CSE 847 Project Intermediate Report
Neural Architecture Search using Evolutionary Optimization Algorithms

Submitted by: Ritam Guha (MSU ID: guharita)

Date: March 22, 2021

1 Introduction

Deep learning algorithms have made immense progress over the years in various domains like image classification, speech processing, language translation, fraud detection, virtual assistance and much more. The backbone of these deep learning models are the novel architectures which are proposed by multiple researchers working in the domains for many years. With years of experience, they are able to identify the architectures which are suitable for particular domains. This kind of approach has two major issues:

1. Proposing a novel architecture in a domain requires years of expertise in the domain which makes it extremely time consuming process. Even after getting a blue print of an architecture, properly optimizing it and customizing it for different problems in the same domain requires hige computational processing. So, it takes a lot of time to come up with an appropriate architecure manually designed by domain experts.

2. Due to the human intervention in the process, it becomes prone to a lot of errors.

The most reasonable way to get rid of these problems is to automate the process of generating a neural architecture customized for a problem in a domain which is known as Neural Architecture Search (NAS). It is not an easy task because the architecture space is unbounded and it is practically impossible to search all the possible architectures to find the best architecture for a problem. So, to make the NAS feasbile, domain expertise is used to restrict the search space and find reasonable architectures using available computational resources within a limited time bound. The NAS methods can be cateogorized using three different components which are described below:

- **Search Space:** The search space of ideal NAS is unbounded but practically infeasible. Here domain expertise is used to reduce the search space so that the researchers are able to find appropriate architectures using limited resources and in reasonable time. Restrictions on components like the number of vertices, number of operations, types of operations etc. may reduce the search space a lot because the size grows exponentially with an increase in these factors. People working in the domains for a long time can propose restrictions on these factors providing a starting point for the search.

- **Search Strategy:** After defining the search space, we need a process to iteratively search through the space to find a suitable architecture. Search strategy is really important to reduce the time it takes to find an appropriate architecture on the defined search space. It also requires a balance in the exploration and exploitation procedures as we want to find an architecture as quickly as possible without facing a premature convergence.

- **Performance Estimation:** In order to proceed with the search strategy and to know if our goal is met, we need a performance metric to evaluate the architectures found in the process and to compare two separate architectures. The simplest option is to train the architecture on the training data and compute the validation accuracy which can be then used as a performance metric. But it is computationally expensive and requires a lot of time which significantly restricts the number architectures that can be evaluated. Researchers are trying to find other ways to approximate the performace of the architectures like lower fidelity estimates [1] (train the architecture partially and use that for computing valdiation accuracy), learning curve extrapolation [2] (after some epochs, stop the learning and extrapolate the learning curve to get an estimate), weight ineritance [3] (Instead of starting from scratch, use some pre-trained weights to speed up the learning process) etc.

So, from the above discussion, we can see that each of these components of NAS can be considered to be huge research topics and requires proper analysis to speed up the NAS process. [4] summarized the process of NAS using Figure 1.

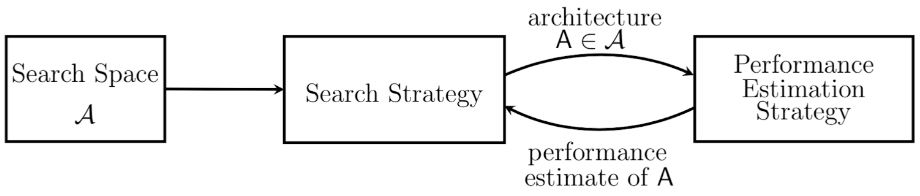
Search Space
$\mathcal{A}$
Search Strategy
architecture
$\mathsf{A} \in \mathcal{A}$
Performance Estimation Strategy
performance estimate of A

Figure 1: Pictorial representation of the Neural Architecture Search Process

From this introduction, it becomes clear that NAS is a tough process and requires proper exploration-exploitation trade-off to reach to a proper solution. Evolutionary Computation can provide us a sophisticated way to search for architectures in an iterative fashion. As a part of this project, I am using a famous evolutionary apporach called Genetic Algorithm (GA) [5] as the search strategy to explore a pre-defined search space and find the global optimum solution. The main concepts of evolutionary optimization which help us derive an appropriate neural architecture are:

- It uses a population of candidate solutions where each candidate solution is a valid architecture in the search space. Due to the introduction of a population, instead of a single solution, it is way better at finding the global optimum instead of getting stuck at some local optimum.

- It uses evolutionary operators like crossover, mutation, selection which helps to generate new and better solutions from the existing pool of solutions. Although it does not gurantee to find the global optimum, it can provide us a near-optimal solutions within a reasonable time duration.

2 Dataset

The process of NAS is extremely time consuming and computationally expensive. So, it is almost infeasible if there is some restriction on the computational resources. But, in 2019, Ying et. al created a benchmark called NAS-Bench-101 [6] which reduced the computational expense of performing NAS on the proposed search space to a large extent.

NAS-Bench-101 is a table mapping a set of architectures in a proposed search space to various metrics like training accuracy, validation accuracy, training time etc. So, the cost of performance evaluation becomes almost negligible. After coming up with an architecture, it can be evaluated just by a query to this table.

2.1 Search Space

NAS-Bench-101 restricts the search to cells which are small feedforward networks acting as building blocks of bigger architectures. Each cell is stacked 3 times followed by a downsampling layer. This pattern is repeated thrice, followed by global avg pooling and final dense softmax layer.

The entire search space contains all possible architecures formed by Directed Acyclic Graphs (DAGs) on V nodes and each node has L possibilities in terms of the type of operation. The allowed operations are 3×3 convolution, 1×1 convolution, 3×3 max-pooling and the maximum size of V is 7 while maximum number of edges is 9. Out of the 7 vertices, 2 are IN and OUT vertices while rest are intermediate vertices.

There maximum number of edges in a graph with 7 vertices is $\binom{7}{2}$=21 and every vertex has 3 possible operations. The maximum number of architectures on such a graph is: $2^{21} \times 3^5 = 510M(approx.)$. But after considering all the restrictions and removing duplications, the space contains $423K$ unique architectures.

2.2 Training Information

Each of the architecture are trained and evaluated on CIFAR-10 dataset. Every architecture has been run for 3 times and using 4 epoch schedules $(4, 12, 36, 108)$. So, for every architecture, there are 12 mappings altogether which gives $423K \times 12 = 5M(approx.)$ models.

2.3 Metrics

Every entry in the table contains 5 evaluation metrics: training accuracy, validation accuracy, testing accuracy, training time in seconds, number of trainable model parameters.

2.4 Complexity in Search

After proper examination of the search space, the authors found out that there are only 11570 models which can be considered as elites among these $510M$ models. So, the probability of hitting one of these elite architectures is 1 to 50000 which makes it a really difficult task.

3 Project Progress

For the project, I have used NAS-Bench-101 as the dataset and formulated a GA to apply over it. For formalizing GA, we need to first specify three very important things: Solution Representation, Fitness Function and Position Update.

- **Solution Representation:** Each candidate solution for NAS-Bench-101 is represented as a (matrix, list) pair. Each matrix is upper triangular binary in nature and it represents a 7-vertex DAG where 2 vertices are for INPUT and OUTPUT while the other 5 vertices are for intermediate operations. The list contains the type of operations to be used. For example, consider the following architecture:

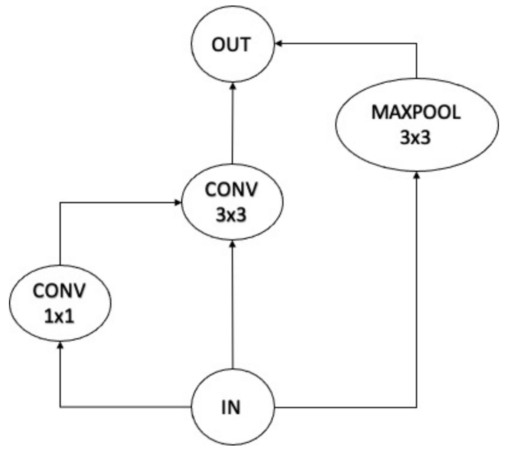

Figure 2: Example of a valid architecture in NAS-Bench-101

This architecture can be represented as a pair of matrix and list as:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

, list=[IN, Conv1x1, Conv3x3, Maxpool3x3, Conv1x1, Cov1x1, OUT
]

Please note that the final 2 Conv1x1 operations have no meaning in the context. They are added to the list to make the sizes of the matrices and lists the same which allows us to apply evolutionary operators like crossover over the architecture represenetations. The corresponding 5^{th} and 6^{th} rows of the matrix are seen to be all 0s which means they don't hold any meaning in the architecture.

- **Fitness Function:** As a fitness function, I have used the validation accuracy for the architectures. To evaluate every candidate solution, the validation accuracy can be obtained just by querying the dataset using the solution representation as the parameter.

- **Position Update:** For position update, I have used three important evolutionary operators: tournament selection, crossover and mutation.

  Tournament Selection: It is a process for selecting parents in the population of candidate solutions. From the population, k random solutions are selected and are said to participate in a tournament. The best solutions out of those k solution in terms of fitness scores are the winners of the tournament and are eligible to become parents.

  Crossover: After the parents are selected, they undergo crossover which is the process of crearting child solutions by intermixing information from multiple parent solutions. For performing crossover, I am selecting a crossover point and the first child is created by taking the operations and corresponding columns of parent 1 till the crossover point and the same things after crossover point from parent 2. In a similar way, the second child is also created by swapping the parent numbers for the first child creation process.

  Mutation: The child solutions created from crossover are mutated by randomly changing the values for the matrices and operators. Sometimes this leads to invalid representations. The process is continued till a valid representation is found.

3.1 Code

The code for the GA implementation is provided below:

```
# !curl -O https://storage.googleapis.com/nasbench/nasbench_only108.tfrecord
# !git clone https://github.com/google-research/nasbench
# !pip install ./nasbench

# Initialize the NASBench object which parses the raw data into memory (this
# should only be run once as it takes up to a few minutes).
from nasbench import api

# Use nasbench_full.tfrecord for full dataset (run download command above).
nasbench = api.NASBench('nasbench_only108.tfrecord')

# Standard imports
import copy
import numpy as np
import matplotlib.pyplot as plt
import random

# Useful constants for NAS-Bench-101
INPUT = 'input'
OUTPUT = 'output'
CONV3X3 = 'conv3x3-bn-relu'
```

```python
CONV1X1 = 'conv1x1-bn-relu'
MAXPOOL3X3 = 'maxpool3x3'
NUM_VERTICES = 7
MAX_EDGES = 9
EDGE_SPOTS = NUM_VERTICES * (NUM_VERTICES - 1) / 2   # Upper triangular matrix
OP_SPOTS = NUM_VERTICES - 2   # Input/output vertices are fixed
ALLOWED_OPS = [CONV3X3, CONV1X1, MAXPOOL3X3]
ALLOWED_EDGES = [0, 1]   # Binary adjacency matrix

"""## Basic usage"""

# Query a cell from the dataset.
cell = api.ModelSpec(
  matrix=[[0, 1, 1, 1, 0, 1, 0],    # input layer
          [0, 0, 0, 0, 0, 0, 1],    # 1x1 conv
          [0, 0, 0, 0, 0, 0, 1],    # 3x3 conv
          [0, 0, 0, 0, 1, 0, 0],    # 5x5 conv (replaced by two 3x3's)
          [0, 0, 0, 0, 0, 0, 1],    # 5x5 conv (replaced by two 3x3's)
          [0, 0, 0, 0, 0, 0, 1],    # 3x3 max-pool
          [0, 0, 0, 0, 0, 0, 0]],   # output layer
  # Operations at the vertices of the module, matches order of matrix.
  ops=[INPUT, CONV1X1, CONV3X3, CONV3X3, CONV3X3, MAXPOOL3X3, OUTPUT])

# Querying multiple times may yield different results. Each cell is evaluated 3
# times at each epoch budget and querying will sample one randomly.
data = nasbench.query(cell)
for k, v in data.items():
  print('%s: %s' % (k, str(v)))

def fitness(solution):
  # Fitness metric for GA
  spec = api.ModelSpec(matrix=solution.matrix, ops=solution.ops)
  if nasbench.is_valid(spec):
    metrics = nasbench.query(spec)
    return metrics['validation_accuracy']
  else:
    return 0

def call_counter(fn):
    # hierarchical function to count number of evaluation calls
    def helper(*args, **kwargs):
        helper.calls += 1
        cur_val = fn(*args, **kwargs)
        if(cur_val > helper.best_val):
          helper.best_val = cur_val
          print('Best Val:{}, Func Eval:{}'.format(helper.best_val, helper.calls))

        return cur_val
    helper.__name__ = fn.__name__
    helper.calls = 0
    helper.best_val = float('-inf')
    return helper

# Associating the function counter with the fitness function
fitness = call_counter(fitness)


class Candidate():
  # Class for representing every candidate solution
  def __init__(self, matrix, ops):
    self.matrix = matrix
    self.ops = ops


def mutation(solution, mutation_rate=1.0):
  # function for performing mutation
  while True:
    new_matrix = solution.matrix.copy()
    new_ops = solution.ops.copy()

    # In expectation, V edges flipped (note that most end up being pruned).
    edge_mutation_prob = mutation_rate / NUM_VERTICES
    for src in range(0, NUM_VERTICES - 1):
      for dst in range(src + 1, NUM_VERTICES):
        if random.random() < edge_mutation_prob:
          new_matrix[src, dst] = 1 - new_matrix[src, dst]

    # In expectation, one op is resampled.
    op_mutation_prob = mutation_rate / OP_SPOTS
    for ind in range(1, NUM_VERTICES - 1):
      if random.random() < op_mutation_prob:
        available = [o for o in nasbench.config['available_ops'] if o != new_ops[ind]]
        new_ops[ind] = random.choice(available)

    new_spec = api.ModelSpec(new_matrix, new_ops)
    if nasbench.is_valid(new_spec):
      return Candidate(new_matrix, new_ops)


```

```
def crossover(parent1, parent2):
  # function for performing crossover over two parent chromosomes
  child1_mat = np.zeros((NUM_VERTICES, NUM_VERTICES),dtype=int)
  child1_ops = []
  child2_mat = np.zeros((NUM_VERTICES, NUM_VERTICES),dtype=int)
  child2_ops = []
  cross_point = np.random.randint(NUM_VERTICES-2)+1

  for i in range(cross_point):
    child1_ops.append(parent1.ops[i])
    child1_mat[:,i] = parent1.matrix[:,i]
    child2_ops.append(parent2.ops[i])
    child2_mat[:,i] = parent2.matrix[:,i]

  for i in range(cross_point, NUM_VERTICES):
    child1_ops.append(parent2.ops[i])
    child1_mat[:,i] = parent2.matrix[:,i]
    child2_ops.append(parent1.ops[i])
    child2_mat[:,i] = parent1.matrix[:,i]

  child1 = Candidate(child1_mat, child1_ops)
  child2 = Candidate(child2_mat, child2_ops)
  return child1, child2


def selection(population, objective):
    # function to perform tournament selection
    K = 5
    num_pop = len(population)
    perm = np.random.permutation(num_pop)
    pop_comb = perm[0:K]
    tournament_fit = np.zeros(K)

    # creating tournament population
    for i in range(K):
        tournament_fit[i] = objective[pop_comb[i]]

    # declaring winners
    idx = np.argsort(tournament_fit)
    parent_id1 = idx[0]
    parent_id2 = idx[1]
    return parent_id1, parent_id2


def initialization(pop_size):
  # function for initialization
    population = []
    for pop_no in range(pop_size):
      matrix = np.random.choice(ALLOWED_EDGES, size=(NUM_VERTICES, NUM_VERTICES))
      matrix = np.triu(matrix, 1)
      ops = np.random.choice(ALLOWED_OPS, size=(NUM_VERTICES)).tolist()
      ops[0] = INPUT
      ops[-1] = OUTPUT
      population.append(Candidate(matrix, ops))

    return population

def GA(pop_size=50, num_gen=100, cross_mut_prob=0.5):
  # main driver function for GA
  population = initialization(pop_size)
  obj_values = np.zeros(pop_size)
  avg_values = np.zeros(num_gen)

  for pop_no in range(pop_size):
    # calculate fitness for all the chromosomes
    obj_values[pop_no] = fitness(population[pop_no])

  for gen_no in range(num_gen):
    avg_values[gen_no] = np.mean(obj_values)

    for i in range(max_cross):
      if(np.random.rand()<cross_mut_prob):

        # parent selection
        parent_id1, parent_id2 = selection(population, obj_values)

        # crossover
        child_1, child_2 = crossover(population[parent_id1], population[parent_id2])

        # mutation
        child_1 = mutation(child_1)
        child_2 = mutation(child_2)

        # child fitness computation
        obj_1 = fitness(child_1)
        obj_2 = fitness(child_2)

        # child replaces the worst solution if applicable
        if(obj_1 > min(obj_values)):
            idx = np.argmin(obj_values)
            population[idx] = child_1
```

```
                obj_values[idx] = obj_1

            if(obj_2 > min(obj_values)):
                idx = np.argmin(obj_values)
                population[idx] = child_2
                obj_values[idx] = obj_2


# calling GA
GA()
```

3.2 Experimental Results

After implementing the code, it was tested on NAS-Bench-101 dataset. The generation-wise convergence graph for the implementation is presented in Figure 3. The Figure represents the situation for 30 independent runs of GA. At each run, we can see that the behavior of the runs are almost the same with little to no variance. It denotes that the GA implementation is pretty robust for NAS-Bench-101. The main goal of the optimization algorithm is to improve the average value over the generations. From the trend of the fitness scores, it can be observed that the GA implementation is working great over NAS-Bench-101 dataset.

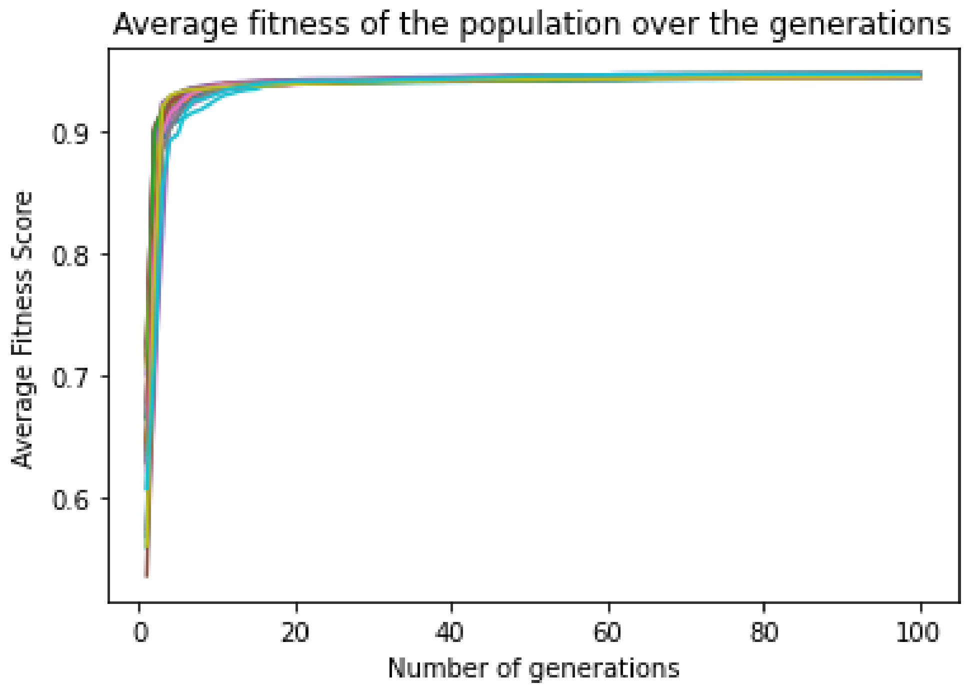

Figure 3: Convergence of the average fitness score over the generations

4 Future Plan

From the results, I noticed that although performing NAS over Bench-101 is difficult, it becomes really easy for GA. Also, NAS-Bench-101 only uses CIFAR dataset which makes it more specific.

So, in the next part of the project, I want use the same GA implementation over other benchamrks like NAS-Bench-201 for testing the generalization capability of this implementation for different Benchmarks and different datasets. If possible, I want to test and compare my GA implementation with other search procedures like Random Search [7], Regularized Evolution [8] etc.

References

[1] Lisha Li, Kevin G Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *ICLR (Poster)*, 2017.

[2] Kevin Swersky, Jasper Snoek, and Ryan Prescott Adams. Freeze-thaw bayesian optimization. *arXiv preprint arXiv:1406.3896*, 2014.

[3] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017.

[4] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55):1–21, 2019.

[5] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[6] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[8] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.