# Neural Architecture Search Based on Differential Evolution for Image Classification Problems

## Deep Learning Final Project Report

**Kaiyu Pei(kp2690), Keng-Ming Chang(kc4536), Jincheng Tian(jt4434)**

**GitHub Link**

https://github.com/gamingzhang/ece7123_final_project

## Introduction

Convolutional neural networks (CNNs) [1] have achieved great success in Computer Vision. Recently, they have been widely used in our life, such as face recognition and vehicle detection. However, the most famous convolutional neural networks were handcrafted from scratches, such as AlexNet [2], VGGNet [3], and ResNet [4]. This approach often relies on extensive statistics and machine learning knowledge and costs a lot of time.

On the other hand, convolutional neural networks become more and more complex as the technology develops. The more extensive networks imply more significant training time and computational resources. Therefore, in recent years, automation of this task has been considered [5].

Also, one of the trends today is the porting of neural networks to the mobile device, where computing resources are limited. Thus, how to design an efficient and lightweight convolutional neural network automatically is a multi-objective optimization problem.

In our final project, we propose a novel algorithm based on differential evolution to automatically design an effective convolutional neural network to solve image classification problems. Our experimental results show that our algorithm can quickly find a fantastic convolutional neural network architecture with high accuracy and low parameters.

## Overview

The framework consists of three basic steps to search for the optimal convolutional nerve network structure:

- initialization of the population
- mutations in individuals within populations
- natural selection of the population

Before starting, we must determine an efficient way to express the structure of the convolutional neural network. Since general CNN networks are stacked by layers, we can represent a layer with a single unit.

We use a sequence to represent a CNN architecture. Typically, a CNN has four kinds of layers: convolutional layer, max pooling layer, average pooling layer, and fully connected layer. Thus, we use four types of units to represent each of the four layers. Figure 1 illustrates a sequence corresponding to a convolutional neural network. 'C' represents the convolutional layer. 'MP' represents the max pooling layer, ''AP' represents the average pooling layer, and 'FC' represents the fully connected layer.

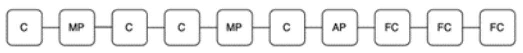

Figure 1

What's more, our optimization goals are the classification performance and the number of parameters of the model. We want to find a model with high classification accuracy and a low number of parameters.

## Algorithm design

### 1. Initialization

Initialization is the first step of the evolutionary process and is performed by the function Initialization(). This step randomly creates multiple CNN.

According to the typical CNN architecture, the fully connected layers form the last few layers in the network because they are used to classify the features extracted from the convolutional or pooling layers. Therefore, we divide a CNN sequence into two parts: Part1 and Part2. Part1 contains the convolutional and pooling layers, while Part2 only contains the fully connected layers. However, due to our limited computational resources, we need to add some restrictions to our CNN models, such as the maximum length of Part1, and the maximum number of convolutional kernels in the convolutional layer.

When creating an individual, we first add convolutional layers as first layer and then add convolutional and pooling layers at random until the number of layers reaches the limitation. We set a probability of adding a convolutional layer or a pooling layer. The ReLU is added next to the convolutional layer if the convolutional layer is added. If the pooling layer is added, we also set a probability of adding the max pooling layer of an average pooling layer. Next, we construct Part2. For Part2, we mimic the structure of AlexNet. In addition to the fully connected layer, we also have ReLU and Dropout layer [6] in Part2. The Dropout layer can temporarily remove the neurons with a certain probability during backpropagation to effectively prevent the occurrence of overfitting.

When we add a convolutional layer, we need to specify some parameters, which are in*_channels*, out*_channels*, *kernel_size*, and padding. For *in_channels*, it is the same as the *out_channels* of the last convolutional layer. For the first convolutional layer, it is 3. For the *out_channels*, as mentioned above, we randomly chose a value. For the *kernel_size*, we provide four types: 1x1, 3x3, 5x5, and 7x7. If padding is not used, the image size will be slightly reduced, and the edge information of the image will be lost too quickly during the continuous convolution process. Therefore, we set padding to 0, 1, 2, and 3, respectively, corresponding to *kernel_size*.

When we add a pooling layer, we also need to specify some parameters, which are *kernel_size* and *stride*. We set *kernel_size* = 2 and *stride* = 2 so that the image size is reduced to half of the original size after each pooling layer. The dataset's image size determines the maximum number of pooling layers we can add. For example, if we use CIFAR10 (3x32x32) as our dataset, there are at most four pooling layers in the CNN. After four pooling layers, the image size will be 2x2.

After constructing a CNN architecture, we need to calculate its performance on our dataset and its parameters. Due to our limited computational resources, we only train ten epochs for each model.

**2. Mutation**

This step is the core of our algorithm. In differential evolution [7], the variation of individuals is performed as a differential operation. We choose the following strategy in figure 2.

$$v_i(g+1) = x_{best}(g) + F \cdot (x_{r2}(g) - x_{r3}(g))$$

$$i \neq r_1 \neq r_2 \neq r_3$$

Figure 2

In this algorithm, $x_{best}\ (g)$ is chosen randomly from the top performer of the previous population, and $x_{r2}\ (g)$ and $x_{r3}\ (g)$ are randomly selected from the previous population.

Next, we define a way to calculate the difference between two convolutional neural networks.

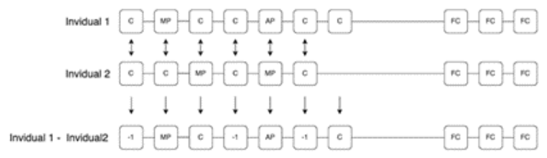

Figure 3

Based on the chart in figure 3, We created two convolutional neural networks, Individual 1 and Individual 2. Let's call them P1 and P2, respectively. There are the following rules:

1. If P1 and P2 are the same, then NEW takes the same value.
2. If P1 has value and P2 is none, or P1 is none and P2 has value, then NEW takes the value of whom is not none.
3. If P1 and P2 have different values, then NEW takes the value of P1

After we have the difference between two individuals, we add the difference to the best individual through the following rules based on figure 4.

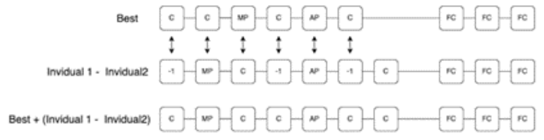

Figure 4

1. We call Individual 1 - Individual 2 as d1. If d1 = -1, then NEW takes the value of the Best
2. If d1 and best are the same, then NEW takes the same value.
3. If d1 and best are different, then NEW takes the value of d1

In this way, we create m new individuals, where $m = n/2$ in our algorithm. Then we add these $m$ individuals to the previous population.

**3. Natural selection**

The next generation of the population will be selected from the merged population. Since the two features involved in this algorithm (accuracy and parameters) have different magnitudes, we need to normalize them. We use min-max normalization with the following formula: $x$ is the normalized value, $x^*$is the original value, and $min$, $max$ are the minimal and maximal values of the sample in this feature, respectively.

$$x^* = \frac{x - min}{max - min}$$

Next, we set a weight to calculate a score for each individual in the population. For our algorithm, score = 0.7 * error rate + 0.3 * parameters. Then we sort the individuals according to their score and keep the top $n$ individuals as the next generation.

## 4. Best CNN architecture

After iteration, we could find the best individual from the last generation.

## Result

### 1. Hyper-parameters

Table 1 shows the hyper-parameters in our algorithm, and Table 2 shows the hyper-parameters of the training.

| | |
|---|---|
| Population size | 20 |
| Numbers of generations | 9 |
| Initial Epochs | 10 |
| Minimal size of part 1 | 7 |
| Maximum size of part 1 | 15 |
| Maximum kernel of convolutional layer | 256 |
| Probability of adding convolutional layer | 0.5 |
| Probability of adding max pooling layer | 0.8 |
| Probability of adding average pooling layer | 0.2 |
| Numbers of new individuals after mutation | 10 |
| Evaluation score | 0.7 * error rate + 0.3 * numbers of parameters |

Table 1 hyperparameters of algorithm

| | |
|---|---|
| Epochs for fully training | 200 |
| Batch size | 64 |
| Loss function | CrossEntropyLoss |
| Optimizer | SGD |
| Learning rate | 0.01 |
| Data Augmentation | RandomCrop RandomHorizontalFlip |

Table 2 hyperparameters of training

### 2. Dataset

We test our algorithm in CIFAR-10 dataset. The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. Each image is one of the following 10 categories: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The images in CIFAR-10 are shown in figure 5.

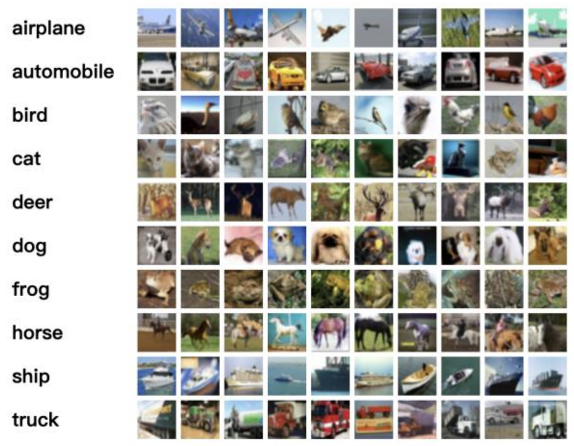

Figure 5

### 3. Results

We randomly generate 20 individuals as the first population. After 9 generations of evolution, we can see that the population evolves towards a direction with fewer parameters and a lower classification error rate (See figure 6). We choose two models in the 9th population as our final models: Model 1 has the best accuracy (After 10 training epochs) in the population, and Model 2 has the lowest number of parameters. After 200 epochs of training, Model 1 achieved 92.14% accuracy, and Model 2 achieved 88.53% accuracy.

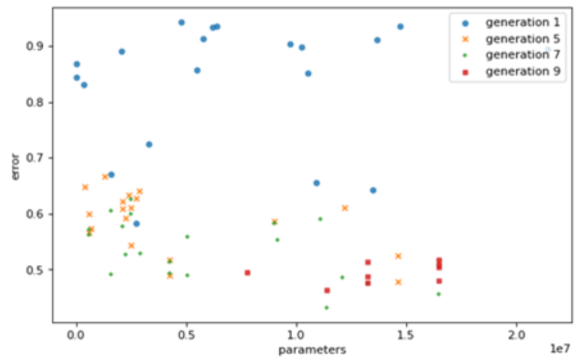

Figure 6

Table 3 shows a comparison between our models and some famous CNN models.

| model | Accuracy on CIFAR-10 | Number of parameters |
|---|---|---|
| AlexNet | 86.5% | 62.3 million |
| VGG16 | 92.64% | 138 million |
| Our Model 1 | 92.14% | 16.4 million |
| Our Model 2 | 88.53% | 7.79 million |

Table 3

Figure 7 shows the loss/accuracy curves of Model 1.

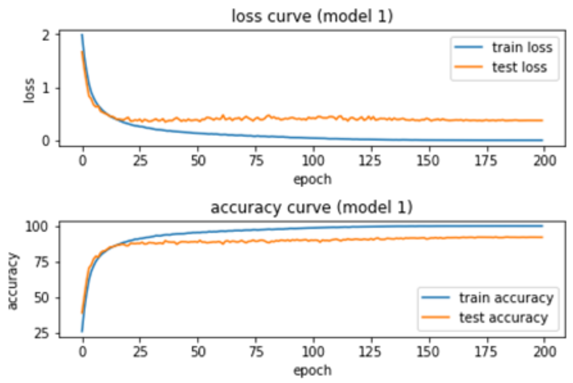

Figure 7

Figure 8 shows the architecture of Model 1.

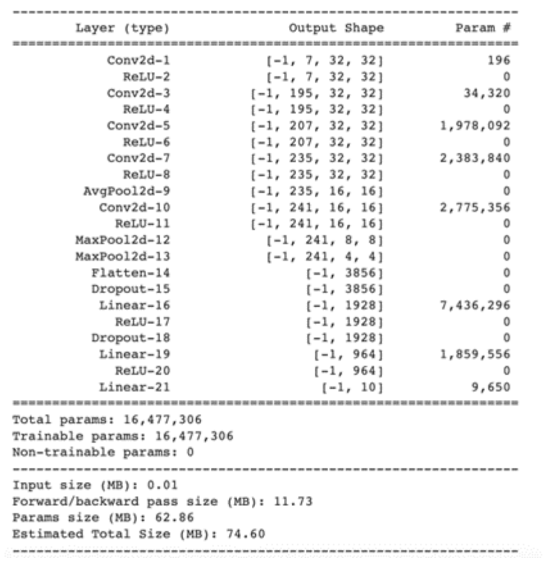
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 7, 32, 32]             196
              ReLU-2            [-1, 7, 32, 32]               0
            Conv2d-3          [-1, 195, 32, 32]          34,320
              ReLU-4          [-1, 195, 32, 32]               0
            Conv2d-5          [-1, 207, 32, 32]       1,978,092
              ReLU-6          [-1, 207, 32, 32]               0
            Conv2d-7          [-1, 235, 32, 32]       2,383,840
              ReLU-8          [-1, 235, 32, 32]               0
         AvgPool2d-9          [-1, 235, 16, 16]               0
           Conv2d-10          [-1, 241, 16, 16]       2,775,356
             ReLU-11          [-1, 241, 16, 16]               0
        MaxPool2d-12            [-1, 241, 8, 8]               0
        MaxPool2d-13            [-1, 241, 4, 4]               0
          Flatten-14                 [-1, 3856]               0
          Dropout-15                 [-1, 3856]               0
           Linear-16                 [-1, 1928]       7,436,296
             ReLU-17                 [-1, 1928]               0
          Dropout-18                 [-1, 1928]               0
           Linear-19                  [-1, 964]       1,859,556
             ReLU-20                  [-1, 964]               0
           Linear-21                   [-1, 10]           9,650
================================================================
Total params: 16,477,306
Trainable params: 16,477,306
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 11.73
Params size (MB): 62.86
Estimated Total Size (MB): 74.60
----------------------------------------------------------------
```

Figure 8

Figure 9 shows the loss/accuracy curves of Model 2.

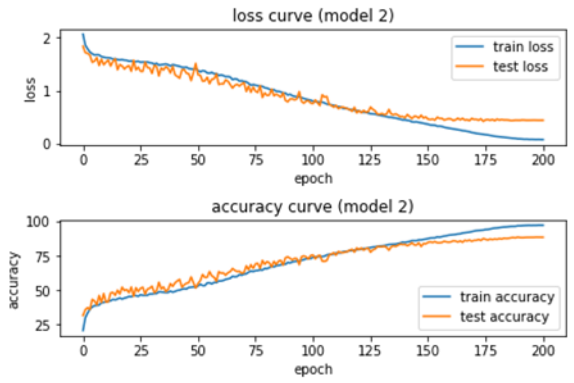

Figure 9

Figure 10 shows the architecture of Model 2.

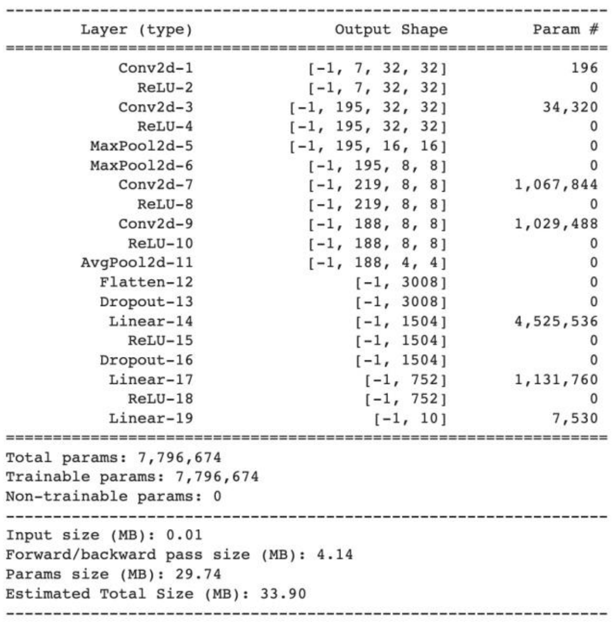
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 7, 32, 32]             196
              ReLU-2            [-1, 7, 32, 32]               0
            Conv2d-3          [-1, 195, 32, 32]          34,320
              ReLU-4          [-1, 195, 32, 32]               0
         MaxPool2d-5          [-1, 195, 16, 16]               0
         MaxPool2d-6            [-1, 195, 8, 8]               0
            Conv2d-7            [-1, 219, 8, 8]       1,067,844
              ReLU-8            [-1, 219, 8, 8]               0
            Conv2d-9            [-1, 188, 8, 8]       1,029,488
             ReLU-10            [-1, 188, 8, 8]               0
        AvgPool2d-11            [-1, 188, 4, 4]               0
          Flatten-12                 [-1, 3008]               0
          Dropout-13                 [-1, 3008]               0
           Linear-14                 [-1, 1504]       4,525,536
             ReLU-15                 [-1, 1504]               0
          Dropout-16                 [-1, 1504]               0
           Linear-17                  [-1, 752]       1,131,760
             ReLU-18                  [-1, 752]               0
           Linear-19                   [-1, 10]           7,530
================================================================
Total params: 7,796,674
Trainable params: 7,796,674
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 4.14
Params size (MB): 29.74
Estimated Total Size (MB): 33.90
----------------------------------------------------------------
```

Figure 10

## Conclusion

Since our model only contains convolutional, pooling, and fully connected layers, it is a basic CNN model without using some techniques like skip connection and parallel layers. Thus, we only choose other basic CNN models like AlexNet and VGG as a comparison.

As shown in Table 3, our best model can achieve 92.14% accuracy with 16.4 million parameters. As a comparison, the famous AlexNet has 86.5% accuracy on CIFAR-10 with 62.3 million parameters. So, our algorithm successfully designed an efficient and lightweight CNN model.

Our algorithm still has some room to improve. For example, our algorithms are not actually automatic because we need to set some limitations due to the limited computational resources. Also, since it is random initialization, the performances of the first generation are not good, so it takes many generations for them to become good architectures. We may use some existing CNN models as our first population, which could improve the algorithm efficiency.

## References

[1] Y. LeCun et al., "Backpropagation Applied to Handwritten Zip Code Recognition," in Neural Computation, vol. 1, no.4,pp.541-551,Dec.1989, doi: 10.1162/neco.1989.1.4.541.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks.

[3] Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, https://arxiv.org/abs/1409.1556

[4] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.

[5] Elsken T, Metzen J H, Hutter F. Simple and efficient architecture search for convolutional neural networks[J]. arXiv preprint arXiv:1711.04528, 2017.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15, 1 (January 2014), 1929–1958.
https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[7] Das, Swagatam & Suganthan, Ponnuthurai. (2011). Differential Evolution: A Survey of the State-of-the-Art.. IEEE Trans. Evolutionary Computation. 15. 4-31.
https://www.researchgate.net/publication/220380793_Differential_Evolution_A_Survey_of_the_State-of-the-Art