# NOVEL PARADIGMS FOR NEURAL ARCHITECTURE SEARCH IN THE HILL CLIMBING DOMAIN

# A PROJECT REPORT

# Submitted in partial fulfilment of the requirements

# for the award of the degree of

BACHELOR OF TECHNOLOGY

In

**INFORMATION TECHNOLOGY**

Submitted By:

**Apoorva Verma**

(2K15/IT/016)

**Karan Goyal**

(2K15/IT/036)

**Mudit Verma**

(2K15/IT/044)

**Pradyumna Sinha**

(2K15/IT/53)

Under the supervision of

**Dr. SEBA SUSAN**



# INFORMATION TECHNOLOGY

# DELHI TECHNOLOGICAL UNIVERSITY

## (Formerly Delhi College of Engineering)

## Bawana Road, Delhi-110042

DECEMBER, 2018

# INFORMATION TECHNOLOGY

# DELHI TECHNOLOGICAL UNIVERSITY

## (Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

**DECLARATION**

# We Apoorva Verma (2K15/IT/016), Karan Goyal (2K15/IT/036), Mudit Verma (2K15/IT/044) and Pradyumna Sinha (2K15/IT/053), students of B. Tech (Information Technology) hereby declare that the Project Dissertation titled “NOVEL PARADIGMS FOR NEURAL ARCHITECTURE SEARCH IN THE HILL CLIMBING DOMAIN” which is submitted by us to the Department of Information Technology, Delhi in partial fulfilment of the requirement for the award of degree of Bachelor of Technology, is original and has not been copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma, Associateship, Fellowship or any similar title or recognition.

**Place: Delhi** Apoorva Verma 2K15/IT/016

Karan Goyal 2K15/IT/036

**Date: 7th December 2018** Mudit Verma 2K15/IT/044

Pradyumna Sinha 2K15/IT/053

# INFORMATION TECHNOLOGY

# DELHI TECHNOLOGICAL UNIVERSITY

## (Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

**CERTIFICATE**

This is to certify that the Project Dissertation titled “Neural Architecture Search”which is submitted by **Apoorva Verma (2K15/IT/016), Karan Goyal (2K15/IT/036), Mudit Verma (2K15/IT/044) and Pradyumna Sinha (2K15/IT/053)** Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of degree of Bachelor of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any degree or diploma to this university or elsewhere.

**Place: Delhi Dr. SEBA SUSAN**

Associate Professor

Department of Information Technology

**Date: 7th December 2018** Delhi Technological University

**ACKNOWLEDGEMENT**

It is a pleasure for us to add heartfelt words for the people who were a part of this project in numerous ways and provided us with constant support right from the beginning of this project.

We express our deep gratitude to **Dr. Seba Susan** for helping us throughout with her helpful guidance, valuable suggestions, provoking discussions and constant supervision.

Our apologies for any oversights or shortcomings in the details provided in this report. Last but not the least we thank our family members and friends for being a constant source of encouragement throughout this period.

**Apoorva Verma (2K15/IT/016)**

**Karan Goyal (2K15/IT/036)**

**Mudit Verma (2K15/IT/044)**

**Pradyumna Sinha (2K15/IT/053)**

**ABSTRACT**

Neural networks have rapidly gained immense popularity over the past few years due to their success in numerous tasks, such as image recognition, speech recognition and machine transation.

**Neural architecture search** (NAS) is a technique that utilizes machine learning to automate the design of artificial neural networks. It involves tasks like model selection, hyper parameter optimization, etc. To be precise, it usually involves learning something like a layer (often called a “cell”) that can be assembled as a stack of repeated cells to create a neural network.

In this project, we have used a Hill Climbing approach that at each step applies a set of Network Morphisms to reduce the size of search space for available networks and then selects the most promising child network which is further optimized by cosine annealing. The best child network of the last iteration serves as a pre-trained network for other models.

# TABLE OF CONTENTS

**1. Introduction**………..………..………..………..………….12

1.1 About

1.2 Problem Statement

1.3 NAS Example

1.4 Inspiration

**2. Background**………..………..………..………..………….14

**3. Related Work**………..………..………..………..……......15

3.1 NASNET

3.2 Efficient NAS

3.3 NAS Hill Climbing

3.4 Neural Architect

3.5 Differentiable Architecture Search

**4. Overview**………..………..………..………..……………..17

4.1 Network Morphism

4.2 NASH Algorithm

**5. Proposed Work**………..………..………..……………….20

5.1 NAS Graph

5.2 NAS Graph Operations

5.2.1 Skip

5.2.2 Deepen

5.2.3 Widen

5.2.4 Maxpool

5.2.5 Add

5.2.6 Merge

5.3 Implementation Improvements

5.3.1 Padding

5.3.2 No Need for Seed Architecture

5.3.3 Addition of Maxpool

5.3.4 Linear Layer Morphism

5.3.4.1 Deepening the Linear Layer

5.3.4.2 Widening the Linear Layer

5.4 Theoretical Contribution

5.4.1 Learning Curve Prediction

5.4.2 Probabilistic Operation Selection

5.4.3 Gradient Stopping

5.4.4 Gradient Starvation

**6. Experiment**………..………..………..………..………….....33

6.1 Baseline

6.2 Retraining from Scratch

6.3 Comparison to Hand-Crafted and Other

Automatically Generated Architectures

6.4 MNIST Dataset

**7. Conclusion**………..………..………..………..……………..37

**8. Future Work**………..………..………..………..…………..38

**9. References**………..………..………..………..……………..39

**10. Appendix**………..………..………..………..………..……41

10.1 Final Neural Architecture

**LIST OF FIGURES**

**Figure 1**. Visualizations of different operations used in NAS Graph.

**Figure 2.**An example of Learning Curve Prediction in action.

**Figure 3.**Visualization of Gradient Stopping for THRESHOLD = 2

**Figure 4.**Gradient Starvation

**Figure 5.**Sample Architecture generated by our NASGraph

**Figure 6.**Architecture generated by NASH

**Figure 7.**Initial Architecture used by NASH

**LIST OF TABLES**

**Table 1.** Baseline experiments.

**Table 2.** Retraining from Scratch

**Table 3.** Comparison between various neural architecture search algorithm run

on CIFAR-10 dataset

**Table 4.** Comparison between various algorithm run on MNIST dataset.

**LIST OF SYMBOLS, ABBREVIATIONS**

1. **NASH –** Neural Architecture Search Hill Climbing
2. **NAS –** Neural Architecture Search
3. **CNN –** Convolutional Neural Network Search
4. **NASNet –** Neural Architecture Search Net
5. **ENAS –** Efficient Neural Architecture Search
6. **DARTS –** Differentiable Neural Architecture Search
7. **RL –** Reinforcement Learning
8. **RNN –** Recurrent Neural Network
9. **RENAS –** Reinforcement Evolutionary Neural Architecture Search

**10. FC –** Fully Connected Layer

**11. DAG –** Directed Acyclic Graph

**12 TopSort –** Topological Sorting

**13. Conv-Block** – Convolutional Block (Convolutional Layer + Batch

Normalization + ReLU)

**14. ReLU –** Rectified Linear Unit

**LIST OF ALGORITHMS**

**ALGORITHM 1**. Network Architecture Search by Hill Climbing

**ALGORITHM 2**. Skip Operation

**ALGORITHM 3.** Widen Morphism

**ALGORITHM 4.** Deepen Morphism

**ALGORITHM 5.** Gradient Stopping

**ALGORITHM 6**. Gradient Starvation

**CHAPTER 1**

**INTRODUCTION**

**1.1 ABOUT**

“Neural Nets to design Neural Nets” is known as Neural Architecture Search**.** Typically, reinforcement learning [3] **or** evolutionary algorithms [6] are used to design the new neural net architectures but in this project we have used a HILL CLIMBING [1] approach based on Network Morphism [2].

**1.2 PROBLEM STATEMENT**

**Neural architecture search** (NAS) [4] is a technique that utilizes machine learning to automate the design of artificial neural networks and optimize the hyper-parameters.

Our problem is, given a neural architecture search space F, the input data X, and the cost metric Cost(·), we aim at finding an optimal neural network f∗ ∈ F with its trained parameter θf\*, which could achieve the lowest cost metric value on the given dataset X .

Mathematically, this definition is equivalent to find f∗ satisfying:

f∗ = argmin min Cost(f(X;θf)), f∈F θf   
where θf ∈ Rw(f) denotes the parameter set of network f , w(f) is the number of parameters in f.

**1.3 NAS EXAMPLE**

Neural architecture search is good for finding new architectures! Google’s AmoebaNet [9] was learned via neural architecture search, and (with the inclusion an aggressive learning schedule and changing the image size as training progresses) is now the cheapest way to train ImageNet [14] on a single machine**.**

AmoebaNet [9] was not designed with a reward function that involved the ability to scale, and so it didn’t scale as well as ResNet to multiple machines, but a neural net that scales well could potentially be learned in the future, optimized for different qualities.

**1.4 INSPIRATION**

* Designing neural nets is extremely time intensive, and requires an expertise that limits its use to a smaller community of scientists and engineers.
* Allows to discover architectures far more complicated than what humans may think to try that can further be optimized for respective goals.
* Parameter Tuning is not at all a trivial task and may take days to perfect.
* It enables the true meaning of Machine Learning, by making the machine learn the “method” (architecture) of learning itself as well.

**CHAPTER 2**

**BACKGROUND**

Neural Architecture Search [4] is an active subset of the field AutoML [10]. The term AutoML [10] has traditionally been used to describe automated methods for model selection and/or hyper-parameter optimization. These methods exist for many types of algorithms, such as random forests, gradient boosting machines, neural networks, and more. There are number of AutoML [10] libraries, the oldest of which is AUTOWEKA, which was released in 2013 and automatically chooses a model and selects hyper-parameters. AutoML [10] provides us a way to selects models and optimize hyper-parameters. It can also be useful in getting a baseline to know what level of performance is possible for a problem.

“Neural nets to design neural nets” is known as neural architecture search [4]; typically reinforcement learning [3] or evolutionary algorithms [6] are used to design the new neural net architectures. It is useful because it allows us to discover architecture far more complicated than what humans may think to try, and these architectures can be optimized for particular goals. Neural Architecture  Search is often very computational expensive.

In Reinforcement learning [3], we use a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set.

In Reinforced Evolutionary Neural Architecture Search (RENAS) [13], which is an evolutionary method with reinforced mutation for NAS. This method integrates reinforced mutation into an evolution algorithm for neural architecture exploration, in which a mutation controller to learn the effects of slight modifications and make mutation actions. The reinforced mutation controller instructs the model population to evolve efficiently in a suitable direction.

**CHAPTER 3**

**RELATED WORK**

**3.1 NASNet[8]**

Learning a model architecture directly on a large dataset is a lengthy process. NASNetaddressed this issue by transferring a building block designed for a small dataset to a larger dataset. The design was constrained to use two types of convolutional cells to return feature maps that serve two main functions when convoluting an input feature map: "Normal Cells" that return maps of the same extent (height and width) and "Reduction Cells" in which the returned feature map height and width is reduced by a factor of two. For the Reduction Cell, the initial operation applied to the cell’s inputs uses a stride of two (to reduce the height and width). The learned aspect of the design included elements such as which lower layer(s) each higher layer took as input, the transformations applied at that layer and to merge multiple outputs at each layer. In the studied example, the best convolutional layer (or "cell") was designed for the CIFAR-10 dataset and then applied to the ImageNet dataset by stacking copies of this cell, each with its own parameters. The approach yielded accuracy of 82.7% top-1 and 96.2% top-5.

**3.2 Efficient Neural Architecture Search (ENAS)[5]**

In ENAS [5], controller discovers neural network architectures by learning to search   for an optimal subgraph within a large graph. The controller is trained with policy gradient to select a subgraph that maximizes the validation set's expected reward. The model corresponding to the subgraph is trained to minimize a canonical cross entropy loss. Multiple child models share parameters, ENAS [5] requires fewer GPU-hours than other approaches and 1000-fold less than "standard" NAS.On CIFAR-10, the ENAS design achieved a test error of 2.89%, comparable to NASNet.On Penn Treebank, the ENAS design reached test perplexity of 55.8

**3.3 Hill Climbing [1]**

A Hill climbing [1] procedure that applies network morphisms [2], followed by short cosine-annealing optimization runs. Surprisingly, The approach yielded competitive results, requiring resources on the same order of magnitude as training a single network. On CIFAR-10, the method designed and trained a network with an error rate below 6% in 12 hours on a single GPU.

**3.4 Neural Architect [9]**

Neural Architect [9] is claimed to be a resource-aware multi-objective reinforcement learning based NAS with network embedding and performance prediction. Network embedding encodes an existing network to a trainable embedding vector. Based on the embedding, a controller network generates transformations of the target network. A multi-objective reward function considers network accuracy, computational resource and training time. The reward is predicted by multiple performance simulation networks that are pre-trained or co-trained with the controller network. The controller network is trained via policy gradient. Following a modification, the resulting candidate network is submitted to both an accuracy network and a training time network. The results of each are combined by a reward engine that passes its output back to the controller network.

**3.5 Differentiable Architecture Search** **[7]**

It addresses the scalability challenge of architecture search by formulating the task in a differentiable manner. Unlike conventional approaches of applying evolution [6] or reinforcement learning [3] over a discrete and non-differentiable search space, this method is based on the continuous relaxation of the architecture representation, allowing efficient search of the architecture using gradient descent.

**CHAPTER 4**

**OVERVIEW**

Neural Architecture Search by Hill climbing (NASH) [1] is a simple iterative approach

that, at each step, applies a set of alternative network morphisms to the current network, trains the resulting child networks with short optimization runs of cosine annealing, and moves to the most promising child network. NASH [1] finds and trains competitive architectures at a computational cost of the same order of magnitude as training a single network; e.g. on CIFAR-10, NASH [1] finds and trains CNNs [17] with an error rate below 6 % in roughly 12 hours on a single GPU. After one day, the error is reduced to almost 5%. Models from different stages of our algorithm can be combined to achieve an error of 4.7 % within two days on a single GPU.

**4.1 NETWORK MORPHISM:**

**Network morphism Type I ->** This morphism can be used to add a fully-connected or convolutional layer, as these layers are simply linear mappings.

**Network morphism Type II ->** The layer can be widened (i.e., increasing the number of units in a fully connected layer or the number of channels in a CNN) by this type of morphism.

**Network morphism Type III ->** Every idempotent function can simply be replaced by:



with the initialization w̃ i = w i . This trivially also holds for idempotent function without weights.

e.g., Relu.

**Network morphism Type IV ->** Every layer fwi is replaceable by i

Screenshot%202018-12-07%20at%201.28.28%20AM.png

with an arbitrary function h and Equation (1) holds if λ is initialized as 1. This morphism can be used to incorporate any function, especially any non-linearities.

We apply n network morphisms, each of them sampled uniformly at random from the following three:

• Make the network deeper, i.e., add a ”Conv-BatchNorm-Relu” block. The position where to add the block, as well as the kernel size (∈ {3, 5}), are uniformly sampled. The number of channels is chosen to be equal to the number of channels of the closest

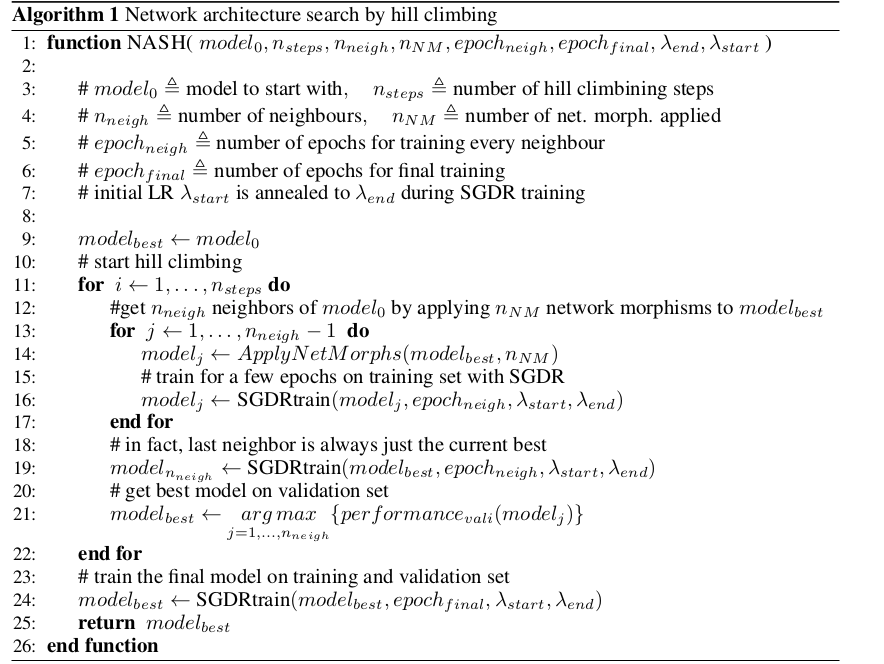
preceding convolution.

• Make the network wider, i.e., increase the number of channels by using the network morphism type II. The conv layer to be widened, as well as the widening factor (∈ {2, 4}) are sampled uniformly at random.

• Add a skip connection from layer i to layer j (either by concatenation or addition – uniformly sampled) by using network morphism type II. Layers i and j are

also sampled uniformly.

**4.2 ALGORITHM:**



**CHAPTER 5**

**PROPOSED WORK:**

This section has been divided into two subsections describing our improvements in better implementation of the NAS Hill Climbing [1] and contributions of our ideas. We propose some novel methods to improve the working of the neural architecture search method within this morphism and hill climbing domain. These techniques boost the execution time and also the improve the designed architecture. The original method incorporates no way to allow to insert padding into a convolutional layer. We incorporated this option into our framework to provide better flexibility to the architecture design. Also, we coded a possibility of allowing max-pool-blocks to be added during a child network generation. Again, this adds to the flexibility of our approach. The original work deals with morphisms being applied only to the convolutional layers. We have, in addition to it, provided functionality to allow morphisms in the linear layers as well. We have also employed learning curve prediction using an extrapolation method described in [11]. All the proposed method has been described in detail in the next section.

**5.1 NASGRAPH**

NASGraph represents an architecture in our state space of neural networks architectures for the given dataset. So, each instance of NASGraph is a state in the state space being fed to the hill climbing algorithm. NASGraph is defined by a certain set of attributes. The NASGraph is, essentially, implemented as a Graph data structure. Each node from this graph can either be a convolutional block (ConvLayer-BatchNorm-ReLU), max\_pool node or a merge node.It has an adjacency list, matrix to define the connections between the nodes. It stores a list of node identifiers and a mapping between the identifiers and the actual instances. Each node also store a list of it parents and children. The operations deepen, widen and skip have been defined on the NASGraph and during the hill climbing process, they are called to morph the parent network into a child network.

**5.2 NASGRAPH OPERATIONS**

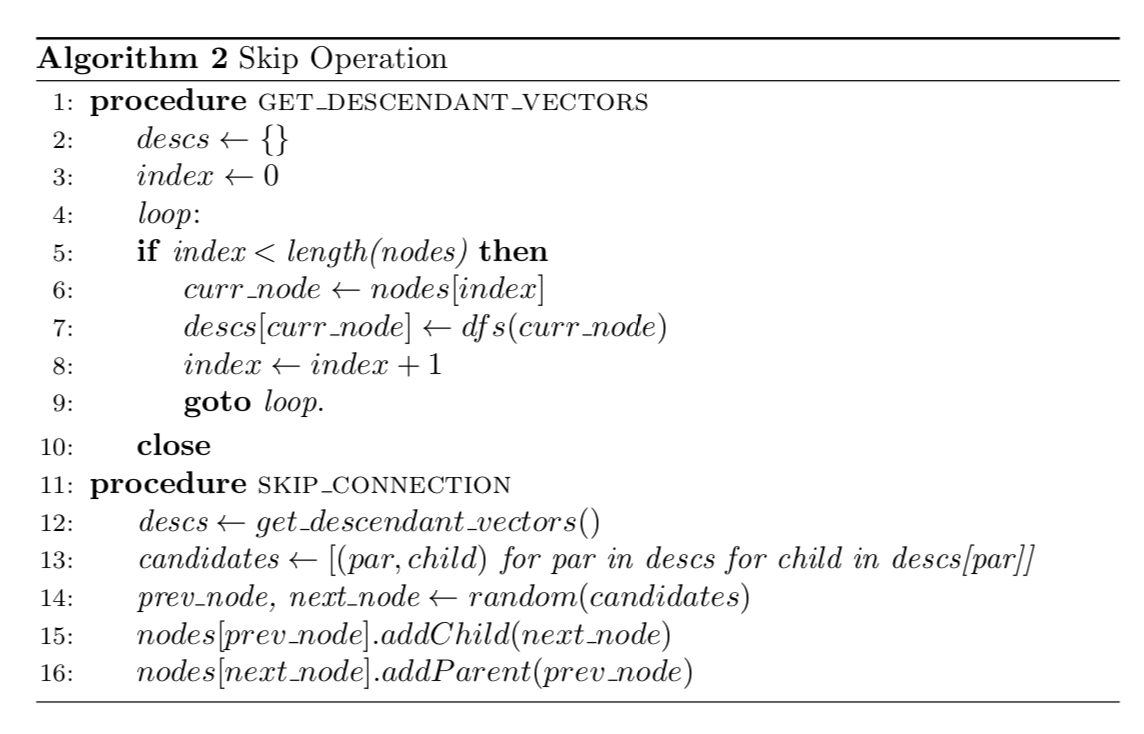
**../Downloads/Untitled%20Diagram.pdf**

**Figure 1:** Visualization of different operations used in NASGraph

Figure 1 gives a diagrammatic overview of the operations in the NASGraph. These operations have been explained in the below subsections.

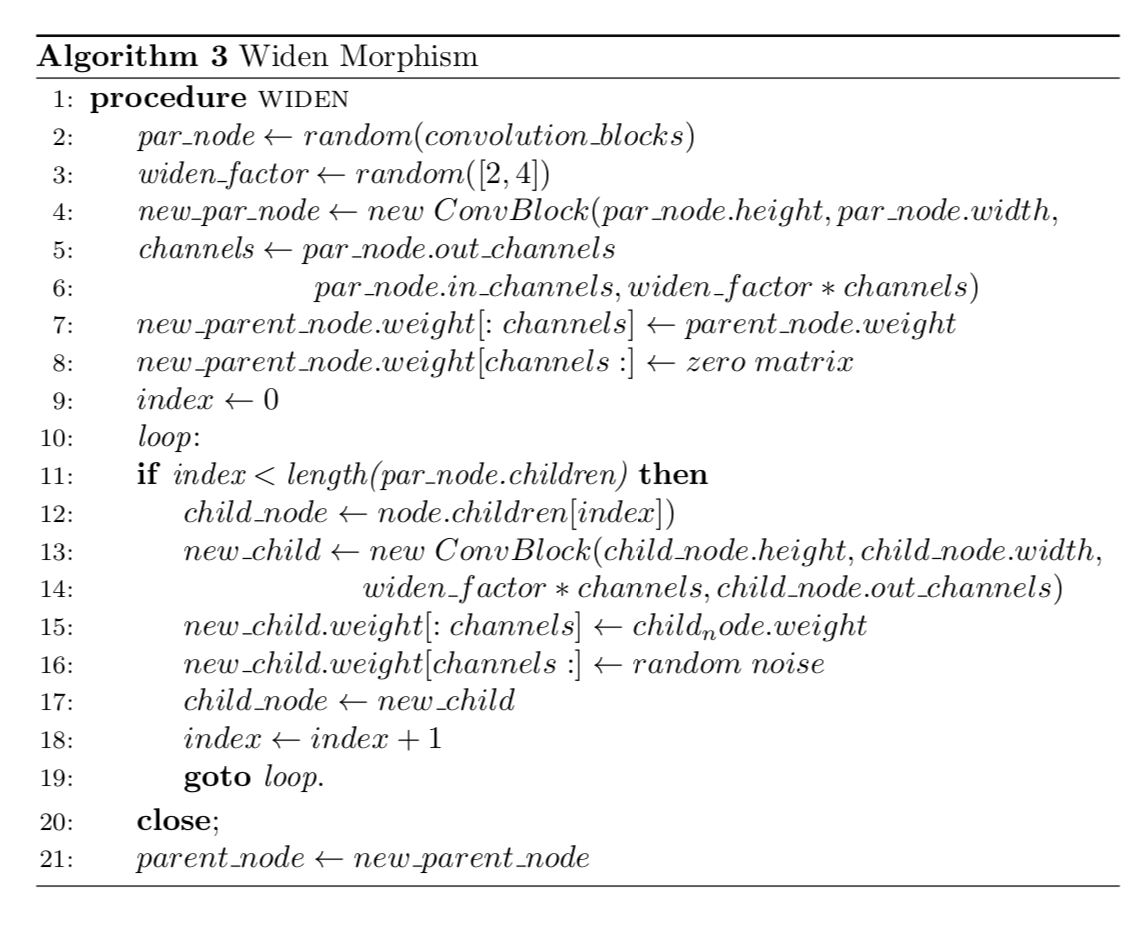
**5.2.1 Skip Operation**

Skip operation corresponds to Morphism Type - 2 The idea for performing skip is first to find all the node pairs in which the first node’s output is same as the second node’s input. This will cause the second node to have multiple parents each proving this node with same dimension inputs, therefore, we simply use an Add node, described below, to Add all such parents and then supply the combined result to this node.



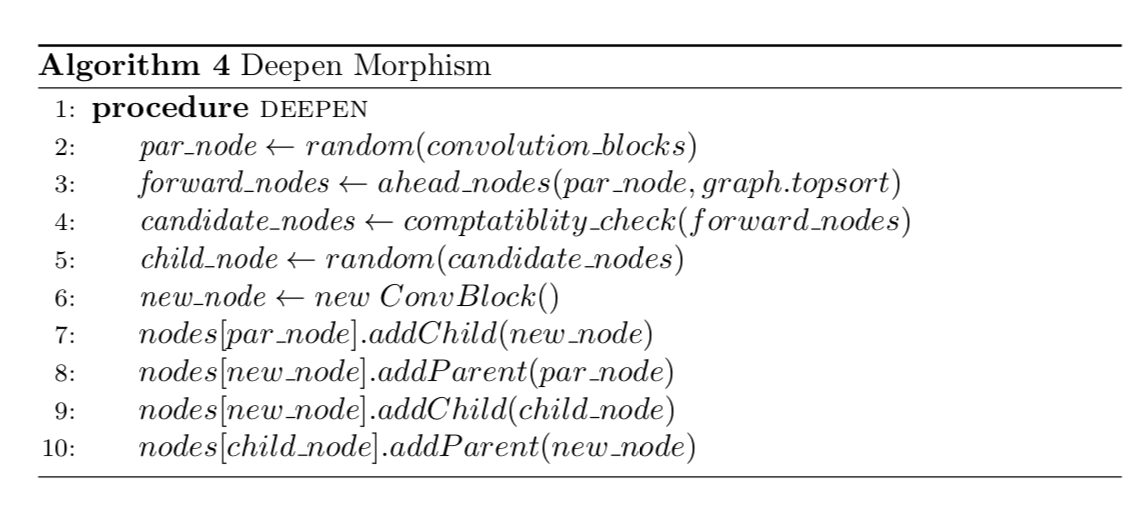
**5.2.2 Widen**

This is consequence of Morphism Type - 2 Property. Widen operation essentially increases the number of weights for a given layer, and this has been implemented for both Convolutional Blocks as well as Linear Layers. Figure1 illustrates a widening operation. The layer 2 gets widened to – 2+2’. The input from the layer 1 is, now, casted to the widened layer 2 – (2+2’). The output from the layer 2 is, then, casted to the layer3.



**5.2.3 Deepen**

Based on the Morphism Type - 1 operation, we implement the Deepen function. Deepen, as the name suggests adds a layer to deepen the existing network. This layer, in our case, is a Convolutional Block composed of a Convolutional Neural Layer, a Batch Normalization Layer, followed by a ReLU activation function. The layer may also be a Linear Layer, that is a fully connected layer. The algorithm to implement deepen is given. We first randomly select a convolution layer (for adding a Conv-Block) and then find a set of all the layers that lie ahead of the selected layer in the topological ordering of nodes, to maintain the directed acyclic graph property, and whose input is same as the output of the new node to be added had it been placed below the selected node. Now, we randomly sample a node out of this list and add the new node between the two.



**5.2.4 MaxPool**

[1] does not provide any operation to reduce the image size, however in our approach we provide two such operations, which not only are able to reduce the image sizes as well as the number of weights, but also are much popular in real life hand crafted neural network architectures, especially in the case of images. These two operations are - capability to use without padding and addition of Max-Pooling layer. Maxpool layer has been used in Image processing tasks for a long time, this motivates us for its addition to our architecture space. Again, for tackling large images, say 512x512 pixels, creating deep models without Maxpooling would cause the depth to be exorbitantly large which is not appreciated, instead by reducing the selection probability of this operation we can use this to generate neural architectures.

**5.2.5 Add**

When multiple parents are present for a single Convolution Block node in the NASGraphwe need some secondary operation to be performed to somehow ‘club’ these weights or outputs and supply to the child node. One such method is ‘Add’ operation. As implied by the name itself, we can simply add the values of all the parent nodes, given that all the parents have output dimension of the same size. Add operation is given in Figure 1.d.

**5.2.6 Merge**

This operation has similar significance to the ‘Add’ operation described in Section 5.2.5, however serves a different purpose. Merge operation can take parents nodes having outputs of different dimensions and then concatenate all of those outputs along a common dimension, usually the channel -1 dimension for 2D images as in our case.

**5.3 IMPLEMENTATIONAL IMPROVEMENTS**

[1] leaves out a lot of implementation details. In addition, we find that their implementation is rigid and limited in terms of the types of layer and selection of operations. We now present our contributions in terms of implementation advancements.

**5.3.1 Padding**

Padding is an essential concept when it comes to architecture generation or selection. It can greatly affect the feature extraction procedure in Convolutional Neural Networks, the main building block of most of Neural Architecture Search algorithms [1,3,6,8]. Padding may be detrimental to use when the image size is of the order of kernel size, and for very deep networks that our NASGraph is able to generate, the image size is likely to become very less in the lower layers. To be able to handle such cases, our implementation involves adding an option to be able to pad in a Convolutional Layer, or not. The decision is made randomly with 50% chance of happening either. Moreover, none of the operations mentioned in [1] can reduce the image size through their architecture since the padding has been applied whenever a new Conv-Block is added. This also increases the search space of our NAS approach.

**5.3.2 No need for Seed Architecture**

[1] requires a trained seed architecture to be supplied to their NASH algorithm as explained in section. This however forms a vicious cycle wherein we need to input some, possibly intelligent or well-performing, architecture and since hill climbing has been used to traverse the search space, the search reduces to mere local optimization. On the other hand, in our approach we do not require any seed architecture or layer, the only requirements are input image dimensions and the number of categorical classes.

**5.3.3 Addition of MaxPool Layer**

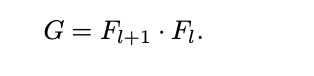
MaxPool layers have been used to reduce number of parameters in a network. [1] is able to exploit the benefits of Maxpooling only if the seed architecture sports such a layer, since there are no operations defined which can add such a layer. Our experience suggests that use of Maxpool can produce promising results, especially when input image sizes are greater than image sizes in CIFAR-10 [14] or MNIST [14] datasets. For real life applications, the input image may be much larger, for these cases [1] would produce a very deep neural network architecture if number of parameters in the architectures are to be reduced. In our approach, however, use of MaxPool can significantly reduce the parameter requirements.

**5.3.4 Linear Layer Morphism**

The network morphisms described in [2] are methods for growing the network while retaining its knowledge. The paper discusses the general case for this and also describes particular equations for the specific case of convolution neural networks. The equations have been mentioned in the (reference to previous section). The approach in [1] only applies these morphisms to the convolution blocks (deepening, widening and skip connections). Our approach, instead, also looks at the morphism in the fully connected layers.

**5.3.4.1** **Deepening the Linear Layer**

The weight matrix between two fully connected layers is decomposed into two weight matrices leading to two linear layers.



One of the weight matrices is initialized as an identity of matrix and the other one is made a copy of the initial matrix.

**5.3.4.2.** **Widening the Linear Layer**

In an architecture of 3 consecutive layers (2 weight matrices) the second layer is widened to include more number of neurons. The added weights are made zero for one of the matrices and they are filled with noise for the other matrix.

During the hill climbing method, one of the different morphisms options is chosen to generate the child node. We give assign a small probability to the outcome being ‘fully connected morphism’ in contrast to the other morphisms. This is because fully connected layers might not require too much growth and have been seen to work well with very simple 2 or 3 layered architectures. Nonetheless, enabling this morphism adds to the power of the neural architecture search technique.

**5.4 THEORETICAL CONTRIBUTIONS**

**5.4.1 Learning Curve Prediction**

**../Downloads/LCP.pdf**

**Figure 2**An example of Learning Curve Prediction in action.

In the approach being follow for the architecture search, we generate child networks from the current network and then select the network which has the best performance over a certain a number of epochs. However, this method might be misleading as it might be possible for an initial promise in the selected network to only be a mirage. Other networks might be able to perform better than the selected network if allowed to train for a larger number of epochs. Due to the enormity of the process, there is only a limited number of epochs that we can afford to train the child networks for. So, we employ a method to predict the competitiveness of a child network based on the a few initial iterations of training. This way we can also stop limit the training of child network as soon as it has been established that it would perform poorly in comparison to the other explored networks until then.

The task here is to extrapolate the initial part of the learning curve to its remainder. Let y1:n denote the initial part of the learning curve for the initial n steps. The work is to fit a model M to y1:n and use them to predict ym, with m > n. We refer to the work in (reference to extrapolation paper) for the technique here. This extrapolation can be used to discard or further explore a child node without the need to train it for a large number of epochs. Essentially, this cuts of a lot of running time in the execution of the algorithm.

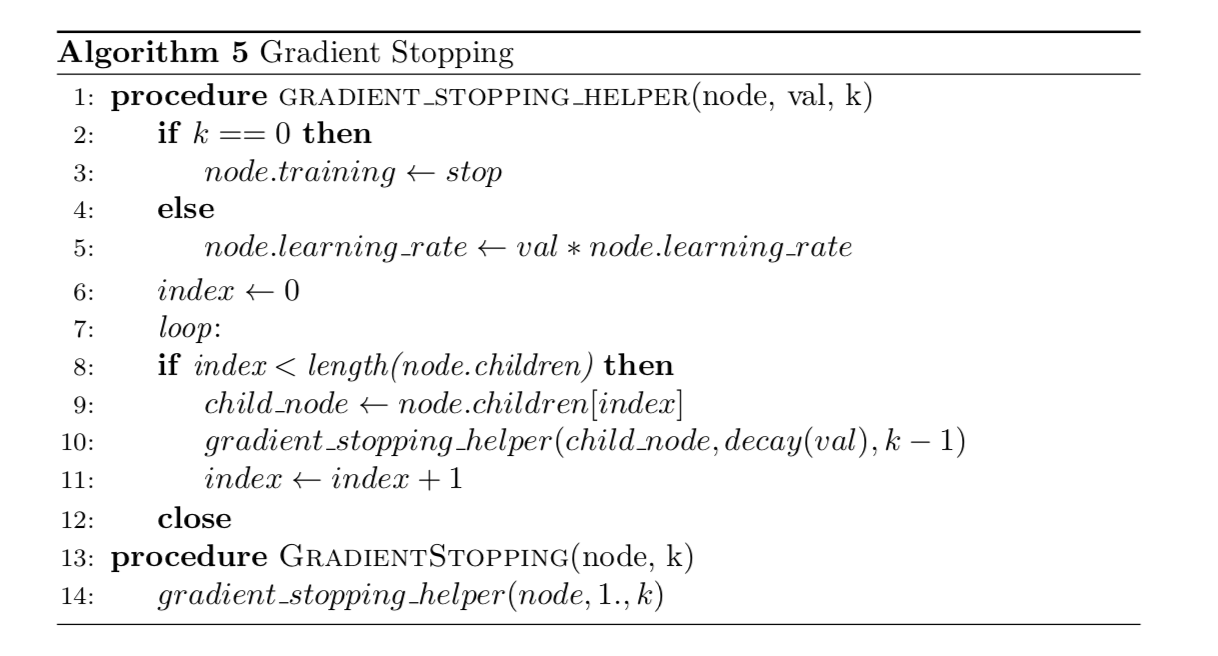
**5.4.2 Probabilistic Operation Selection**

The operations defined for our NASGraph are widen, deepen, skip and maxpool. However, a random selection of operation type is infeasible and unnecessary. For example, in the initial stages of architecture generation, when very few layers have been added via appropriate Morphism technique, it may not be possible to perform a ‘skip’ operation, this is because, for a skip operation to be performed, the output of the first selected NASGraph node must match with the input dimensions of some other node and such a connection must not result in a cycle. In the initial phases (add a diagram to support this text) such node pairs are difficult to be generated therefore recurrent calls to ‘skip operation’ is computational wastage. To curb this issue, we take the last 20 counts of the operations which have been successfully applied divided by the total number of times those operations have been requested, this value becomes the ‘score’ of each operation. Finally, all the scores are normalized so that these act as probabilities and probabilistic sampling is performed based on the newly generated probability mass function.

**5.4.3 Gradient Stopping**

**../Downloads/GradStop.pdf**

**Figure 3**Visualization of Gradient Stopping for THRESHOLD = 2

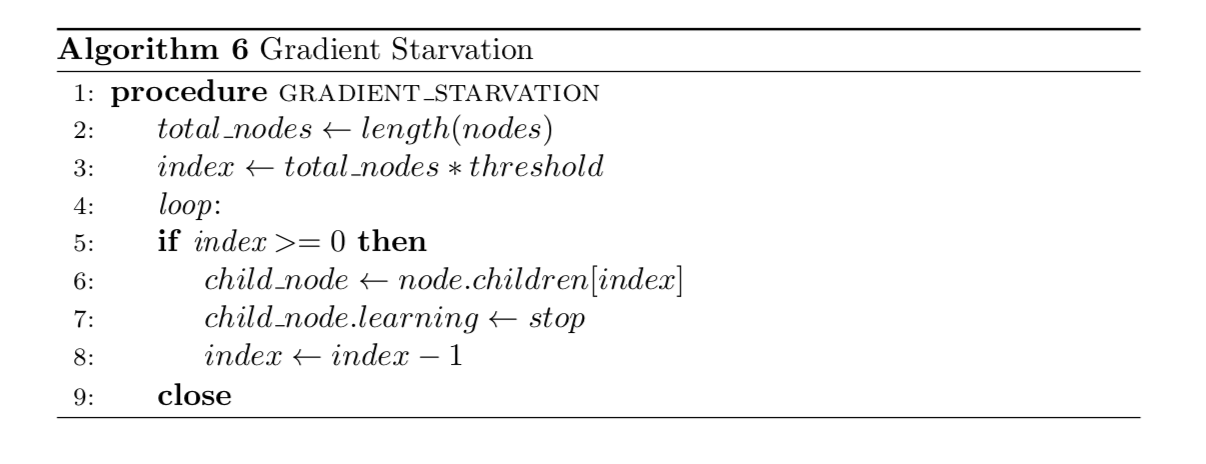


Gradient Stopping tries to achieve a similar purpose as Gradient Starvation, but in a different manner. Whenever a new node is added to the NASGraph, we only allow training of the layers ‘around’ this added layer. To achieve this, we define a THRESHOLD parameter as in the algorithm below, and then set train parameter of the layers below and above THRESHOLD ‘distance’ in the Graph as False. For ‘distance’ we can simply assume each edge weight as one in the NASGraph, and treat this as finding the nodes at a distance k in a Graph data structure. The intuition is that for a very deep neural network, training the whole of the network for an addition of a single layer, is computationally very expensive. We may explain this from the Water Drop analogy. When a drop of water falls in large pool of water, the ripples are confined to only a small region around the location where the drop fell and the far away regions of the pool remains largely unaffected. Similarly, an addition of a small layer should not affect the layers far apart from it and thus only the nearby ones may be considered for training along with the newly added node.

**5.4.4 Gradient Starvation**

**../Downloads/GradStarv.pdf**

**Figure 4**Gradient Starvation. Darker Shades of nodes will be allowed to be trained for more number of epochs, since lighter shade nodes have been trained for longer periods earlier.



For nodes which have been generated, say, in the initial morphism operations, the number of times those nodes have underwent training procedure, that is weight updating, is much higher than the newly created nodes. This difference may even be more than a hundred epochs for a small hill-climbing tree, even with few Morphism operations [2] and lesser training epochs. This causes an implicit discrimination based on the chronological order in which nodes were added to the NASGraph. Hence some of the newer nodes may ‘starve’ for a gradient descent update unlike others. To alleviate this we perform our Gradient Starvation function, which stops the training of certain layers (usually the upper layers) as they have been trained for a long time earlier. For example, if the current model needs to be trained for 20 epochs, an upper layer, A, which has been trained for 100 epochs already, would be trained only for a first few epochs out of these 20. However, for newly added nodes, they would be trained for the complete 20 epochs. The algorithm for gradient starvation is given in the Fig. This method is different from Gradient Stopping explained previously as here we strive to reduce the implicit disparity between NASGraph nodes training caused by the time of their addition to the graph, whereas in Gradient Stopping we want to train only the nearby nodes of the added new node.

**CHAPTER 6**

**EXPERIMENTS**

We evaluate our method on CIFAR-10 and CIFAR-100. First, we investigate whether our considerations from the previous chapter coincide with empirical results. We also check if the interplay of modifying and training networks harms their eventual performance. Finally, we compare our pro- posed method with other automated architecture algorithms as well as hand crafted architectures.

We use the same standard data augmentation scheme for both CIFAR datasets used by [Loshchilov & Hutter (2017)] in all of the following experiments. The training set (50.000 samples) is split up in training (40.000) and validation (10.000) set for the purpose of architecture search. Eventually the performance is evaluated on the test set. All experiments where run on Nvidia Titan X (Maxwell) GPUs, with code implemented in Keras (Chollet et al., 2015) with a TensorFlow (Abadi et al., 2015) backend.

**6.1 BASELINE**

[1] generates models based on [14]. That is, since it requires seed architectures before applying the Morphism operations [2], it has to begin with some manually crafted architecture (or randomly generated architecture). Moreover, as we have seen, [1] cannot change image sizes as it cannot add ‘new layers’ that can change the image dimensions. Hence, it is imperative that similar models are likely to be not generated by our approach. Therefore, we first implemented their scheme and trained on parameters mentioned, and then compared it with our results as in Table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Runtime (hrs)** | **Parameters (mil.)** | **Avg. Error (%)** |
| NASH Nsteps = 5, Nneigh = 8, SGDR | 13.1 | 5.7 | 6.2 |
| NASH Nsteps = 5, Nneigh = 8, No SGDR | 11.3 | 5.8 | 7.4 |
| Ours – NO LCP | 25 | 6.1 | 5.54 |
| Ours – LCP | 19 | 6.1 | 5.82 |

**Table 1:**  
Baseline experiments. Runtime, #params, and error rates are averaged over 10 runs (for nneigh = 8) and 30 runs (nneigh = 1) runs, respectively. Error rates for our method has been averaged over 20 runs.

**6.2 RETRAINING FROM SCRATCH**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Taken (hrs)** | **Error (%)** |
| Ours | 19 | 5.82 |
| Ours – Scratch | 4.8 | 4.23 |

**Table 2:**  
Comparing Our model when trained from scratch vs when trained via Morphism Operations

The above table shows a comparison between Our model when trained from scratch versus when generated via our algorithm. The table shows that had the architecture been known previously it could be comparable to the state of art Error rates.

**6.3 COMPARISON OF HAND-CRAFTED AND OTHER AUTOMATICALLY GENERATED ARCHITECTURES**

In this section we compare our results with those obtained in the original implementation of [hill climbing paper], state-of-the-art models by Gastaldi (2017) and Loshchilov & Hutter (2017) as well as other automated architecture search methods.

The proposed method can generate competitive network architectures within just 12 hours and by another 12 hours it is able to outperform most of the automated search architecture methods (though the require much more time and GPU). We are unable to perform as good as the two handcrafted architectures as well as the one found by Zoph & Le (2017). However, it is important to note that the number of resources spent by Zoph & Le (2017) are humongous in comparison to ours.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Resources Days** | **Resources GPU** | **Parameters (mil.)** | **Avg. Error (%)** |
| NASH Nsteps = 5, Nneigh = 8, SGDR | 0.5 | 1 | 5.7 | 6.20 |
| Shake-Shake | 2 | 2 | 26 | 2.91 |
| WRN 28-10 | 1 | 1 | 36.5 | 3.86 |
| Baker et al. (2016) | 8-10 | 10 | 11 | 6.93 |
| Cai et al. (2017) | 3 | 5 | 19.7 | 5.72 |
| ENAS | 7 | 800 | ? | 2.91 |
| Genetic | 20 | 2 | 16.4 | 7.10 |
| Ours - No LCP | 1 | 1 | 6.1 | 5.54 |
| Ours – LCP | 0.7 | 1 | 6.1 | 5.82 |

**Table 3:**

Comparison of various Neural Network Solutions to CIFAR-10

**6.4 MNIST DATASET**

We also ran our CIFAR-10 generated architecture with the MNIST dataset to evaluate it performance. When compared with the results of the **MCDNN (correct it if wrong)** algorithm, we were able to achieve an accuracy of about 98% which is only 1% better than ours.

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Parameters**  **(mil)** | **Error (%)** |
| MCDNN | ? | 0.23 |
| Ours | 6.1 | 2.02 |
| ENAS | ? | 1.23 |

**Table 4:**

Comparing our Architecture when generated from CIFAR-10 and trained on MNIST with other State of the Art**CHAPTER 7**

**CONCLUSION**

# We proposed “NOVEL PARADIGMS FOR NEURAL ARCHITECTURE SEARCH IN THE HILL CLIMBING DOMAIN” which is an improvement over the Hill climbing and Network Morphism based Neural Architecture Search proposed in [1]. Our method is not just simple and fast, but it also outperforms most of the existing automated architecture search methods with lesser GPU days and resources. Our experiments ran on the CIFAR-10 dataset and yielded emulous results.

In addition to the morphism techniques used by the Hill Climbing algorithm in [1] we have also proposed two more methods (Max Pooling, Linear Layer Morphism). We have further made six different kinds of improvements, namely, Padding (added an option to be able to pad in a Convolutional Layer randomly), Elimination of seed architecture ( our only requirements are input image dimensions and the number of categorical classes in the beginning of the algorithm), Learning curve prediction (to discard or further explore a child node without the need to train it for a large number of epochs), Probabilistic operation selection (associating scores with each operation of NASGraph), Gradient Stopping (only allow the training of k(threshold) layers ‘around’ the newly added layer) and Gradient Starvation (weight updation for old nodes must be comparable to the newly created ones).

**CHAPTER 8**

**FUTURE WORK**

Having mentioned the above extensions, we hope that our approach can deliver as a basis for the development of better more sophisticated techniques that yield further improvements in the existing performance. There are different ways to initialize the weight in the morphism operations. We would like to explore this further and find other literature materials that would with respect to this. We would also want to learn some parameters to intelligently choose which convolution blocks would be picked for a particular morphism operation. Currently, this process is completely random. An important criteria would be to pick convolution blocks from those regions of the network which have been having significant changes in their weights in the recent iterations. Other such criteria have to be looked at which might be able to contrast some convolution blocks from other ones. The operation selection process i.e, picking one of the morphism operation for child generation also has to be more intelligent than how it is now. This is an important aspect to be looked at that and seems to show good promise in improving the method. This method only looks at automatically generating convolutional neural networks. However, an ideal neural architecture search method should be generic and should have the ability to generate a network of any type and for any sort of dataset. This, too, has to be explored. A lot of good methods in this field are based on genetic algorithms. We would like to bring the goodness of those methods into our technique too. Hill climbing is one of the most basic algorithms for discrete optimization. Though effective in our technique, we would like to explore replace this algorithm with a more powerful method like iterative deepening A\* or simulated annealing.

**REFERENCES:**

1. **Elsken, T., Metzen, J. H., & Hutter, F. (2017). Simple and efficient architecture search for Convolutional Neural Networks. *arXiv preprint arXiv:1711.04528*.**
2. **Wei, T., Wang, C., Rui, Y., & Chen, C. W. (2016, June). Network morphism. In *International Conference on Machine Learning* (pp. 564-572).**
3. **Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.**
4. **Elsken, T., Metzen, J. H., & Hutter, F. (2018). Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*.**
5. **Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., & Dean, J. (2018). Efficient Neural Architecture Search via Parameter Sharing. *arXiv preprint arXiv:1802.03268*.**
6. **Xie, L., & Yuille, A. L. (2017, October). Genetic CNN. In *ICCV*(pp. 1388-1397).**
7. **Liu, H., Simonyan, K., & Yang, Y. (2018). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.**
8. **Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L. J., Fei-Fei, L., ... & Murphy, K. (2017). Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*.**
9. **Zhou, Y., & Diamos, G. (2018). Neural Architect: A Multi-objective Neural Architecture Search with Performance Prediction. In *Proc. Conf. SysML*.**
10. **Le, B. Z. Q., & Zoph, B. (2017). Using machine learning to explore neural network architecture. *Google Research Blog*.**
11. **Domhan, T., Springenberg, T., & Hutter, F. (2014, June). Extrapolating learning curves of deep neural networks. In *International Conference on Machine Learning AutoML Workshop*.**
12. **Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929-1958.**
13. **Chen, Y., Zhang, Q., Huang, C., Mu, L., Meng, G., & Wang, X. (2018). Reinforced Evolutionary Neural Architecture Search. *arXiv preprint arXiv:1808.00193*.**
14. **Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2017). A downsampled variant of ImageNet as an alternative to the CIFAR datasets. *arXiv preprint arXiv:1707.08819*.**
15. **Baker, B., Gupta, O., Naik, N., & Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.**
16. **Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model**

**architecture search through hypernetworks. arXiv preprint, 2017.**

1. **Xavier Gastaldi. Shake-shake regularization. ICLR 2017 Workshop, 2017.**
2. **A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter. Learning curve prediction with Bayesian**

**neural networks. In International Conference on Learning Representations (ICLR) 2017 Conference, Track, April 2017.**

1. **Snoek, H. Larochelle, and R.P. Adams. Practical Bayesian optimization of machine learning**

**algorithms. In NIPS, 2012.**

1. **Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. arXiv preprint,**

**2016.**

**21. J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization.**

**22. Y. Bengio. Practical recommendations for gradient-based training of deep architectures. In**

**G. B. Orr, and K. Müller, editors, Neural Networks: Tricks of the Trade (2nd ed.) from**

**pages 437–478. Springer, 2012.**

**23. J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-**

**Parameter Optimization. In Advances in Neural Information Processing Systems 24, 2011.**

**24. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images.**

**(2009)**

**25. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for**

**image recognition. CVPR, 2016.**

**26. Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary**

**algorithm that constructs recurrent neural networks. IEEE transactions on neural networks**

**27. James Bergstra, Dan Yamins, and David D. Cox. Making a science of model**

**search: Hyperparameter optimization in hundreds of dimensions for vision architectures.**

**In ICML, 2013.**

**28. Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture**

**search by network transformation. In Association for the Advancement of Artificial Intelligence, 2018a.**

**29. Terrance Devries and Graham W. Taylor. Improved regularization of convolutional**

**neural networks with cutout. arXiv preprint, abs/1708.04552, 2017.**

**30. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for**

**Image Recognition. In Conference on Computer Vision and Pattern Recognition, 2016.**

**31. Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely Connected Convolu-**

**tional Networks. In Conference on Computer Vision and Pattern Recognition, 2017.**

**32. F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for gen-**

**eral algorithm configuration. In LION, pages 507–523, 2011.**

**33. Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric**

**Xing. Neural Architecture Search with Bayesian Optimisation and Optimal Transport.**

**arXiv:1802.07191, February 2018.**

**34. A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter. Learning curve prediction with Bayesian**

**neural networks. In International Conference on Learning Representations, 2017a.**

**APPENDIX**

**10.1 FINAL NEURAL ARCHITECTURE**

Following gives the <NodeNumber, InputDimension, OutputDimension>

2 (4, 16, 28, 28) (4, 16, 28, 28)

1 (4, 32, 30, 30) (4, 16, 28, 28)

11 (4, 32, 30, 30) (4, 32, 30, 30)

5 (4, 16, 28, 28) (4, 16, 28, 28)

22 (4, 32, 30, 30) (4, 32, 30, 30)

21 (4, 16, 28, 28) (4, 16, 28, 28)

4 (4, 16, 28, 28) (4, 512, 26, 26)

0 (4, 3, 32, 32) (4, 32, 30, 30)

17 (4, 32, 30, 30) (4, 32, 30, 30)

3 (4, 16, 28, 28) (4, 16, 28, 28)

9 (4, 16, 28, 28) (4, 16, 28, 28)

15 (4, 16, 18, 18) (4, 512, 16, 16)

24 (4, 32, 30, 30) (4, 32, 30, 30)

12 (4, 512, 22, 22) (4, 16, 18, 18)

10 (4, 512, 26, 26) (4, 512, 26, 26)

6 (4, 512, 26, 26) (4, 512, 22, 22)

26 (4, 16, 14, 14) (4, 16, 4, 4)

20 (4, 512, 16, 16) (4, 16, 14, 14)

13 (4, 32, 30, 30) (4, 32, 30, 30)

19 (4, 32, 30, 30) (4, 32, 30, 30)

16 (4, 16, 28, 28) (4, 16, 28, 28)

18 (4, 16, 28, 28) (4, 16, 28, 28)

25 (4, 16, 28, 28) (4, 16, 28, 28)

8 (4, 16, 28, 28) (4, 16, 28, 28)

7 (4, 32, 30, 30) (4, 32, 30, 30)

23 (4, 16, 28, 28) (4, 16, 28, 28)

14 (4, 16, 28, 28) (4, 16, 28, 28)

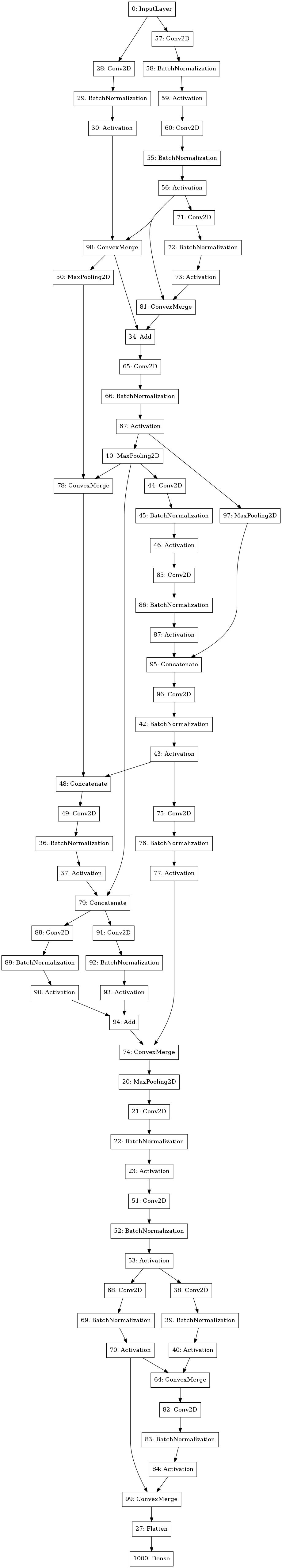
SAMPLE testing forward

INPUT: (4, 3, 32, 32)

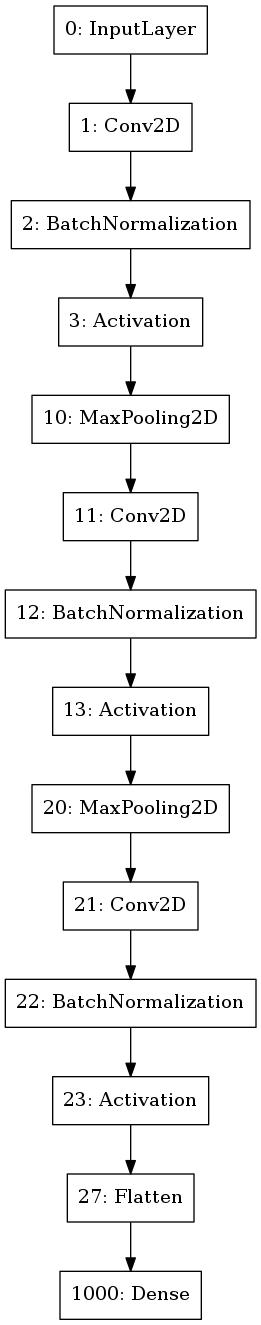
OUTPUT: (4, 16, 4, 4)

**COLLEGE/College7thSem/BtechProject/project/pyProject/unix.gv.pdf**

**Figure 5:** Sample Architecture generated by our NASGraph



**Figure 6:** Architecture generated by NASH



**Figure 7:** Initial Architecture used by NASH