MiniGen: Efficient Evolutionary Search for Convolutional Neural Networks

Wenxuan Zhang Arshia Ebadi Yuqi Jing

***Abstract*—**

Convolutional neural networks (CNNs) are commonly used for image classification. CNN architectures can be manually constructed, but another option is to automatically learn the best architecture for a given task.

In this paper, we propose MiniGen – a genetically-inspired search algorithm – to find optimal CNN architectures for image classification. Since the space of architecture expands exponentially with the complexity of the architecture, we propose an architecture space consisting of modularized encoding structures. In each generation, MiniGen evaluates these architectures on small stratified datasets sampled from CIFAR10. The best architectures then (possibly) undergo mutation and are used to generate new architectures which make up the new generation. This cycle repeats until certain stop criteria are met.

The focus (and novelty) of this algorithm is to learn architectures with low complexity, which alongside the smaller subsets used for evaluation, saves time and lowers computational cost, thus combatting a prominent issue in previous genetically-based search algorithms. Through experiments, we show that MiniGen was able to find CNN architectures with 0.9 million parameters that achieve an accuracy of 74.4% in CIFAR10 in just 25 generations.

The code is available at https://github.com/yqjing/Neural_architecture_search.

I. Introduction

We will discuss our novel MiniGen algorithm in the context of finding optimal convolutional neural network (CNN) architectures for the purpose of classification. This algorithm is inspired by the biological framework of natural selection, in addition to previous research done by Xie and Yuille (2017) and Liu et al. (2018).

Biologically, genes are the building blocks of individuals and (somewhat) control how good the individual will be at performing certain functions. Strictly speaking, this is an oversimplification and the environment also affects individuals' abilities. However, that is not the focus of this project so it will not be discussed. Given the assumption that certain genes allow individuals to perform better at certain tasks (which we will shortly justify), natural selection works to increase the proportion of these advantageous genes in the population from one generation to the next. Through this mechanism, a population (which can be thought of as a collection of either individuals or genes) can evolve through time to become better at performing a given task. To justify our assumption, we can invoke another biological concept called mutation. Every time a gene is passed down from a parent to an offspring, there is a (small) chance that the gene will mutate, thus impacting its function. This mutation is random, in that it can either improve or hinder the gene's function. Natural selection will help the gene propagate throughout the population if the mutation is beneficial, and (ideally) prevent its propagation otherwise.

Capturing the interplay between mutation and natural selection is the focal point of our algorithm. Our approach focuses on the efficiency of the model rather than simply trying to achieve the highest possible accuracy. In this context, efficiency means using less computational power in searching for the optimal architecture. Biologically, natural selection takes a long time to act as changes occur on a generational time scale (e.g. for humans, a generation is about 25 years, so even the smallest changes will take several decades if not centuries to propagate through a population). This issue manifests a bit differently in architecture search, where the time constraint is replaced by computational cost, as showcased by Xie and Yuille (2017).

iterating through several generations requires training many different architectures (each representing an individual) numerous times (each round of training representing a generation). We seek to combat this issue from different angles, as will be explained in further sections.

II. Related Works

In their genetic CNN algorithm, Xie and Yuille (2017) use a block-encoded structure and stack identical blocks to construct a full network. Therefore, they simply search for the optimal architecture for the block encoding in their genetic algorithm. They also implement a decay factor in the evaluation of individual networks, which is a concept that we drew inspiration from for our algorithm, and will discuss later.

We also examined the progressive search algorithm proposed by Liu et al. (2018). They used a search algorithm that is similar to evolutionary search but is different in terms of the search space. Instead of searching for a network architecture with lots of parameters, they started the search by looking for architectures with small parameter sizes, and incrementally increasing the parameter sizes of the neural architectures in the search space, which greatly improved the efficiency of the search algorithm. This inspired us to

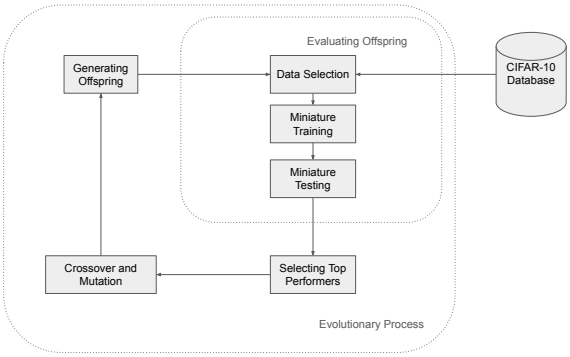

Fig. 1. Overview of the MiniGen evolutionary algorithm process, highlighting the main stages of offspring generation, evaluation, crossover and mutation, and selection for the next generation, with the CIFAR-10 database serving as the input for genetic operations.

design an encoding structure that is modularized and has several layers. We then delved into the highly cited paper EfficientNet (Tan and Le, 2019). In this paper, the authors tried to find a neural architecture that has a small parameter size and high classification accuracy, so that it can do inference efficiently on mobile devices. They introduced a penalty for the number of FLOPS performed by an individual network in the inference stage when doing evaluation in the neural architecture search. The larger model will have a larger inference FLOPs number and will be penalized more in the testing stage. However, the shortcoming of this approach is that when other regularizations (for example, decay factor) are taken into account in the evaluation stage of a particular neural architecture, the evaluation metric can get very complicated, and requires numerous trials and human engineering to make the evaluation metric feasible. This motivated us to come up with a simpler inductive bias for selecting neural architectures with small parameter sizes.

III. MiniGen Methodology

A. Notation

In the classification task, the database CIFAR10 is denoted as $\mathcal{D}$. The $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ denote the training and testing datasets. The small subset of data, sampled from the full training dataset using stratification, is denoted as $\mathcal{D}_{mini\,train}$. The small subset of data, sampled from the full testing dataset using stratification, is denoted as $\mathcal{D}_{mini\,test}$.

For the encoding part of the methodology, b_i denotes the number of blocks inside the cell i, c_i denotes the channel size inside the encoding of cell i. An action list in a cell encoding of a neural architecture is denoted as al_i, where i is the ith cell.

B. Encoding

Each neural network within the population is represented by an encoding that details its architecture, consisting of cells, blocks, and operations. Specifically, the encoding structure

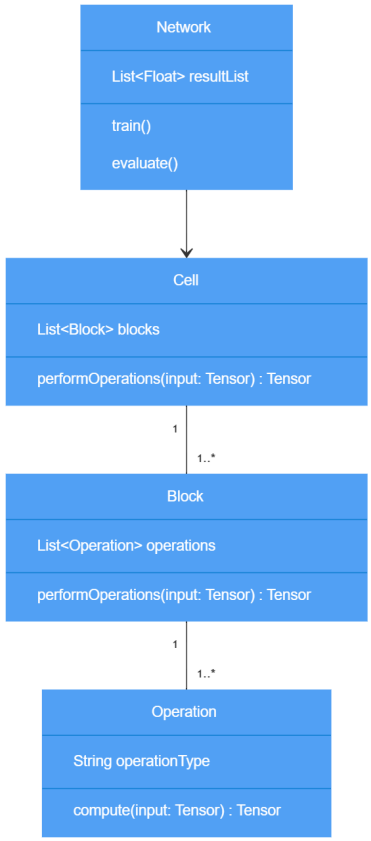

Fig. 2. A UML diagram showing the structure of a network object. A network is composed of a cell, a cell can be made up of one or more blocks, and a block can be made up of one or more operations.

stipulates that each cell contains a single block, which in turn is comprised of a variable number of operations. This design choice facilitates a focused and efficient search within the architecture space.

Specifically, the encoding for an individual network in the MiniGen has the following structure:

$$[[., 1, al_0], [c_1, b_1, al_1], [c_2, b_2, al_2], [c_3, b_3, al_3], [c_4, b_4, al_4]],$$

where the network is denoted as a list of 5 elements. Biologically this mimics the structure of DNA. Each element is called a cell with the index i and the following structure:

$$[c_i, b_i, al_i],$$

where c_i is the channel number of the cell; b_i is the number of blocks inside a cell; al_i is the action list for the cell. We give a detailed explanation for the available encodings and their corresponding actual operations in Table I.

There are two types of cell structures in a network. The first one is the input cell which corresponds to the first cell in the

TABLE I
AVAILABLE OPERATIONS FOR THE ACTION LIST

Operation Encoding	Actual Operation
"3*3 dconv"	Depthwise Separable convolution with kernel size 3 by 3
"5*5 dconv"	Depthwise Separable convolution with kernel size 5 by 5
"3*3 conv"	Convolution with kernel size 3 by 3
"5*5 conv"	Convolution with kernel size 5 by 5
"1*7-7*1 conv"	Convolution with kernel size 1 by 7 followed by convolution with kernel size 7 by 1
"3*3 dil conv"	Convolution with kernel size 3 by 3 and dilation equal to 2
"identity"	Identity connection (skip connection), with identity map if the stride is 1 and 1 by 1 convolution if the stride is 2
"3*3 maxpool"	3 by 3 max pooling
"3*3 avgpool"	3 by 3 average pooling

network encoding and only has one block inside the cell; the action list al_0 for the input cell has the following options:

- "identity"
- "3*3 dconv"
- "5*5 dconv"
- "3*3 conv"
- "5*5 conv"

The second cell type is the normal cell type, which corresponds to the remaining 4 elements in the network encoding list. There can be multiple blocks within one cell, i.e. $b_i \in \mathbb{N}$. The action list al_i for the normal cell has the following options:

- "identity"
- "3*3 dconv"
- "5*5 dconv"
- "3*3 conv"
- "5*5 conv"
- "1*7-7*1 conv"
- "3*3 dil conv"
- "3*3 maxpool"
- "3*3 avgpool"

For every operation in Table I apart from 'identity', we apply batch normalization and RElu activation function after the main operation for default. We design an identity operation for the encoding given the excellent performance of ResNet in the computer vision tasks (He et al., 2016). It performs an identity map from the input of a cell to the output of a cell. If stride is 1 then the 'identity' operation will simply be an identity map with `lambda: x`. If stride is 2 then the 'identity' operation will be a 1 by 1 convolution.

Inside the network, the spatial dimension of the images remains unchanged, which is 32 by 32-pixel size. We perform a stride equal to 2 in the input cell (however the spacial dimension is kept in 32 by 32 using padding throughout the network). The first cell has fixed channel size flow $3 \rightarrow 32 \rightarrow 64$ and fixed action list length 2 or 3. Therefore, the channel size c_0 for the input cell is denoted as '.' since it is not changeable. If the 'identity' operation is selected as the first operation, then the action list will have length 3 and the next two operations will change the channel size of the input image from $3 \rightarrow 32$ and $32 \rightarrow 64$. If the 'identity' operation is not selected as the first operation, then the action list of the input cell will have length 2 and the two operations will still change the channel size of the input image from $3 \rightarrow 32$ and $32 \rightarrow 64$. For example, the input cell has the encoding [., 1, ['identity', '3*3 conv', '3*3 conv']] in one of the neural architectures found in our experiments.

For the normal cells, the strides are all fixed to be 1 and the channel size c_i is the same for all operations in action list al_i. The channel size has the following options:

$$[24, 40, 64, 80, 128, 256],$$

when the encoding is initialized, cell i will get assigned randomly a channel size from the above channel list. During the evolutionary search, the channel size c_i and action list al_i will get passed on to the next generation if the individual neural architecture is selected in the current generation. One example for the normal cell encoding will be [256, 1, ['3*3 avgpool', '3*3 maxpool', '3*3 dil conv', '3*3 dil conv']], where the channel size is 256; the number of blocks inside the cell is 1.

Such a highly modularized encoding structure enables us to scale up/down the neural architecture easily to suit the need of the desired parameter size. In our experiments, we used block number 1 for all cells in the network to keep the computational cost small and network architecture compact, but in future work, we will scale up the model size to explore optimal architecture with larger parameter sizes ($\sim 5M$).

C. Evolutionary Process

The MiniGen algorithm progresses through four distinct stages: Evaluating performance, Selecting top performers, applying crossover and mutation, and creating offspring for the next generation.

1) Generating Offspring: In this initial generation, offspring are generated randomly until the population limit is reached. This method ensures diversity within the population, crucial for exploring a wide range of architectural possibilities.

2) Evaluating Offspring: A novel heuristic function is introduced to estimate the performance of individual networks. This process involves three key steps: data selection, miniature training, and miniature testing.

Data Selection: A small subset of data, $\mathcal{D}_{mini\,train}$ is sampled randomly from $\mathcal{D}_{train}$ using stratification. In the same sense, a small subset of data $\mathcal{D}_{mini\,test}$ sampled randomly from $\mathcal{D}_{test}$. This subset, comprising n training samples and m

testing samples, is deliberately small to ensure rapid evaluation cycles and is balanced across classes to minimize bias. A more detailed description of the data selection can be found in Section IV-A.

Miniature Training: Each network is trained on $\mathcal{D}_{mini\,train}$ for a limited number of epochs, denoted by E_{mini}. This abbreviated training period is not intended to fully optimize the networks but rather to distinguish their potential efficacy quickly.

Miniature Testing: Following training, networks are assessed on $\mathcal{D}_{mini\,test}$. The performance outcomes are recorded in a result list for each network, which maintains a history of its performance scores. An average of the most recent scores, adjusted by a decay factor, gives each network's performance score ps. The metric ps for a net x becomes $ps_x = acc_x - decay_x$, where acc_x is the average accuracy of the individual network x in the testing dataset, and the $decay_x$ is defined as:

$$decay_x = 0.01 \times |gen_c - gen_x|, \tag{1}$$

where gen_c is the current generation number, $gen_c \in \{0, 1, 2, ...\}$; gen_x is the generation number where net x was created, $gen_x \in \{0, 1, 2, ...\}$, $gen_x <= gen_c$. For example, a net that was created in generation 3 will have decay 0.05 in generation 8, if it still stays in the population.

The decay mechanism helps to push the evolution along. As can be seen from Figure 3, the left graph was trapped in the early generation 3, which did not have the generation decay. In contrast, the right graph kept evolving until generation 11. This demonstrates the capacity of the generation decay to stop evolution from stopping in early generations.

3) Crossover and Mutation:

a) Crossover: Networks are first ranked based on their performance scores, and a selection of the top-performing networks is made to continue to the next generation. The remaining networks are removed from the population. The crossover process occurs at the block level within the network encoding. Two parent networks contribute operations to the offspring, with the specific operations chosen randomly from each parent at each crossover point. The probability Pr_c of an individual network c being selected for crossover is given by:

$$\text{Pr}_c = \text{SoftMax}(ps_c) = \frac{\exp(ps_c)}{\sum_j \exp(ps_j)}, \tag{2}$$

where ps_c is the performance score of the candidate network and the denominator is the sum of the performance scores of all networks in the population. $j \in \{0, 1, ..., 4\}$. Intriguingly, we tried simple weighted probability at the beginning, but the result was not satisfying since the simple weighted probability cannot distinguish the neural architecture with a small performance difference. However, after we change the Pr_c to be softmaxly weighted. The performance improved dramatically.

b) Mutation: There are three levels of mutation happening in the evolutionary search. The most basic level of mutation happens in the operation stage. For a particular cell in the create_new_child stage, we go through the action list and each operation has a probability P_o to mutate (reassign a random operation from the operation list to that index in the action list). The second level of mutation happens to the channel size of a cell. Each cell has a probability P_c to be randomly reassigned a channel size from the channel size list. The final level of mutation happens to the action list of a cell. In the create_new_child stage, for each cell, the action list has a probability P_b to mutate. In this case, we re-generate a completely new random action list encoding for the cell from the available operation list.

Using the three levels of mutation, our encoding structure can find the ideal component from the micro scale to the macro scale of a neural network.

4) Creating the Next Generation: The top k performing networks, as determined by their performance scores, are retained for the next generation. From them, $n - k$ new networks are produced via crossover and mutation to replenish the population, maintaining diversity. For example, given that we want a population size of $n = 15$ for each generation, and choose to keep $k = 5$ best performers, then after each round of evaluations we create 10 offspring from the top 5 performers.

IV. Experiments

For the classification task, we performed experiments on the CIFAR-10 dataset (Alex, 2009). All the models are implemented using PyTorch 2.2 and trained on one Nvidia Tesla A100 GPU.

A. Stratified Mini CIFAR10 in Evolutionary Search

CIFAR-10 is a dataset of colour images containing $50,000$ training images and $10,000$ testing images. It has 10 different classes with $6,000$ images for each class. Specifically, the 10 classes are *airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships*, and *trucks*. Each image is of 32×32 pixel format with 3 colour channels.

In the traditional NAS literature, the whole CIFAR10 dataset is used for training and selecting the best network architectures from a population of networks (Liu et al., 2018; Tan and Le, 2019; Xie and Yuille, 2017). This leads to tremendous computational costs for conducting one round of search. Inspired by the Reptile (Nichol et al., 2018), a meta-learning algorithm which generalizes from a small number of examples from the test task, we sample a stratified subset from the training and testing instances of CIFAR10 to form the *Mini CIFAR10*. Specifically, for the training instances of Mini CIFAR10, we first stratify the $50,000$ training instances of CIFAR10 into 10 sub-populations based on the 10 classes and randomly sample 100 images from each of the 10 sub-populations. We apply the same sampling procedure to form the testing instances of Mini CIFAR10 and sample 100 images from the testing instances of the CIFAR10 in a stratified fashion. Therefore, our Mini

TABLE II
PROPERTIES OF THE TRAINING DATASETS USED IN THE EXPERIMENTS.

Data Set	Training instances	Testing instances	Colour channels	Pixel format	Classes
CIFAR10	$50,000$	$10,000$	3	32×32	10
Mini CIFAR10	$1,000$	100	3	32×32	10

CIFAR10 consists of 1000 training instances and 100 testing instances. We present the detailed comparison of the CIFAR10 and Mini CIFAR10 datasets in Table II.

Several advantages were observed by using the Mini CIFAR10 dataset for training and selecting in the evolutionary search. First, by using a much smaller training dataset, each individual network was able to be trained much faster. This allows us to conduct a deeper search in terms of generations compared to previous genetic algorithms. Specifically, given the same time and hardware, the genetic search on Mini CIFAR10 was able to run 12.5 times more generations than the genetic search on the full CIFAR10 dataset (we compared training using 50,000 images for 5 epochs with training using 1,000 images for 20 epochs for each neural architecture).

Second, stratification ensures that the accuracy in the testing instances still represents the overall performance of the particular neural architecture in the full CIFAR10 dataset. In other words, the Mini CIFAR10 permits the neural architecture search to generalize from a small number of images from the full dataset.

Third, a small training dataset introduced the inductive bias that a neural architecture with a very large parameter size would tend to perform badly in the testing instances. Hence, the Mini CIFAR10 penalizes undesirably large architectures. Intriguingly, we observed the phenomenon that for a training epochs of 20 the accuracy of the neural architecture that is larger than 10 million parameters goes down with the increase of parameters. This is consistent with our goal to find a small neural architecture with around 5 million parameters.

B. Controlled Experiments

The controlled experiment was the critical element for the effectiveness of the MiniGen algorithm. There were two main controlling challenges we met during the experiment.

GPU randomness control: During the experiment, we found that the accuracy of a particular net on the testing instances of Mini CIFAR10 can be quite volatile (ranging from 20% to 40%) even after we controlled for the training dataset, testing dataset, training epochs, batch size and the random seed for training dataset shuffle. This made the selection process in the genetic algorithm fruitless. Through extensive studying and researching, we found out that it was caused by the random algorithm used in the CUDA backend of PyTorch, and because we were using a small training dataset, the randomness in training was magnified. Therefore, we tackled the problem by setting the `torch.cuda.manual_seed(2809)` and `torch.use_deterministic_algorithms(True)` before the training of each neural architecture. In addition, we fixed the configuration of cuBLAS (the CUDA Basic Linear Algebra Subroutine library) by setting `CUBLAS_WORKSPACE_CONFIG=:16:8 python3 mini_gen.py` in the shell file. This change provides us with the consistent accuracy result of a particular neural architecture trained on the Mini CIFAR10 dataset.

Dataset randomness control: We further controlled the Mini CIFAR10 for each experiment, which was randomized and selected from the full CIFAR10 dataset. Specifically, we applied `torch.save` to reserve a particular Mini CIFAR10 dataset to pickle file, and used `torch.load` to load it in the later experiments. This allowed us to compare fairly the performance of the neural architectures selected from different experiments.

C. Model Scaling

We applied a simplified encoding structure in the MiniGen for searching efficiency. This significantly improved the searching speed of MiniGen and in the meantime kept the accuracy of the neural architectures found by the algorithm.

Specifically, the particular encoding structure we used in the experiments was the following:

$$[[., 1, al_0], [c_1, 1, al_1], [c_2, 1, al_2], [c_3, 1, al_3], [c_4, 1, al_4]]$$

where there are 5 cells in one network; c_i and al_i represent the channel size and action list of the cell i, $i \in \{0, 1, ..., 4\}$; However, in each cell, there was only one block. The simplified encoding greatly reduced the number of parameters in the neural architecture by more than half but still represented the overall accuracy of the scale-up models. In future work, after the optimal architecture is found, we can scale it up by increasing the number of blocks inside the cells. One ideal encoding for the scaled-up models will be:

$$[[., 1, al_0], [c_1, 1, al_1], [c_2, 2, al_2], [c_3, 2, al_3], [c_4, 4, al_4]]$$

By changing the length of the encoding and the number of blocks inside each cell, we enabled the NAS to adapt to architectures with different targets of parameter ranges (e.g. small models ($\sim$ 1M), medium model ($\sim$ 50M)).

D. Implementation Details

We set the population size to 15 and number of generations to 25. In each generation, we train each individual net 20 epochs on the Mini CIFAR10 training instances and test it on the Mini CIFAR10 testing instances. At generation 0, we initialize the population randomly, with each individual net being assigned a random encoding. For each generation, we picked the top 5 architectures based on the evaluation metric ps as the parents generate child networks by crossover and mutation. The metric ps was described in section III-C2. At the beginning of each generation (after generation 0), we generated the population by creating the remaining 10

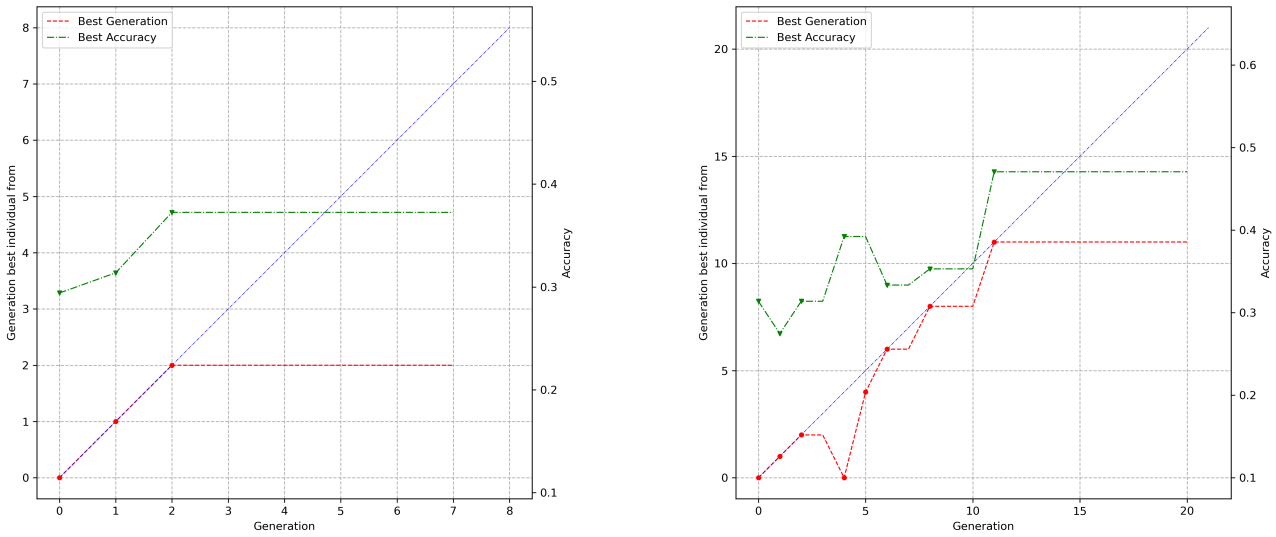

Fig. 3. Comparison of evolution generated by **selection without decay** (left) and **selection with decay** (right) trained on the Mini CIFAR10 dataset. In each graph, the red line represents the generation number of the highest-accuracy individual net in the population for a particular generation in the x-axis. The generation number corresponds to the left y-axis in the graph. The green line represents the accuracy of that individual (also the highest) in the population for a particular generation in the x-axis. The accuracy value corresponds to the right y-axis in the graph.

TABLE III
10 NEURAL ARCHITECTURES FOUND BY MINIGEN IN 25 GENERATIONS

Neural Architecture Index	**Acc.**	**Parameters**	Acc-to-Param Ratio
1	74.7%	0.9M	0.83
8	71.6%	0.9M	0.80
7	64.6%	0.9M	0.72
2	71.8%	1.1M	0.66
0	73.4%	2.4M	0.31
5	67.3%	1.9M	0.35
9	75.0%	3.0M	0.25
4	70.4%	4.0M	0.18
3	69.7%	5.3M	0.13
6	71.6%	5.4M	0.13

architecture from the top 5 architectures in the previous generation. The mutation probability for P_c, P_o and P_b were set to 0.3, 0.5 and 0.2.

We observed that the accuracy of the best individual in the population stopped improving after approx. 25 generations. Therefore, we set the generation number to be 25. After 25 generations, the generation was reset to zero. If the accuracy of the best individual network at generation 25 is better than 35%, we keep the selected top 5 architectures at that generation and use them to initialize the population (all network's generation attribute is reset to zero). Otherwise, we randomly initialize the population.

We kept running the MiniGen until 10 individual nets were found to have accuracy better than 40% on the Mini CIFAR10 testing instances. Then we trained the 10 individual nets on the full CIFAR10 training set for 340 epochs and tested its accuracy on the full CIFAR10 testing set.

E. Results & Analysis

We report the results on CIFAR10 classification for the 10 architectures found in Table III. The neural architecture index is assigned based on the order in which the network was found during the evolutionary search. The **Acc.** refers to the testing accuracy of each network in the full CIFAR10 dataset. The **Parameters** column lists the corresponding penalized size of the network in millions. We sort the table in descending order of the Acc-to-Param Ratio.

As can be observed from the table, MiniGen is able to find neural architectures that possess decent accuracy ($\sim$ 75%) but with relatively small parameter sizes ($\sim$ $1M$). As the parameter sizes get larger, the MiniGen didn't manage to find architectures that possess increasing accuracies. For the

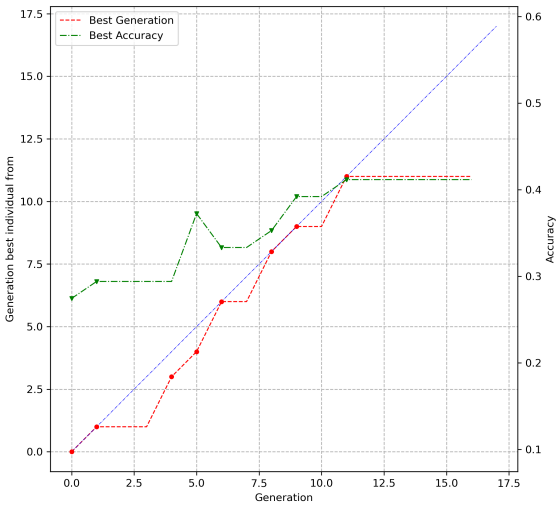

Fig. 4. A diagram showing the best Generation & best Accuracy change of the MiniGen for Neural Architecture with Index 1 in Table III.

architecture with the largest number of parameters ($\sim 5M$), the accuracy still remains around 75%. Our assumption is that the Mini CIFAR10 dataset introduces the inductive bias that, for the particular encoding used in our experiment, model architecture with parameters around 1 million will be effectively searched and found. However, for larger model architecture with parameters around 5 million, the Mini CIFAR10 penalizes their training process and causes an unsuccessful search in the architecture space. We further drew the conclusion that the 1% training dataset size of the CIFAR10 dataset is useful for finding convolutional neural architectures with around 1M parameters for our particular encoding configuration.

In future work, we aim to scale up the encoding configuration (by including more cell numbers and block numbers per cell) in order to make the Mini CIFAR10 dataset suited for finding ideal neural architectures in the parameter size range around 5M. This can be easily experimented with given our modularized and flexible encoding structure.

We also present the Best Genration & Best Accuracy evolving graph 4 for the Neural Architecture with Index 1 found in our experiment. The detailed encoding is showed below:

[128, 1, ['identity', '3*3 conv', '3*3 conv']],
[256, 1, ['3*3 avgpool', '3*3 maxpool', '3*3 dil conv', '3*3 dil conv']],
[40, 1, ['3*3 avgpool', '3*3 dil conv', '5*5 dconv']],
[128, 1, ['5*5 dconv', '3*3 conv']],
[64, 1, ['1*7-7*1 conv', '3*3 avgpool', '3*3 avgpool']]

As can be seen from the figure, during the evolutionary search, the generation number where the best individual net was created kept increasing as the current generation number increased. In the meantime, the corresponding accuracy of the best individual also increases. The architecture in index 1 was found at generation 11. The searching process only took around 10 minutes to run in our A100 GPU, where previous genetic algorithms normally take around hours to find a ideal architecture trained on hundreds of GPUs. The positive correlation between the best generation number and the best accuracy demonstrates the effectiveness of our MiniGen algorithm. The short search time showed the high efficiency of our proposed search algorithm and its capability to find a descent architecture in a short time span.

F. Limitations

Several limitations of MiniGen were observed during the experiments. First of all, given the working range for parameter size ($\sim$ 1M), the accuracy of the neural architectures found by MiniGen is competitive in the task of CIFAR10 classification. In the future, we aim to improve this by modifying the encoding structure to accommodate larger parameter sizes.

Second, the evolutionary search algorithm converges in around 25 generations even with the presence of decay penalization for networks with older generations. We intend to tackle this problem by incorporating continuous training and improving the design of decaying factor to the genetic algorithm, i.e. the parameter of a particular network from the previous generation can be passed to the next generation if it gets selected but its performance score will be penalized by multiplying its average accuracy with the redesigned decaying factor.

V. Conclusion

We have proposed a novel genetic algorithm called MiniGen for finding optimal architectures in image classification based on the idea that the performance of a neural architecture evaluated (trained and tested) on a small dataset can generalize to the performance evaluated on the full dataset. The approach is far simpler and more efficient than previous genetic algorithms, and reflects a inductive bias that is beneficial for selecting neural architecture with small parameter sizes. We show that the method works particularly well for finding neural architectures with parameters around 1M. We further demonstrate how the MiniGen can efficiently find ideal architectures in just a few generations (around 25 generations) and in surprisingly short time (around 10 mins). A natural direction for future work is to scale up our highly modularized encoding structure to adapt the search space to larger neural architectures ($\sim$ 5M). Overall, the simplicity and effective inductive bias of MiniGen makes it a promising approach for neural architecture search in the area of small neural networks.

ACKNOWLEDGEMENTS

We would like to thank Professor Jeff Orchard for encourging us to explore the idea of using mini CIFAR10 for performing neural architecture search. If it wasn't his blief for the potential effectiveness of the inductive bias, we wouldn't have picked and carried out the project from the beginning to the end. This work was supported by the MCFC server for providing us with 1 NVIDIA A100 80G GPU.

REFERENCES

Alex, K. (2009). Learning multiple layers of features from tiny images. *https://www. cs. toronto. edu/kriz/learning-features-2009-TR. pdf*.

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*, pages 770–778.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018). Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34.

Nichol, A., Achiam, J., and Schulman, J. (2018). On first-order meta-learning algorithms. *arXiv preprint arXiv:1803.02999*.

Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR.

Xie, L. and Yuille, A. (2017). Genetic cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1379–1388.

APPENDIX

Encodings in Results: The encodings for the 10 architectures mentioned in section IV-E can be found in https://github.com/yqjing/Neural_architecture_search.