Efficient Automatic Optimization of Neural Network Architecture

Dissertation Submitted In Partial Fulfillment Of The Requirements For The Degree Of

Master of Technology
in
Computer Science

by

Harsharaj Pathak
[Roll No: CS1819]

Under the Guidance of

Prof. Ashish Ghosh
Professor
Machine Intelligence Unit(MIU)

Indian Statistical Institute
Kolkata-700108, India

Acknowledgment

It was a great privilege and learning experience, to work with Prof. Ashish Ghosh. He had been a constant source of support. I want to sincerely thank all the research scholars in MIU. Thanks to all the friends who had been there with me for the past two years.

Contents

List of Figures

List of Tables

Abstract

Neural Networks are at the heart of deep Learning Frame works which have yielded excellent results in various complex problem domains. But the design of neural network architecture is a challenging task. Judicious selection of network architecture and manual tuning of network parameters is a tedious and time consuming process. There has been a substantial effort to automate the process of neural network design using various heuristic algorithms. Evolutionary algorithm are amongst the most successful methods to automate the network architecture search process. But these algorithms are very computation intensive. Thus we try to explore a technique that could lead to faster evolutionary algorithms to find optimal neural network architecture.We also do a survey of various alternative methods.

Chapter 1

Introduction

1.1 Introduction

Machine Learning is a field in Computer Science that aims to teach a computer to perform tasks without explicit instructions by creating a program that can learn from examples. It has emerged as an important field of Computer Science due to its ability in solving complex and fuzzy tasks such as object recognition, language translation, etc. Out of the various machine learning techniques, deep learning as been particularly successful and has achieved record breaking accuracy in various tasks. At the heart of deep learning there is something called an Artificial Neural Network(ANN)[1] which is an abstract representation of a signal processing system whose design has been inspired by biological brains. These networks are very powerful in there capability to approximate complex functions in high dimensions. But their internal working continue to remain a mystery and hence proper design of their architecture is a challenging task. Mostly this task is done using heuristics and by trail and error methods. But there has been a substantial effort to automate the design process of these ANN that can be widely categorized under Automatic Architecture Design[2]. The techniques that have come out of these efforts have given very good results but they have their own drawbacks. In this work we attempt to do a survey of these techniques and identify their limitations. Then we will propose an algorithm that tries to addresses some of these issues.

1.2 Thesis Outline

The thesis is arranged as follows. In chapter 2 we discuss the preliminary concepts. In chapter 3 we survey a number of existing methods. In chapter 4 we do a comparison between the methods. In chapter 5 we propose a new algorithmic approach. In chapter 6 we provide experimental results. In chapter 7 we conclude the topic while discussing future scope.

Chapter 2

Preliminaries

2.1 Supervised Machine Learning

In this section we present a brief overview of supervised machine learning which is the main focus of initial ANN models. In supervised machine learning we try to estimate a function:

$$f : \varepsilon_x \rightarrow \varepsilon_y$$

where $\varepsilon_x \subseteq R^m$ and $\varepsilon_y \subseteq R^n$. Here we give an algorithm training data of the form $(x_i, y_i)_{i=1...N}$ and $y_i \approx f(x_i)$. The learning algorithm will typically try to minimize some loss function L:

$$L : \varepsilon_x \times \varepsilon_y \rightarrow R^{\geqslant 0}$$

L describes the discrepancy between the function f and our estimated $f^{'}$ over the training data by successively modifying $f^{'}$. It must be noted that minimising L on this training data is usually not a challenge but the success of the learning algorithm depends on how closely $f^{'}$ approximates $f^{'}$ out of the training data. A process known as generalization. This turns out to be the major challenge of supervised machine learning.

2.2 The Perceptron

The first model of ANN is the perceptron[1]. The model is loosely based on the neuron in the brain. Here we have a n dimensional input connected to a non linear model of a neuron called the McCullus Pitt model of the neuron[]. The output of the perceptron for a particular $x_i = (x_1, x_2, ...x_m)$ is given by

$$v\left(x_i\right) = \sum_{j=1}^{m} w_j x_j + b$$

where wi are the synaptic weights weights and b is the bias. The goal is to update the weights so that v = 0 represents a desired decision boundary for the training data. i.e $v(x_i) > 0$ when $y_i = 1$ and $v(x_i) \leq 0$ when $y_i = -1$. And thus the perceptron acts a binary classifier that can classify vectors in X into two classes C_1 (where y = 1) and C_2 (where y = -1) .To train the perceptron using training data D : $(x_i, y_i)_{i=1...N}$ an algorithm called the perceptron learning algorithm is used. As per this algorithm the weights are updated according to the following rules:

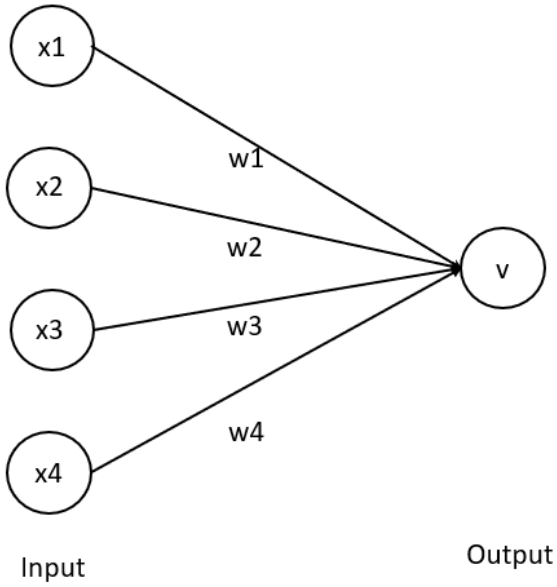

Figure 2.1: Simple diagram of a perceptron

$w(n+1) = w(n) - \eta x(n)$ if $w^T(n)x(n) > 0$ and $x(n) \in C_1$
$w(n+1) = w(n) + \eta x(n)$ if $w^T(n)x(n) \leqslant 0$ and $x(n) \in C_2$

Is has been proved that if the classes c_1 and c_2 are linearly separable then the algorithm converges. However the perceptron is limited by the fact that it cannot learn non linear boundaries[3]. Due to this reason this approach to supervised learning was considered fruitless for a long time

2.3 Multilayer Perceptron

The problem of non linear class class boundary was ultimately with the introduction of the Multilayer Perceptron(MLP) and the back propagation algorithm[4]. Here an ANN is built by stacking multiple layers of neurons such that each layers output is fed into the neurons in the next layer. We can think of each pair of adjacent layers as a mapping. Each neuron applies an activation function $\sigma : \Re \to \Re$ to the weighted sum at its input.The layers between the input and output alyers are called hidden layers. It was shown that if non linear activations are used then for any given function f their exists some MLP that can approximate it[5]. Thus MLP can find highly non linear decision boundaries in the data space.This enables MLP to perform complex tasks such as handwritten digit classification. But to achieve this the network weights have to be optimized. For this a popular algorithm called back propagation was developped. It is a gradient descent algorithm where the error gradient values at each node is back propagated all the way to the first hidden layer. The full details of the algorithm will not be discussed, but the basic weight update rule for a connection weight w_i is given by:

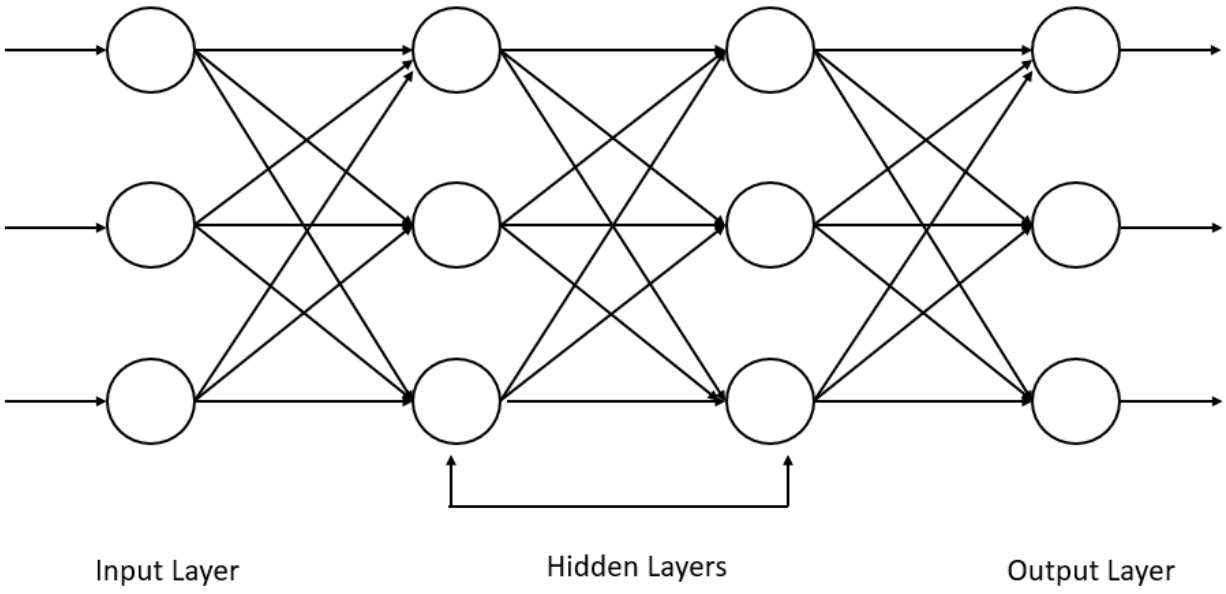

Figure 2.2: Simple diagram of a multi layer perceptron

$$\Delta w_i = -\eta \partial E / \partial w_i$$

where E is a measure of error based on the loss function and η is the learning rate. Thus the algorithm tries to minimise the error by descending down the error surface.

2.4 Challenges with training neural Network

It was shown that a neural network with only a single hidden layer can approximate any function given it has sufficiently many nodes with non linear activations[5]. However it is not feasible to practically solve any learning problem using just one hidden layer as for complex problems the size of hidden layer required can grow prohibitively large. It was shown[6] that if a neural network with k layers require O(n) nodes to express a function. Then an MLP with k - 1 layers will require O(2^n) nodes to express the same function. Thus the expressive power of a ANN grows very fast with increase in depth. This makes problem of finding architecture difficult. If a network has too few parameters, then it will not have sufficient expressive power to find the right input output mapping. This phenomenon is called under fitting and is marked by high training error. On the other hand, if a network has too many parameters then it may find a mapping more complex then the one that is needed which will lead to low training error but poor validation error(out of sample error). This phenomenon is called over fitting. Thus a network architecture that captures the precise structure of the function we are trying to approximate will give the best generalization possible. It can be argued that the smallest network that can give the minimum training accuracy is the optimal network for the given problem. This idea is even supported by Learning theory which states that the out of sample error of a learning algorithm (assuming training data and test data is drawn from

the same distribution) is bounded as[7]:

$$Pr\left(error_{test} \leqslant error_{training} + \sqrt{1/N[(\log(2N/D)+1)] - \log(\eta/4)}\right) = 1-\eta$$

Where D is the VC dimension of the hypothesis set which in case of a neural network is roughly based on the number of parameters in the network, $0 < \eta < 1$ and N is the number of training samples.
But the problem complexity is not known in advance and hence finding the right architecture is difficult. However finding the architecture that fits the problem complexity does not necessarily solve the model selection problem of ANN. If back propagation is used for training then there is no guarantee that the algorithm will converge to the global optimum instead it may stuck at a local optimum where the gradient of weight vector becomes close to zero. Various solutions have been proposed to overcome this problem such as using momentum[8]. But local minima is still a problem especially for small ANN. Therefore practically it is seen that deeper networks tend to optimize better as with large number of weights, the chances of a local optima gets smaller[9]. Therefore there is a focus on using over parameterised networks to learn a function and additional techniques are used to prevent overfitting such as weight decay[10], dropout[11], etc. However these methods do not entirely solve the model selection problem as overfitting is still possible and each technique come with additional parameters that needs to be tuned. Thus deciding the proper architecture for any given learning scheme is still a design challenge

2.5 Human designed architectures

Owing to the significance of network architecture, a great deal of research has gone into finding good neural architectures for various problem domains. Some notable developmemts are Convolutional Neural Network(CNN)[12], that tries to capture the spatial correlation in image data and Long Short Term memory(LSTM)[13], that tries to capture temporal correlations in sequential data such as text. Although CNN is not just an architectural solution but also a modification to the optimization problem as it imposes weight sharing. But the advent of CNN has led to lots of explorations on architectural patterns which has led to the development of very deep NN such as AlexNet[14], VGG[15], GoogleNet[16] , Resnet[17],etc. These networks have achieved record breaking accuracy in vision tasks and the main distinction of these modes lies in finding novel architectures that not only captures the problem structure but also the ease of optimization. For example resnet introduces the idea of residual connections that circumvents the vanishing and exploding gradient problem[18] in very deep neural networks. Usually architecture choice for a given problem is done by manual tuning by experts based on domain knowledge and there are no reliable mathematical rules to decide the architecture given a certain data set.

Chapter 3

Automatic Architecture Design

Given the importance of network architecture and the difficulty in finding good architectures researchers have tried to develop methods that automate the process of finding network architectures. The set of algorithms that perform this task can be broadly categorized as Automatic Architecture Search Algorithms. These algorithms can be divided into Network Architecture Search(NAS) algorithms and Dynamic Learning Algorithms. These paradigms contain a large number of algorithms and it is difficult to do an exhaustive survey so we only mention a few algorithms that covers a wide range of flavours.

3.1 Network Architecture Search

This class covers algorithms that tries to search for optimum network architectures over a large search space. Most of these algorithms use either reinforcement learning[19] or Evolutionary strategies[20]. Few of these algorithms are Re enforcement Learning based NAS, Efficient Neural Architecture Search(ENAS), Progressive Neural Architecture Search(PNAS), Auto CNN, Learning Evolutionary AI Framework(LEAF), Neural Architecture Search with Bayesian Search and Optimal Transport(NASBOT). We will briefly describe each method without going into technical details since we are only concerned with the advantages and drawbacks of the methods.

3.1.1 Re enforcement Learning Based NAS

In this technique[21] a Recurrent Neural Network(RNN)[22] is used as a parent network. A RNN is a variant of ANN with feedback loops and they can process sequential inputs, The RNN is used as a controller and outputs a sequence of values these values are treated as hyper parameters of a neural network of a certain template.A a child network is created with these hyper parameters and trained on training data. Then its accuracy on validation data is measured and the resulting error is treated as the sample error for the controller network. Since the child network architecture is non differentiable, a policy gradient method is used to update the parameters of the controller RNN. Then a new child network is generated and the process is repeated until a child network of desired validation accuracy is found.Networks found using this method reportedly achieved a test error of 96.35% on the CIFAR10 image classification data set using 37.4 M parameters and 2000 GPU days of computation.

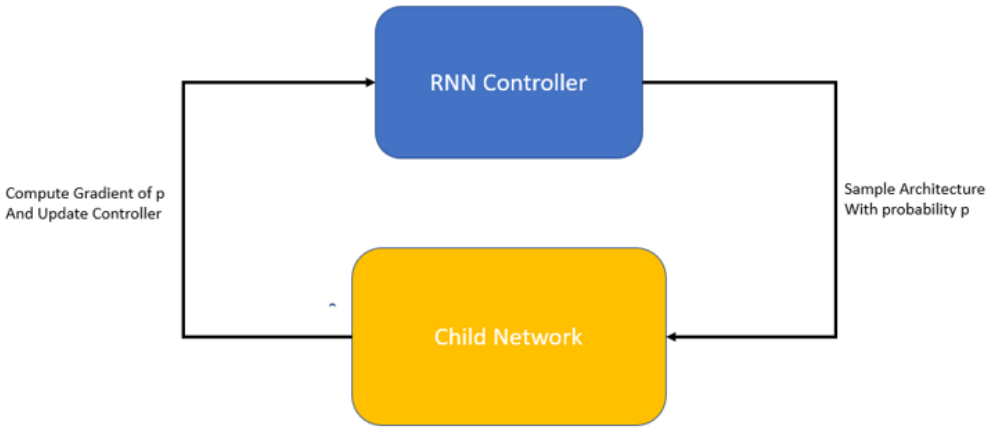

Figure 3.1: Control Flow of RL NAS

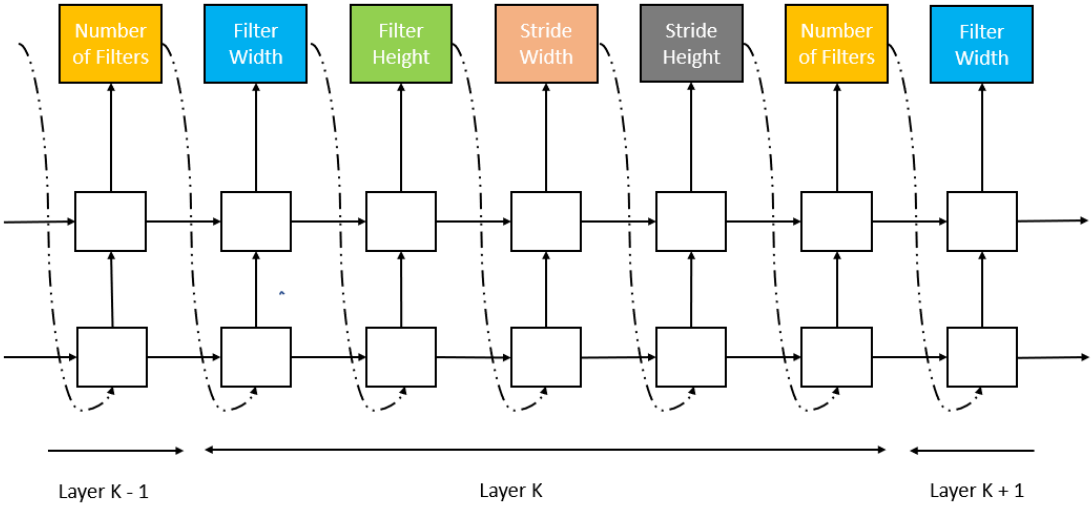

Figure 3.2: Controller RNN predicting network parameters for each Layer. The white blocks are network units in the Controller RNN. The colored blocks are the outputs that define parameters of the child network.

3.1.2 Progressive Neural Architecture Search(PNAS)

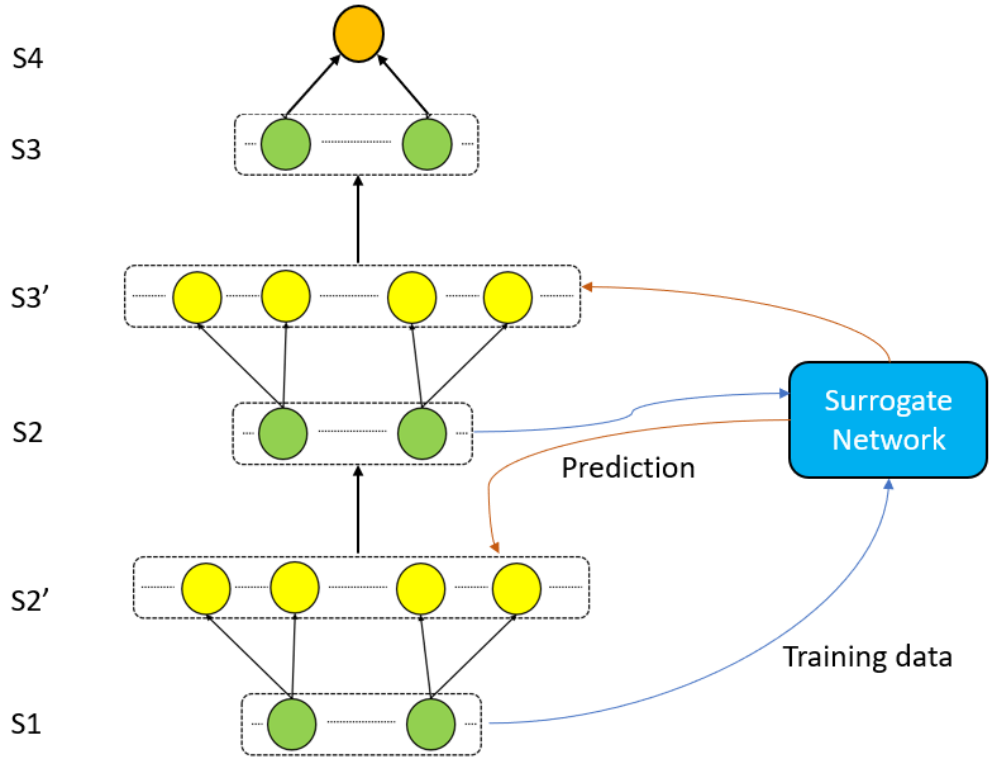

Figure 3.3: The above diagram shows the stage wise progression of network search. The green circles represent networks in a certain generation which are evaluated to train the surrogate model. The yellow circles represent the set of expanded versions of the networks in the previous generation whose performances are predicted using the surrogate model. The orange circle is the final selected network

This technique[23] is used to grow networks gradually. The method has been used to design CNN. Here the whole CNN architecture is not searched , instead a CNN block is optimized and the final CNN is formed by connecting a number of this blocks in a predefined way. Also their is no RNN controller here and re enforcement learning is not used but a surrogate network is used to predict the performance of a model without training it. The algorithm starts with a random set of k small blocks. A CNN corresponding to each block is constructed and trained. Their validation accuracy is measured. The surrogate network is then trained using the evaluation data. Then each block is expanded into larger blocks in multiple possible ways leading to a larger set of candidate blocks. The performance of these blocks are then predicted using the surrogate model and the top k are selected for the next round. This process is repeated until a network of desired accuracy is found. This technique reported 96.59% accuracy on CIFAR10 using 225 GPU days of computation.

3.1.3 Efficient Neural Architecture Search ENAS

ENAS[] is a more efficient and flexible version of NAS.ENAS is used to design an individual cell of a network.In ENAS, the designer first specifies a large cell network made of various computational units. A computational unit may be an activation, filter[12], pooling layers[12], etc. The task is now to pick a sub graph of this graph and also to specify the type of computation performed at each node. This sub-graph is taken as the final architecture of choice. This sub-graph is selected using a controller RNN which outputs defines which edges to chose from the large graph and what computation to perform at each node.

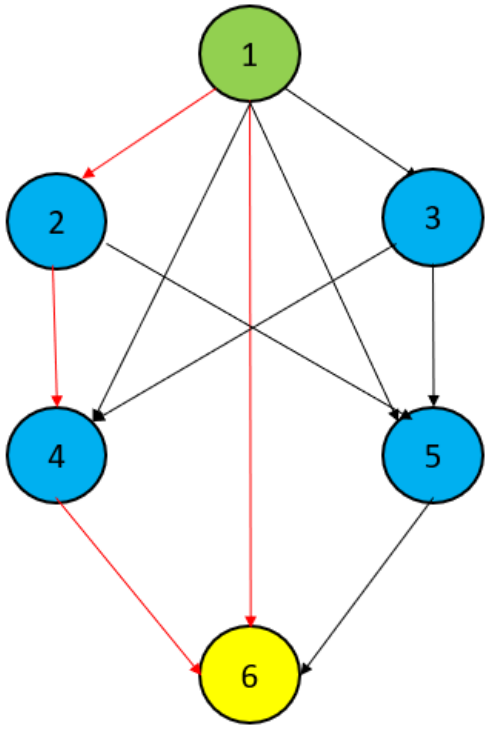

Figure 3.4: The graph above is the super-graph from which architecture are sampled. The green circle is the input unit and the yellow circle is the output unit. The blue circles are hidden units. The red edges mark a certain sub graph that is used to denote a possible network architecture

The trick that makes it efficient is that weights in the large graph are shared among all sub-graphs. That is during training the controller RNN, the child network does not have to be trained from scratch each time, rater it initiates training the relevant weights from their existing values in the super graph and as it trains, the weights in the super-graph are also updated accordingly so that they can be used by some other network sampled from the same super graph. This technique reported 97.09% test accuracy on CIFAR10 with 4 GPU days of computation. Its success is surprising since a given set of weights are expected to perform different tasks when put in different overall architectures and made to act on different computational units, yet this kind of weight sharing still gives reliable results. A proper explanation of this phenomenon has not been found so far.

3.1.4 Auto CNN

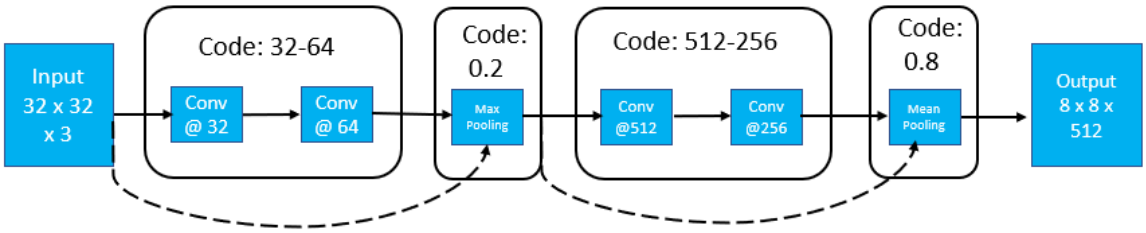

Figure 3.5: An example of Auto CNN encoding and corresponding network architecture

This NAS method[24] is based on Evolutionary algorithm. This method is also used to find CNN architectures. In this method a network architecture is encoded as a gene that contains a string of numbers. Each pair of consecutive numbers represents either a skip layer(successive convolutional layers with a skip connection) or a pooling layer along with their parameter.There are no fully connected layers. The search starts with a random initial population of genomes. Each genome is decoded into a network and its fitness is evaluated by training it on a training data and then finding its validation accuracy. The next generation is chosen based on binary tournament selection. Then 1 point cross over is done on the population with a certain probability. This is followed by mutation operation on each genome that can involve adding or removing a layer or altering the parameters. The parameters are chosen from a discrete set of possible parameter values. This is repeated until a network of desired accuracy is found. Then this network is further trained. To reduce computational requirements, the evaluations are done on partial data. Also they keep a database of all architecture evaluated so far to prevent the same architecture from being evaluated multiple times. Using this technique, test accuracy of 96.78% was reported on CIFAR10 using 40 GPU days of computation.

3.1.5 Learning Evolutionary AI Framework(LEAF)

This[25] is perhaps the most sophisticated NAS algorithm.It is an evolutionary algorithm. A detailed description of this algorithm is beyond the scope of this work so only the basic idea will be discussed. The strategy includes three layers: the algorithmic layer, the system layer and the problem domain layer.

The algorithmic layer searches for hyper parameters and network architectures. The system layer is used for parallel model evaluation through cloud computing and the problem domain layer is used for parameter tuning. The architecture search algorithm is based on a neuro evolutionary technique called NEAT[26](discussed later).The NEAT based search technique enables arbitrary graph topology but contrary to the original NEAT, here a node in the graph represents an entire layer in the network and its corresponding parameters while edges represent connection pattern between layers.

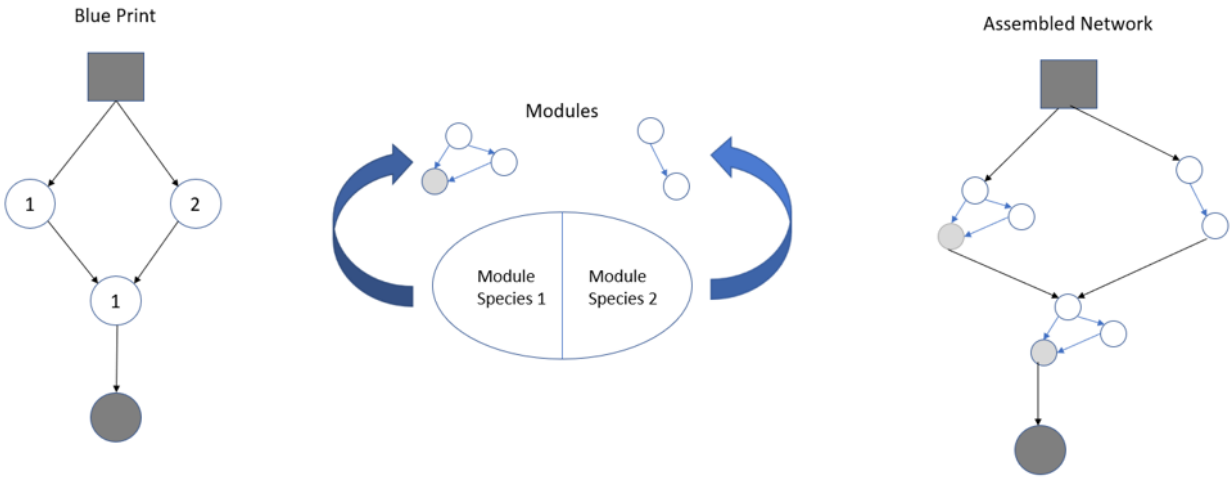

Figure 3.6: The above diagram shows how modes of designated species to into corresponding sites in an evolved blueprint

The evolution happens at two levels. At on level a blueprint of the overall network architecture is searched while at the the other level populations of modules that go into various components of the blue print are evolved. A network is formed by taking a blueprint and substituting its components by individuals of module population each component points to and then this network is evaluated by training on training data. With each generation the fitness of the population of blueprints as well as that of the modules improve, thus giving overall better networks. Using this technique the authors surpassed the best known accuracy on the Wikedetox problem achieving an accuracy of 99.97% with 375 CPU days of computation.

3.1.6 Neural Architecture Search with Bayesian Search and Optimal Transport(NASBOT)

NASBOT[27] is a NAS technique which is also based on evolutionary optimization. Here networks are encoded as strings to indicate the various layers and the parameters associated with each layer. The mutation operators enable adding or removing layers, modifying the layer parameters or creating duplicate paths from existing paths. The main contribution of this algorithm is that it uses Bayesian Optimization[28] to decide which networks are most likely to give best validation accuracy. For this they propose a distance metric to find the distance between two networks called OTMANN distance. This enables selective evaluation on network models.This technique achieved a test accuracy of 91.31% on CIFAR10 using 1.68 GPU days of computation.

3.2 Dynamic Learning Algorithms

Dynamic Learning algorithms are the class of algorithms that optimizes the architecture during the training process.

3.2.1 Cascade-Correlation Networks

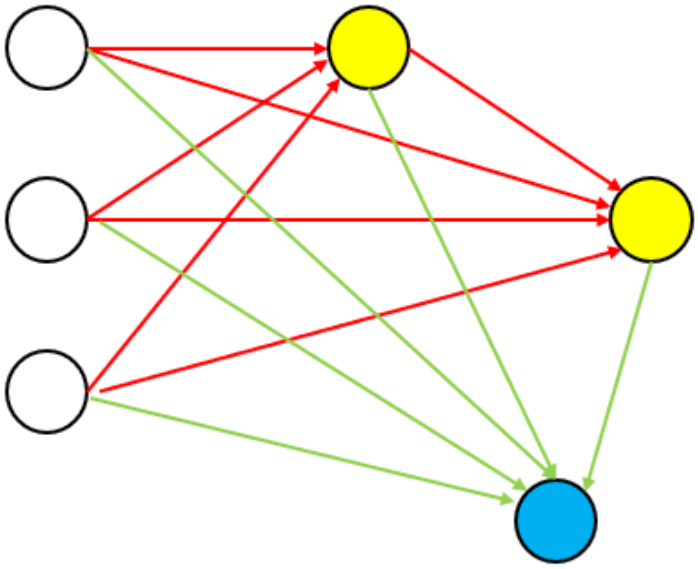

Figure 3.7: Diagram of Cascade. The white circles are inputs. The Blue circle is the output unit. The yellow circles are the hidden units. The red edges denote frozen weights while green edges denote trainable weights

The Cascade - Correlation Network[29] adds hidden units to the network during training. Here the network starts with no hidden units and the input layer is fully connected to the output layer. On training if the error becomes stagnant and if the validation error is not satisfactory, then a hidden unit is added. A newly added hidden unit has incoming connections from all inputs and all existing hidden nodes. Initially there is no outgoing connection from this hidden node. The incoming weights to this newly added node are then trained to maximize the sum of the correlation between the node output and the error at each output of the network. The idea is that the output of this node should be able to cancel out the network error when connected to the outputs. This node is then connect to the output nodes and its input weights are frozen. The training then resumes. Like this successive hidden nodes may added depending on the complexity of the problem. Since input weights of the hidden nodes are frozen , training does not involve back propagation and only the weights connected to the output layer are trained. The authors reported solving the 8 bit parity problem using a network with 4 to 5 hidden units.

3.2.2 Node Splitting

The Node Splitting Technique[30] is also based on adding nodes during the training process. In this method additional nodes are generated by splitting existing nodes into two during training. The network starts from some minimal configuration and as the learning stagnates, a node is chosen for splitting. The choice of node to split is done as follows: When the training error stops falling, all weights are frozen.

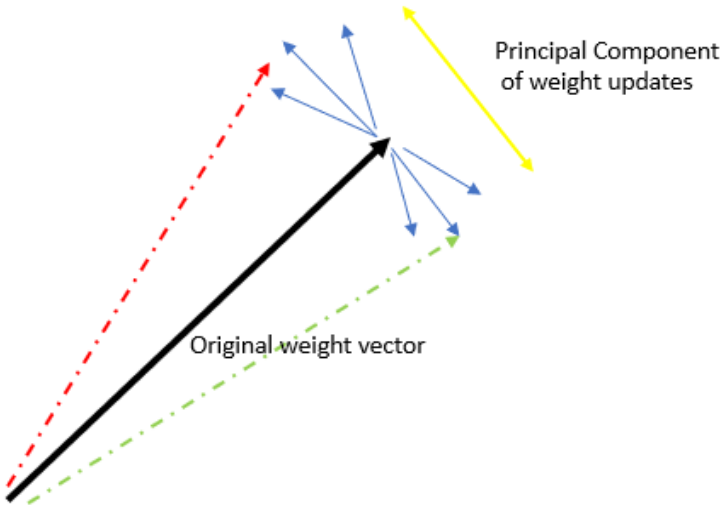

Figure 3.8: The black arrow initial weight vector of a node. The blue arrows are the weight updates. The red and green arrows denote the weight vectors assigned to the two child nodes after splitting

Then the weight update values based on gradient descend is calculated for each weight for each training sample(although weights are not actually updated). Thus for each node we have a set of weight update vectors for the weight vector associated with that node corresponding to each training sample. The node which has the maximum variance for these set of update vectors(calculated by Principal Component Analysis)[31] is chosen as the node to split. The the connections to the child nodes are same as that of the parent. But the weights assigned to the two resulting nodes are one standard deviation apart on either direction in the direction of the principal component of the weight updates. The author did not find any good result of using this technique to neural networks but the adaptation of the technique to Gaussian Mixture Model[32] was found successful, with the models being able model a circle and an enclosing annulus.

3.2.3 Forward Thinking

Forward thinking[33] is a simple dynamic learning scheme where the network grows one layer at a time. The authors have described a general framework for greedy layer wise learning. which is not not restricted to neural networks.

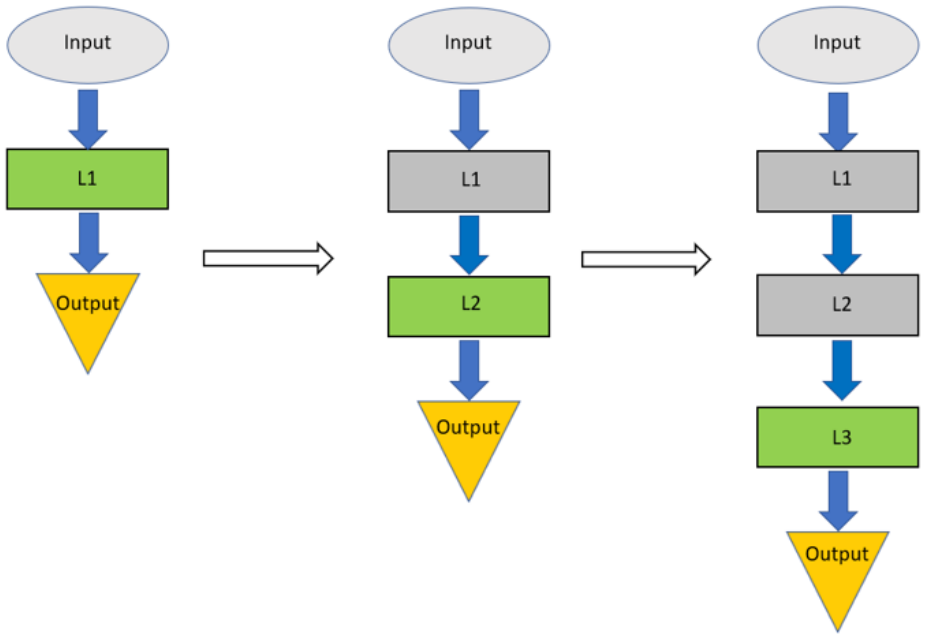

Figure 3.9: Step wise progressing on forward thinking. The grey layers have frozen weights. The training is confined to the yellow and green blocks

In particular, the algorithm for neural networks is as follows: First a network is constructed with a single hidden layer and trained. If the learning stagnates, the input to the first hidden layer are frozen and the inputs are pushed through this layer to create a new set of synthetic inputs. The weights going out of this layer are discarded and a new layer with random weights is sandwiched between this layer and the output layer. Now the training resumes but is confined to the newly added weights. This way even as new layers are being added, the training only happens in a shallow network with only one hidden layer. The idea is that each layer makes the effective training data for the subsequent layers more separable. But due to shallow training, the training time is reduced. Also because most weights are frozen, it does not suffer from moving target problem or vanishing gradient problem. This technique yielded a 99.72% test accuracy on the MNIST data set.

3.2.4 Neuro Evolution of Augmenting Topologies(NEAT)

NEAT[26] is a very successful algorithm that automatically searches network architectures along with learning weights. It is not a dynamic learning algorithm in the conventional sense as it does

not involve weights training. Rather contrary to all algorithms discussed so far, NEAT learns both weights and architecture using evolution.The idea of using genetic algorithm as an alternate to gradient based training for learning weights of a neural network was introduced by Darrell Whitley[34]. In this technique, networks are encoded in a manner that reflects both connection pattern and associated weights.

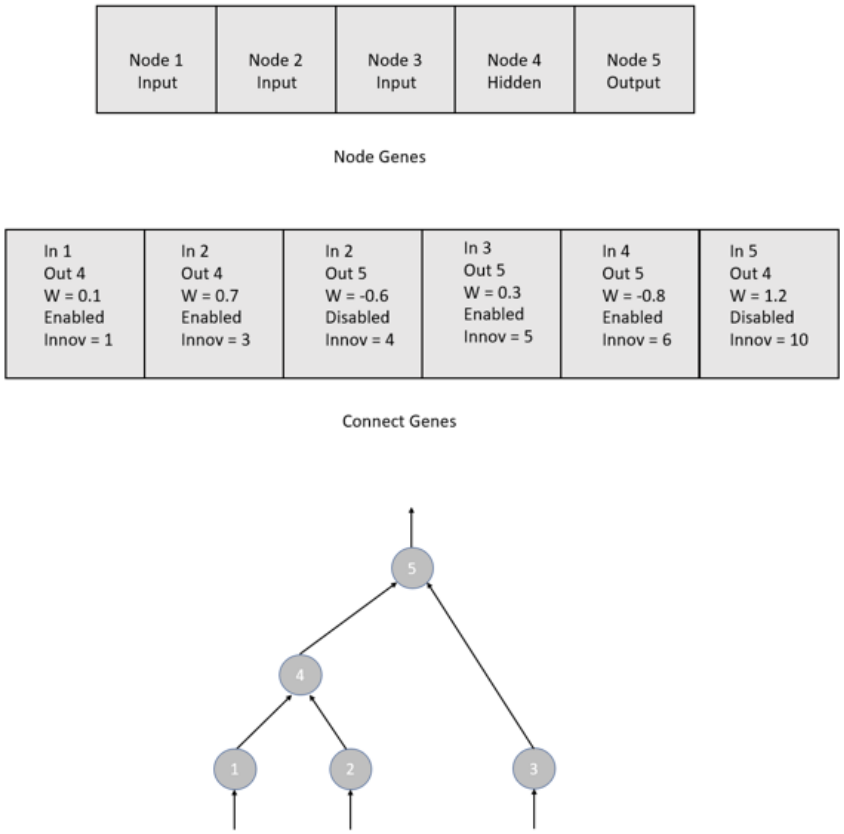

Figure 3.10: Encoding Scheme and the corresponding network in NEAT

NEAT can evolve arbitrary graph architectures. At each generation, the networks are evaluated on a specific task and a fitness is assigned based on performance. Then a new generation is created based on certain selection scheme. Mutation operations[20] can alter weights and add new nodes and connections. There is non deletion. Cross over[20] is done by aligning the genes based on innovation numbers. An innovation number is assigned to each distinct connection in the whole pool of networks so that cross over can happen meaningfully. Also the networks are separated into species based on amount of dissimilarity so that each species can evolve separately and novel species do not eliminated before they can flourish. The authors demonstrated NEAT by solving the "double pole balancing task without velocities".

3.2.5 HyperNEAT

HyperNEAT[35] is an extension of NEAT that can evolve much larger networks by utilizing symmetries and spatial regularities. Unlike NEAT, HyperNEAT uses an indirect encoding ,i.e each node and connection in the network is not explicitly encoded rather the connectivity pattern is encoded using a technique called Compositional Pattern Producing Network(CCPN). This leads to networks that have a regular structure and can be scaled up to increase model complexity. As in NEAT, the weights are also learned through evolution. HyperNEAT was demonstrated to perform well on a visual discrimination task and a robot food gathering task.

3.2.6 Differentiable Architecture Search(DARTS)

DARTS[36] is a very efficient Architecture Search method that treats the network architecture as a continuous function. Some would regard DARTS as a NAS strategy but owing to the fact that in DARTS weight and Architecture are updated concurrently, we have categorized it under dynamic learning. In DARTS, the architecture search is treated as a problem of continuous relaxation. A network architecture is described as a repeated group of cells connected in a predefined manner. The goal is to find the best architecture for each cell. For this a mother cell is defined in the form of a directed acyclic graph network with a set of nodes and connections. Each node node represents a feature map and each connection represents a particular type of operation(e.g convolution, pooling, reduction). But there can be multiple connection between two nodes. Each connection in the graph has an associated weight.In the actual network, all operations connecting two nodes occur simultaneously and resulting output is taken as the soft max weighted sum over the results of each operation as per the weights associated with the operations. Now the learning process happens in this network as a bilateral optimization problem. The optimization objective is as follows: $min_{\alpha} L_{val}\left(w^{*}(\alpha), \alpha\right)$ s.t

$$w^{*}\left(\alpha\right) = argmin_{w} L_{train}\left(w, \alpha\right)$$

where L_{val} is the validation loss, α is the outer parameter refering to the weights given to the set of operations, w is the inner parameter refering to the internal weight vectors of each operation. To make the optimization computationally efficient, the following approximation is used:

$$\nabla_{\alpha} L_{val}\left(w^{*}(\alpha), \alpha\right) \approx \nabla_{\alpha} L_{val}\left(w - \xi \nabla_{w} L_{train}\left(w, \alpha\right), \alpha\right)$$

where ξ is the learning rate for the inner optimization step. Thus $w^{*}(\alpha)$ is adapted by using a single training step without solving the complete inner optimization.

At each step first the external weights of the DAG is fixed and the internal weights of the network are improved with respect to the training error. Then the internal weights are fixed and the external weights are improved with respect to the validation error. When for a pair of nodes in the DAG, one weight dominates the rest, that connection having that weight is taken as the connection of choice between the nodes and the remaining connections are pruned. Thus finally we are left with a DAG with exactly one connection between every two nodes. The cell architecture corresponding to this DAG is the chosen architecture for cells in the network. The original authors however did not give any convergence guarantee of the algorithm. This technique gave 97.24% accuracy on CIFAR10 with 4 GPU days of computation.

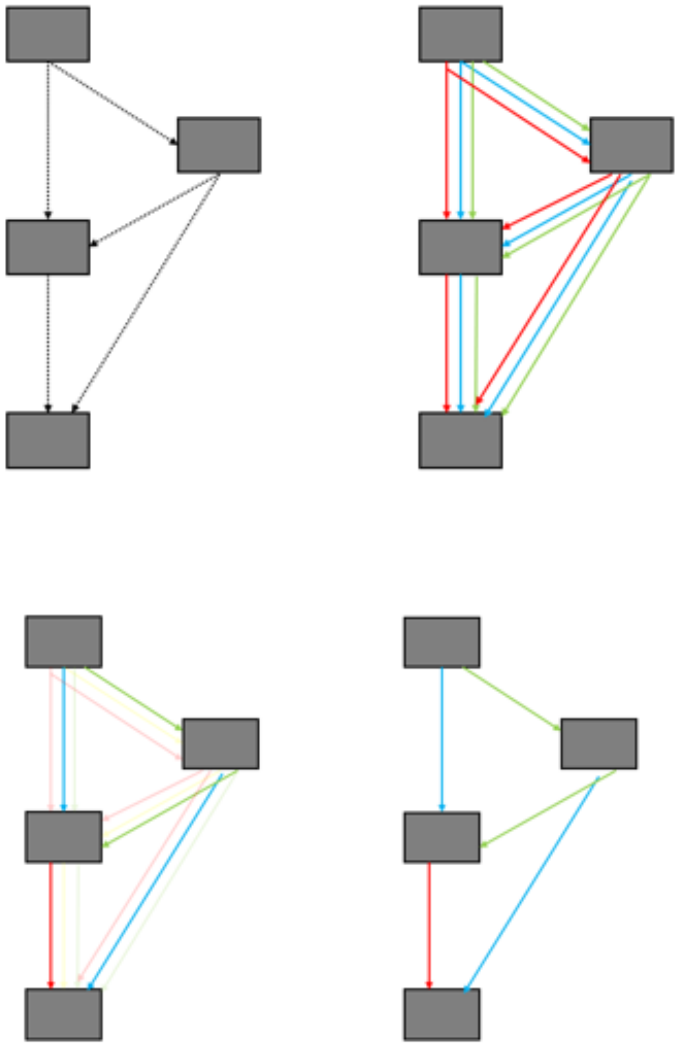

Figure 3.11: The dotted lines in the first figure shows the unknown operations between various computational units. The second figure shows a network where various operations co exist. The third figure shows some operations fading away due to lowering of associated weights. The fourth figure shows the final network architecture.

Chapter 4

Comparison between Automatic Architecture Search Models

In this chapter we do a comparison between the wide range of Automatic Architecture Search models discussed in the previous chapter and propose an idea called greedy neuro evolution. The comparison is mainly based on the algorithmic nature of the models and not based their precise performance statistics.

4.1 Comparison between models

From the discussion in the previous chapter it is clear that dynamic learning schemes are very favourable due to due their efficiency. However, the old techniques such as cascade cascade, node splitting have not gained much popularity and are rarely used any more. This is mainly due to the fact that modern practise revolves around very large networks with millions or billions of parameters where the search space of architectures in enormous whereas the above two methods only allow slow augmentation of network size and hence cannot yield network architectures for complex tasks such as object recognition and machine translation. Forward thinking is a newly introduced method and can grow network indefinitely. But this method requires the designer to set the width and type of each layer. Also the method allows only linear search on the number of layers thus is very limited in searching for different connection patters. Shallow training is an advantage but the networks may grow very deep creating problems in storage requirements and test time speed. So far the technique has only been demonstrated in limited problems and it is not clear how it will hold up across domains. Dynamic Evolutionary schemes such as NEAT and HyperNEAT have found successful applications in policy based tasks such as robot control where error gradients are not readily available.NEAT searches in the space of architecture and weights simultaneously thus exploring a very large search space. Thus NEAT is only able to evolve networks of modest size. HyperNEAT tries to solve this by growing large modular networks and has been quiet successful but in problems with labelled data, gradient based methods are still more successful. Both NEAT and Hyper NEAT spend most of the time searching in weight space leading to limited architecture search.DARTS is so far the best Automatic Architecture Search algorithm in terms of accuracy and efficiency in difficult supervised learning tasks. However, the level of automaticity and novelty

search is lacking in these method. First an initial graph has to be defined from which a relaxed sub graph is the final cell architectute. This super graph requires domain knowledge to build. Also the algorithm cannot grow the network beyond this super graph or discover novel connection patterns. It is not easy to apply DARTS to a new domain without existing architectural research in that domain. Coming to the NAS models, their main disadvantage is their high algorithmic complexity. It is already being argued that RL methods to find architecture parameters is very inefficient. Other methods using weight sharing or progressive search are relatively efficient. But ENAS is also confined to finding architecture within a supergraph while PNAS requires training of large number of networks to feed the surrogate network with enough data to accurately predict network performance. The Evolutionary Algorithms have the highest level of automaticity. Evolutionary Algorithms are very flexible in their design can in principle find any architecture. The LEAF model has the advantage of finding very novel network architectute without much domain expertise which gradient based methods like DARTS are unable to find. Also the power of evolutionary search can be seen from LEAF's record breaking performance in the very abstract problem of toxic comment classification. In the future as machine learning tries to tackle more and more complex problems, the ability to search for novel architecture and connection patterns is likely to grow in importance. This would give evolutionary schemes an edge over other schemes. But the large computational requirements of this schemes may make these methods impractical for bigger problems. As of now all Evolutionary Architecture Search schemes to the best of our knowlege rely on evaluating the fitness of a model based on is performance on the given problem. This is very computationally intense. There have been methods such as using Bayseian Optimization to limit the number of evaluations such as in NASBOT, yet this may not be enough while while searching for larger architectures and also NASBOT failed to give state of the art performance on CIFAR10 and in the same paper the authors found that their search only beats random network search by a small margin. Others have tried to to evaluations of fewer data and for fewer epoch but if the algorithm is fundamentally dependant on assigning fitness based on expected performance, such short cuts will give less reliable evaluations. Thus for novel neural architecture search to be feasible, one line of development would be to improve the algorithmic efficiency of evolutionary architecture search algorithms. In the following section we discuss one possible approach.

4.2 Greedy Neuro Evolution

As discussed above, all evolutionary NAS algorithms to the best of our knowledge use a fitness criteria that involves training a network on the training set and testing its accuracy on the validation set. This is a kind of global search and has high time complexity. Parallelization of evaluation is possible but will still have the same computational requirements. However evolutionary algorithms are inspired by biological evolution in nature and in nature evolution does not happen in this way. Complex behavior and structures evolve in nature in successive stages of evolution where at each stage some immediate problem is addressed. For example the birds of flight evolved from simple creatures which in turn evolved from unicellular organisms. But these organisms were not selected by the evolutionary process on the basis of their ability to fly. Similarly intelligence in living creatures did not directly evolve based on complex intelligence based selection. The idea of incremental evolution is not new and even neuro evolutionary strategies based on incremental evolution has been proposed[37]. But these strategy relies on the idea of transfer learning, i.e evolving a network for a particular task and then using that as the starting point for learning a

related (often more difficult) task. How ever the evolution itself does not happen greedily and also it requires manual intervention and explicit definition of different tasks. A more natural approach to evolution would be to make incremental evolution implicit. Like in case of natural evolution, the criteria of selection should not be the final performance but some locally observable criteria. We call this greedy neuro evolution. There are potentially other advantages of such greedy approach. It has been recently observed[38] that deep neural networks have far more parameters than what they need. Pruning algorithm has been developed that can prune networks to 5% of their initial size and still maintain the same level of performance. However if gradient based training is initially done in such small networks then they are unable to give the same performance. This suggests that the network which is optimum testing is not necessarily optimum for training. They feasibility of training a network(at least in case of gradient based techniques) is dependant on the nature of the error surface. Thus a useful solution would be to dynamically modify the network during the training process so that the optimization of parameters can be done suitably while at the same time the optimum mapping is achieved eventually. To see if such an approach can be effective we have experimented with a neuro evolutionary scheme that does evolution in multiple phases and evaluates fitness based on the immediate training gains.

Chapter 5

Proposed Algorithm

In this section we propose a simple algorithm to test whether evolving a network architecture in multiple stages based on immediate training gains can lead to any long term benefits. Although architecture search in general refers to network topology and activation function, in our case we confine the attention to network topology search. Owing to the success of NEAT based algorithms, we use the NEAT scheme as our basis. Also we will only search for general network architecture at the level of individual nodes and connections and not modular networks composed of specialised building blocks. In practise even basic MLP can grow quite deep(more then 10 layers) in certain applications and so MLP architecture search is also a difficult problem. The evolutionary scheme we are employing will be described in the following sections

5.1 Network Encoding

The encoding scheme we are using is similar to NEAT as shown in figure There are two genomes for each network. One is the genome containing the list of nodes in the network. The other genome contains the information of each connection which includes the incoming and outgoing node, the connection weight, the active marker which tells whether that connection is active. The innovation number, which is a unique identifier for each unique connection throughout the population. We have one additional marker called novel, which tells if this connection was added in the most recent mutation.

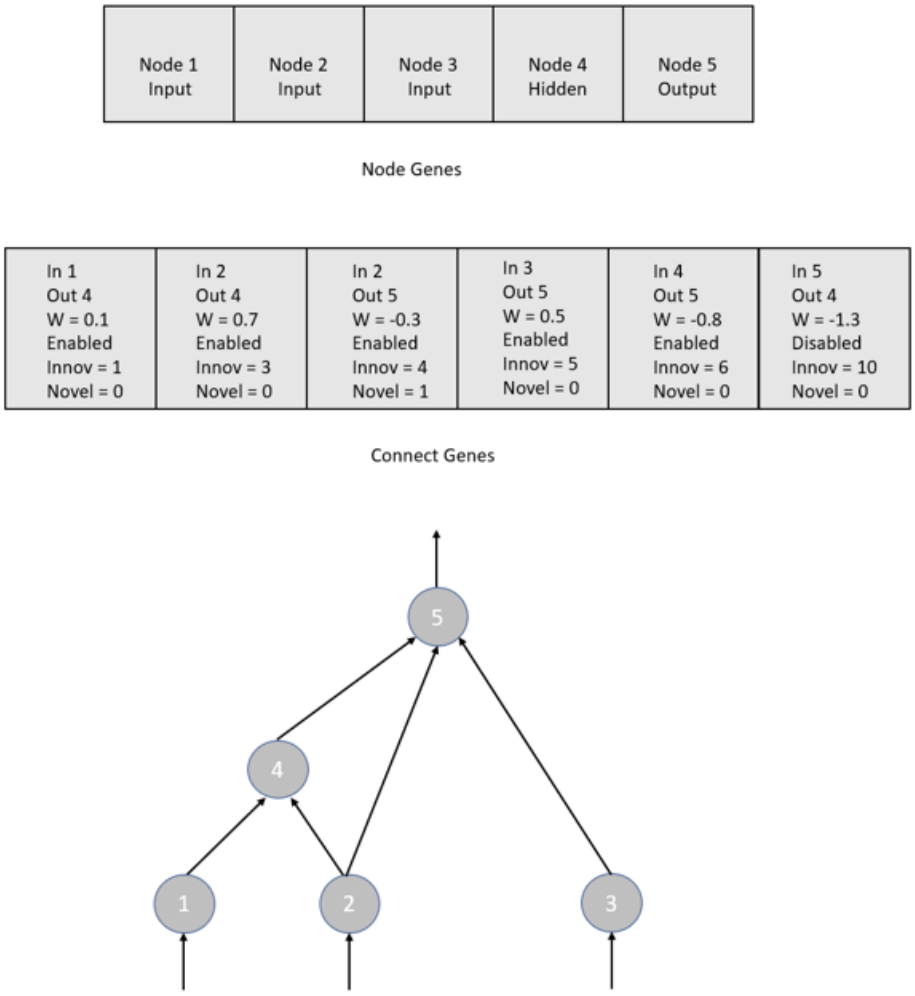

Figure 5.1: Encoding Scheme and the corresponding network.

5.2 Mutation Operation

There are two types of mutations that can occur in a genome. The first is a connection mutation that introduces a new connection between two existing nodes. This is done by adding a new entry to the connection genome with relevant parameters. The position of this entry is such that the innovation numbers are ordered. If this connection already exists in the population then the existing innovation number for this connection is used otherwise the global highest innovation number is incremented and assigned to this connection. The node mutation involves adding a new node on

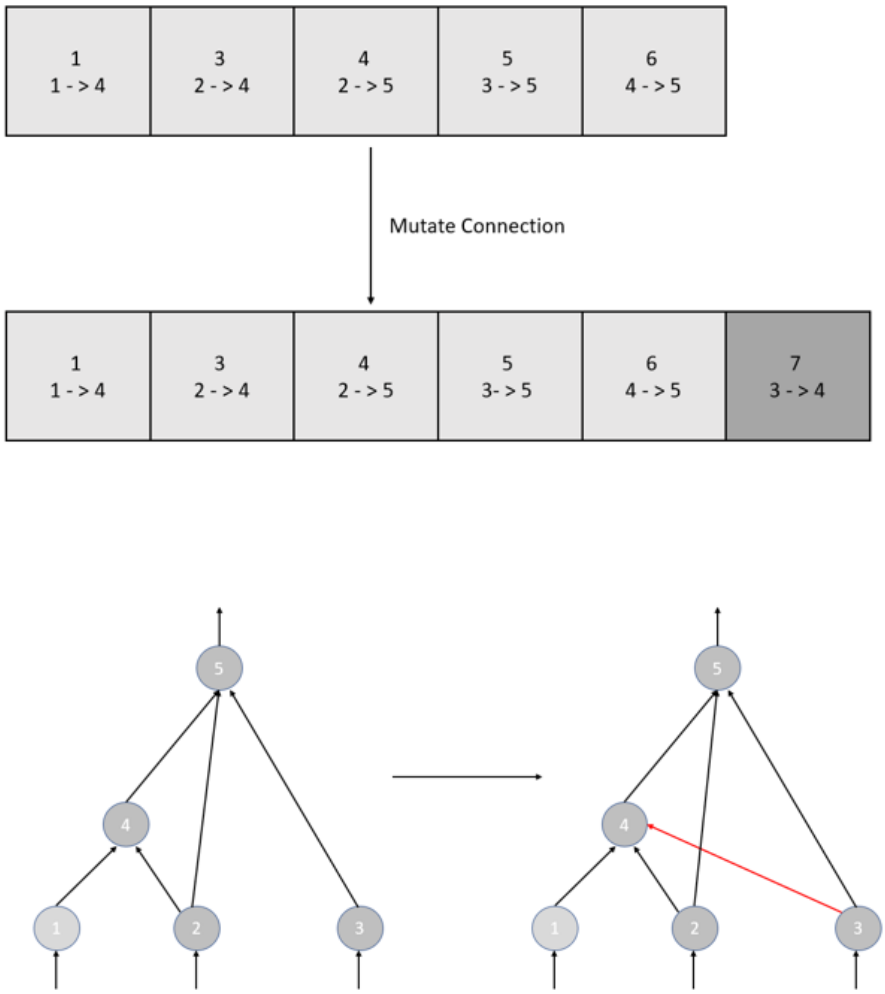

Figure 5.2: Mutate Connection operation.

an existing connection. This node splits the connection into two. The weight incoming to this node is assigned as 1. The out going weight gets the same value as that of the original connection. In the connection genome, this is reflected by adding to new entries for the two new connections and setting the active marker of the existing connection to 0. It can be easily observed that if a ReLU activation is used in the newly added node and if the activation coming into this node is always non

negative then adding a node in this manner does alter the behavior of the network immediately.

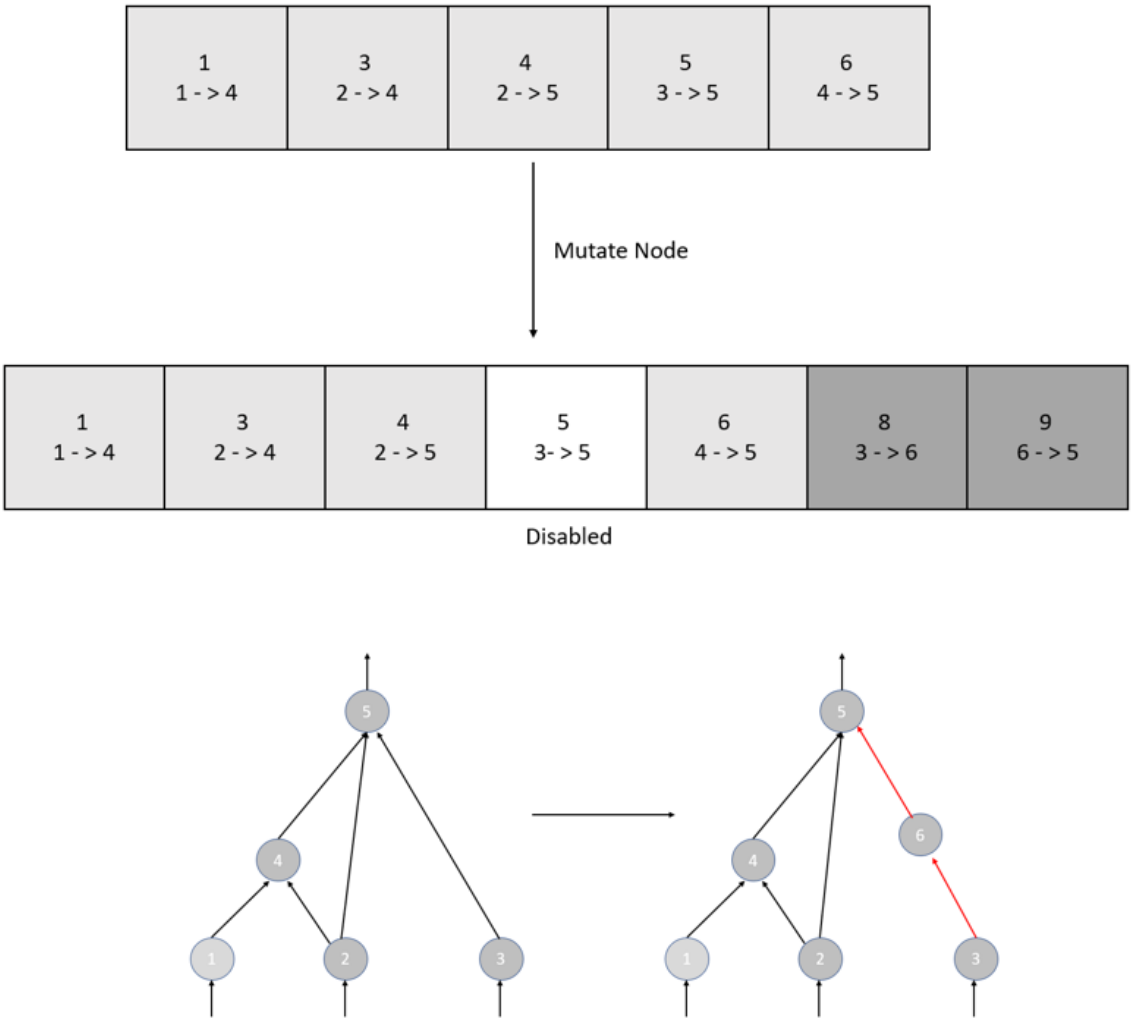

Figure 5.3: Mutate Node operation.

5.3 Cross Over Operation

The cross over operation in this scheme is different from NEAT. In NEAT the resulting offspring from crossover always has the architecture of the fitter parent thought the weights can come from either parent. Since we are only evolving architecture such approach would be redundant(although the weight mixing may be useful). In our scheme for each innovation number in the union of the set of innovation numbers of both the parents, the child inherits a connection from either parent at random. If a particular innovation number is absent from one of the parent, it is assumed that that parent has a null connection corresponding that innovation number. Inheriting a null

connection means that connection is absent in the child. This scheme can temporarily lead to useless connections as seen in the figure.

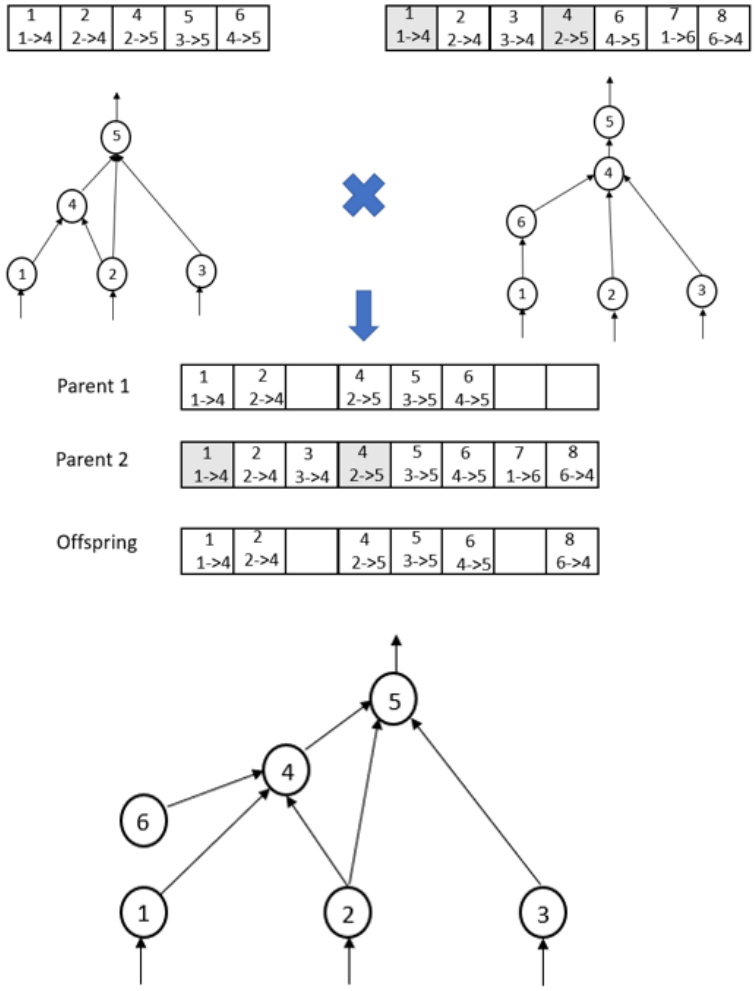

Figure 5.4: The figure above explains the cross over operation. The grey blocks are the active connections

5.4 The Algorithm

5.4.1 Overview

The algorithm has two distinct phases: The Learning Phase and The Evolution Phase. In the learning phase, a network of a fixed architecture is trained using a gradient based technique such as back propagation. In the evolution phase, the existing network is subject to evolutionary modifications, which in this case are addition of nodes and connections with randomly initialised weights and evaluated based on immediate training progress. The best architecture found by evolution is

then returned to the training phase. This process repeats until a satisfactory architecture is found. What makes evolution greedy is that the evaluation is done based on training progress instead of final accuracy. Also evolution act upon a pre trained architecture rather than an untrained one. Although multiple phases of evolution can be a draw back, since modifications made in any phase is limited, the overall number of generations does not grow large.

5.4.2 Detailed Description of the Algorithm

We know explain the algorithm in detail. Unlike NEAT, here there is no weight mutation(other than the random weights that come along with added connections). The algorithm starts with a fully connected perceptron whose input and output size depends on the dimensionality of the training data. Here we describing a version of the algorithm that is simply used to fit the training data. In practise as previously mentioned, fitting of data can usually be done using any large enough network and the main purpose of architecture search is generalization. However fitting the training data with constrained number of parameters is non trivial and being able to fit the the with few parameters increases the probability of good generalization. We will however propose another version that tries to account for generalization. The training set is D.

$$(x_i, y_i)_{i=1,2,3,,...N}$$

The network is first trained on the data. When the training error ceases to improve, the network is encoded as a genome and passed to the Evolution Phase. In the evolution, a population of encoded networks is initialized with replicas of the initial network. Each network is subjected to mutation operation. The cross over is skipped in the first round. All networks are then evaluated using data D' sampled randomly from D. The fitness evaluation consists of the following process. First using the novel marker the newly added weights are trained for a few epochs on D'. Then the hole network is trained on D' for a few epochs. If the error on D' after partially training the whole network is E_2 then the fitness of the network is given by

$$F = (1/E_2)$$

. This fitness function favours network that have hight reduction in error. Then we apply a simple median selection where two copies are made for all networks for fitness above the median and passed to the next generation. The same process repeats except in all subsequent generations, mutation is followed by a cross over operation. The amount of mutation and crossover is implementation dependant. But unlike NEAT since we are doing purely architecture search, the mutations are more aggressive and multiple nodes and connections may be added in each generation. Also to make sure that the selection is done based on connectivity patter, the same number of nodes and connections and nodes are added to all networks in a generation. The evolution is done for a fixed number of generations. There is not required fitness criteria. The fittest network fond is then returned and training resumes on this network on data set D. This process is repeated until a desired training accuracy is attained or a maximum network size is reached.

5.5 Generalization and Pruning

The above algorithm was designed only with the aim of fitting a training set with a minimal architecture. In real life problems, the data is often noisy and we don't want our network to fit the

noise. While lowering the complexity of the architecture implicitly tries to solve this problem, we can add additional steps to prevent over fitting. First during evaluation, along with D' we can take another data set D" sampled from a validation set and instead of calculating fitness on the error on D', we can calculate it on error on D"(without training on D"). Apart from that our architecture can only grow, however to make the whole space of architecture accessible, we also need to prune connection. As discussed before, a network may sometimes need certain connections to progress in training but not need them for the final mapping. Thus we add another type of evolution phase called the pruning phase. It is the same as the previously discussed evolution phase, except in the pruning evolution, only one type of mutation is allowed where a certain fraction of connection get deactivated. The fitness criteria is same as before.

Chapter 6

Experimental Results

6.1 Fitting synthetic data

In this section we test the ability of our algorithm to find networks that can fit some synthetic curves with one dependant variable and one independent variable. Here we are interested in finding whether the evolution scheme is at all finding something useful. Thus we compare the performance of our scheme with a random scheme where instead of the prescribed fitness value, a random fitness value is assigned to each network. It should be noted that the random scheme does not return a random network as it too causes network to augment when the training stagnates but its fitness evaluation is random. Three synthetic data sets are generated with one input and one dependant variable: The piece wise linear curve, The Rectified sinusoidal Curve and The sine cos curve.

6.1.1 The piece wise linear curve

This curve is defined as:

$y = x$ if $0 \leq x < 0.25$
$y = 2x$ if $0.25 \leq x < 0.5$
$y = 3 - 3x$ if $0.5 \leq x < 1$

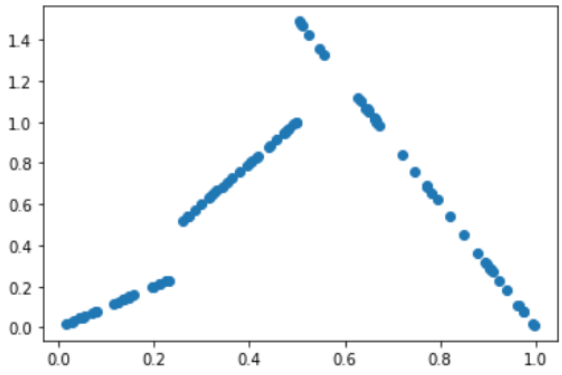

Figure 6.1: The piece wise linear curve

6.1.2 The Rectified sinusoidal curve

This curve is defined as:

$$y = |sin(x)| \text{ for } 0 \leq x < \pi$$

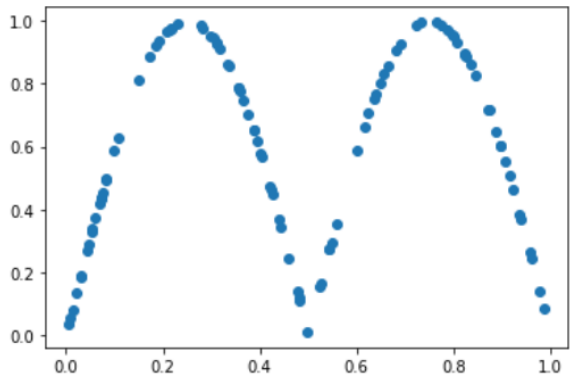

Figure 6.2: The Rectified sinusoidal Curve

6.1.3 The sin-cos wave

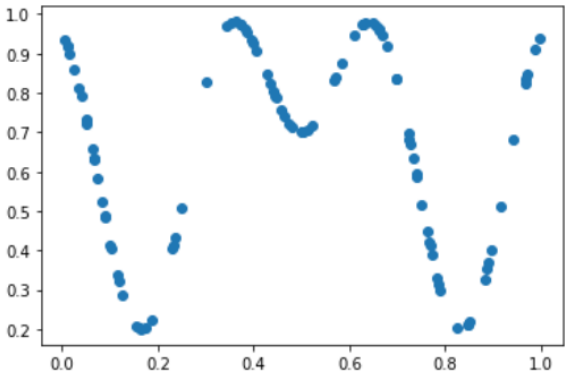

Figure 6.3: The sine cos curve

This curve is defined as:

$$0.7 - 0.25 \times sin(3\pi x) - 0.25 \times cos(6\pi x)$$

Table 6.1: Error comparison of proposed method with random search for ReLU output

Problem	Random Search	Proposed method
Piece Wise	0.110	0.073
Rectified Sine	0.097	0.063
Sine Cos	0.072	0.054

Table 6.2: Error comparison of proposed method with random search for Sigmoid output

Problem	Random Search	Proposed method
Piece Wise	0.135	0.115
Rectified Sine	1.02	0.086
Sine Cos	0.090	0.073

6.1.4 Methodology

For each curve fitting problem. A network is initialized with only one input, one bias and one output node. Training starts on 100 data points from the respective curves. The evolution process occurs as described above. There is no pruning as here we are only trying to fit the data. The population size is 51. The number of nodes added in a single mutation is $max(1, 0.008 \times numberofconnections)$.The number of connections added in a single mutation is $max(1, 0.1 \times numberofconnections)$. Each evolution phase occurs for 3 generations. Each fitness evaluation consists of 6 epochs of training. For the piece wise linear curve maximum number of hidden nodes is 4. For the rectified sinusoid and the sin cos curves the maximum number of hidden nodes is kept 9 and 15 respectively. When maximum network size is reached, the network continues training for 1000 epochs. Gradient descent with momentum is used with learning rate 0.1 and momentum 0.9. The experiment is repeated once with ReLU activation in hidden and output nodes and once with ReLU activation in hidden nodes and sigmoidal activation in output node

6.1.5 Results

Table 6.1 and Table 6.2 show the average values of the per sample error of the network evolved using the proposed strategy and a random evolved randomly on the three curve fitting problems over 50 runs. The test shows that networks found using evolution fits the data better on average than network found using random search. The difference is however more profound while using ReLU outputs.

6.2 Testing on real data

In this section we discuss the experimentation of the proposed algorithm on some simple benchmark machine learning problems. We run two regression experiments and two classification experiments. In this tasks both augmentation and pruning evolution are used. Augmentation happens when training error stagnates. Pruning is done when training error is reducing but validation error stagnates or increases.

Table 6.3: Error comparison for Balance regression data

Parameter	MLP1	MLP2	MLP3	Proposed method
Average Error	0.282	0.230	0.244	0.206
No. of parameters	61	171	281	73

Table 6.4: Error comparison for Abalone regression data

Parameter	MLP1	MLP2	MLP3	Proposed method
Average Error	1.64	1.68	1.73	1.53
No. of parameters	61	171	281	43

6.2.1 Balance scale data

In this test we take the balance scale dataset from the UCI repository. The original dataset os meant for classification. The are 4 attributes. Left weight, Left distance, Right weight, Right distance. The goal is to classify weather the system is left leaning, right leaning or balanced. We converted the problem to a regression problem in which given the inputs we have to calculate the net torque on the system. The regression labels are generated by calculating the torque. More over only 50 training samples are used to make generalization difficult. 50 samples are also used for validation and 100 samples are used for testing. The problem is solved using the proposed scheme and also using 3 different fixed topology multilayer perceptrons with 1, 2 and 3 hidden layers respectively with 10 nodes in each hidden layer. No regularization method is used to keep the comparison fair. The proposed algorithm can search for architectures indefinitely so a limit is imposed that at most 10 evolutionary phases will be allowed(augmentation and pruning combined).All activations are ReLU. The multilayer perceptrons are each trained 5 times with random initial weights and the best test accuracy is taken. The population for evolution is taken as 21. Remaining parameters are same as in the previous section. When no more evolution is allowed, the network is trained until no improvement in validation accuracy is seen. Table 6.3 shows the comparative results of per sample average error on test data.

6.2.2 Abalone rings prediction

In this experiment another dataset from UCI repository called Abalone dataset is considered. The goal here is to predict the number of rings in an abalone based in 8 attributes.This problem can be seen as both classification and regression. We are solving it as a regression problem. We are taking only 250 samples in training set. 500 samples are used in validation and 1000 are used in testing. As before few training samples are taken so that avoiding over fitting is difficult. The same setting is used as the previous problem. Table 6.4 shows the relative performance.

6.2.3 Balance classification

Here we again test on the balance dataset but this time we solve as a 3 class classification problem as already described. In the output layer ReLU is now replaced by softmax. Remain setting is the same.Results are now shown on misclassification percentage. Table 6.5 shows the results.

Table 6.5: Error comparison for Balance Classification data

Parameter	MLP1	MLP2	MLP3	Proposed method
Error %	12.8	11.6	8.4	14.8
No. of parameters	61	171	281	35

Table 6.6: Error comparison for Iris Classification data

Parameter	MLP1	MLP2	MLP3	Proposed method
Error %	1.33	2.67	2.67	1.33
No. of parameters	31	61	91	14

6.2.4 Iris classifiaction

Owing to the poor performance of the proposed method on the above classifiaction problem. We ran a test on the simpler Iris dataset. This dataset has 4 attributes and 3 output class. We used 75 samples for training.This time the perceptron layers have 5 nodes each instead of 10. Table 6.6 shows the results.

6.3 Discussion of Results

The proposed method outperformed the 3 basic fixed architecture networks on the regression problems even considering that the best accuracy out of 5 runs was taken for each network. But the method did poorly on the classification version of the same balance problem. All methods worked well on the Iris dataset which can be attributed to the simplicity of the problem.

Chapter 7

Conclusion and Future Work

We have discussed various schemes proposed for Automatic Network Search. We have identified that evolutionary approach has the maximum level of automaticity and can find novel architectures. We argued that these evolutionary methods suffer from high computational requirements and hence proposed that a greedy approach should be used to to carry out efficient evolution. We attempted to develop a simple greedy evolutionary scheme and demonstrated its superiority over random search on synthetic curve fitting problems. We then tested this method and compared its with certain basic fixed topology layered neural networks on real data while adding evolutionary pruning. The proposed method appears to perform better on the the regression problems although it fails on the classification version of similar problem. It performs well on Iris but this is likely due to the simplicity of the problem. The fact that even in synthetic curve fitting the gap between evolution and random search was more in case of ReLU outputs leads us to hypothesise that the proposed method is effective for ReLU and linear activation but not for sigmoidal or softmax activation and hence is applicable to regression problems. This may be related to the piece wise linear nature of ReLU outputs.Although the greedy neuroevolution problem is far from solved, our results are encouraging. Preferably evolving networks based on local criterion does appear to translate to global advantage in certain cases. The future challenges is now to formulate a mathematical understanding of how local performance improvements can be used to predict long term performance. This could lead to better fitness criteria which captures more information about the effect of an architectural modification. Our fitness criteria is very basic. Other more sophisticated fitness functions were tried but no noticeable improvements. Also our fitness function has a very direct relation with the end goal but ideally it should not be the case. Another challenge is to scale up these line of methods so that it can be applied to advanced learning problems. One suggested technique is to combine greedy evolution with forward thinking. The forward thinking algorithm requires manual choice of each layer parameter. Instead each layer can be evolved greedily. The real solution would however be to evolve novel global structures based on greedy local evolution. This is a difficult problem and further research is required.

Chapter 8

Bibliography

[1] F. ROSENBLATT. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1988.

[2] Steven Abreu. Automated architecture design for deep neural networks. *arXiv*, 2019.

[3] Seymour Papert Marvin Minsky. Perceptrons: an introduction to computational geometry. *The MIT Press*, 1969.

[4] Geoffrey E. Hinton Ronald J. Williams David E. Rumelhart. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[5] Maxwell Stinchcombe Halbert White KurtHornik. Multilayer feedforward networks are universal approximators. *Neural Networks*, 323:359–366, 1989.

[6] Maithra Raghu2 Jascha Sohl-Dickstein2 Surya Ganguli Ben Poole1, Subhaneil Lahiri1. Exponential expressivity in deep neural networks through transient chaos. 2016.

[7] V. N. Vapnik and A. Ya. Chervonenkis. On the importance of initialization and momentum in deep learning. *THEORY OF PROBABILITY AND ITS APPLICATIONS*, 1971.

[8] George Dahl Geoffrey Hinton Ilya Sutskever, James Martens. On the importance of initialization and momentum in deep learning. 2013.

[9] Haochuan Li-Liwei Wang Xiyu Zhai Simon S. Du, Jason D. Lee. Gradient descent finds global minima of deep neural networks. *arXiv*, 2019.

[10] John A. Hertz Anders Krogh. A simple weight decay can improve generalization. *Advances in Neural Information Processing Systems 4 (NIPS 1991)*, 1991.

[11] Alex Krizhevsky Ilya Sutskever-Ruslan Salakhutdinov Nitish Srivastava, Geoffrey Hinton. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 2014.

[12] Yoshua Bengio Yann LeCun. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361, 1995.

[13] Jurgen Schmidhuber Sepp Hochreiter. Long short-term memory. *Neural Computation*, pages 1735–1780, 1997.

[14] Geoffrey E. Hinton Alex Krizhevsky, Ilya Sutskever. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 2012.

[15] Karen Simonyan ∗ Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2015.

[16] Yangqing Jia Pierre Sermanet-Scott Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew Rabinovich Christian Szegedy, Wei Liu. Going deeper with convolutions. *arXiv*, 2014.

[17] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Deep residual learning for image recognition. *arXiv*, 2015.

[18] Roger Grosse. Exploding and vanishing gradients.

[19] Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv*, 2013.

[20] Annu Lambora ; Kunal Gupta ; Kriti Chopra. Genetic algorithm- a literature review. *IEEE*, 2019.

[21] Quoc V. Le Barret Zoph. Neural architecture search with reinforcement learning. *arXiv*, 2017.

[22] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. *Artificial neural networks: concept learning*, pages 112 – 127, 1990.

[23] Maxim Neumann Jonathon Shlens Wei Hua Li-Jia Li Li Fei-Fei Alan Yuille Jonathan Huang Kevin Murphy Chenxi Liu, Barret Zoph. Progressive neural architecture search. *arXiv*, 2017.

[24] Mengjie Zhang Gary G. Yen Yanan Sun, Bing Xue. Automatically designing cnn architectures using genetic algorithm for image classification. *arXiv*, 2020.

[25] Babak Hodjat Dan Fink Karl Mutch Risto Miikkulainen year = 2019-month = pages = title = Evolutionary Neural AutoML for Deep Learning volume = journal = arXiv Jason Liang, Elliot Meyerson.

[26] Risto Miikkulainen Kenneth O Stanley. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10:99 – 127, 2002.

[27] Jeff Schneider Barnabas Poczos Eric Xing Kirthevasan Kandasamy, Willie Neiswanger. Neural architecture search with bayesian optimisation and optimal transport. *arXiv*, 2019.

[28] Hugo Larochelle Jasper Snoek. Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 2012.

[29] SE Fahlman. The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems*, 1990.

[30] Mike Wynne-Jones. Node splitting: A constructive algorithm for feed-forward neural networks. *Advances in Neural Information Processing Systems*, 1992.

[31] Jonathon Shlens. A tutorial on principal component analysis. 2005.

[32] Douglas Reynolds. Gaussian mixture models. *Encyclopedia of Biometrics*, 2009.

[33] Ben Ehlert Jeffrey Humpherys Tyler Jarvis Sean Wade Chris Hettinger, Tanner Christensen. Forward thinking: Building and training neural networks one layer at a time. *arXiv*, 2017.

[34] Darrell Whitley. Applying genetic algorithms to neural network learning. 1989.

[35] Kenneth O. Stanley ; David B. D'Ambrosio ; Jason Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. *Artificial Life*, 15:185 – 212, 2009.

[36] Yiming Yang Hanxiao Liu, Karen Simonyan. Darts: Differentiable architecture search. *arXiv*, 2019.

[37] Adam Stanton. ncremental neuroevolution of reactive and deliberative 3d agents. *European Conference on Artificial Life*, 2015.

[38] Michael Carbin Jonathan Frankle. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv*, 2019.