EECS 592: A Gene Expression Programming approach to designing Convolutional Neural Network Architectures

Deepak Nagalla
University of Michigan
Ann Arbor, MI, USA
Email: ndeepak@umich.edu

Ishan Kapnadak
University of Michigan
Ann Arbor, MI, USA
Email: kapnadak@umich.edu

Naman Bhargava
University of Michigan
Ann Arbor, MI, USA
Email: namanb@umich.edu

I. Introduction & Related Work

Convolutional Neural Networks (CNNs) have risen sharply in popularity over the last decade or so due to their versatility and success in solving a variety of computer vision as well as language tasks. Moreover, CNNs present themselves as a versatile AI solution to many problems due to the flexibility of their architectures. A typical state-of-the-art CNN architecture today consists of several convolutional layers, pooling layers, and fully connected layers. With the tasks at hand becoming more and more challenging, CNN architectures have also become more and more complex. Today it is not too uncommon to see an extremely deep CNN architecture with several millions of parameters. Although versatile, this also poses a huge challenge since the problem of finding the right CNN architecture also becomes more and more difficult.

Traditionally, the architecture of the convolutional network was often designed by hand, after taking into consideration the problem statement, and the data. However, there has been a lot of recent work that tries to automate this process in a bid to come up with an intelligent approach that can efficiently pick a CNN architecture that is near-optimal for the task at hand. One approach that has been taken is to optimize the hyperparameters using a machine learning approach. Here, some pre-defined properties of the network (such as the number of layers and sizes, activation functions, etc.) are optimized using approaches such as grid search or gradient methods [1], random search [2], or Bayesian method [3]. However, such methods are often restrictive since they assume a fixed architecture (in terms of what layers are used and how they are connected) and only optimize the parameters of those layers.

Another approach that has been growing in popularity is the use of evolutionary algorithms to optimize networks. [4] provides a thorough survey of a lot of these approaches. There has also been work that uses Cartesian Genetic Programming (CGP) to design CNN architectures [5]. However, CGP is a tree-based approach and is quite resource-intensive. Another popular evolutionary approach to optimization is through the use of Gene Expression Programming (GEP) [6]. The aim of our project is to replicate the approach in [5] using both CGP as well as GEP. We will use the CGP approach as a baseline and compare the performance of the GEP approach with respect to this baseline. Although there has been some previous work that uses GEP in conjunction with CNN architectures [7], this has been focused on applications in cybersecurity. Our approach, in contrast, is focused on image classification (similar to [5]).

II. Problem Set-Up

The goal of our rational agent is to find the optimal convolutional neural network architecture for a given image classification problem. The agent is given access to all the training images and their labels. The agent is expected to return the CNN architecture that has optimal performance for this task.

The agent is tested on the CIFAR-10 dataset [8] which is one of the most common datasets used for the task of image classification. The CIFAR-10 dataset consists of $50,000$ training images and $10,000$ test images. Images are categorized into 10 different categories such as aeroplanes, automobiles, birds, cats, etc. Further, all categories are equally represented in the dataset with each category having $6,000$ images corresponding to it. Moreover, the categories are also mutually exclusive, so that each image belongs to exactly one category. The images all have size 32×32 pixels.

Before the training dataset is passed to the agent, we perform some pre-processing steps on the images. Firstly, all images are normalized so that each pixel value is between 0 and 1. Further, to reduce computation time, all images are also converted to grayscale. Finally, we also apply histogram equalization to the images to enhance contrast. Moreover, we randomly split the training data further into $45,000$ training images and $5,000$ validation images. The agent uses the training images to train itself and the validation images to compute the accuracies of various architectures that it generates as part of its algorithm.

III. Methodology

A. Gene Expression Programming

1) Gene Encoding and Decoding: For Gene Expression Programming, we use the method proposed in [7]. For the purpose of this project, we restrict our network to have two kinds of layers – (i) Convolutional layers (denoted C), and (ii) Dense layers or Fully connected layers (denoted F). Further, the network is organized so that the input layer first passes through all the convolutional layers and then through all the dense layers. Each convolutional layer is followed by a max pooling layer. At the end of the network, we add a dense layer with 10 outputs since our image classification problem has 10 categories or labels. A CNN architecture following these properties can then be encoded as a *multiset* (a set in which the order of elements does not matter but their multiplicity matters). For two arbitrary elements A and B, we define the following properties that we will use.

1) *Commutativity.* $A \cup B = B \cup A$, $A \cap B = B \cap A$.
2) *Union, Intersection.* $A \cup B = \{A, B\}$, $A \cap B = \emptyset$.
3) *Null Operations.* $\emptyset \cup A = A$, $\emptyset \cap A = \emptyset$.
4) $A \cup A = \{A, A\}$, $A \cap A = \{A\}$.
5) $A \cup \{A, B\} = \{A, A, B\}$, $A \cap \{A, B\} = \{A\}$.

Note that crucially, we have $A \cup A = \{A, A\}$ which differs from the usual notion of union. This is because the multiplicities of elements matter.

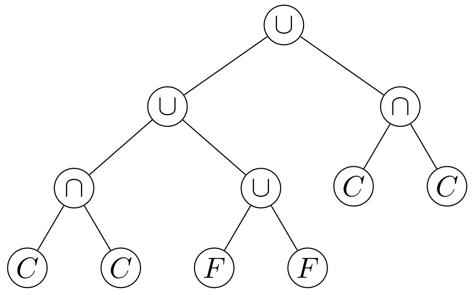
A GCNN gene is defined as a vector $G = \langle G_h, G_t \rangle$ where G_h denotes the gene head and G_t denotes the tail head. We use the *terminal set* $\mathcal{T} := \{C, F\}$ and the *function set* $\mathcal{F} := \{\cup, \cap\}$. That is, elements of the CNN architecture representation are drawn from $\mathcal{T}$ and combined using operations from $\mathcal{F}$. The gene head G_h draws its elements from $\mathcal{F}$ and the gene tail G_t draws its elements from $\mathcal{T}$. Assuming that each operation in $\mathcal{F}$ takes exactly 2 operands, we have the relation

$$l_t = l_h + 1$$

where l_h, l_t denote the length of the gene head and gene tail. An example of a random gene of length 11 (with $l_h = 5$ and $l_t = 6$) is shown below.

$$\begin{array}{c|ccccc|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ G_1 & \cup & \cup & \cap & \cap & \cup & C & C & F & F & C & C \end{array}$$

Once a gene sequence has been created, it can be decoded using an expression tree. We first parse the head of the gene and arrange all the operations seen into a binary tree. Once this binary tree is created, we parse the tail of the gene and assign all elements to the leaves of the operation tree going from left to right. For example, the gene G_1 depicted above is converted to the following expression tree.



The final layout of the resulting CNN can be computed by decoding this expression tree. For example, the above tree can be decoded using a recursive approach. The resulting multiset is

$$\begin{aligned}(C \cap C) \cup (F \cup F) \cup (C \cup C) &= \{C\} \cup \{F, F\} \cup \{C\} \\ &= \{C, C, F, F\}.\end{aligned}$$

Since both $\mathcal{T}$ and $\mathcal{F}$ have two elements, we have encoded a gene vector G as a bit string. A 0 in the head represents a $\cup$ and a 1 in the head represents a $\cap$. Similarly, a 0 in the tail represents a C and a 1 in the tail represents an F. Thus, we have shown how a bitstring can be encoded into an expression tree and further decoded into the resulting CNN architecture. One redundancy that arises in this method is that a bitstring may give rise to an architecture that has only convolutional layers or only dense layers, or in the worst case, has no layers at all! A simple workaround is to check if the resulting multiset contains both kinds of layers, and if not, add one of each type to the architecture.

2) Genetic Algorithm: Once the genes are formalized, it remains to define the evolution algorithm that we use. We have used two variants for our project.

1) Crossover Mutation
2) $(1 + \lambda)$ Evolutionary Algorithm

The first algorithm we implemented was crossover mutation. Here, we maintained a population size of 10, with the initial 10 genes being picked uniformly at random. At each evolutionary stage, we first compute the fitness of all the individuals in the population (the fitness is given by the accuracy of the trained network on the validation data). Once the fitnesses are computed, we define a probability distribution on these 10 individuals that assigns a weight proportional to the fitness. We then generate a new generation of individuals in the following manner. We first sample two parents from the population according to the above-defined distribution. These parents then reproduce (via picking a random crossover point and merging the two genes at the crossover point) to give a child gene. This child gene is then mutated where each bit is flipped randomly with some mutation probability. We keep track of the optimal fitness and the optimal gene. After some fixed number of generations, we return the optimal gene and declare the architecture defined by this gene as our final designed architecture.

The second genetic algorithm we implemented was the $(1 + \lambda)$ Evolutionary Algorithm which is studied in [9], [10]. Here, we maintain just one parent. At each iteration, we generate λ offsprings from the parent using *forced mutation*. In contrast to the mutation defined in the previous algorithm (which we call *neutral mutation*), here we pick one bit from the bitstring at random and forcibly flip it. This has the added advantage that each offspring is guaranteed to be different from its parent. Once the offsprings are generated, we also mutate the parent using neutral mutation. These $1 + \lambda$ individuals are then evaluated and the gene with the highest fitness is chosen as the parent of the next generation.

B. Cartesian Genetic Programming

We also implement Cartesian Genetic Programming (CGP) based on [5] to serve as our baseline. The CGP encoding scheme represents a CNN architecture with a two-dimensional grid with N_r rows and N_c columns (these determine the depth and width of the CNN architecture). We also use highly functional computational blocks as our node functions. The algorithm proposed by the authors uses six functional blocks – ResBlock, ConvBlock, Average Pooling, Max Pooling, Summation, and Concatenation. CNN architectures are represented via genotypes and phenotypes. Genotypes are arrays of fixed lengths where each element corresponds to a node in the grid and holds information about its function and connections. the phenotype is the actual CNN architecture that is generated by decoding the genotype (this may have variable length due to some nodes being inactive). An example of a 2×3 grid is shown in Figure 1 with Figure 2 showing the corresponding genotype and 3 showing the corresponding phonetype.

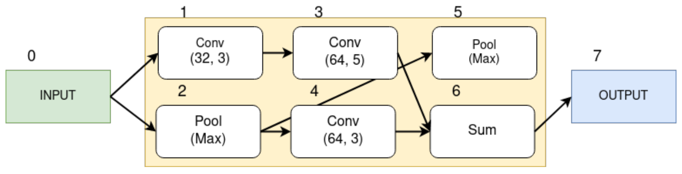


Fig. 1. Example 2x3 grid used by CGP

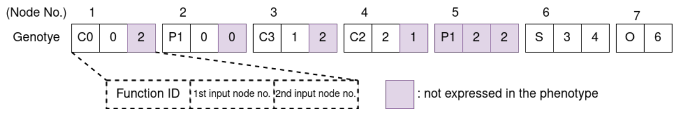


Fig. 2. Genotype Corresponding to grid in Fig. 1

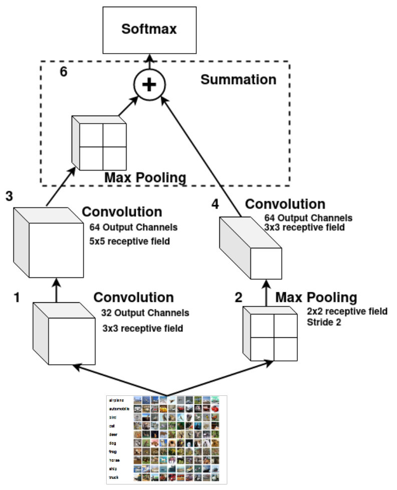


Fig. 3. Phenotype Representation Corresponding to grid in Fig. 1

The authors use a $(1+\lambda)$ evolutionary algorithm with point mutation, as describe in detail in the previous subsection. We also allow the ConvBlock and ResBlock some flexibility in parameters, with the number of filters being chosen from $\{32, 64, 128\}$ and kernel size being chosen from $\{3 \times 3, 5 \times 5\}$. We use a mutation rate of 0.05, and grid size of 5×30. While the authors ran the algorithm for 500 generations with each architecture being trained for 500 epochs, we did not have the computational resources to do so. As a result,

C. Parameter Settings and Evaluation for GEP

For the crossover mutation algorithm, we maintain a population size of 10. We allow the evolution algorithm to go up to 50 generations. We use a mutation probability of 0.1 and maintain a gene length of 11. For the $(1 + \lambda)$ EA, we use $\lambda = 2$. We again use a gene length of 11, mutation probability of 0.1, and maximum number of generations 50.

For the CNN architecture, the first convolutional layer has 32 filters whereas each subsequent convolutional layer has 64 filters. Each convolutional layer uses a kernel size of 3×3, with ReLU activation and padding. The first dense layer is preceded by a flatten layer, and each dense layer has an output size of 64. The architecture ends with a dense layer having 10 outputs, each corresponding to one of the labels. For each evolutionary algorithm, we also experiment with two variants of the architecture – one with dropout [11], and one without dropout.

Each CNN architecture is evaluated by training it for 10 epochs on the training data, and then using its accuracy on the validation data as the fitness. We use an ADAM optimizer [12] with categorical cross-entropy loss. The final CNN architecture is then tested on the test data, and the accuracy is the main metric used to compare the different algorithms.

IV. Results

The accuracies obtained on the CIFAR-10 dataset for each method that we tested are shown below in the table below.

Method	**Accuracy**
GEP with Crossover Mutation	73.16%
GEP with Crossover Mutation and Dropout	78.62%
GEP with $(1 + 2)$ EA	78.18%
GEP with $(1 + 2)$ EA and Dropout	80.97%
CGP-CNN	75.22%

In Figure 4, we see that for both genetic algorithms, adding Dropout starkly increases performance. Due to the computational burden associated with CGP CNN, we could not obtain enough information to confidently conclude if our proposed algorithms performed better. However, initial evidence does suggest our algorithms perform at least competitively with CGP CNN despite having lesser flexibility and a more simplified architecture.

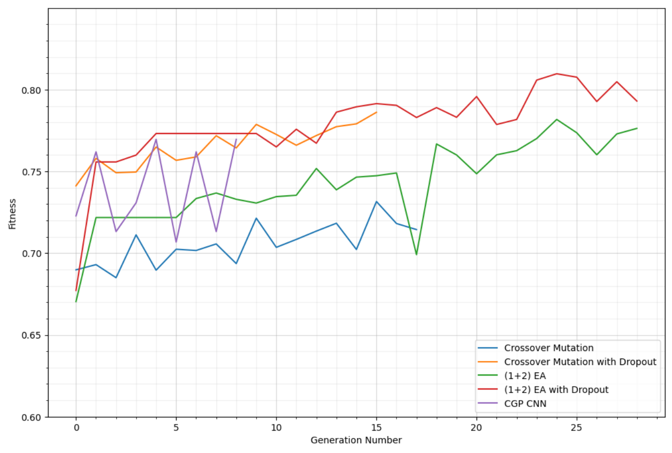


Fig. 4. Maximum accuracy across various algorithms.

V. Future Work

Although we got some promising initial results, there are a lot of places that can see improvement. Firstly, we were not able to implement the CGP algorithm to its full extent due to computational problems. For example, we trained CNN architectures for 40 epochs in the CGP algorithm as opposed to the 50 that the authors used. We were also only able to run the algorithm for 9 generations as opposed to the 100 that the authors used. As a result, we were not able to recreate the baseline performance of 93.25% stated in [5].

There is also a lot of room for improvement in the GEP algorithm. As with CGP, we were only able to train each architecture for 10 epochs and run the algorithm for less than 30 generations. We have so far only used two kinds of layers. Our future work could focus on integrating more highly functional blocks like the CGP algorithm. We also plan to experiment with different values of λ in the evolutionary strategy. Moreover, the function set for GEP is currently restricted to just $\{\cap, \cup\}$. It might be worthwhile to investigate how using some more sophisticated ways to combine layers can improve our performance.

Overall, GEP proves to be a promising avenue of methods for designing CNN architectures, and with the right tuning of parameters, can provide a robust and flexible way of finding near-optimal CNN architectures for a variety of tasks.

References

[1] Y. Bengio, "Gradient-based optimization of hyperparameters," *Neural Computation*, vol. 12, no. 8, pp. 1889–1900, 2000.

[2] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012.

[3] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[4] J. D. Schaffer, L. D. Whitley, and L. J. Eshelman, “Combinations of genetic algorithms and neural networks: a survey of the state of the art,” *[Proceedings] COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 1–37, 1992.

[5] M. Suganuma, S. Shirakawa, and T. Nagao, “A genetic programming approach to designing convolutional neural network architectures,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO ’17, (New York, NY, USA), p. 497–504, Association for Computing Machinery, 2017.

[6] C. Ferreira, “Gene expression programming: a new adaptive algorithm for solving problems,” 2001.

[7] D. Song, X. Yuan, Q. Li, J. Zhang, M. Sun, X. Fu, and L. Yang, “Intrusion detection model using gene expression programming to optimize parameters of convolutional neural network for energy internet,” *Applied Soft Computing*, vol. 134, p. 109960, 2023.

[8] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[9] T. Jansen, K. A. D. Jong, and I. Wegener, “On the Choice of the Offspring Population Size in Evolutionary Algorithms,” *Evolutionary Computation*, vol. 13, pp. 413–440, 12 2005.

[10] B. Doerr, C. Gießen, C. Witt, and J. Yang, “The (1+λ) evolutionary algorithm with self-adjusting mutation rate,” *CoRR*, vol. abs/1704.02191, 2017.

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[12] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2017.