. In particular, CGP-CNN (ResSet) outperforms all hand-crafted models, and the architectures constructed by using our method have a good balance between classification errors and the number of parameters. The Neural Architecture Search achieved the best error rate, but this method used 800 GPUs for the architecture search. Our method could find a competitive architecture with a reasonable machine resource.

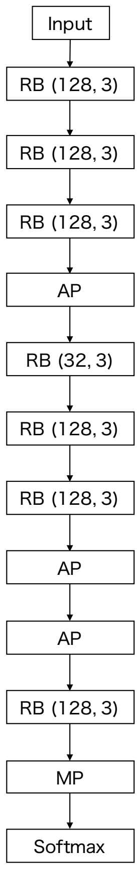
Figure 4 shows the architectures constructed by CGP-CNN (ConvSet) and CGP-CNN (ResSet). We can observe that these architectures are quite different; the summation and concatenation nodes are not used in CGP-CNN (ResSet), whereas these nodes are frequently used in CGP-CNN (ConvSet). These nodes lead the wide network; therefore, the network of CGP-CNN (ConvSet) is a wider structure than that of CGP-CNN (ResSet).

Added to this, we observe that CGP-CNN (ResSet) architecture has a similar feature with the ResNet [10]. The ResNet consists of a repetition of two types of modules: the module with several convolutions with the shortcut connections without down-sampling,

[1]We have implemented the VGG net [27] for the CIFAR-10 dataset because the VGG net is not applied to the CIFAR-10 dataset in [27]. The architecture of the VGG is identical with configuration D in [27]. We denote this model as VGG in this paper.
[2]The mean error rate and the number of parameters of the top five models are shown.



(a) CGP-CNN (ConvSet)



(b) CGP-CNN (ResSet)

**Figure 4: The CNN architectures designed by our method on the default scenario.**

and down-sampling convolution with a stride of 2. Although our method cannot perform down-sampling in the ConvBlock and the ResBlock, we can see from Fig. 4 that CGP-CNN (ResSet) uses average pooling as an alternative to the down-sampling convolution. Furthermore, CGP-CNN (ResSet) has some convolutions with the shortcut connections, such as ResNet. Based on these observations, we can say that our method can also find the architecture similar to one designed by human experts, and that model shows a better performance.

Besides, while the ResNet has a very deep 110-layer architecture, CGP-CNN (ResSet) has a relatively shallow and wide architecture. We guess from this result that the number of output channels of ResBlock in the proposed method is one of the contributive parameters for improving the classification accuracy on the CIFAR-10 dataset.

## 4.4 Result of the Small-data Scenario

In the small-data scenario, we compare our method with VGG and ResNet. We have trained VGG and ResNet models by the same setting of the re-training method in the proposed method; it is based on the training method of the ResNet [10].

Table 4 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. It is obvious that VGG and ResNet are inadequate for the small-data scenario because these architectures are designed for a relatively large amount of data. Meanwhile, our method can tune the architecture depending on the data size. Figure 5 illustrates the CGP-CNN (ConvSet) architecture constructed by using the proposed method. As seen in Fig. 5, our method has found a wider structure than that in the default scenario.

Additionally, we have re-trained this model with the 50,000 training data and achieved a 8.05% error rate on the test data. It suggests that the proposed method may be used to design a relatively good general architecture even with a small dataset.

**Table 4: Comparison of error rates on the CIFAR-10 dataset (small-data scenario).**

| Model | Error rate | # params ($\times 10^6$) |
|---|---|---|
| VGG [27] | 24.11 | 15.2 |
| ResNet [10] | 24.10 | 1.7 |
| CGP-CNN (ConvSet) | 23.48 | 3.9 |
| CGP-CNN (ResSet) | 23.47 | 0.83 |

## 5 CONCLUSION

In this paper, we have attempted to take a GP-based approach for designing the CNN architectures and have verified its potential. The proposed method constructs the CNN architectures based on CGP and adopts the highly functional modules, such as ConvBlock and ResBlock, for searching the adequate architectures efficiently. We have constructed the CNN architecture for the image classification task with the CIFAR-10 dataset and considered two different data size settings. The experimental result showed that the proposed method could automatically find the competitive CNN architecture compared with the state-of-the-art models.

However, our proposed method requires much computational cost; the experiment on the default scenario needed about a few weeks in our machine resource. We can reduce the computational time if the training data are small (such as in the small-data scenario in the experiment). Thus, one direction of future work is to develop the evolutionary algorithm to reduce the computational cost of the architecture design, e.g., increasing the training data for the neural network as the generation progresses. Another future work is to apply the proposed method to other image datasets and tasks.

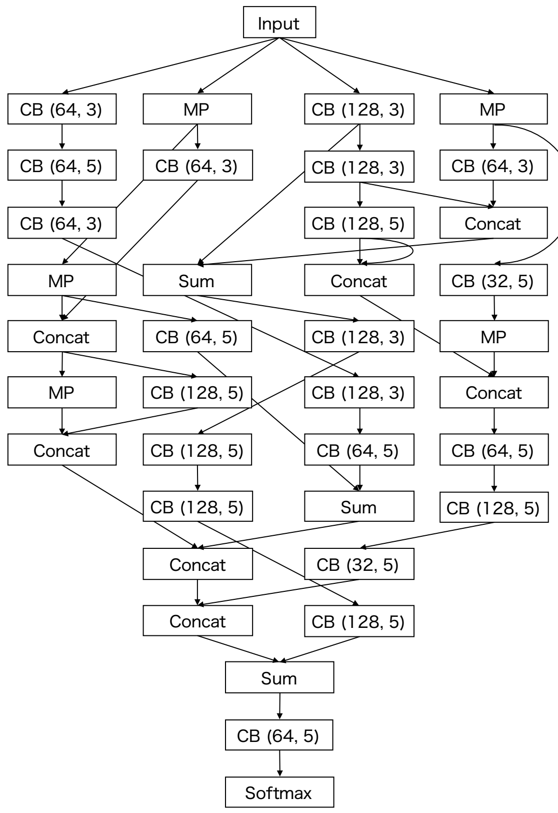


**Figure 5: The architecture of CGP-CNN (ConvSet) constructed in the small-data scenario.**

## REFERENCES

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2016. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167* (2016).

[2] Yoshua Bengio. 2000. Gradient-based optimization of hyperparameters. *Neural computation* 12, 8 (Aug. 2000), 1889–1900.

[3] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, 1 (Jan. 2012), 281–305.

[4] James Bergstra, Daniel Yamins, and David D. Cox. 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 (ICML '13)*. Atlanta, Gerorgia, 115–123.

[5] James S. Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems 24 (NIPS '11)*. Granada, Spain, 2546–2554.

[6] Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. 2016. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016 (GECCO '16)*. Denver, Colorado, 109–116.

[7] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. 2013. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*. Atlanta, Gerorgia, 1319–1327.

[8] Simon Harding. 2008. Evolution of image filters on graphics processor units using cartesian genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '08)*. Hong Kong, China, 1921–1928.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV '15)*. Santiago, Chile, 1026–1034.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*. Las Vegas, Nevada, 770–778.

[11] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and others. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (Nov. 2012), 82–97.

[12] Gao Huang, Zhuang Liu, Kilian Q. Weinberger, and Laurens van der Maaten. 2016. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993* (2016).

[13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*. Rome, Italy, 507–523.

[14] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML '15)*. Lille, France, 448–456.

[15] Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representaions (ICLR '15)*. San Diego, 2452–2459.

[16] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images. *Technical report* (2009).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS '12)*. Nevada, 1097–1105.

[18] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov. 1998), 2278–2324.

[19] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network in network. In *Proceedings of the 2nd International Conference on Learning Representaions (ICLR '14)*. Banff, Canada.

[20] Ilya Loshchilov and Frank Hutter. 2016. CMA-ES for hyperparameter optimization of deep neural networks. *arXiv preprint arXiv:1604.07269* (2016).

[21] Julian F. Miller and Stephen L. Smith. 2006. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation* 10, 2 (April 2006), 167–174.

[22] Julian F. Miller and Peter Thomson. 2000. Cartesian genetic programming. In *Proceedings of the European Conference on Genetic Programming (EuroGP '00)*. Scotland, UK, 121–132.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. In *Proceedings of Neural Information Processing Systems (NIPS '13) Workshop on Deep Learning*. Nevada.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, and others. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533.

[25] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML '10)*. Haifa, Israel, 807–814.

[26] J. David Schaffer, Darrell Whitley, and Larry J. Eshelman. 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 (COGANN '92)*. Baltimore, Maryland, 1–37.

[27] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[28] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25 (NIPS '12)*. Nevada, 2951–2959.

[29] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Mr Prabhat, and Ryan P. Adams. 2015. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML '15)*. Lille, France, 2171–2180.

[30] Kenneth O. Stanley. 2007. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines* 8, 2 (June 2007), 131–162.

[31] Kenneth O. Stanley, David B. D'Ambrosio, and Jason Gauci. 2009. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life* 15, 2 (2009), 185–212.

[32] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[33] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '15)*. Boston, Massachusetts, 1–9.

[34] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Neural Information Processing Systems (NIPS '15) Workshop on Machine Learning Systems (LearningSys)*. Montreal, Canada.

[35] Phillip Verbancsics and Josh Harguess. 2013. Generative neuroevolution for deep learning. *arXiv preprint arXiv:1312.5355* (2013).

[36] Phillip Verbancsics and Josh Harguess. 2015. Image classification using generative neuro evolution for deep learning. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision (WACV '15)*. Hawaii, 488–493.

[37] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. In *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR '15)*. Boston, Massachusetts, 3156–3164.

[38] Richard Zhang, Phillip Isola, and Alexei A. Efros. 2016. Colorful image colorization. In *Proceedings of the 14th European Conference on Computer Vision (ECCV '16)*. Amsterdam, The Netherlands, 649–666.

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