Genetic Alogrithm for CNN Hyperparametr Optimization

Tianrui Guo

Department of Applied Mathematics

April 20, 2019

Abstract

In this big data era, we developed many efficient and effective ways to analyze our database. Among various techniques, computer vision is a fancy and important one. One of the most significant technique in modern computer vision is convolutional neural network which is a typical method in deep learning. When we process image classification or image recognition, we inevitably encounter a problem of determine the hyperparamters such as the number of Conv channels, the size of filters or the dimension of the dense layers. The process of finding a set of hyperparameters which can improve the performance of convolutional neural network is known as hyperparameter optimization, and it can be also called as tuning. In this Capstone project we are using genetic algorithm for finding an optimal CNN architecture. When the convolutional neural network goes deeper and more complicated, the number of hyperparameters increases dramatically. In order to increase the performance of tuning, we no longer use manual tuning methods like grid search, instead we use genetic algorithm which is a kind of random search method for tuning, because we want the optimization algorithm searching automatically in a larger hyperparamter space. To test the performance of genetic tuning, we implement genetic algorithm on MNIST which is a relative small data set include 60,000 examples in training set and 10,000 examples in test set. If this algorithm performs well in this dataset, we will have possibility to implement it on other significative datasets.

1 Introduction

Machine learning is the cutting edge technique in modern data science, and it is dominated by deep neural network. These great inventions enable us have much better performance in data analysis. Inspired by the deep neural network architecture, we obtain a rapid development in computer vision. Computer vision is an cutting edge technique making computer to see visual object and extract information from images and videos in the similar way as human does. Image classification and image recognition are the art-of-state field in computer vision, and they have significant contribution to human face recognition, automatic driving and neural style transfer. In this capstone project, we are using convolutional neural network to train MNIST dataset, and this is a kind of image classification. MNIST dataset is a large database of handwritten digits that is commonly used for training various image processing systems. The examples in MNIST dataset are black and white images of human handwritten digits and they are normalized to fit into a 28×28 pixel bounding box and anti-aliased, which introduced grayscale levels. Hence, we can encode the images into matrices which have all entries equals to one or zero, this problem is transferred as matrix regression. The framework we use to process image classification of MNIST dataset is convolutional neural network (we called CNN or Conv neural network later). Convolutional neural network (CNN, or ConvNet) is a class of deep neural

networks, most commonly applied to analyzing visual imagery. Convolutional networks were inspired by biological mechanism in that the connectivity pattern between neurons resembles the organization of the animal visual cortex. The advantage of CNN of image classification is CNN has a weight-sharing feature. Comparing with fully connected neural networks, CNN has conspicuous fewer parameters which makes CNN extract information from dataset at lower computational cost.

The main purpose of this project is to find an optimal hyperparameter set for CNN framework. In deep learning, a hyperparameter is a parameter whose value is set before the training begins. Different models have different types of hyperparameters, in our CNN model we have filter size, Conv layer channel, pooling size, drop out rate and dense layer size as hyperparameters. Hyperparameters are integers and continuous numbers in our model, leading to mixed-type optimization problems. The choice of hyperparameters can affect the performance of deep neural network significantly. The accuracy and the time required for training process will depend on the hyperparameters we choose. To improve performance of our CNN model, we can apply various tuning methods to find a set of optimal hyperparameters. In this case we are applying genetic algorithm for hyperparameter optimization. Genetic algorithm for CNN tuning is a kind of population based training which is inspired by the process of natural selection. We consider a

tuple of hyperparameters as a chromosome of an individual. In our model, we assume every individual has only one chromosome. Since one chromosome represents a tuple of hyperparameters, a gene on the chromosome represents one hyperparameter. If a model has ten hyperparameter, the individual will have one chromosome consists ten genes. For our genetic algorithm, all the individuals in the same generation constitute the whole population. In next section, we will discuss each gene on the chromosome and how genetic algorithm works. For the first generation, we randomize the initial population using the hyperparameter distribution that is pre-defined. In this case, we regulate the distribution of hyperparameter to save computational resource, since genetic algorithm is at high level of computational cost, and we will also discuss later. Compare to using grid search method, for the first generation, we use random method to generate will enable us to explore more combination of hyperparameters. In most cases, we know little about the properties of deep neural ntework model, so that it is hard to directly give empirical combinations which have great performance. Even though grid search method seems more robust, we will not use it here. Figure 1 shows difference between grid search and random search.

The remainder of this Capstone project is organized as follows. In Section 2 we will discuss the model and tuning method we use. Section 2.1 is about CNN model for this problem, and Section 2.2 is about genetic algorithm. Results after

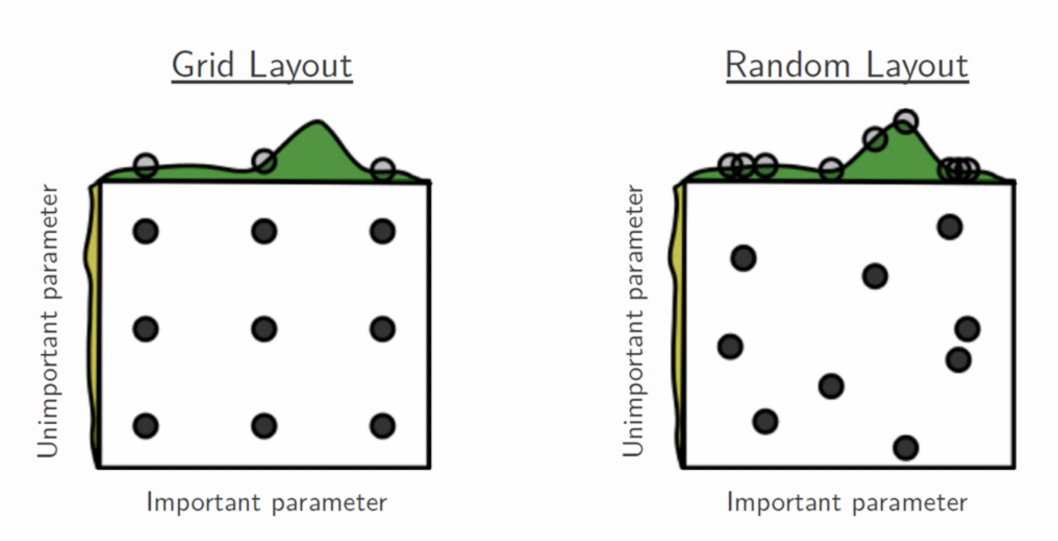

Figure 1: Grid Search vs Random Search

testing by MNIST dataset are included in Section 3 and we draw conclusion in Section 4.

2 Modeling

In the first part of this section, we will discuss the structure of our CNN model, and explain the functions of every part of CNN architecture. In the second part of this section we will introduce genetic algorithm, and we will explain how genetic algorithm works for CNN hyperparameter tuning. In our model for training MNIST dataset, CNN model is implemented as the inner part for genetic algorithm, and genetic algorithm works as outer structure for selecting optimal hyperparameter combination.

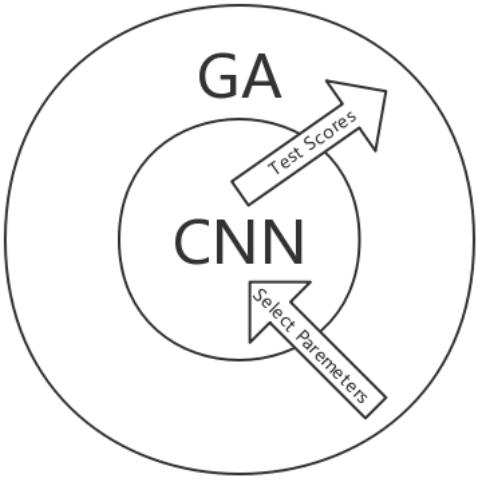

Figure 2: Relationship bewteen genetic algorithm and CNN in our model

2.1 CNN Model

As the inner structure of our model, CNN model is the classifier that test the performance of the model by a tuple of given hyperparameters from genetic algorithm. In our experiment, we use fixed layer structure to train MNIST dataset. In other words, our neural layers such as Conv layers, max pooling layers, drop-out layers, dense layers and softmax layers are in fixed order and the number of each layer is also fixed. The reason that we are not using a dynamic structure CNN model is because we are tuning the model by genetic algorithm, therefore the individual in a certain generation will have the same type of chromosome which indicate they have the same type of genes representing

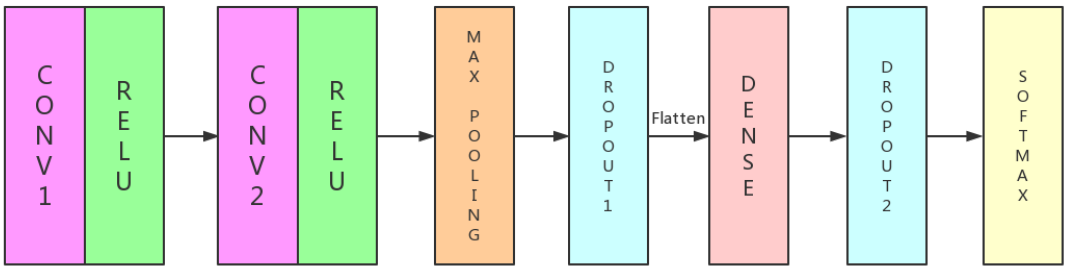

Figure 3: The structure of our CNN model

hyperparameters. If we change the order or number of our hidden layers in CNN model, the chromosome of individuals will change simultaneously. Individual with different type of chromosome cannot reproduce the next generation.

The structure that we are applying is shown in Figure 3. This model is inspired by LeNet-5, which is developed by Yann Lecun. The first hidden unit is consisted of a convolution operator and a ReLU activation function. The convolution operator calculates the summation of the production between each entry in the filter matrix and the corresponding area in the input matrix. Here we applied 1 as the strides of filter since our input is just a 28×28 gray scale image, we do not use bigger strides. Consider A is the input matrix, A_R is the sub-matrix of A whose upper left entry is A_{ij} and dimension equals to m which is the dimension of filter maxtix and F is the filter matrix. Then we have:

$$Conv(A)_{ij} = \sum_{p=1}^{m} \sum_{q=1}^{m} A_{Rpq} F_{pq}$$

Figure 4 shows how the convolution operator works. In this case, the first hidden layer provides us two hyperparameters which are the size of filter(or kernel) and the number of filters. It worth to notice that we actually apply same padding to the input matrix. For MNIST dataset, the inputs of data 28×28 pixel image, this is of a very small dimension. We basically don not want the image shrink when processing Conv layers, so we just add zero entries around the input matrix to get the output matrix ad the same dimension after Conv layers, and this is what we do for same padding. Additionally, if we do not use any padding, the entries near the boundaries of input matrix will become less import.

After the convolution operation, we put the matrix into an activation function. In our cases, we use ReLU. This activation function was first introduced to a dynamical network by Hahnloser et al. in 2000. The mathematical notation of ReLU is:

$$f(x) = max(0, x) = x^{+}$$

ReLU function helps us map the output of previous layer to the range of 0 to 1. The reason that we use ReLU as activation function is:

- **Sparsity**: When the input of ReLU is non-positive, the output turns to be zero, which sparse activation.

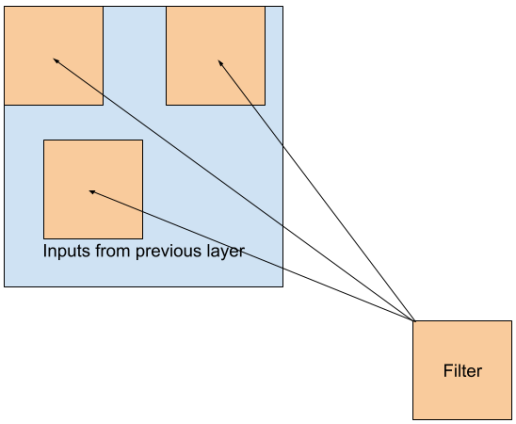

Figure 4: How CNN filter and max pooling layer works

- **Better gradient propagation**: ReLU can effectively reduce vanishing gradient problem. For example, the gradient of sigmoid activate function vanishes when the absolute value of input goes very large.

- **Easy to compute**: ReLU save a lot of computational resource when doing propagation.

For the second hidden unit, it is totally the same with the first one, and also gives us another two hyperparameters which are the size of the filers and the number of filters. This hidden unit is following with a max pooling layer. The max pooling layer works similarly as Conv layers, the Conv layers calculate the summation

of the productions of each pair of entries in input matrix and filter matrix. The max pooling layer just selects the max entry of each sub-matrix in the same size as the max pooling kernel. In Figure 4, max pooling layer just select the max values in those yellow boxes. The purpose that we add pooling layers in our neural network is the pooling layers can reduce the size of the representations and keep the significant values which accelerate our computation. Notice that the max pooling layer also provide us a new hyperparameter which is the size of max pooling kernel. The next layer of our model is a dropout layer. This layer works in a quite simple way but it will help our model get better convergence. The dropout layer randomly drop some hidden units in our neural network, in other words, we ignore some neurons in forward or backward propagation. By doing this we can effectively prevent over-fitting, and this dropout layer gives us a hyperparameter of the random drop rate. Until this part, we finished the feature learning of our model, and the next thing we need to do is the classification part, which enable us to get the relationship between images and labels. Firstly we need to apply a fully connected layer to the output from previous convolutional layers. Before doing this, we need to flatten the output matrices from Conv layers. This means we reshape tensor into vector in order to proceed the fully connected layer. A fully connected layer is an linear operation in which each input is connected to each output by a weight and a bias. Suppose $a^{[l-1]}$ is the input vector of the $l-1^{th}$ layer, $W_i^{[l]}$ is the i^{th} weight vector in the next l^{th} layer,

and b_i is the i^{th} bias, then we will have $a_i^{[l]}$ which is the i^{th} output of l^{th} dense layer(fully connected layer is also called dense layer) is:

$$a_i^{[l]} = ReLU(W_i^{[l]} a^{[l-1]} + b_i)$$

$ReLU$ is the activation function we use before. This dense layer also provides a hyperparameter of the output vector size. This means the number of the weight vector multiplied by the flattened input vector. The first fully connected layer is also followed by a dropout layer that deactivate some neurons to improve the convergence a prevent over-fitting which gives us a dropout rate as another hyperparameter.

The final layer of CNN model is also a dense layer, but different from the previous dense layer, we are applying another activation function which is Softmax Function. The reason that we use Softmax function is because we are doing multiple classification. MNIST dataset has ten categories of labels for human-written images, so the out put of our CNN model should give us corresponding outputs. The activation function such as ReLU and sigmoid can only do binary classification, and that is why we cannot use them in the final layer to generate outputs. Suppose $z_i^{[l]}$ is the value after linear operation $W_i^{[l]} a^{[l-1]} + b_i$ for the l^{th} dense

layer. The mathematical equation of Softmax function is:

$$Softmax(z_i^{[l]}) = \frac{e^{z_i^{[l]}}}{\sum_{i=1}^{m} e^{z_i^{[l]}}}$$

where m is the number of categories, in our case $m = 10$. The Softmax activation function gives us output of multiple catrgories. From the equation above, we will get an output of probabilities of each label, and then the Softmax layer return us a corresponding label. For instance, in the MNIST problem, if the probability of number 1 is the maximum value among all probabilities it will return the label as 1. By using Softmax function, we got the final output of forward propagation.

For deep neural network, the choice of loss function and optimizer can significantly affect the performance of the model. When we use Softmax as activation function, the categorical Cross-Entropy loss can be a good choice for our regression. The reason we choose categorical Cross-Entropy loss is we are doing a multi-class classification, and categorical Cross-Entropy loss cooperates with Softmax very well. The formula of categorical Cross-Entropy loss in our case is:

$$CELoss = -\sum_{i}^{c} t_i log(Softmax(s)_i)$$

$$t_{i=p} = 1, t_{i\neq p} = 0$$

In this formula, c is the number of classes(labels), t is a c dimension vector of $0s$ and $1s$. For the output is one-hot, only when the p^{th} class keeps its term, the

p^{th} entry of t equals to 1, otherwise the entries of t is zero and s is the output of CNN model. In our case, the output is a vector with probabilities so the formula can be written as:

$$CELoss = -\sum_{i}^{n} \hat{y_{i1}} log(y_{i1}) + \hat{y_{i2}} log(y_{i2}) + \cdots + \hat{y_{ic}} log(y_{ic})$$

where n is the number of samples and c is the number of categories. The optimizer we apply is ADADELTA method. We actually do not tuning our learning rate, therefore we use ADADELTA since it has a robust performance even if using the default learning rate in most cases. It is also worth to mention that we are using mini batch gradient descent method for training, this will help us get a better and faster convergence. Hence we get another hyperparameter of batch size.

Until now, our model basically needs to optimize nine hyperparameters, they are:

- The size of filters for the first and second Conv layers
- The number of filters for the first and second Conv layers
- The size of max pooling layer
- The dropout rates for the two dropout layers
- The dimension for the first fully connected layer

Notice that we do not take the number of epoch and mini batch size as hyperparametes because there will be a great difference between different epoch and mini batch size, these two hyperparameters will vanish the influence of other hyperparameters. In the second part, we will discuss how does genetic algorithm work with hyperparameter optimization.

2.2 Genetic Algorithm

The genetic algorithm of our code works as the outer part of our model. We use genetic algorithm for selecting optimal hyperparameter set for our inner CNN model. Genetic algorithm is inspired by the theory of natural selection by Darwin. Individuals with the better genes will survive in the process in natural selection, the individuals with better genes will have better fitness, therefore they have more chance to reproduce the next generation. As a result, the individuals in next generation will inhere competitive genes from their parents. The population selects better individual by this way. In our genetic algorithm, we consider a set of hyperparameter for CNN as a individual, hence one hyperparameter represents a gene on the individual's chromosome(We assume one individual only has one chromosome). The first step of our algorithm is selecting the candidates with higher fitness from our initial generation, and the selected individuals make up the mating pool for reproducing the next generation. In the next step, the individuals

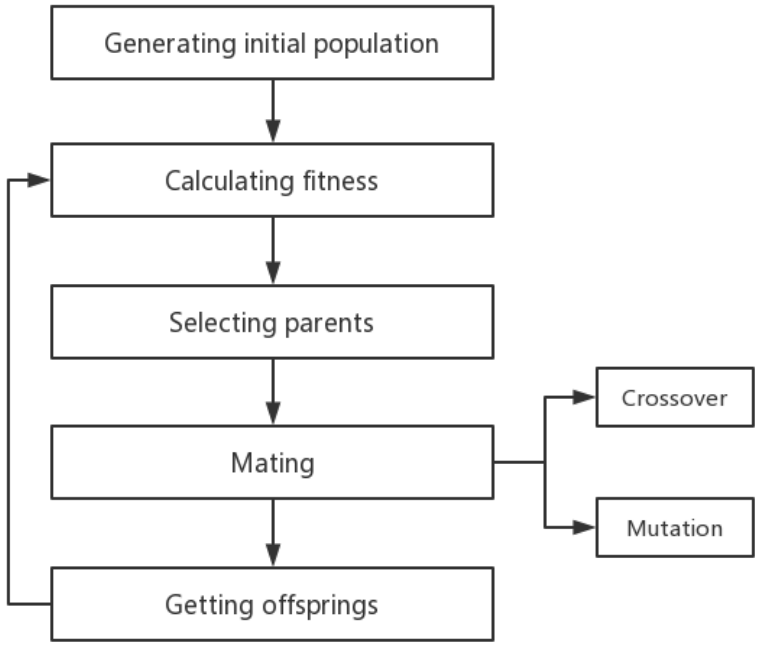

Figure 5: The process of genetic algorithm

in the mating pool start a mating process to generate the offspring set. The mating process basically follows the same rules as natural creatures: the parents' chromosomes crossover with each other and mutation take place on the new chromosomes. Figure 5 shows the steps for our genetic algorithm. Then we will explain how every step works with detail.

2.2.1 Generating initial population

The purpose of this project is automatically search a optimal hyperparameter set for our CNN model. To begin with, we randomly generate eight sets of hyperparameter, so that we can get an initial population matrix P , and the dimension of matrix P is 8×8 since each set contains eight hyperparameters. In the initial population matrix, the column size indicates the number of candidates in the first generation, and the row size indicates the number of genes on one individual which represents the number of hyperparameters. The thing we need to notice is that we cannot choose the hyperparameter arbitrarily, since the algorihtm of CNN model also follows some rules. For example, the number of filter in Conv layers should be positive integer, the size of filter can not exceed 28 because the dimension of our input is 28×28, and the dropout rate should be float number between 0 to 1. To ensure the CNN model run correctly, we design a initialize distribution for generating the initial population as Table 1. The process of generating the initial is technically easy, we use numpy build-in function random to generate the population. We choose the distribution of hyperparameters that contains the empirical optimal hyperparameter set. This is for checking whether genetic algorithm can choose the hyperparameter combination which approaches the empirical optimized hyperparameter set.

Table 1: Distribution of Initial Hyperparameters

Hyperparameter	Value
Size of Filter 1	2-10 int
Number of Filter 1	2-64 int
Size of Filter 2	2-10 int
Number of Filter 2	32-128 int
Size of Max Pooling	2-10 int
Size of Dense Layer	63-256 int
Dropout Rate 1	10%-90%
Dropout Rate2	10%-90%

2.2.2 Calculating fitness & selecting parents

Once we get our initial population we need to select the competitive candidate for generating the next generation. We should know that the individuals with higher fitness which means the genes of competitive candidates have better performance. In this case, the hyperparamter set with better performance will be selected as parents of the next generation. We evaluate the performance of candidates in two aspects. The first criterion is the accuracy of the test dataset. For judging the performance of a machine learning project, we usually consider the bias and variance of the model. The accuracy of test set can reflect both bias and variance, when the test accuracy is high, we can say both bias and variance is low which means our model is workable and robust. The second aspect is the value of the loss function. As we all know, in an optimization problem, our aim is to minimize the value of loss function(or cost function). We are using categorical Cross-Entropy loss function as we discuss before. Therefore if we can get smaller value of our

categorical Cross-Entropy loss function, we can say the performance of the model is better. In the genetic algorithm we need to design a function which can quantify the level of fitness into a real number. This function is known as the fitness function for genetic algorithm. In this CNN hyperparameter optimization model, we basically want the input of fitness function is one tuple of hyperparameters, and the output is the fitness value. In genetic algorithm, we always want to maximum the fitness value, because the bigger fitness value means the individual is fitter and more competitive. Using accuracy of test set and value of loss function, we define the fitness function as following:

$$Fitness(V_{hyperparemeter}) = CNN_{acc} - CNN_{loss}$$

Where $V_{hyperparemeter}$ is a vector of one set of hyperparameters. CNN_{acc} is the accuracy of test set and CNN_{loss} is the value of loss function. It is obvious that a set of hyperparameter with better performance returns a larger fitness function value. Hence, for the selecting process, we need to set a number m which is the number of parents we want to select, and then we rank the row vectors of the initial population with respect to the fitness value. Finally we choose the first m rows as our parents pool.

2.2.3 Mating: crossover and mutation

For genetic algorithm, the most import part is mating. This is the process that we use parents pool to generate the new population. In this mating process we applied two genetic operators: crossover and mutation. These two operators typically enable the offspring to share many of the characteristics from their parents. Next, we will explain how crossover and mutation works.

Crossover In genetic algorithm, crossover can be also called as recombination. This means crossover is used to combine the genetic information of two parents. This operator is inspired by the crossover process which happens during sexual reproduction in biology. The first step in crossover is to designate a crossover point, and then we swap the partial chromosome in the same side of the crossover point. By doing this, the two offspring will carry the genetic information from both of the parents. In general, there are many ways to design the pattern of crossover, the number and the position of crossover points can be variable. In our cases, we do not randomize the number and position of crossover points. We just pick the middle point of each chromosome and exchange the same side of the chromosome, because we only have eight hyperparameters which means the chromosome has eight genes, therefore the middle point of chromosome is enough for the recombination. Figure 6 shows the crossover of our genetic algorithm.

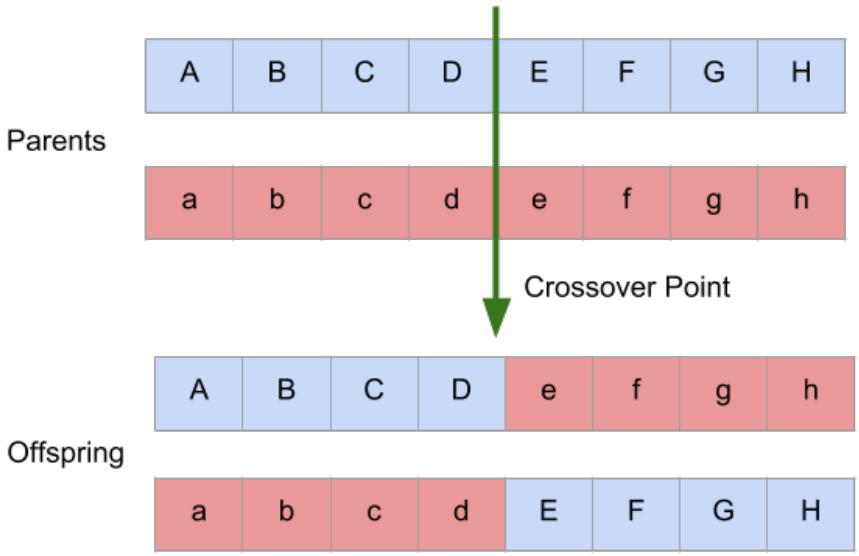

Figure 6: The process of crossover, the **A** to **H** genes corresponds to our eight hyperparameters.

Mutation After crossover process, it is necessary to add some genetic diversity to the offspring, so we apply mutation after crossover. Mutation is also an important genetic operator which is analogous to biological mutation. Notice that we used random method for generating our initial population, so it is possible that the initial points are far away from our optimal points. If we only use crossover to generate next generation, we can hardly get a good convergence or we have a terrible convergence rate. Adding some diversity to genes can help us avoid this problem. In our model, for each offspring, we randomly pick one gene and apply a small perturbation to the value of gene, which means we slightly change the value of one hyperparameter for the new-generated hyperparameter

Figure 7: The process of mutation, **CNEW** is gene **C** after mutation and **dnew** is gene **d** after mutation.

set. One thing that is worth mentioning is we should choose the rate of mutation low, because the search will turn to a primitive random if the mutation happens too frequently. Figure 7 shows how mutation works.

2.2.4 Why genetic algorithm

There are many advantages of using genetic algorithm for hyperparameter optimization:

- For multiple local optima cases, using genetic algorithm prevents us trapping in only one solution. In our case, it is possible to have multiple optimal hyperparameter sets.
- When objective function is not derivable. For some cases the objective function is not smooth or we cannot get the formulated expression of our objective function. In our case, the outputs of CNN model is not derivable.
- GA enables us for multiple types optimization, for CNN the hyperparameters are in multi-type.

- The iterations of GA is independent with each other, therefor we can implement parallel computing.
- GA enables us tuning a huge hyperparameter set, even though our project have relative small hyperparameter set.
- GA can also work even though the objective function is noisy of stochastic.

Until now, we have introduced our model with detail. In the next section, we will discuss the results of our numerical simulation.

3 Result

4 Conclusion