Spring 2020

Computational model for neural architecture search

Ram Deepak Gottapu

COMPUTATIONAL MODEL FOR NEURAL ARCHITECTURE SEARCH

by

RAM DEEPAK GOTTAPU

A DISSERTATION

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

SYSTEMS ENGINEERING

2020

Approved by

Dr. Cihan H Dagli (Advisor)
Dr. Ruwen Qin
Dr. Benjamin Kwasa
Dr. Zhaozheng Yin
Dr. Donald Wunsch

ABSTRACT

A long-standing goal in Deep Learning (DL) research is to design efficient architectures for a given dataset that are both accurate and computationally inexpensive. At present, designing deep learning architectures for a real-world application requires both human expertise and considerable effort as they are either handcrafted by careful experimentation or modified from a handful of existing models. This method is inefficient as the process of architecture design is highly time-consuming and computationally expensive.

The research presents an approach to automate the process of deep learning architecture design through a modeling procedure. In particular, it first introduces a framework that treats the deep learning architecture design problem as a systems architecting problem. The framework provides the ability to utilize novel and intuitive search spaces to find efficient architectures using evolutionary methodologies. Secondly, it uses a parameter sharing approach to speed up the search process and explores its limitations with search space. Lastly, it introduces a multi-objective approach to facilitate architecture design based on hardware constraints that are often associated with real-world deployment.

From the modeling perspective, instead of designing and staging explicit algorithms to process images/sentences, the contribution lies in the design of hybrid architectures that use the deep learning literature developed so far. This approach enjoys the benefit of a single problem formulation to perform end-to-end training and architecture design with limited computational resources.

ACKNOWLEDGMENTS

I would like to express my heartful thanks to my advisor, Dr. Cihan Dagli, since it has been a great honor and privilege working with him and with the group, and for his conscientious guidance throughout my five years of research life. I would also like to thank my committee members, Dr. Ruwen Qin, Dr. Benjamin Kwasa, Dr. Zhaozheng Yin, Dr. Donald Wunsch for their perceptive suggestions during my study and research.

Thanks go to all the faculty and staff at the EMSE department as well for their efforts to create an efficient and friendly working environment in SESL lab. I would also like to thank my family and friends for their endless motivation and support to me.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

1. INTRODUCTION

1.1. MOTIVATION

The last few years have seen much success of deep neural networks (DNNs) in many challenging applications such as image recognition, speech recognition, and machine translation. DNNs learn by using multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level [16]. With the composition of enough such transformations, very complex functions can be learned from data using a general-purpose learning procedure. The key aspect of deep learning is that the composition, placement, and connectivity between the modules within the architecture play an important role in regard to its performance. For example, Figure 1.1 shows two 2-hidden layered architectures with their performances and total trainable parameters. The architecture on right has better performance even though it uses less parameters than on the left because of its complex connectivity and composition. As deep learning models generally consist of multiple layers (even more than 100 in some cases [8, 9, 12]), architecting them to find a design that gives optimal performance is an arduous task and requires a lot of expert knowledge and ample time. It is therefore not surprising that in recent years, techniques to automatically design these architectures have begun gaining popularity. These methodoligies usually build a search space and evolve the design within the constraints of the space to eventually find an optimized architecture [19, 22, 24, 34, 35]. This dissertation investigates the state-of-the-art techniques used for designing deep learning models and present new approaches to improve the automation. Concretely, the research is focused on searching large search spaces both quickly and efficiently to produce optimal architectures for a given dataset.

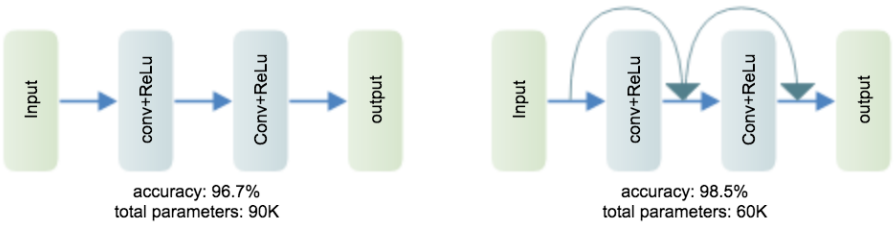


Figure 1.1. Two architectures trained on MNIST dataset. Both are trained with learning rate: 0.01, batch size: 62, epochs: 70. The architecture on right uses more connections between layers with compressions. This gives better performance with fewer parameters when compared with the architecture on left.

1.2. CHALLENGES

Even though it is evident that deep learning models learn by extracting features at multiple levels, the exact approximation that happens within different layers is difficult to understand, much less tune the architecture based on that. As such it is essential that the search space used to build efficient architectures must accommodate the complex interactions while having finite number of combinations so that the search algorithm can find the best suitable combination for the given dataset. In addition to search spaces, deep learning networks are often associated with huge computation as the trainable parameters range from hundreds of thousands to millions. Therefore, any search associated with them involves training each modified architecture from scratch which is highly computationally expensive and not feasible for practical applications. In order to be able to perform search efficiently, novel search approaches are necessary to control the computation costs so that they can be applied for any given dataset.

1.3. ENCOURAGING PROGRESS

Despite the difficulty of the task, there is a rapid progress in the area of architecture search. In particular, the state-of-the-art models related to architecture search have been able to find efficient designs using novel search spaces while being computationally efficient. For example, the initial designs used as many as 800 GPUs to search for an efficient architecture [34] while the latest models use only 1 GPU with similar performance [19, 22].

1.4. OUTLINE OF CONTRIBUTIONS

The approaches in this dissertation improve upon the existing state-of-the-art deep learning models in the field of search space and computational requirements. Sections 2 and 3 provide necessary background related to the concepts presented in this dissertation.

Section 4 presents a generalized platform which uses system architecting principles to find an efficient deep learning architecture. System architecting approaches are generally used for developing very complex systems with multiple objectives. Therefore, it is a provides a very optimal way to experiment with different search spaces, objectives and even participating layers. The platform described in this part will be used in the following sections.

Section 5 introduces a parameter sharing approach which greatly reduces the computation during the architecture search process. The approach focuses on modifying the existing parameter dimensions as per the requirement instead of creating new parameters for each architecture generated during the search process.

Section 6 extends the NAS to multi-objective problem where the optimal architectures are designed based on device configurations in addition to validation accuracy.

1.5. LIMITATIONS OF USING THIS APPROACH

Although the approaches presented in this dissertation are capable of searching efficient architectures, it is still not capable of finding perfect architectures for a given dataset. For instance, the weight modification approach presented in Section 5 is able to reduce the computation time required to perform the search. However, it ignores the fact that no two architectures use the parameters in the same fashion. This shows that there is still a room for improving the way the parameters are shared during the search process. In addition to that, the approaches presented in this dissertation only improve the search process but they do not increase the current best accuracies. Therefore, in order to build a perfect classification model for any given dataset, it is mandatory that more robust search designs are required.

2. LITERATURE REVIEW

This section insvestigates the progress made in designing deep learning architectures by describing how different contributions helped the transition into neural architecture search approaches. It also descrbes state-of-the-art search spaces and techniques used for current architecture search designs.

2.1. DEEP LEARNING HISTORY

The history of deep learning algorithms starts with the MARK I Perceptron in 1957 by Frank Rosenblatt which uses a weight, bias and update rule to solve a binary classification problem. Later in 1960, Widrow and Hoff developed Adaline/Madaline using the concept of stacking to generate multi-layer perceptrons which closely resembles the modern day neural networks. In 1986, Rumelhart introduced backpropogation which gave a principled way to train the multi-layered perceptrons and neural networks [25]. However, these approaches were not popularly used as they are not suitable for large problems mainly due to lack of computation during that time. In 2006, Hinton et al. showed that it is possible to train a deep neural network if sufficient computation is provided [10]. Finally around 2010-2012 (see Figure 2.1), with the improvements made in computational resources (GPUs), the first strong results for deep learning architectures were produced in acoustic modelling [20], speech recognition [6] and image classification [15]. Since then a significant amount of research was made towards the exploration of deep learning architectures to design more efficient architectures.

This dissertation mainly concentrates on the exploration part of the architectures specifically for images (CNNs) and natural language (RNN/LSTM) to some extent, as they are the most popular applications. The following sections describe different case studies

of CNNs from 1998, which led to state of the art models. Many of these models used ImageNet Large Scale Visual Recognition Challenge (ILSVRC) as a standard reference as shown in Figure 2.1.

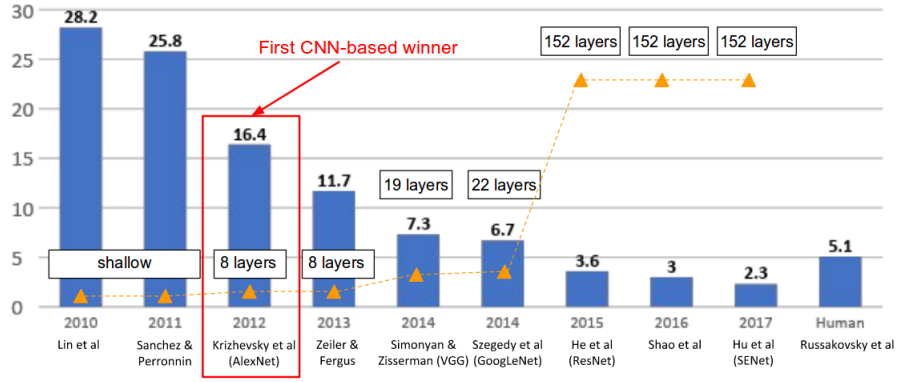


Figure 2.1. Histogram plots showing the evolution of architecture performances over the last couple of years. Each architecture was a winner of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in respective years.

2.1.1. Lenet-5. It is the first instantiation of CNN by LeCun which was successfully used in practical application i.e. digit recognition [17]. It used the format: [CONV-POOL-CONV-POOL-FULLY CONNECTED-OUTPUT] as shown in Figure 2.2.

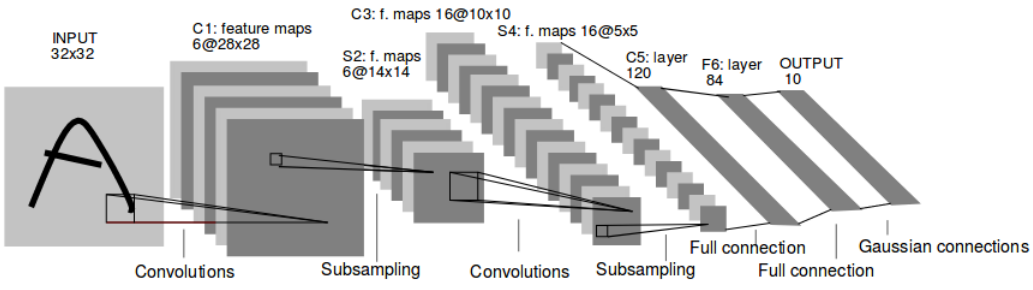


Figure 2.2. LeNet architecture. The architecture was used for MNIST dataset and the first practical application of CNN model [17].

2.1.2. AlexNet. It is the first large scale CNN designed by Krizhevsky which outperformed all the previous non-deep learning based algorithms by significant margin in IMAGENET classification challange [15]. It is also responsible for starting the research related to CNN architecture design. The architecture as shown in Figure 2.3, consists of 8 layers and was trained on IMAGENET dataset (which has 227x227x3 images) using two GTX 580 GPU each with only 3GB memory.

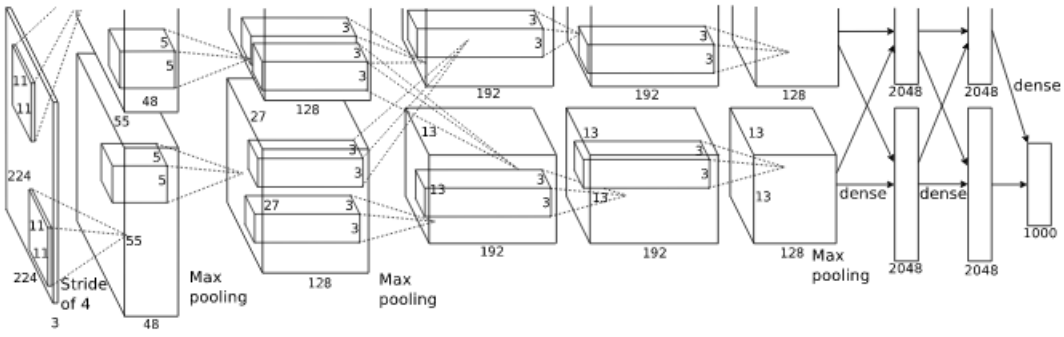


Figure 2.3. AlexNet architecture. The architecture was separated into two parallel processes to distribute the computation [15].

2.1.3. ZFNet. It is the winner of 2013 IMAGENET challange and developed by Zeiler and Fergus [33]. It has the same structure as AlexNet but with improved hyper-parameters. Figure 2.4, show the overall implementation of ZFNet. This implementation showed that it is possible to significantly improve the architecture performance just by tuning the hyper-parameters.

2.1.4. VGG and GoogLeNet. In 2014 IMAGENET challange, there are two important architectures that are very close in terms of performance. The VGGNet [28] by Oxford university consisted of 19 layers deep (significantly deeper than previous models) and uses small filters (3x3) instead of large filters (say 7x7) unlike its predecessors. They showed that a stack of three 3x3 conv filters has the same effective receptive field as one

7x7 conv filter but has added advantage of more non-linearities. Figure 2.5 shows different version of VGG net. Note that the results in [28] are achieved after using ensembles of different versions.

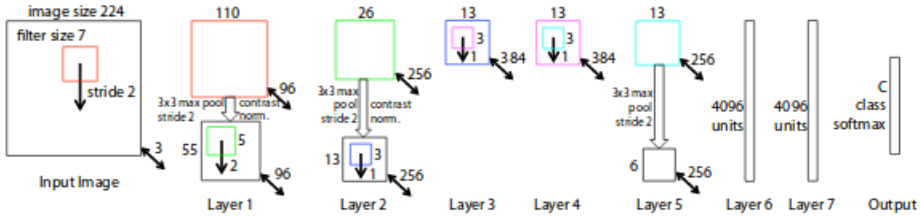


Figure 2.4. ZFNet architecture. It shows the tuning of hyperparameters [33].

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.5. VGGNet architectures of different versions/layers. The best results were obtained by taking ensembles of one/many versions [28].

Similar to VGG net, the GoogLeNet [30] is also a deep network with 22 layers but designed for computational efficiency. The efficiency was achieved by stacking up inception modules instead of convolutional layers as shown in Figure 2.6. The inception module consists of multiple layers in parallel built on top of single input layer whose outputs are concatenated depth-wise at the concat node. It used only 5M parameters (which is 12x less than AlexNet). Note that the naive inception module is very computationally expensive (uses 854M ops). In order to reduce the computational complexity, they used 1x1 conv (also called bottleneck layers) to reduce the feature depth.

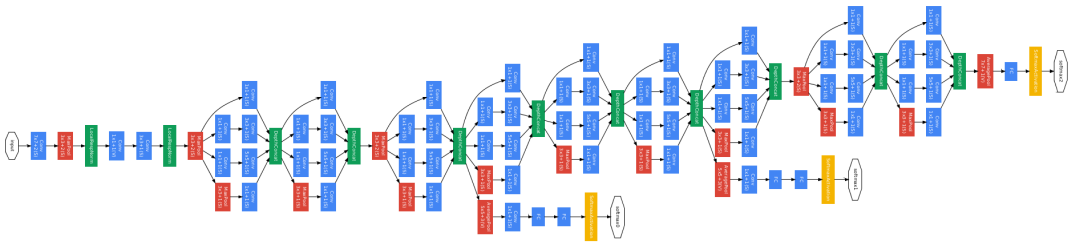


Figure 2.6. Complete architecture design of GoogLeNet. The architecture is made of multiple inception modules stacked on top of each other [30].

2.1.5. Observations for Very Deep Architectures. Since deeper architectures are capable of giving high accuracies, the authors in [8] tried to further increase the depth without increasing the width. The plots in Figure 2.7 show that the 56 layered network has bad performance when compared to 20 layer network in both training and validation error. In order to overcome this problem, they hypothesized that the degraded performance may be related to optimizing very deep networks and they used the concept of blocks which contain residual mapping instead of direct mapping as shown in Figure 2.8. This ensures better information flow in very deep architectures during optimization which led to the development of ResNet.

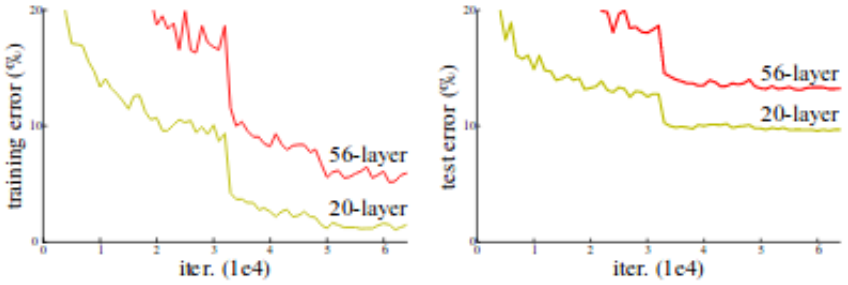


Figure 2.7. Performance of "plain deep CNN networks" at two different depths. Both training and test plots show degraded performance with the increase in depth [8].

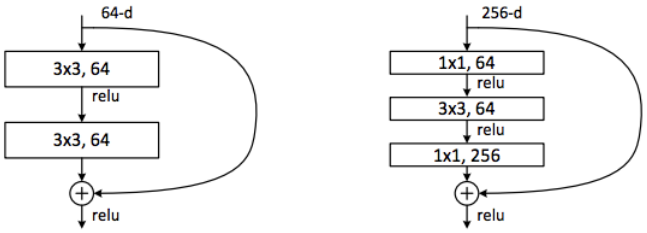


Figure 2.8. Residual mapping for ResNets. [8].

2.1.6. ResNet. Similar to GoogLeNet which increased the width of the architecture, the ResNet [8] increased the depth of the architecture using the hypothesis defined above i.e. it used 152 layers with blocks having 1x1 and 3x3 convolutions and residual connections as shown in Figure 2.9. In 2015 IMAGENET challange, it beat all the previous architectures.

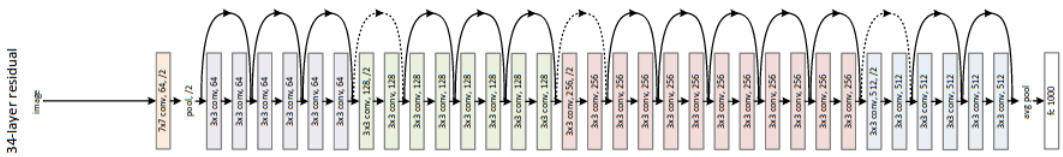


Figure 2.9. ResNet architecture. The skip connections act as identity mappings [8].

Improvements on ResNets: There are many architectures that were developed based on ResNet aiming to further improve the performance. These models did not participate in IMAGENET challenge but they did provivide improved results on the dataset.

2.1.7. Identity Mappings in Deep Residual Networks. In this paper [9], the authors improved ResNet block by creating more direct path for propagating information throughout network (see Figure 2.10). The addition of activation improved the performance of ResNet.

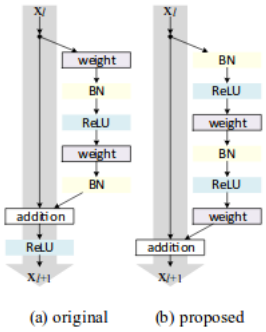


Figure 2.10. Improved ResNet. The identity mappings in ResNet blocks have additional layers. [9].

2.1.8. Wide Residual Networks. In this architecture [32], the authors argued that that very deep residual networks has a problem of diminishing feature reuse, which makes these networks very slow to train. To avoid this, they used wider residual blocks (FxK filters instead of F filters in each layer) as shown in Figure 2.11.

2.1.9. ResNext. In this architecture [31], the authors tried to combine the inception module and ResNet block to get a design with fewer parameters. A ResNext block is shown in Figure 2.12. This design secured 2nd place in the 2016 ILSVRC challange.

2.1.10. Deep Neworks with Stochastic Depth. Since very deep networks also suffer from vanishing gradient probles, the authors in this paper [13] used short networks during training. This is achieved by randomly dropping a subset layers of a very deep

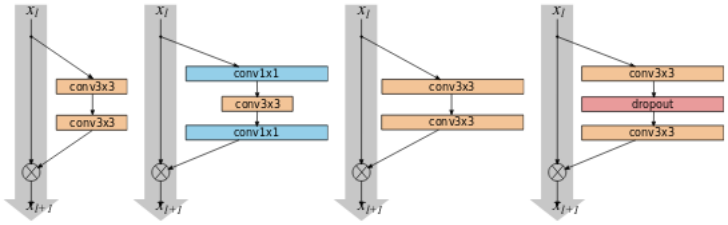


Figure 2.11. Wide ResNet. Modifications made on filters for the blocks used in ResNet architecture [32].

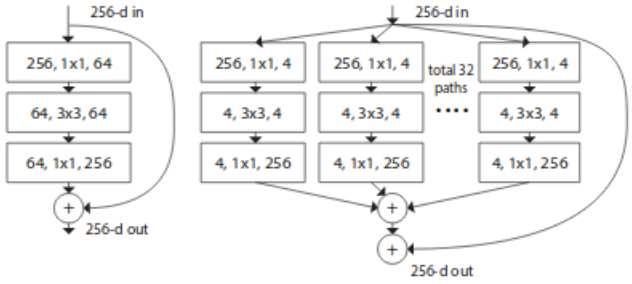


Figure 2.12. ResNext architecture. It shows the combination of ResNet and Inception module [31].

network during each training pass and bypassing them with identity function as shown in the Figure 2.13. However, during the test time, the complete architecture is used. This approach allowed to train very deep networks (beyond 1200 layers) while also reducing the training time.

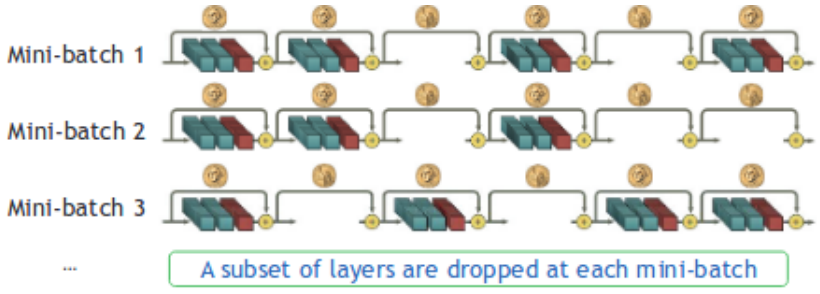


Figure 2.13. Deep architectures trained by randomly dropping layers.

2.1.11. DenseNets. In this architecture [12], each layer is connected to every other layer in the network as shown in the Figure 2.14. This approach alleviates vanishing gradient, strengthens feature propagation and encourages feature reuse. The architecture not only beat all the previous designs in performance but did it with significantly lower parameters.

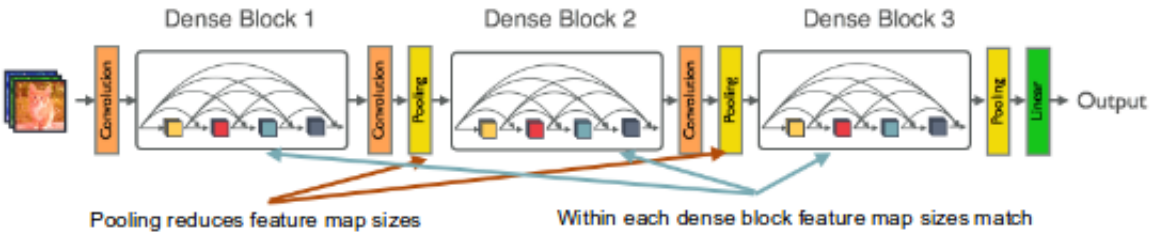


Figure 2.14. DenseNet architecture. It shows blocks with all possible connections between its layers [12].

2.1.12. Takeaways from the Models. The following points describe the overall observations from different models:

- Tuning the parameters improves model accuracy.
- Increasing width increases learning capability.
- Bottleneck layers improve computational efficiency.
- Increasing depth eventually degrades the accuracy.
- Increasing depth with residual connections allows us to create deeper networks without degrading accuracy to a certain extent.
- Innovative connections between the layers improve accuracy

2.2. ARCHITECTURE SEARCH APPROACHES

From the models described above, it is now clear that there are many factors that influence the overall performance of an architecture for a given dataset. Also, manual search approaches considering all the above mentioned discoveries is not feasible because of the large search space associated with it. For example, designing a hybrid 100+ layer architectures while choosing appropriate depth, width, filter composition and connections among them will have millions of combinations and manual search is not possible to get an optimal architecture considering all the factors for a given dataset. Therefore, a learning to learn approach is essential in order to find efficient architectures using the available choices. The popular approaches used in this regard are the evolutionary algorithms (EA) and reinforcement learning (RL). Many attempts were made to design efficient architectures using these algorithms [12,13,14] but they were not able to perform better than man-made architectures. One reason for such performance can be attributed to computational requirements as each new architecture generated in the search process has to be trained from scratch which is both time consuming and expensive.

2.2.1. NAS. In [34], the authors conducted a massive experiment using reinforcement learning with 800 GPUs to search for efficient architecture that can out perform all the man made architectures. The experiment took 24 days using all GPUs with the goal of finding the best filter dimensions, filter layers and connections for each layer in a fixed length architectures (15 layers). The resulting architecture did not out perform DenseNet [12] but was able to produce results very similar to it. Figure 2.15 shows the first automatically generated architecture that was able to compete with human-designed architectures.

2.2.2. NASNET. The above experiment showed that it is possible to design efficient architectures using learning to learn process. However, the approach is not feasible as it required huge computational resources. In order to reduce the computation required for searching the architectures, the authors in [35] came up with a new search space call neural architecture search (NAS) where, instead of searching the complete architecture, they used

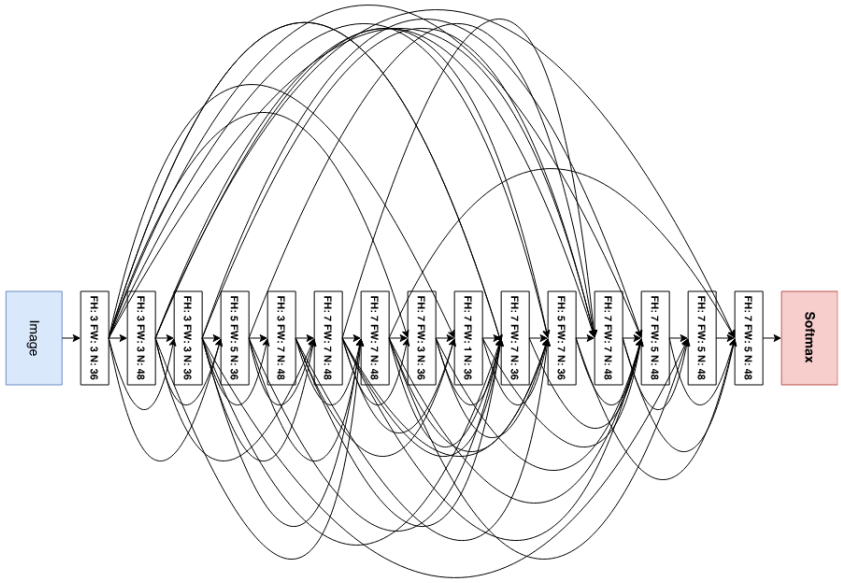


Figure 2.15. NAS with reinforcement learning. It is the first architecture designed using evolution from reinforcement learning [34].

cell based approach. The search is applied with respect to these cells which are stacked upon each other in a pre-determined architecture as shown in Figure 2.16. Figure 2.17 shows the normal cell and reduction cells that are learnied using this search space. The normal cells maintain the dimensions of input while the reduction cell acts as pooling layer. This approach also uses reinforcement learning to evolve the population. The experiment took approximately 48 hours using 400 GPUs which is significant improvement over the previous approach. Even though it is still not feasible for general usage because of the number of GPUs it used, it is the first architecture that was able to out perform all the man made architectures.

2.2.3. ENAS. Finally in [22], the authors present efficient neural architecture search using parameter sharing. The approach uses the same RL approach, but instead of training each new architecture from scratch, they considered each operation as a node in a graph and the goal is to find the optimal sub-graph (see Figure 2.18). Therefore, whenever a particular node is selected, it uses the previously trained weights instead of training from scratch.

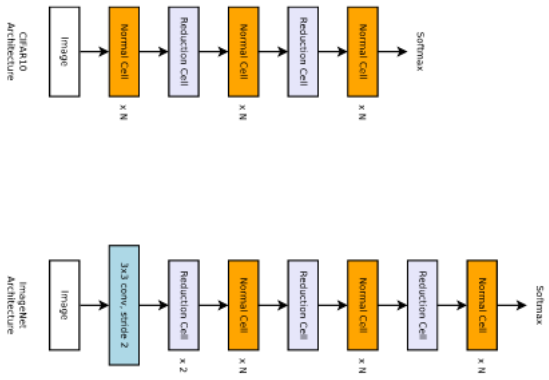


Figure 2.16. NasNet template. Pre-determined design for architecture search [35].

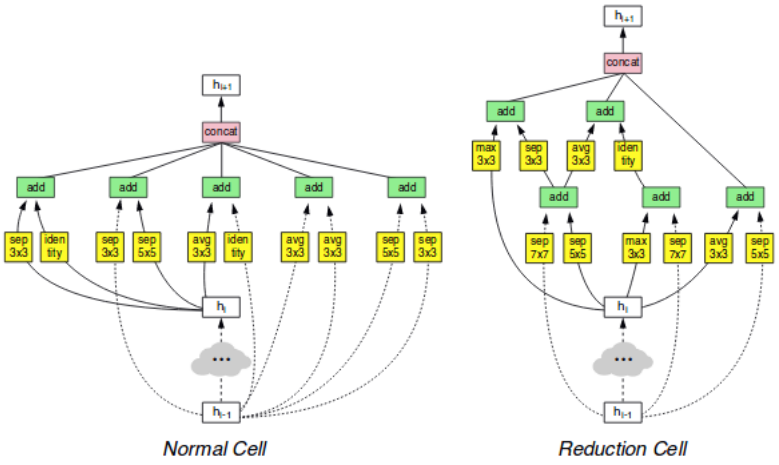


Figure 2.17. NasNet results. Cells designed from improved search space for NasNet [35]. Computation reduced from approx 800 GPUs to 400 GPUs.

This gave performance similar to that of NAS but used only 1 GPU. This is a massive improvement from previous approaches since we are able to find efficient architecture using only one GPU which is feasible for general purposes.

2.2.4. DARTS. Following ENAS, the authors in [19] proposed DARTS: Differentiable Architecture Search which used a differential optimization approach by converting the NAS into a bi-level optimization problem. This is achieved by continous relaxation of discrete architecture representation and performing search using gradient descent. Once the

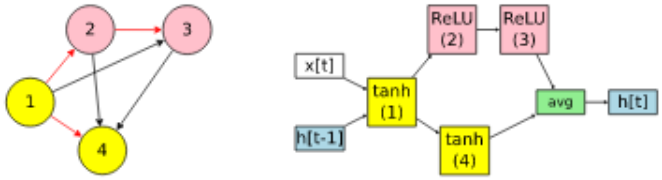


Figure 2.18. Graph representation of cells in ENAS approach. The computation reduced from 400 GPUs with 2 day training to 1 GPU with 1 day training [22].

architecture is designed, the continous representation is converted back to discrete architecture. This approach also used parameter sharing and was able to produce slightly better results than ENAS. Figure 2.19 shows the steps used in DARTS.

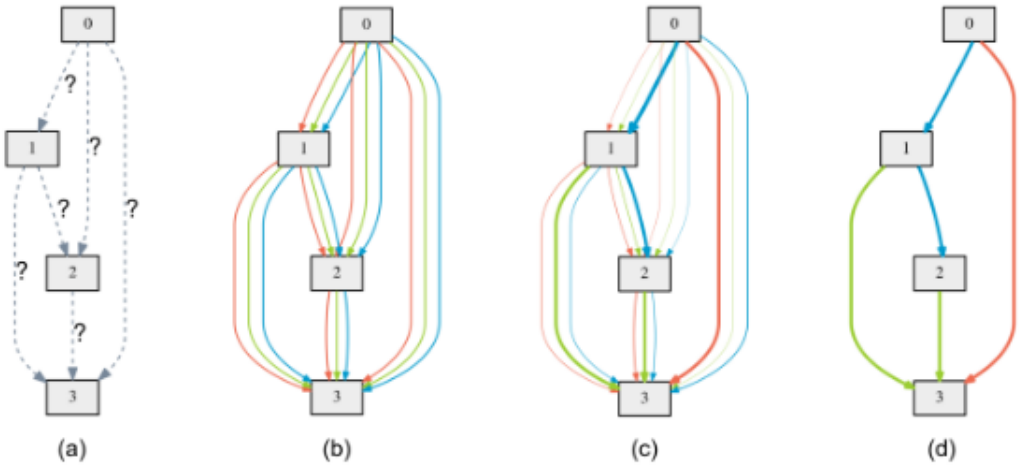


Figure 2.19. An overview of DARTS. (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities. [19].

Considering all these improvements in architecting deep learning models, it is obvious that the direction of research is towards using efficient learning to learn approaches. This dissertation provides solutions to further improve the search speed and also to search large search spaces more efficiently.

3. THEORITICAL BACKGROUND

This section provides a brief introduction to all the concepts and algorithms used in this dissertation. For a more thorough and slower-paced introduction on deep learning, please refer to [7].

3.1. SUPERVISED LEARNING

Supervised learning is a branch of machine learning that performs a mapping $f : X \rightarrow Y$ using a set of collected examples, where X is input space and Y is output space. In case of images, $X \in$ collection of images and $Y \in$ corresponding labels which are labelled manually.

Objective: Consider a training dataset of n samples $\{(x_1, y_1)(x_2, y_2)...(x_n, y_n)\}$ made up of independent and identically distributed (i.i.d) samples from a data generating distribution D; i.e. $(x_i, y_i) \in D \ \forall \ i$. If $L(\hat{y}, y)$ is some loss function that calculates the disagreement between predicted label $\hat{y_i} = f(x_i)$ for some $f \in F$ and true label y_i, then the objective is to find $f^* \in F$ which satisfies:

$$f^* = \underset{f \in F}{\operatorname{argmin}} E_{(x,y)\sim D} L(f(x), y) \tag{3.1}$$

In other words, the goal is to find a function f^* that minimizes the expected loss over the data generating distribution D.

Limitations and Assumptions: Unfortunately, the optimization problem defined in Equation 3.1 is unmanageable as it is not possible to have access to all possible elements in D and therefore cannot evaluate the expectation (E) without making unrealistically strong

assumptions on D, L, f. However, under the i.i.d assumption we can approximate the expected loss in Equation 3.1 above with sampling by averaging the loss over available training data.

$$f^* = \underset{f \in F}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) \tag{3.2}$$

In other words, the loss is optimized only over training samples and hope that it is a good proxy equation that closely resembles the actual objective equation in Equation 3.1.
Example: Neural Network Classification

In order to understand the objective function described in Equation 3.2, consider the neural network which consists of N data points each with dimension of H and three possible classes C. The prediction for the network will be: $f(x) = W_2\ tanh(W_1^T\ x\ +\ b_1)\ + b_2$ where W_2 is a KxC matrix (C = 3 in this example). The output $f(x)$ will therefore be a 3-dimensional vector. Note that it is common practice to represent the output vector as logits by passing them through a softmax function when dealing with classification problems. The softmax function takes a vector z as input and outputs the same dimensional vector p, where $p_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$. Note that the vector z can contain arbitrary real-valued quantities, but the vector p is normalized so that all of its values are in the range 0-1 and they sum to 1. This vector p is essentially the prediction ($\hat{y}_i$) from the network and the correct label y is a 3-dimensional vector that is all 0 except for a single 1 at the index of true class (E.g. [0,0,1] for class 3). The most commonly used loss function for classification problem is cross-entropy loss as shown in following equation.

$$L(\hat{y}, y) = -\sum_{k=1}^{K} y_k log\hat{y}_k = -log\hat{y}_y = 1 \tag{3.3}$$

where the first equality is the actual cross-entropy definition and second equality is a simplified form for classification. Since the interpretation of the network output is a probability distribution for C=3 different classes, the minimization in Equation 3.2 is basically the minimization of negative log probability of correct class.

3.2. BACKPROPAGATION

By evaluating the gradients of the loss function and using stochastic gradient descent (SGD) to minimize the loss, it is possible to find a mapping function $f \in F$ which can map $X \rightarrow Y$ consistent with the training data. This section describes the process of backpropagation which efficiently computes gradients of scalar valued functions with respect to their inputs. For any feed forward network including deep neural networks, the prediction y$\hat{y}$ for any input x is made by passing the information forward through the network. During training, the forward pass continues onward until it produces a scalar cost value. The backpropagation, allows the information from the cost to flow backwards in order to compute the gradients. Consider the example shown in Figure 3.1.

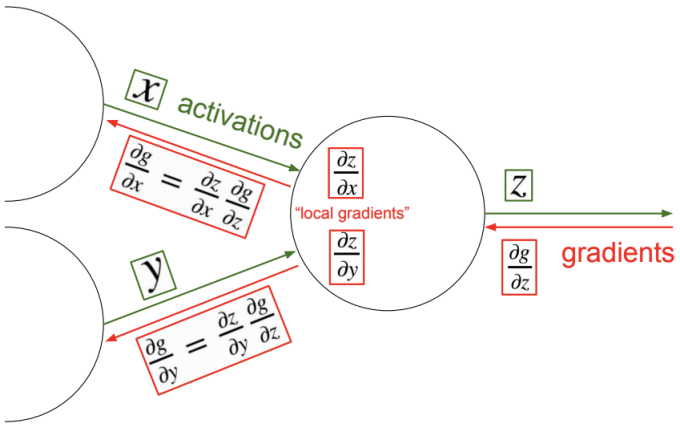


Figure 3.1. An example of backpropagation along a computational graph.

During forward pass x and y take on specific (numerical) values and the vector (or scalar) z is computed using some fixed function (e.g. $z = xy$). Notice that it is possible immediately compute the Jacobian matrices dz/dx and dz/dy of this transformation using calculus, because of the knowledge of what function z is computing in the forward pass. These tell us what first order influence x and y have on the value of z. The value z goes off into a computational graph and eventually at the end of the graph the total loss g (a scalar) is computed. The backward pass proceeds in the reverse order, recursively applying the chain rule to find the influence of all inputs of the graph on the final output. In particular, this computational unit finds out what dg/dz is, telling us how z influences the final graph output. The chain rule states that to backpropagate this we should take the global gradient on z, dg/dz and multiply it onto the local gradients for each input. For example, the global gradient for x will become $dg/dx = dz/dx\ dg/dz$. If x, z are vectors then this is a single matrix-vector multiplication. The gradient is then recursively chained, in turn, through the functions that produced the values of x and y until the inputs are reached. In neural network applications, the inputs we are interested in are the parameters, and their gradient tells us which way they should be nudged to decrease the loss.

Computational Graph Representation: Instead of thinking of the computational process of a Neural Network as a linear list of operations it is more intuitive to think about the function from inputs to outputs as a directed acyclic graph (DAG) of operations, where vectors flow along edges and nodes represent differentiable transformations that consume some number of vectors and combine them to one vector that then flows to other nodes. Most implementations of backpropagation organize the code base around a Graph object that maintains the connectivity of operations (also called gates, or layers) and a large collection of possible operations. Both Graph and Node objects implement two functions: forward() and backward(). The Graph's forward() iterates over all nodes in the topological order and calls their forward(), and its backward() iterates in the reverse order and calls their backward(). Each Node computes its output with some function during its forward() call,

and in backward() it is given the gradient of the loss function with respect to its output and returns the 'chained' gradients on all of its inputs. Here the 'chained implies taking the gradient on its output and multiplying it by its own local gradient (the Jacobian of this transformation). This gradient then flows to its children nodes that perform the same operation recursively. As a last technical note, if the output of a node is used in multiple places in the graph then correct behavior by the multivariable chain rule is to add the gradients during backpropagation along all branches. This would be handled in the Graph object.

3.3. REGULARIZATION

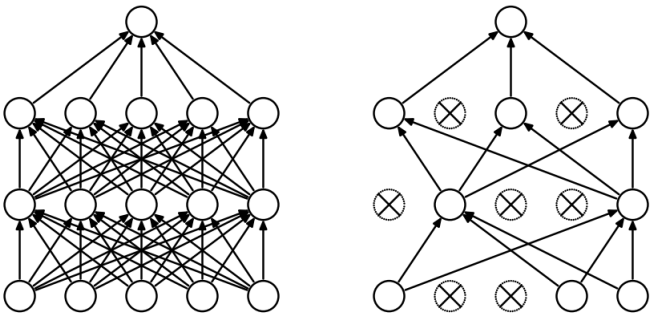
Unfortunately, optimizing Equation 3.2 instead of Equation 3.1 poses challenges. For instance, consider a function f that maps each x_i in the training data to its y_i but returns zero everywhere else. This would be a solution to Equation 3.2 (for any sensible loss function L that achieves a minimum value when $y = \hat{y}$), but we would expect very high loss for all other points in D that are not in the training set. In other words, we would not expect this function to generalize to all $(x, y) \sim D$. An additional concern is that there may be many different functions that all achieve the same loss under Equation 2.2 (so there is no unique solution), but their generalization outside of the training data could vary. If there is only training data then the challange is to choose among an entire set of $f \in F$ that all achieve the same loss in Equation 3.2? Both of these concerns can be alleviated by using regularization. There are a couple of regularization types in machine learning, but the followin are popular in deep learning.

3.3.1. L2 Regularization. This regularization works by adding the term R to the objective:

$$f^* = \underset{f \in F}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} L(f(x_i), y_i) + R(f) \tag{3.4}$$

where R is a scalar-valued function that encodes preference for some functions over others, regardless of their fit to the training data. This addition can be partly justified as following the principle of Occam's razor, which could be stated as: 'Suppose there exist two explanations for an occurrence. In this case the simpler one is usually better'. Put in another way, the regularization is a measure of complexity of a function. Together with the regularization term, the objective in Equation 2.3 encourages simple solutions that also fit the training data well, and its intended effect is to some extent compensate for the discrepancy between the objective in Equation 3.2 and Equation 3.1.

3.3.2. Dropout. The key idea is to randomly drop units (along with their connections) from the neural network during training as shown in Figure 3.2. This prevents units from co-adapting too much. During training, dropout samples from an exponential number of different 'thinned' networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.



(a) Standard Neural Net (b) After applying dropout.

Figure 3.2. Dropout Neural Net Model. *Left:* A standard neural net with 2 hidden layers. *Right*: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

3.4. CONVOLUTIONAL NEURAL NETWORK

Convolutional neural networks (CNNs) are a class of deep neural networks designed for handling data with some spatial topology (e.g. images, videos, sound spectrograms in speech processing, character sequences in text, or 3D voxel data). In each of these cases an input example x is a multi-dimensional array (i.e. a tensor). CNNs use relatively little pre-processing compared to other supervised learning methods. This implies that the network learns the filters that in traditional algorithms were hand-engineered which gives a major advantage over other traditional algorithms.

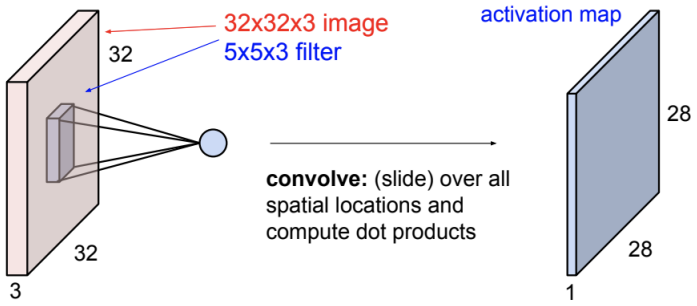


Figure 3.3. CNN model. Illustration of convolving a 5x5 filter (which we will eventually learn) over a 32x32x3 input array with stride 1 and with no input padding. The filters are always small spatially (5 vs. 32), but always span the full depth of the input array (3). There are 28 x 28 unique positions for a 5 x 5 filter in a 32 x 32 input, so the convolution produces a 28 x 28 activation map, where each element is the result of a dot product between the filter and the input. A convolutional layer has not just one but a set of different filters (e.g. 64 of them), each applied in the same way and independently, resulting in their own activation maps. The activation maps are finally stacked together along depth to produce the output of the layer (e.g. 28 x 28 x 64 array in this case).

3.4.1. Conv Layer. The core computational building block of a Convolutional Neural Network is the Convolutional Layer (or the CONV layer) which takes an input tensor and produces an output tensor by convolving the input with a set of filters. For instance, consider the example shown in Figure 3.3. The input is a color image with dimensions

32x32x3 while the choice of filter is 5x5x3 (= 75 parameters which are trainable). The convolution happens by sliding the filter across the spatial points of the input tensor (one step at a time i.e. stride = 1) and computing the dot product between local values of X and w. This produces an activation map which will have the dimension 28x28x1. It is common to use multiple such filters in each layer (say 32) and the output will have a dimension of 28x28x32. Also, for very deep CNNs (which is common for CNNs) it is necessary to maintain the dimensions so as to be able to stack multiple layers. For this purpose, the input at each layer is padded with a border of zeros. If a pad of 2 is added to the input tensor x in Figure 3.3 and the convolution is performed with same stride value (stride = 1), the resulting activation will have the same height and width as that of input tensor x i.e. 32x32x32. As the complete architecture is optimized using loss, each filter will develop the capacity to look for certain local features in the input tensor x. Steps performed during conv operation:

1. Input a tensor x with dimensions W_1xH_1xC_1
2. Define configuration of conv: stride s, pad p, number of filters K and spatial extent of filter F.
3. Produce a activation map with dimensions W_2xH_2xC_2 where $W = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)/S + 1$ and $C_2 = K$.

3.4.2. Pool Layer. In addition to convolutional layers, to further control overfitting it is common to use pooling layers that decrease the size of the representation with a fixed downsampling transformation (i.e. without any parameters). In particular, the pooling layers operate on each channel (activation map) independently and downsample them spatially. A commonly used setting is to use 2x2 filters with stride of 2, where each filter computes the max operation (i.e. over 4 numbers). The result is that an input tensor is downscaled exactly by a factor of 2 in both width and height and the representation size is reduced by a factor of 4, at the cost of losing some local spatial information.

3.4.3. ConvNet Architectures. Finally, a convolutional network is built by stacking convolutional layers and possibly introducing pooling layers to control the computational complexity of the architecture. A typical convolutional neural network architecture that processes images might take the form [INPUT, [CONV, CONV, POOL] x 3, F C, F C]. Here, INPUT represents a tensor of a batch of images (e.g. [100 x 32 x 32 x 3] for a batch of 100 32 x 32 color images) CONV is a convolutional layer with 3x3 filters applied with padding of 1 and stride of 1, POOL stands for a typical 2x2 filter max pooling layer with stride of 2, and FC are fully-connected layers, where the last one computes the logits of different classes just before a softmax classifier. In this architecture the spatial size of the input is reduced by a factor of 2 in both width and height after each POOL layer, so after the third POOL layer the spatial size of the representation along width and height would be 4x4.

3.5. RECURRENT NEURAL NETWORK

In many practical applications the input or output spaces contain sequences. For example, sentences are often modeled as a sequence of words, where each word is encoded as a one-hot vector (i.e. a vector of all zeros except for a single 1 at the index of the word in a fixed vocabulary). A recurrent neural network (RNN) is a connectivity pattern that processes a sequence of vectors $\{x_1, ..., x_T\}$ using a recurrence formula of the form $h_t = f_\theta(h_{t-1}, x_t)$, where f is a function that we describe in more detail below and the same parameters θ are used at every time step, allowing us to process sequences with an arbitrary number of vectors. The hidden vector h_t can be interpreted as a running summary of all vectors x until that time step and the recurrence formula updates the summary based on the next vector. It is common to either use $h_0 = 0$, or to treat h_0 as parameters and learn the starting hidden state. The precise mathematical form of the recurrence $(h_{t-1}, x_t) \rightarrow h_t$ varies from model to model as shown in Figure 3.4.

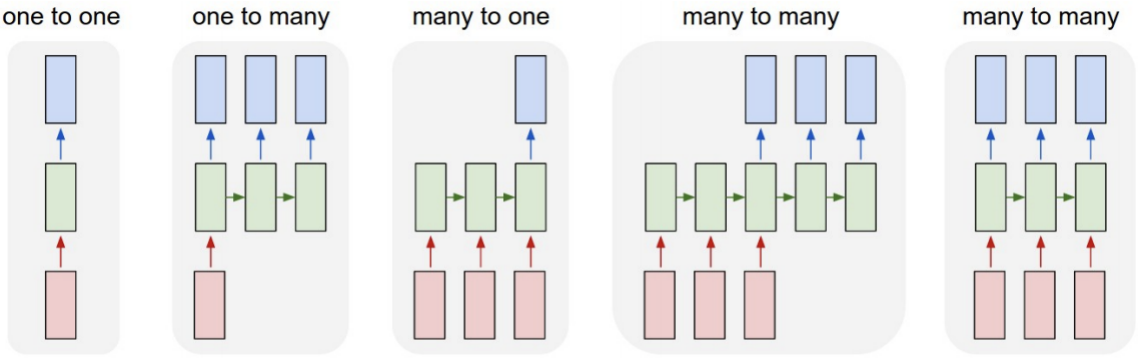


Figure 3.4. RNN architecture designs. An ordinary neural network (left) might take an input vector (red), transform it through some hidden layer (green), and produce an output vector (blue). In these diagrams boxes indicate vectors and arrows indicate functional dependencies. Recurrent neural networks allow us to process sequences of vectors, for example: 1) at the output, 2) at the input, or 3) both either serially or in parallel. This is facilitated by a recurrent hidden layer (green) that manipulates a set of internal variables ht based on previous hidden state h_{t-1} and the current input using a fixed recurrence formula $h_t = f_\theta(h_{t-1}, x_t)$, where θ are parameters we can learn.

That is, the previous hidden vector and the current input are concatenated and transformed linearly by the parameters W. Note that this is equivalent to instead writing $h_t = tanh(W_{xh}x_t + W_{hh}h_{t-1})$, where the two matrices W_{xh}, W_{hh} concatenated horizontally are equivalent to the matrix W above. These equations omit the additional bias vector for brevity. The tanh nonlinearity can also replaced with ReLU. If the input vectors x_t have dimension D and the hidden states dimension H then W is a matrix of size [H x $(D + H)$]. Interpreting the equation, the new hidden states at each time step are a linear function of elements of x_t, h_{t-1} and squashed by non-linearity. The vanilla RNN has a simple form, but unfortunately, the additive interactions are a weak form of coupling [32,33] between the inputs and the hidden states and the functional form of vanilla RNN leads to undesirable dynamics during backpropagation [6] (In particular, the gradients tend to either vanish or explode over long time periods). The exploding gradient concern can be alleviated with a heuristic of clipping the gradients at some maximum value [7], but the RNNs still suffer from the vanishing gradient problem.

3.6. LONG SHORT-TERM MEMORY

The LSTM [8] recurrence is designed to address the limitations of the vanilla RNN. Its recurrence formula has a form that allows the inputs x_t and h_{t-1} interact in a more computationally complex manner that includes multiplicative interactions, and the LSTM recurrence uses additive interactions over time steps that more effectively propagate gradients backwards in time [8]. In addition to a hidden state vector h_t, LSTMs also maintain a memory vector c_t. At each time step the LSTM can choose to read from, write to, or reset the cell using explicit gating mechanisms. The precise form of the update is as shown in Figure 3.5.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} \qquad \begin{aligned} c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned}$$

Figure 3.5. LSTM Equations.

Here, the sigmoid function sigm and tanh are applied element-wise, and if the input dimensionality is D and the hidden state has H units then the matrix W has dimensions [$4H$ x (D+H)]. The three vectors $i, f, o \in R^H$ are thought of as binary gates that control whether each memory cell is updated, whether it is reset to zero, and whether its local state is revealed in the hidden vector, respectively. The activations of these gates are based on the sigmoid function and hence allowed to range smoothly between zero and one to keep the model differentiable. The vector $g \in R^H$ ranges between -1 and 1 and is used to additively modify the memory contents. This additive interaction is a critical feature of the LSTMs design, because during backpropagation a sum operation merely equally distributes gradients. This

allows gradients on the memory cells c to flow backwards through time uninterrupted for long time periods, or at least until the flow is disrupted with the multiplicative interaction of an active forget gate.

Example: Character-level Language Modeling. Character-level language models are a commonly studied interpretable testbed for sequence learning. In this setting, the input to the RNN is a sequence of characters from some text (e.g. 'cat sat on a') and the network is trained to predict the next character in the sequence ('m' in this example, the first letter of 'mat'). Concretely, assuming a fixed vocabulary of K characters we encode all characters with K-dimensional one-hot vectors $\{x_t\}, t = 1, ..., T$, and feed these to the network to obtain a sequence of H-dimensional hidden vectors $\{h_t\}$. To obtain predictions for the next character in the sequence we further project the hidden states to a sequence of vectors $\{y_t\}$, where $y_t = W_y h_t$ and W_y is a [K x H] parameter matrix. These vectors are the logits of the next character (so the probabilities are obtained by passingthese through a softmax function) and the objective is to minimize the average cross-entropy loss over all targets.

3.7. TRAINING SUMMARY

Data preparation. First, obtain a dataset that is made up of a set of pairs (x, y) where x is some input example and y is a label. The data is then split into three folds, commonly a training, validation and test fold (common proportions could be 80%, 10%, 10% respectively). The training fold is used for optimizing the parameters with backpropagation, the validation fold for hyperparameter optimization, and the test fold for evaluation (discussed below in more detail). Data preprocessing. Preprocessing the data can help improve convergence of neural networks [5]. For images, common preprocessing techniques involve standardizing the data (subtracting the mean and dividing by the standard deviation individually for every input dimension of x), or at the very least subtracting the mean. It is

critical to estimate these statistics only on the training data, and using these fixed statistics to process the validation and test data, as this appropriately simulates the deployment of the final system into a real-world application.

Optimization. A default recommendation is to use Adam [4] for optimization, with learning rates of approximately 1e3, the coefficient of first moment of 0.9 and second moment 0.99. It is often beneficial to anneal the learning rate during the course of training by approximately a factor of 100 by the very end of training, but it is common to decay the first time only after half of the training is finished. During optimization it is almost always a good idea to use Polyak averaging [7], where we keep track of an averaged parameter vector θ_{avg} (e.g. $\theta_{avg} = 0.999\theta_{avg} + 0.001\theta$) after every parameter update and use θ_{avg} to compute the validation performance and when saving the model checkpoint to file.

4. SYSTEM ARCHITECTING APPROACH FOR DEEP LEARNING

This section introduces the framework required to simplify the steps necessary to perform the neural architecture search. It also presents a case study showing the use of framework to effectively explore large design space by automating certain model construction, alternative generation, and assessment. The framework exploits the system architecting principles for exploring different types of deep learning architectures related to computer vision and natural language processing. It provides the architect necessary capabilities to modify the search space, participating layers and objectives based on the requirements.

The application of framework to find an optical architecture for CIFAR-10 dataset show the evolution of neural architectures with minimized human participation. Despite significant computational requirements, it is now possible to design architectures as good as a state-of-the-art models simply by defining the search space, layers and objectives. Experimental results show that evolving architectures on a well defined search space can produce multiple architectures that are on par with state-of-the-art models while removing the manual hours spent on designing the architecture. These results also show that there is a promise to further the improve the overall architecture search by improving the evolutionary process.

4.1. OUTLINE

With the success of recent neural architecture search (NAS) approaches in deep learning, it is evident that the evolved designs are capable of outperforming the human made designs. So far, all the NAS approaches use almost identical search spaces and optimized on objective/s to produce an optimal architecture. In general, the neural architecture search (NAS) consists of three phases as shown in Figure 4.1. (1) The search space (S) encapsulates the logical representation of different architectures that are possible using the

defined operations. The range of values used for representing different architectures depends on the architect. The current NAS approaches either use binary format or integer format to define the architectures. (2) The search strategy defines the optimization technique and responsible for building the graphs for different architecture based on the search space (S). The most popular search strategies include genetic algorithms, reinforcement learning and differential approach. (3) The performance evaluation performs training & validation. It also calculates the overall objective value which is used by the search strategy to generate better architectures. This step is the most time consuming part of the entire search process as each architecture is trained and validated for a set of epochs. Once the architectures are evaluated, the validation accuracy is combined with other objectives to get an overall objective score. This process is repeated until a stopping criteria is reached.

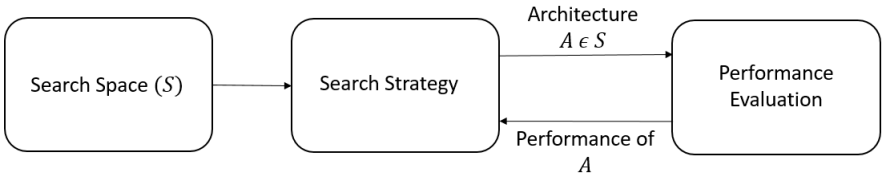


Figure 4.1. Illustration of neural architecture search method. The search strategy generates an architecture from search space S. The performance is calculated using the performance evaluation method and the value is returned to the search strategy.

While this approach is successful on standard datasets such as CIFAR-10/100 and IMAGENET, the real-world application of these approaches usually face far more constraints in the form of computational complexity, resources and time. Therefore, it is essential to have the capability to change the parameters of NAS depending on the requirements of the architect. This implies that for successful application of NAS on new datasets, there should be (1) Fleible architecture search space: Based on the dataset and available computation, the architect should have the freedom to specify the search space rather than to use a space that is constrained to a block. The framework presented in this section allows the architect

to explicitly design the search space based on problem requirements. (2) Multi-objective capability: Models that are designed for deployment on real-world devices are generally less expensive in terms of computation. This is achieved by having a trade-off between the accuracy and compute. To facilitate this process, the framework incorporates multi-objective optimization such that the designed models follow the trade-off constraints. (3) Flexible layer combinations: Based on type of deep learning model and dataset features, the choice of layers may vary in terms of filter size, number of filters and type of layers. The framework provides this capability to the architect by inputting the layer configurations before the start of the search process.

4.2. RELATED WORK

System architecting approaches, applied in design, analysis and optimization have flourished in various domain specific disciplines [4, 14, 21]. Our approach aims to use these capabilities in the field of deep learning to reduce the human effort required to design the architectures by considering each layer as a system and finding optimal connectivity between input and output.

The idea of using evolution to learn neural networks dates back to 2001[11]. Attempts were made to design or optimize neural networks using evolutionary algorithms, reinforcement learning and other machine learning approaches[23, 29]. In recent years, these approaches resurfaced as the deep learning architectures became deeper thereby increasing the search space significantly. Initial approaches include the design architectures that can overcome the man-made models[2, 26, 27]. The prominent feature of these attempts is to design complete architectures and optimize them based on the validation accuracies. Since it included the design of complete architecture, they are computationally expensive and also they did not have much success when compared to the state-of-the-art models except for [2, 24].

In order to reduce the computational complexity and be able to search for optimized architectures in the design space, it is more advantageous to build a novel search space that has the capability to do both. In [35], the authors showed that by using a small and efficient search spaces, we can build highly efficient architectures. Our approach takes inspiration from [35] and [24] to develop a search space that is capable of designing very deep architectures. By using a system architecting approach, we can have the desired search space for multiple objectives [1, 5] which is more generic.

4.3. FRAMEWORK

The systems architecting approach to build a framework for NAS is implemented in four phases as shown in Figure 4.2. The configuration of each phase will define the overall NAS problem that needs to be solved.

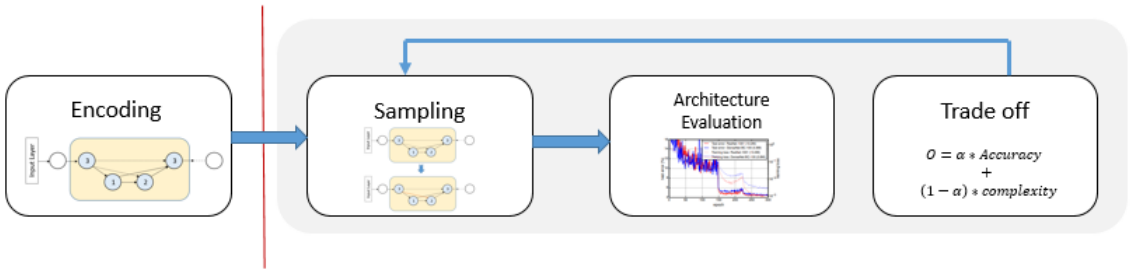


Figure 4.2. Outline of NAS framework. The encoding defines the search space for the architectures. The mutation generates new architectures based on search space and objective values. The multi-objective function calculates overall objective value based on different objectives provided by the architect. Lastly, the trade-off defines helps the architect to choose the most optimal solution.

4.3.1. Encoding. Since deep learning models can be viewed as stack of computational blocks that define layer wise computation, each layer can be encoded using a unique representation. Similarly, the inputs to each layer can just be the output of previous layer

[3] or outputs of multiple layers as in [4]. Therefore, the entire encoding scheme of the architecture is a combination of both operation encoding (A_{ops}) for layer wise operations and interface encoding (A_{if}) for connectivity between the layers.

$$A = A_{ops} + A_{if} \tag{4.1}$$

4.3.1.1. Operation encoding. The operation encoding is represented in a sequential format as whose length is equal to the number of layers on which the search is applied. If O represent the set of chosen operations, the encoding is given by

$$A_{ops} = [o_i^{(1)}, o_i^{(2)}, o_i^{(3)}, ..., o_i^{(L)}] \tag{4.2}$$

where $i \in O$ and L is the number of layers.

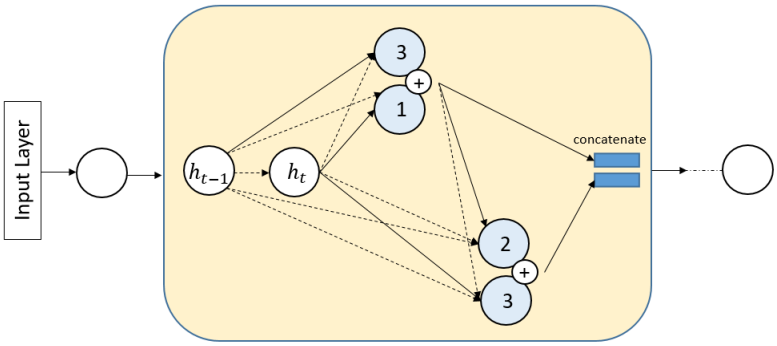


Figure 4.3. Encoding for wide architecture. It is the same formulation used in state-of-the-art NAS approaches.

4.3.1.2. Interface encoding. The interface encoding (A_{if}) defines the connectivity among the layers. So far, state-of-the-art deep learning models are based on two types: (1) wide models such as inception & googleNet and (2) deep models such as ResNet and DenseNet. Combining both of these designs explodes the search space and it becomes

difficult to find optimal architecture using reasonable computation. To prevent such scenario, the framework uses either a wide design or deep design based on the architect's requirement. Since both these designs have significant differences in connectivity, the framework provides unique encoding approach for both of them as descibed below:

Wide Design: The wide design is used in almost all of the state of the art NAS approaches [12]. Each operation chooses only one input from the set of available options to generate a complete architecture as shown in Figure 4.3. In general, only two previous layers h_t and h_{t-1} are used to design the overall NAS layer. Therefore, the choice of inputs can be represented by unique integers for each layer. If T represents the set of previous layers, the general formulation for wide design interface encoding is given by

$$A_{if} = [c_j^{(1)}, c_j^{(2)}, c_j^{(3)}, ..., c_j^{(L)}] \tag{4.3}$$

where $j \in T$ and L is the number of layers.

Deep Design: The deep design capability in this framework is inspired by the popular ResNet[16] and DenseNet[17] architectures. Unlike wide design, deep architectures have a lot of interconnections and it is difficult to represent the architectures using integer format. Therefore, a one-hot encoded approach is more suitable as shown in Figure 4.4. In this encoding, the presence of connection is represented by 1 while 0 represents no connection. The formulation for deep design is given by

$$A_{if} = [(p^1), (p^1, p^2), ..., (p^1, p^2, ..., p^L)] \tag{4.4}$$

where $p \in 0/1$

4.3.2. Sampling. The main idea of the sampling is to (1) generate feasible architectures based on search strategy and encoding. (2) improve architecture design using past knowledge and prevent local minima.

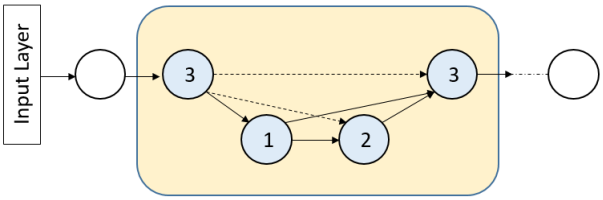


Figure 4.4. Encoding for deep architecture.

4.3.3. Architecture Evaluation and Trade-off. Each new architecture generated using the sampling process is trained and evaluated on validation accuracy by default. The optimal architecture is therefore the one which has highest validation accuracy. If additional objectives are added, the architect is responsible for identifying the trade-off between the validation accuracy and other objectives. Section 6 discusses this concept in detail.

4.4. CASE STUDY

In this section, the proposed NAS framework is used to design a dense architecture for an image classification problem to demonstrate the effectiveness of framework for finding optimal architecture. For the sake of quantitatively comparing the performance with the state-of-the-art models, only validation accuracy is used as the overall objective.

4.4.1. Dataset. The CIFAR-10 dataset is a collection of images each consisting of one of the 10 different classes - airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. It consists of 60,000 32x32 images that are uniformly distributed among all classes (6000 images for each class) and is commonly used to develop deep-learning object detection algorithms. Because of the low resolution images (32x32), the CIFAR-10 dataset allows the architects to test different types of algorithms and compare their performance with minimal computation. Figure 4.5 shows few examples of CIFAR-10 dataset.

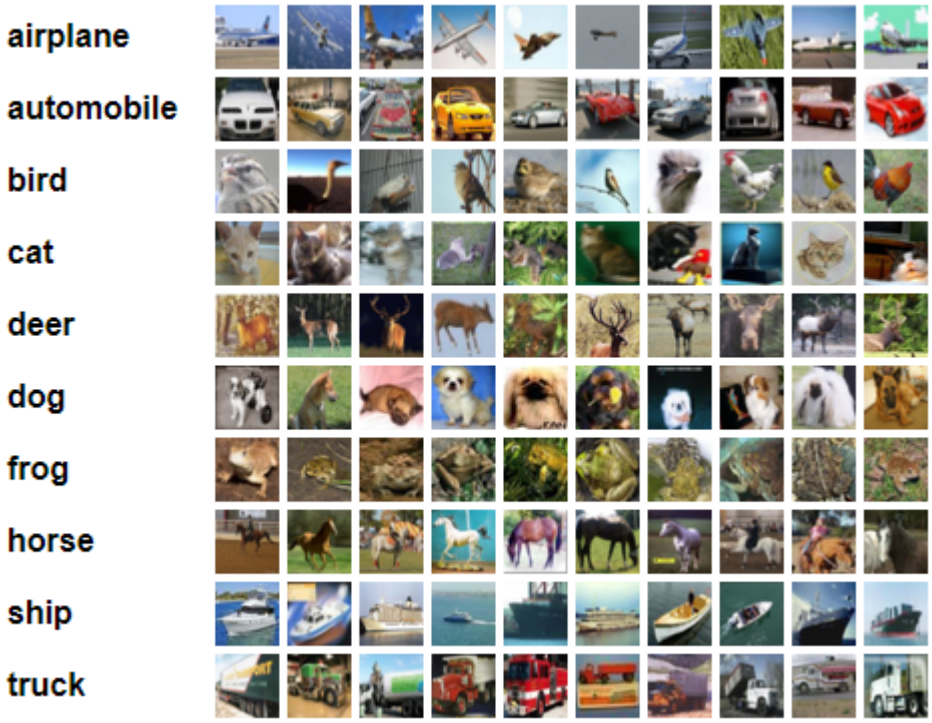


Figure 4.5. Samples from CIFAR-10 dataset.

4.4.2. Modelling the Architecture Using Framework. The image classification models generally consist of different types of layers that provide different capabilities to the overall architecture. Table 4.1 shows the capabilities provided by different layers that are used in CNN. The goal is to use the capabilities of the layers mentioned in Table 4.1 and design a hybrid architecture that can give optimal performance i.e. maximum validation accuracy.

4.4.2.1. Encoding. As shown in Figure 4.2, the first step in using the framework is choose an encoding format for the architecture search. Since the current objective is to design a deep and dense model, it uses the encoding format defined in Equation 4.1 where A_{if} uses a deep design.

Table 4.1. Table showing layers and their capabilities

System/Layer	Linear transformation	Non-linear transformation	Regularization	Parameter scaling
convolutional layer	x	-	-	-
pooling layer	x	-	-	-
activation layer	-	x	-	-
dropout	-	-	x	-
batch-normalization	-	-	x	x

In addition, as described in Section 2.2.2, using architecture search on a pre-determined template is more computationally feasible than on complete architecture. Therefore, the current architecture search is performed on the template as shown in Figure 4.6. The search process is applied only on a small part of the architecture called e-block. The e-blocks are separated by a transition layers which offer the necessary hierarchy necessary for the overall architecture.

Equation 4.2 shows that encoding requires a set of pre-defined operations which can be used in the architecture search. Based on the literature developed so far, each convolutional layer in a CNN design is often accompanied by a Relu activation - dropout with a probability of 0.8/1 and batch normalization. Therefore, the possibility of generation different operations for the search space lie in the convolutional layer.

If a convolutional layer is represented by conv_w_h_n where w is filter width, h is filter height and n is number of filters. For this case study, the w, $h \in [1, 3, 5, 7]$ while $n \in [12, 24, 33, 34]$. This generates a total of 62 unique operations O such that $o_i \in$ {1-62} in Equation 4.2.

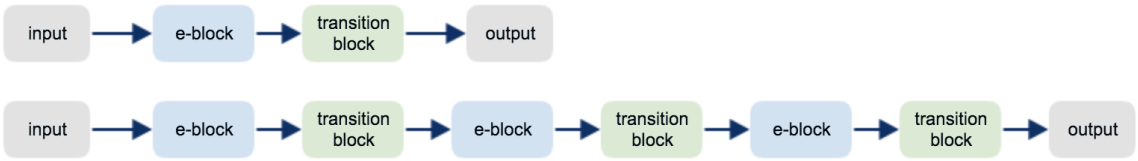


Figure 4.6. Pre-Detemined architecture designs. The top model was used for evolution and the bottom model was used to train complete architecute.

4.4.2.2. Genetic algorithm for search strategy. The second step in the framework is to design a search strategy for performing the architecture search. Some of the popular search strategies include genetic algorithms, reinforcement learning and differential approaches. This dissertation uses genetic algorithms as they are relatively simple to use and require less hyper-parameters as compared to other techniques.

Genetic algorithms are a type of evolutionary algorithms that are inspired by the theory of natural evolution. It reflects the process of natural selection by choosing only the fittest individuals for reproduction in order to give offspring for next generation. The individuals are represented by a genotype and the algorithm produces the offspring either by mutation or crossover. From the frameworks perspective, each sampled encoding represents a genotype which can be identified as a unique deep learning model and the evolution implies the generation of new offspring models from the best performing models.

Mutation: The mutation process of an architecture (A), involves changing only a single random bit q_i. This is done to preserve the good properties of a survived architecture while providing an opportunity of trying out new possibilities. Since the gene representing the architecture has both binary and integer values, the change is performed based on location of the bit. If the location of the bit represents a binary format, the the bit is flipped. For the case of integer format, a new value is chosen from the available range of values. Figures 4.7,4.8, 4.9 and 4.10 show examples of different possible mutation based on encoding formats.

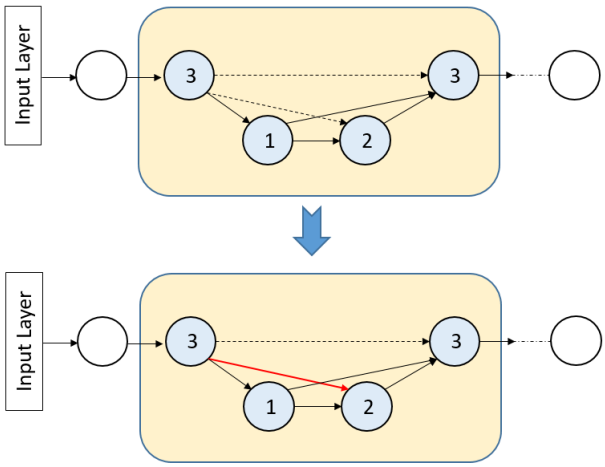


Figure 4.7. Interface mutation on a deep design.

4.4.3. Experiments and Results. In this section, we describe the experiments conducted to learn e-blocks using the method mentioned above. We used CIFAR-10 dataset to learn multiple top performing e-blocks. All our experiments were performed using an NVIDIA TITAN X GPU.

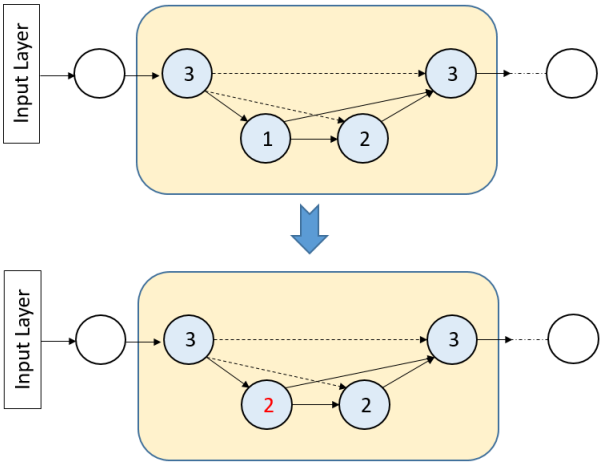


Figure 4.8. Operation mutation on a deep design.

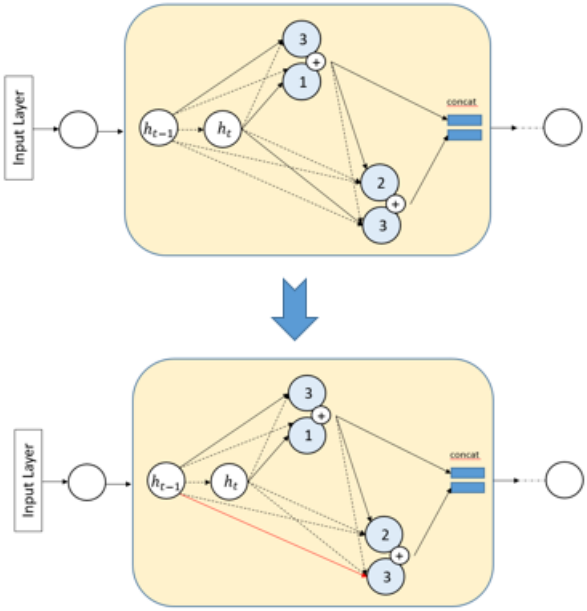


Figure 4.9. Interface mutation on a wide design.

The search process took over 2 weeks using a single GPU and covered 500 architectures. We ran our approach for 50 generation with a population size of 10. The crossover and mutation operations are performed in such a way that repeated architectures are not generated.

Pre-Processing on CIFAR-10: The CIFAR datasets consist of colored natural images with 32×32 pixels. CIFAR-10 (C10) consists of images drawn from 10 classes. The training and test sets contain 50,000 and 10,000 images respectively, and we hold out 5,000 training images as a validation set. We adopt a standard data augmentation scheme (mirroring/shifting) that is widely used for this datasets [12,17,18,32]. For pre-processing, we normalize the data using the channel means and standard deviations. For the final run we use all 50,000 training images and report the final test error at the end of training.

Training conditions: Throughout the evolution process, we train each architectures using stochastic gradient descent (SGD) with a learning rate of 0.1 and batch size of 64. We also observed that running each architecture of 25 epochs gave optimal results. Data augmentation was applied on the dataset so that each architecture will learn on noisy data.

Evolution: After defining the configuration of framework, it makes use of evolutionary search to find an optimized architecture for the given dataset. The approach starts with a random population whose fitness (objective) values are calculated. The top-n accuracies are then used to generate offspring using mutation operations. These steps are repeated for n iterations until the stopping criteria is achieved.

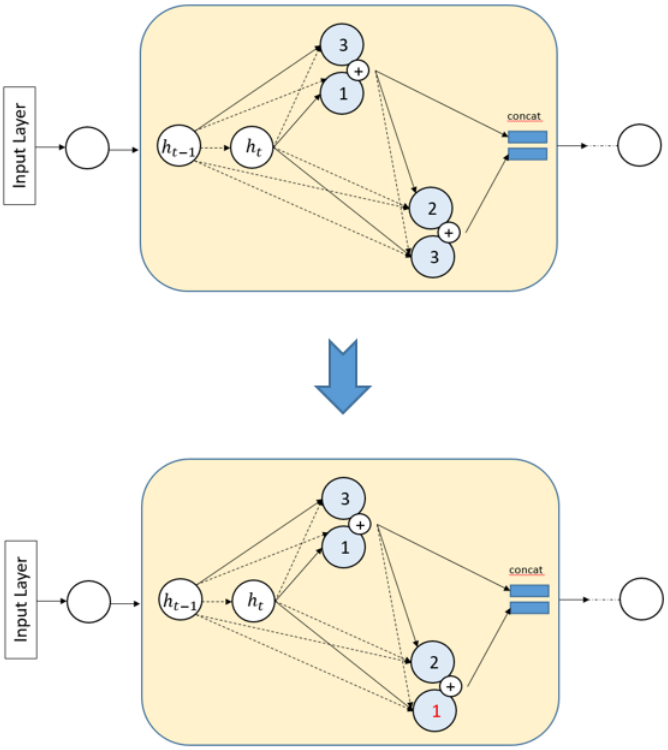


Figure 4.10. Operation mutation on a wide design.

Table 4.2. Results showing the performance of framework on CIFAR-10 with respect to state of the art models

Method	Depth	Params	CIFAR-10
Network in Network[11]	-	-	8.81
All-CNN[12]	-	-	7.25
Deeply Supervised Net[18]	-	-	7.97
FractalNet[19]	21M	38.6	5.22
ResNet[8]	110	1.7M	6.61
Wide ResNet[32]	16	11M	4.81
ResNet (pre-activation)[9]	164	1.7M	5.46
DenseNet[12]	100	0.8M	4.51
e-block@1	100	1.2M	5.23
e-block@2	100	2.8M	4.61

To be able to evolve the e-block using the genetic algorithm, we encoded the complete e-block into a binary chromosome. The chromosome for an e-block has the structure shown in Figure 4.4. If c is the number of choices and d is the depth of the e-block, then the total length of the chromosome is given as $n*c+(d*(d+1)/2)$ where n is the number of options in each choice. The first half of the equation defines the number of bits required to choose the layer configuration details while second half of the equation defines the inputs. The inputs to each layer can be any number of inputs from previous layers concatenated along the dimensional axis. Therefore, all the skip connections will be covered in this layer.

Figure 4.11 shows the top-performing e-block obtained from our experiments. Note that the final generation yields multiple top performing architectures. We observed that all these architectures can be tuned to maximize their accuracies. We consider the top two e-blocks and use them for comparison. We call these blocks e-block@1 and e-block@2 whose accuracies and parameters (in millions) are on par with the top human designed architectures.

4.4.4. Results on CIFAR-10 Dataset. For the task of image classification on CIFAR datasets, we used three e-blocks separated by a transition layer (see Figure 4.6). The test accuracies of the best architectures are reported in Table 4.2 along with other state-of-the-art models. As can be seen from the Table, architecture using e-block@1 and e-block@2 with data augmentation achieves a performance that is similar to state-of-the-art models.

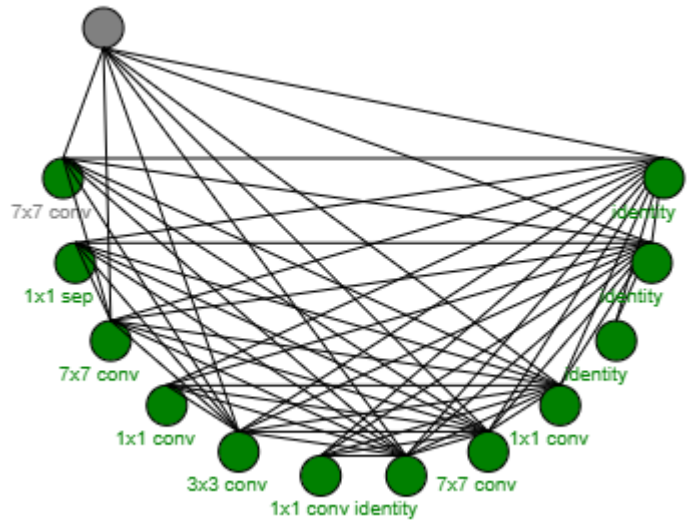


Figure 4.11. Model of best architecture found on CIFAR-10 dataset with an error rate of 4.61%.

4.4.5. Additional Experimentation on CIFAR-100. In addition to CIFAR-10, the framework is also used to design a architecture on CIFAR-100 dataset which has 100 classes instead of 10. The experimental setup is shown below

Table 4.3 shows that the framework was able to achieve competitive accuracy using only 1 GPU.

Table 4.3. Results showing the performance of framework on CIFAR-100 with respect to state of the art models

Method	Params (M)	C100
ResNet (reported by [3])	1.7	27.22
Wide ResNet	11	22.07
With Dropout	1.7	20.50
ResNet (pre-activation)	36.5	24.33
DenseNet	0.8	22.27
e-block@C100	3.7	22.64

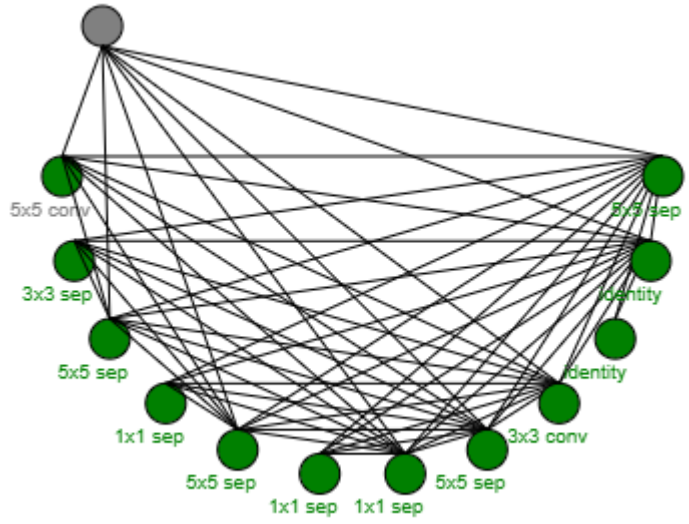


Figure 4.12. Model of best architecture found on CIFAR-100 dataset with an error rate of 22.64%.

4.5. DISCUSSION

This section defines the handling of certain constraints that are associated with the NAS process.

4.5.1. Feasible Architectures. During the search process (evolution), there is a possibility to end up with architectures that are not feasable. For example, dimension of output became less than 1 due to too many pooling layers. Such architectures are allocated a fitness value of -1 by default to prevent the generation of such architectures.

4.5.2. Limitations. Even though our approach is capable of producing deep and efficient architectures, the size of each e-block is pre-determined. In our experiments we used the size of 12, however, if we need to experiment with different size e-block after each transition layer, the entire search process has to be repeated.

4.5.3. Discussion. In this section, we showed the possibility of using system architecting methodologies for designing deep learning models. The approach allows the user to define his own objectives and find the best suitable deep learning model while optimizing all the objectives. Note that we did not use the latest approaches for the architecture search as our main intention is not to improve the accuracy but to show the possibility of using system architecting for deep learning.

5. EFFICIENT ARCHITECTURE SEARCH FOR DEEP NEURAL NETWORKS

This section addresses the scalability challenge of neural architecture search by utilizing a parameter sharing approach with regularized genetic algorithm (RGE). The key idea is to use a regularized genetic algorithm (RGE) on a pre-determined template and discover a high-performance architecture by searching for the optimal gene. During evolution, each model corresponding to a discovered chromosome is trained for a fixed number of epochs to minimize a canonical cross-entropy loss on a given training dataset. Meanwhile, the performance of the trained model on validation dataset is used as a fitness value to perform the evolutions. Because of parameter sharing the trained weights in each generation are carried to the next, thereby reducing the GPU hours required for maximizing the validation accuracy. On the CIFAR-10 dataset, the approach finds a novel architecture that outperforms the best human-invented deep architecture (DenseNet). The CIFAR-10 model achieved a test error of 4.22% with only 0.96M parameters which is better than DenseNet of 4.51% with 0.8M parameters. On CIFAR-100 dataset, the approach was able to compose a novel architecture that achieved 20.53% test error with 3.7M parameters which is on par with 20.50% test error of wide ResNet with 36.5M parameters.

5.1. OUTLINE

Automatic neural architecture design has shown the capability to discover high-performance neural networks, which are significantly better than human-designed models. Existing methods, irrespective of whether they use reinforcement learning (RL) [8,10], evolutionary algorithms (EA) [9] or gradient descent (GD) [11,12] rely on Neural Architecture Search (NAS) space, first published in [8]. The NAS space took its inspiration from inception [25], a wide-architecture that has better performance than previous simple linear architectures. [26]. In later stages, very deep architectures with skip connections such as

ResNet [1-4] and DenseNet [5] achieved even higher performance that inception [25]. However, in the case of automatic architecture design, the inception style (wide-architecture) is preferred because of its transfer capability: use cells designed on a small dataset on a larger dataset as the search process on large datasets is computationally expensive. For example, initial approaches which did not use parameter sharing required as many as 800 GPUs and took approximately 24 days to find a single architecture using NAS space on the CIFAR-10 dataset. Therefore, implementing a similar approach upon a larger dataset is not feasible but by using the cells designed on CIFAR-10 inside deep architectures gave very high accuracies. [8-12]. With the use of parameter sharing in [10], the computation time on CIFAR-10 was reduced to approximately 18 hours, and the overall process used only 1 GPU to find an optimal architecture. Thus, the parameter sharing technique opened the possibility to work on architecture search for deep designs without consuming a lot of GPU hours.

The main contribution of this work is to use the search space introduced in [27] for deep architecture design and compose a novel architecture using parameter sharing, which is significantly better than the human-designed deep architectures. There are three main components to achieve this: (1) Define the encoding scheme for the search space such that its combinations correspond to different possible architectures. (2) Train the generated architectures with training data and test the performance with validation data. The fitness value for the RGE will be the accuracy of the trained model on validation data. (3) For each new architecture generated during evolution, assign its parameters with most recent trained values. If a particular parameter configuration is new, then perform initialization. Even though the architecture design in this paper is implemented using a genetic algorithm, it can also be implemented using reinforcement learning or gradient descent. This section does not include the performance comparison between different optimization techniques as its main emphasis is to show an efficient way to conduct deep architecture search.

The experimental results show that the top performing architectures found on CIFAR-10 and CIFAR-100 datasets all have a performance that is significantly higher than best human-designed deep architectures with minimal parameters. The best design on CIFAR-10 gave a test error of 4.22% with 0.96M parameters which is better than DenseNet with 4.51% test error and 0.8M parameters. Additionally, compared to the approach used in [27], which requires approximately 60 days to compose a deep architecture, the technique presented in this paper required only 2 days to design a new architecture from scratch which is a significant improvement.

5.2. RELATED WORK

The exploration of neural network architectures has played a vital role since their initial discovery [20]. With the rise in popularity of neural networks, several techniques were proposed to improve the performance of the models while keeping the parameters to a minimum [1-5, 13-18]. Recently the design of architectures has shifted from purely human-designs to a combination of human-knowledge and automatic approach called neural architecture search (NAS) [6-12, 19-24].

Optimization technique: The successful optimization techniques that composed state-of-the-art neural network designs relied on (1) reinforcement learning (2) genetic algorithm and (3) gradient descent. Even though there is no concrete evidence which can prove that these techniques can guarantee convergence, their application on NAS space was able to produce high-performing architectures [8-12]. The approach presented in this paper uses a variant of genetic algorithm called regularized genetic algorithm (RGE), first published in [9] but reinforcement learning or gradient descent approach should also work.

Search Space: Initial attempts of automatic neural network design were not able to generate architectures that are better than the human-made designs [21-22]. However, the use of heavy computation for architecture search pushed the performance of automatic architectures in recent years [6,7]. Even though the performance of the architecture improved

on standard datasets, it is not feasible to implement on new datasets because of the extensive computational requirements. In [8], the NAS space was introduced to address this issue, which used a pre-determined architecture design, and the search process got reduced to finding the optimal cell. The approach was not only able to generate better architectures than their predecessors but also used much fewer GPU hours. This experiment showed that innovative search spaces are capable of reducing the computation time to some extent when compared to the full architecture search.

Parameter sharing: Even though the NAS space reduced the computation time, it is still not feasible to apply to new datasets as it required approximately 800 GPUs to compose a new architecture. In [10], the architecture search is implemented using both parameter sharing and NAS space, which reduced the number of GPUs from 800 to 1, thereby making this approach feasible to any new dataset. Even though different architectures use parameters differently, the results from [6, 10-12] established that parameters learned for a particular model can be used for other models, with little to no modifications. Therefore, the approach in this section uses parameter sharing with its architectures to compose high-performing architectures.

5.3. APPROACH

This section describes the different techniques used in composing the high-performing architecture. Section 3.1 describes the search space in general form, and its encoding to generate the chromosomes. Section 3.2 then introduces the regularized genetic algorithm which maximizes the validation accuracy using the search space. Finally, Section 3.3 describes the parameter sharing process, which prevents the parent architectures from discarding their trained parameters by sharing them across the generation.

5.3.1. Parameter Sharing. The architectures generated during evolutions share the parameters for training rather than initializing them for each mutation. Namely, if a layer has a matching shape to one of previously trained architecture, the weights are inherited.

Therefore, the initial population will not have any inheritance, and the later generations will have it from their parents, except for the mutated layer. The mutated layer will inherit if its shape was used previously by other trained architectures. Eventually, all possible weight configurations will have an initialization, and the convergence depends on finding optimal chromosome using shared parameters. Figure 5.1 shows the parameter sharing between three simple architectures. The red lines indicate the transfer of parameters whenever same convolution configurations are used. For all the new configurations, new parameters are initialized.

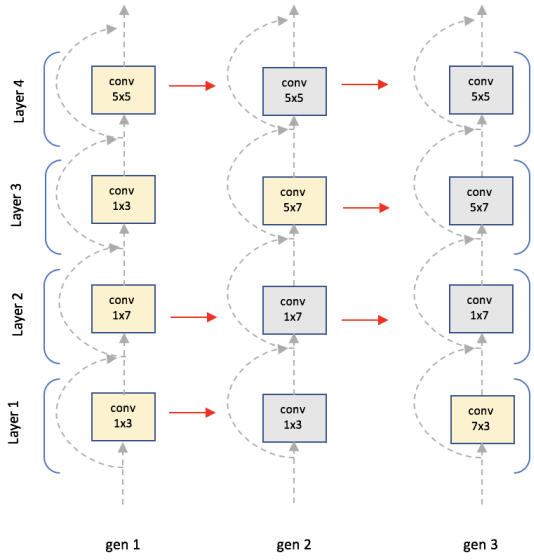


Figure 5.1. The representation of parameter sharing between three different architectures. The yellow nodes show the initialization of parameters. The first architecture is all yellow as there are no previous parameters. The second and third architectures have only 1 yellow nodes as the rest of the parameters are inherited from previous architectures.

The evolutionary process and parameter training are alternatively implemented to maintain a balance between them. The architectures train for few epochs (say 2) and then mutated. After a few hundreds of generations (say 500), the average validation accuracy/fitness will gradually maximize. The evolution will stop either when the maximum

accuracy of each generation no longer improves or when any architecture achieves the state-of-the-art accuracy. Since there is no possibility to discard any parameters, the overall search process will be able to complete using only one GPU.

5.3.2. Encoding. The central idea of this approach is the observation that all architectures that the algorithm ends up iterating over the generations are a unique combination of chromosomes. Figure 4.4 illustrates a generic chromosome structure. The length of the chromosome (n) determines the length of the block, and its values denote the operations at different layers.

The choice of operations is limited to a set of convolutions (best configurations based on previous literature) which are derived from NAS [8]. The list below shows different types of convolutions used in this paper. Since there are eight choices, the values of the chromosome will have a range between 1 and 8. An example of such representation is shown in Figure 5.2 where n=3.

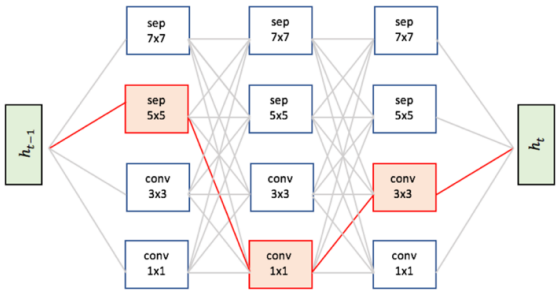


Figure 5.2. Chromosome example. A simple design chosen for a chromosome with n=3 and values [1,3,5,7]. The graph shows all the possible connections between the layers. The red lines indicate the chosen operations at each layer based on values in the chromosome.

New architectures are generated during each generation of the evolution process based on the chromosome and pre-determined template, as shown in Figure 5.3. As can be seen in Figure 5.3C, each layer inside the e-block is based on a set of operations apart from the chosen convolutions. The 1x1 convolution control the dimensionality of the filters. The

batch normalization (BN) operations prevent the explosion of gradients during training and, the ReLu operations provide the required non-linearity inside the architecture. Also, the transition layers which have the form BN -> Conv 1x1 ->ReLu -> pooling, before e-blocks provide the necessary hierarchy.

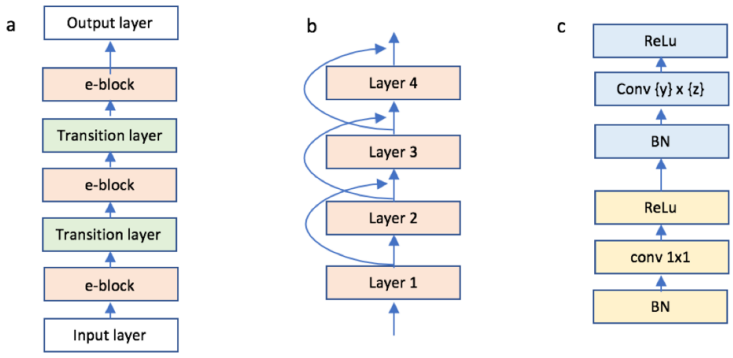


Figure 5.3. Overall template design. a) The template used for architecture search; b) each e-block in the template has n layers which are densely connected; c) Each layer has multiple fixed operations except for one which will be chosen based on chromosome value.

5.3.3. Regularized Genetic Algorithm. Algorithm 1 gives a summary of the evolutionary algorithm used in this paper. The process starts with random initialization of population (P), which represent random initial designs. The population size (P) is always maintained and all architectures that conform to the search space described, are possible and equally likely. At each generation, the algorithm samples S random models from the population (with replacement) and selects the model with highest validation accuracy as a parent. A new architecture, called the child, is constructed from the parent by the application of a transformation called a mutation (described below). The new child architecture gets trained, evaluated, and added to the population. This process is called tournament selection, and the oldest model in the sample (S) is discarded (or killed) to maintain the population size at P.

Algorithm 1 Aging Evolution

```
population ← empty queue                          ▷ The population.
history ← ∅                                       ▷ Will contain all models.
while |population| < P do                         ▷ Initialize population.
    model.arch ← RANDOMARCHITECTURE()
    model.accuracy ← TRAINANDEVAL(model.arch)
    add model to right of population
    add model to history
end while
while |history| < C do                            ▷ Evolve for C cycles.
    sample ← ∅                                    ▷ Parent candidates.
    while |sample| < S do
        candidate ← random element from population
                ▷ The element stays in the population.
        add candidate to sample
    end while
    parent ← highest-accuracy model in sample
    child.arch ← MUTATE(parent.arch)
    child.accuracy ← TRAINANDEVAL(child.arch)
    add child to right of population
    add child to history
    remove dead from left of population           ▷ Oldest.
    discard dead
end while
return highest-accuracy model in history
```

Figure 5.4. Steps of Regularized Genetic Algorithm (RGE).

Mutation: The mutation process changes a random value in the parent chromosome to generate a child architecture. These random modifications make the genetic algorithm traverse the search space to find an architecture that maximizes the fitness value/validation accuracy.

5.4. EXPERIMENTS AND RESULTS

This section complements the methods in Section 5.3 with the details necessary to reproduce the experiments. The first part describes the datasets and pre-processing used on the datasets. The latter part describes the parameters used for RGE and training the generated models. Finally, Table 5.1 shows a comparison with the previous best deep architectures along with results from the approach presented in this paper.

Datasets: The two CIFAR datasets consist of 60k colored natural images with 32×32 pixels. CIFAR-10 (C10) consists of images drawn from 10 and CIFAR-100 (C100) from 100 classes. The training and validation sets contain 50k and 5k images respectively, and the remaining 5k images are held as a test set. The experiments adopt a standard data augmentation scheme that is widely used for these two datasets [13-18] and denoted by a "+" mark at the end of the dataset name (e.g., C10+). For preprocessing, the images were normalized using the channel means and standard deviations. During evolution, the training and validation are implemented using corresponding datasets. The final results were reported on the test images using the best architecture.

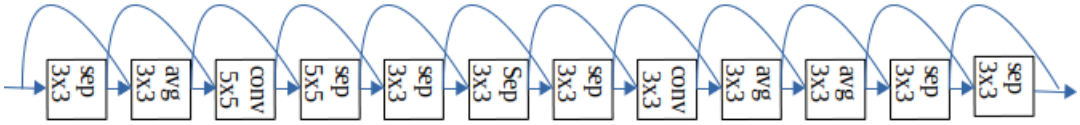


Figure 5.5. e-block of best architecture

Training details: The population for the RGE algorithm is set at P=100 and the sample size S is set at 10. The weights of all the architectures are initialized uniformly between -0.08 and 0.08. Once the RGE samples a population, a child model is constructed using mutation and trained for 2 epochs. The validation accuracy/fitness from trained model is used for selecting parents. The settings for training the CIFAR-10 and CIFAR-100 child models are the same as those used in [5]: momentum Optimizer with a learning rate of 0.1, weight decay of 1e-4, momentum of 0.9 and used Nesterov Momentum [28].

Results on CIFAR: For the task of image classification, the length of e-block is set to N = 12. The test accuracies of the best architectures are reported in Table 5.1 along with other state-of-the-art deep architecture models. As can be seen from the Table, a deep model using e-block with data augmentation achieves an error rate of 4.22% on CIFAR-10, which

Table 5.1. Error rates % on CIFAR datasets

Method	Params (M)	C10	C10+	C100	C100+
ResNet	1.7	-	6.61	-	-
ResNet (reported by [3])	1.7	13.63	6.41	44.74	27.22
ResNet with Stochastic Depth	1.7	-	4.91	-	-
Wide ResNet	11	-	4.81	-	22.07
With Dropout	1.7	-	4.17	-	20.50
ResNet (pre-activation)	36.5	11.26	5.46	35.58	24.33
DenseNet	0.8	5.92	4.51	27.15	22.27
e-block@C10	0.96	4.69	4.22	-	-
e-block@2C100	3.7	-	-	21.13	20.53

is slightly better than the previous deep architecture of 4.51%. Similarly, on CIFAR-100, it achieves an accuracy of 20.53% which is on par with the previous best deep model of 20.50%.

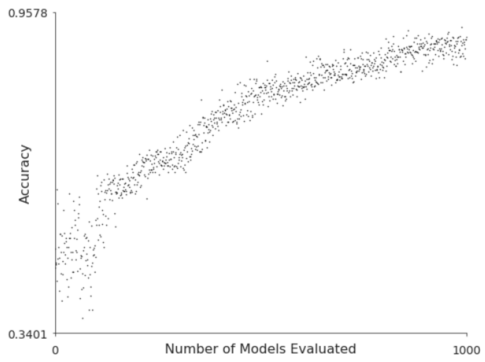


Figure 5.6. Plot showing the evolution of architectures.

5.5. DISCUSSION

This section shows an efficient approach to search for deep architectures. It is the first attempt where a genetic algorithm and parameter sharing were used together for composing architectures. The results show that this approach is indeed capable of discovering architectures that are better than human made designs while using much less parameters than their predecessors. However, the results were not trained to beat the state-of-the-art results as its main focus is to show that deep search spaces are indeed capable of generating high performance architectures. Therefore, the future work will be targeted to generate architectures that are better than the current best architectures.

6. MULTI-OBJECTIVE NEURAL ARCHITECTURE SEARCH VIA NSAGA-II

While the previous two sections introduced an efficient way to perform neural architecture search (NAS), they ignored device related objectives such as inference time, memory usage and power consumption. Finding optimal architectures based on device related objectives is extremely important when the computing resources are limited. This section introduces a device aware Pareto-optimal NAS that is capable of optimizing for both device-related and device-agnostic objectives. It uses the wide search space with parameter sharing that is defined in Section 4 on multiple objectives rather than a single objective. Experimental results on CIFAR-10 demonstrate the effectiveness of Pareto-optimal architectures by comparing the results on two different GPUs: (1) workstation with NVIDIA Titan X GPU. (2) workstation with NVIDIA K80 X GPU. Compared to the results in Sections 3 and 4, this approach provides multiple solutions which provide trade-offs between multiple objectives.

6.1. OUTLINE

Deep Learning architectures designed using optimization techniques have demonstrated impressive performance in many machine learning tasks like image recognition and language modeling. While different techniques such as reinforcement learning (RL) and genetic algorithms (GA) were be used for optimization, they all use only one objective (e.g. accuracy) to search for optimal architecture while ignoring other important objectives (e.g inference time).

While there is some previous work where multiple objectives were used to design the optimal architectures, the search space is considerably small as the choices are limited to previously designed blocks (e.g. denseblock, inception and mobilenet). In addition, the

search space is also computationally expensive as parameter sharing cannot be implemented. To this end, there is a significant gap between the NAS approaches used for single-objective and multiple objectives.

To address the above mentioned drawbacks, this section introduces a multi-objective NAS that can perform the architecture search with both computational efficiency and multiple objectives. In this way, the architect can explore the trade-off space and select the most suitable architecture under the specific case. Figure 6.1 shows an example different pareto-optimal architectures designed on two different devices with two objectives. Even-though same dataset is used for search on both devices, the results are significantly different because of the device configuration. This shows that the approach is capable of designing various architectures at the Pareto-front for the corresponding device.

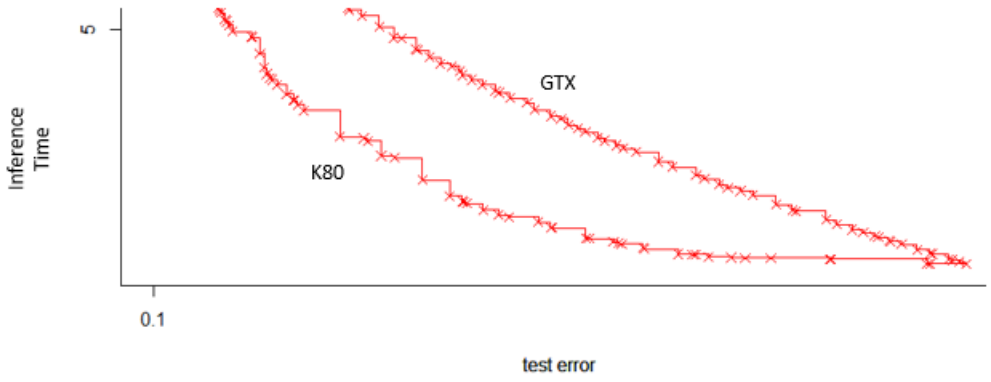


Figure 6.1. Comparison of inference times for NVIDIA GTX and K80. Since K80 is more powerful, the inference times are lower.

6.2. RELATED WORK

Multi-objective optimization: Multi-objective optimization deals with the problems that have multiple complementary objectives. If S represents the solution space for all the feasible architectures and f_1, f_2,..., f_n represent the N objectives then the goal of multi-objective optimization is to find a solution $s \in S$ which minimizes all the objectives. However, it is usually unlikely that only one s exists that minimizes all the objectives. Instead, multiple Pareto-optimal solutions exist in such a way that one cannot reduce an f_i without increasing f_j. Formally, a solution $s^{(1)}$ Pareto-dominates another solution $s^{(2)}$ if $\forall_i \in 1,2, ..., n : f_i(s^{(1)}) \leq f_i(s^{(2)})$ and $\forall_j \in 1,2, ..., n : f_j(s^{(1)}) \leq f_j(s^{(2)})$. The Pareto-optimal solutions S^* are exactly those solutions that are not dominated by any other $s \in S$. The set of Pareto-optimal solutions are called Pareto-front.

Multi-objective Neural Architecture Search: Not much of the literature exist in multi-objective NAS as most of NAS approaches focus on searching models that achieve highest performance (e.g.classification accuracy) regardless of the model complexity. [19] proposed to treat neural network architecture search as a multi-objective optimization task and adopt an evolutionary algorithm to search models with two objectives, run-time speed, and classification accuracy. However, the performances of the searched models are not comparable to handcrafted small CNNs, and the numbers of GPUs they required are enormous.

6.3. MULTI-OBJECTIVE GENETIC ALGORITHM (MOGA)

Being a population-based approach, GA are well suited to solve multi-objective optimization problems. A generic single-objective GA can be modified to find a set of multiple non-dominated solutions in a single run. The ability of GA to simultaneously search different regions of a solution space makes it possible to find a diverse set of solutions for difficult problems with non-convex, discontinuous, and multi-modal solutions

spaces. The mutation operator of GA may exploit structures of good solutions with respect to different objectives to create new non-dominated solutions in unexplored parts of the Pareto-front. In addition, most multi-objective GA do not require the user to prioritize, scale, or weigh objectives. Therefore, GA have been the most popular heuristic approach to multi-objective design and optimization problems. Jones etal.[4]reported that 90% of the approaches to multi-objective optimization aimed to approximate the true Pareto front for the underlying problem. A majority of these used a meta-heuristic technique, and 70% of all meta-heuristics approaches were based on evolutionary approaches.

NSGA-II: It is one of the most popular multi objective optimization algorithms with three special characteristics, fast non-dominated sorting approach, fast crowded distance estimation procedure and simple crowded comparison operator [2]. Deb et al. [7] simulated several test problems from previous study using NSGA-II optimization technique, and it is claimed that this technique outperformed PAES and SPEA in terms of finding a diverse set of solutions [3,5].

The objective of the NSGA algorithm is to improve the adaptive fit of a population of candidate solutions to a Pareto front constrained by a set of objective functions. The algorithm uses an evolutionary process with surrogates for evolutionary operators including selection, genetic crossover, and genetic mutation. The population is sorted into a hierarchy of sub-populations based on the ordering of Pareto dominance. Similarity between members of each sub-group is evaluated on the Pareto front, and the resulting groups and similarity measures are used to promote a diverse front of non-dominated solutions.

Figure 6.2 provides a pseudocode listing of the Non-dominated Sorting Genetic Algorithm II (NSGA-II) for minimizing a cost function. The Sort-By-Rank-And-Distance function (steps 15) orders the population into a hierarchy of non-dominated Pareto fronts. The Crowding-Distance-Assignment (step 12) calculates the average distance between members of each front on the front itself. Refer to NSGA-2 algorithms in [23] for a clear presentation of the Pseudo-code and explanation of these functions. The Crossover-and-Mutation

Procedure NSGA-II

Input: $N', g, f_k(X) \triangleright N'$ members evolved g generations to solve $f_k(X)$

1 Initialize Population $\mathbb{P}'$;
2 Generate random population - size N';
3 Evaluate Objectives Values;
4 Assign Rank (level) based on Pareto - *sort*;
5 Generate Child Population;
6 Binary Tournament Selection;
7 Recombination and Mutation;
8 **for** $i = 1$ *to* g **do**
9 **for** *each Parent and Child in Population* **do**
10 Assign Rank (level) based on Pareto - *sort*;
11 Generate sets of nondominated solutions;
12 Determine Crowding distance;
13 Loop (inside) by adding solutions to next generation starting from the *first* front until N' individuals;
14 **end**
15 Select points on the lower front with high crowding distance;
16 Create next generation;
17 Binary Tournament Selection;
18 Recombination and Mutation;
19 **end**

Figure 6.2. NSGA-2 Algorithm.

(steps 7 and 18) function performs the classical crossover and mutation genetic operators of the Genetic Algorithm. Both the Select-Parents-By-Rank-And-Distance (steps 9 -14) and Sort-By-Rank-And-Distance functions (step 15) discriminate members of the population first by rank (order of dominated precedence of the front to which the solution belongs) and then distance within the front (calculated by Crowding-Distance-Assignment).

6.4. SEARCH STRATEGY AND ENCODING

Even though NSGA-II deals with multiple objectives, the encoding strategy of the algorithm is independent of objectives. Therefore, both wide and deep designs described in Section 4 can be directly used for the search procedure. However, unlike regularized

genetic algorithm, NSGA-II generates multiple architectures in each generation. Therefore, parameter sharing should be performed for each gene in the population. Therefore, if N represents the population size, then there will be equivalent unique parameter sharing, to pass the trained parameters through the generations. Figure 6.3 shows an example of parameter sharing across two generations.

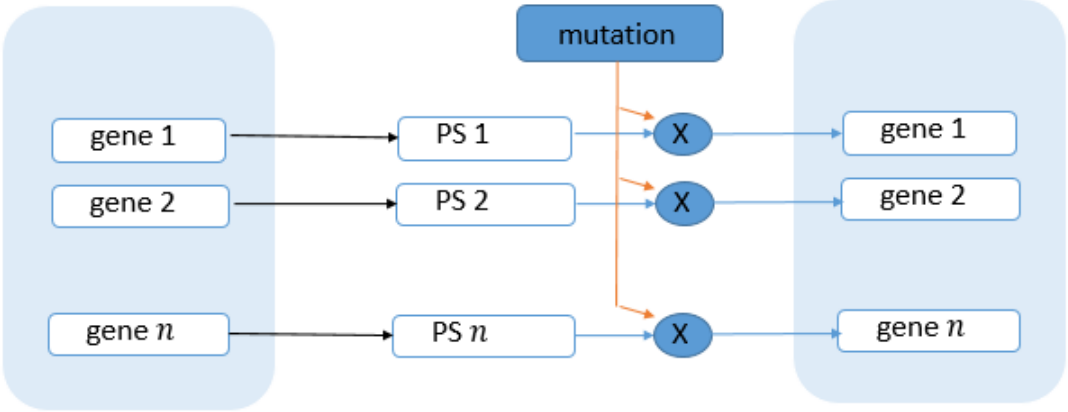


Figure 6.3. Parameter sharing for NSGA-II. Each gene in the population has its own shared parameters which will be forwarded to next generation.

6.5. EXPERIMENTS AND RESULTS

This section presents results for using NSGA-II on searching neural architectures for CIFAR-10 and CIFAR-100. The implementation is done using two different settings: (i) optimize 2 objectives and search for architectures using a wide design as defined in Section 5.1 and, (ii) we optimize 2 objectives and search for architectures based on deep design as defined in section. The progress of architecture search for both settings is visualized in Figure 6.4. The Pareto front improves over time, reducing the validation error while covering a wide regime of, e.g., model parameters, ranging from 10000 to 1000000

We train the full model end-to-end in a single step of optimization. We initialize the CNN with weights pretrained on ImageNet [84] and all other weights from a gaussian with standard deviation of 0.01. We use stochastic gradient descent with momentum 0.9 to train the weights of the convolutional network, and Adam [46] to train the other components of the model. We use a learning rate of 1e-6 and set momentum = 0.9, decay rate = 0.99. We begin fine-tuning the layers of the CNN after 1 epoch, and for efficiency we do not fine-tune the first four convolutional layers of the network. Our training batches consist of a single image that has been resized so that the longer side has 720 pixels. Our implementation uses tensorflow [12]. One mini-batch runs in approximately 300ms on a Titan X GPU and it takes about 1.5 days of training for the model to converge.

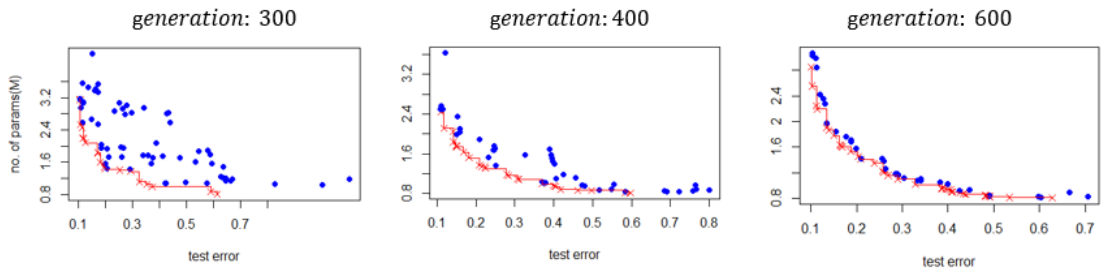


Figure 6.4. Progress of the Pareto front of NSGA-II during architecture search. The Pareto front gets more and more densely settled over the course of time.

The experiment aims at solving the following multi-objective problem: minimize the two objectives (i) performance on CIFAR-10 (expensive objective) and, (ii) number of parameters (cheap). We think having two objectives is a realistic scenario for most NAS applications. Note that one could easily use other, more sophisticated measures for resource efficiency. In this experiment, we implement wide design as defined in Section 6.4. It natively handles this unconstrained, arbitrarily large search space, whereas other methods are by design a-priori restricted to relatively small search spaces (Bender et al., 2018; Liu et al., 2018b). Also, NSGA-II is initialized with trivial architectures rather than

networks that already yield state-of-the-art performance. The set of operators to generate child networks we consider in our experiments are the three network morphism operators (insert convolution, insert skip connection, increase number of filters), as well as the three approximate network morphism operators (remove layer, prune filters, replace layer). The operators are sampled uniformly at random to generate children. The experiment ran for approximately 1.5 days on a single GPU. The resulting Pareto front consists of approximately 100 neural network architectures.

Comparison to Published Results on CIFAR-10: We compare against different-sized NASNets and other hand-crafted architectures. In order to ensure that differences in test error are actually caused by differences in the discovered architectures rather than different training conditions, we do not use stochastic regularization techniques, such as Shake-Shake or Scheduled-Drop-Path in this experiment as they are not applicable to all networks out of the box. The results are visualized in Table 6.1. When considering the performance on CIFAR-10 versus the number of parameters NSGA-Net is on par with NASNets and hand-crafted designs for resource-intensive models while it outperforms them in the area of very efficient models (e.g., less than 100,000 parameters). We highlight that this result has been achieved based on using only 1.5 GPU days for NSGA-Net compared to others which took approximately 80 days.

6.6. DISCUSSION

This section presents a multi-objective evolutionary algorithm for architecture search. The algorithm employs a NSGA-II mechanism along with parameter sharing to speed up the training of novel architectures. Moreover, it exploits the fact that evaluating several objectives, such as the performance of a neural network, is orders of magnitude more expensive than evaluating, e.g., a model's number of parameters. Experiments on CIFAR-10 show that NSGA is able to find competitive models and cells both in terms of accuracy and of resource efficiency. We believe that using more sophisticated concepts

Table 6.1. Comparison of NSGA-Net with other NAS methods on CIFAR-10 for different-sized models.

Method	Params (M)	Error(%)
ResNet	1.7	6.61
ResNet (reported by [3])	1.7	6.41
ResNet with Stochastic Depth	1.7	4.91
Wide ResNet	11	4.81
With Dropout	1.7	4.17
ResNet (pre-activation)	36.5	5.46
DenseNet-BC	0.8	4.51
NASNet@C10	3.3	2.65
ENAS@C10	4.6	2.89
PLNT@C10	5.7	2.49
DPP-Net@C10	11.4	4.36
NSGA-Net@C10	3.9	2.93

from the multi-objective evolutionary algorithms literature and using other network operators (e.g., crossovers and advanced compression methods) could further improve search process in the future.

7. CONCLUSION AND FUTURE WORK

7.1. CONCLUSION

Compared to the existing literature, the approaches presented in this dissertation have the following advantages: (1) framework to perform NAS using the user defined templates (2) modifiable search spaces so that the architect can perform searches based on the type of deep learning problem (3) multi-objective solutions that provide alternative solutions for real-world deployment.

This dissertation is expected to simplify the architecture development process on real-wold datasets thereby reducing the manual effort and computational resources. It provides a modeling tool to enhance the optimal use of computational resources by considering multiple objectives associated with real-world scenarios.

7.2. FUTURE WORK

The approaches defined in this dissertation are focused mainly in the field of computer vision classification problems. However, the deep learning has applications in many other domains such as natural language processing and medical imaging. This implies that the approaches presented in this dissertation can be experimented with other domains to improve the overall capabilities of the framework. Lastly, the goal of this research is to encourage the researches to further enhance the transition from tradition human-based deep learning designs to search based approaches.

APPENDIX

DENSENET FOR ANATOMICAL BRAIN SEGMENTATION

Automated segmentation in brain magnetic resonance image (MRI) plays an important role in the analysis of many diseases and conditions. In this section, we present a new architecture to perform MR image brain segmentation (MRI) into a number of classes based on type of tissue. Recent work has shown that convolutional neural networks (DenseNet) can be substantially more accurate with less number of parameters if each layer in the network is connected with every other layer in a feed forward fashion. We embrace this idea and generate new architecture that can assign each pixel/voxel in an MR image of the brain to its corresponding anatomical region. To benchmark our model, we used the dataset provided by the IBSR 2 (Internet Brain Segmentation Repository), which consists of 18 manually segmented MR images of the brain. To our knowledge, our approach is the first to use DenseNet to perform anatomical segmentation of the whole brain.

1. OUTLINE

Quantitative analysis of brain magnetic resonance image (MRI) is necessary for the diagnosis of many neurological diseases and conditions. For instance, abnormal shapes of certain anatomical regions of brain have been found to be associated with many disorders such as Alzheimer's and Parkinson's [4,5]. Segmentation i.e. labelling of each pixel (2D)/voxel (3D) plays a critical role in this analysis which led many researchers to design mathematical models that can accurately automate the process of segmentation. A popular approach for segmentation before the deep learning algorithms includes the use of atlases

[7,8] and pattern recognition methods [6]. In order to perform anatomically accurate segmentation, these methods use explicit spatial and intensity information which are manually created features.

The explicit definition of such features can be avoided with the use of convolutional neural networks (CNNs). CNNs have shown excellent results in many computer vision tasks such as ImageNet [9]. In recent years, convolutional neural networks (CNNs) have become a dominant machine learning approach for MRI segmentation because of their self-learning ability and generalization over large amounts of data [17]. CNNs were initially used for medical image analysis in [10-15], where they were able to provide accurate segmentations of various abnormalities without the use of hand crafted features. This motivated the researchers to develop many new CNN architectures for various segmentation tasks on MRI.

CNNs have also been used for brain MRI segmentation in [16], where the authors presented a CNN architecture to accurately segment the white matter, gray matter and cerebrospinal fluid in infants. In addition, a number of competitions were created by MICCAI in last few years where datasets were provided to segment different brain MRIs. In the recent MICCAI challenges of neonatal and adult [2], the best accuracy was not produced by single architecture. It was shown that different architectures are good at segmenting different aspects of the brain. In [2], the authors tried to develop a more general approach that is not biased towards one aspect of the brain. Considering the results from all these challenges, it was obvious that there are a lot of inaccuracies in the developed architectures and there is a lot of space for improvement.

In this paper, we present a new architecture for automated brain segmentation based on the concept presented in DenseNet [1]. DenseNet is the present state-of-the-art CNN architecture which gave the best results on CIFAR-10, CIFAR-100 and ImageNet datasets using only a fraction of parameters compared to its predecessors [1]. The idea behind DenseNet is to connect each layer to every other layer behind it to improve the information

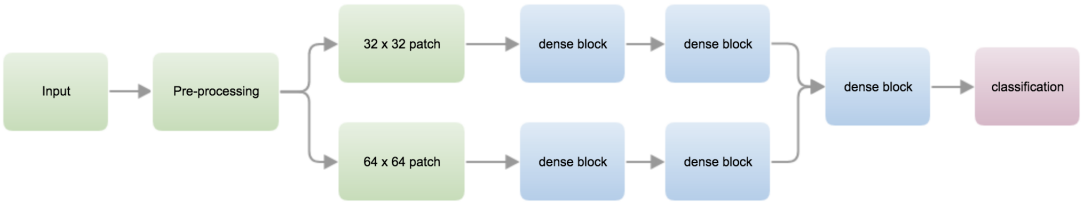


Figure 1. Architecture for brain MRI segmentation.

flow within the architecture. This gives CNN the capability to make decisions based on all of the layers rather than a single final layer. By using this approach for brain segmentation, we perform the segmentation which can further be used to identify abnormalities in brain.

2. APPROACH

Figure 1 shows an overall schematic of our proposed architecture. Considering the complexity of brain MRI, we use patches for inputs rather than sending the whole image as input to the architecture. The pre-processing steps include the normalization of the MRI images and generating concentric patches. The central pixel/voxel will be target which the architecture has to classify using the patches. The larger patch provides spatial information as they are large enough to understand where the central pixel is located. The smaller patch provides the detailed information about the neighborhood of the central pixel. Figure 2 shows an example of concentric patches.

There are two parallel paths for each patch where it goes through two dense blocks. The two paths are then concatenated and passed through a final dense block before the classification layer. Figure 3 shows the schematic of the dense block. The layers in dense block form a composite function [1]. It consists of three consecutive operations: batch normalization (BN) [18], followed by a rectified linear unit (ReLU) and a 3×3 convolution (Conv). The transition layers are placed in between the blocks for spatial reduction. It consists of batch normalization and an 1x1 convolution followed by 2x2 pooling layer.

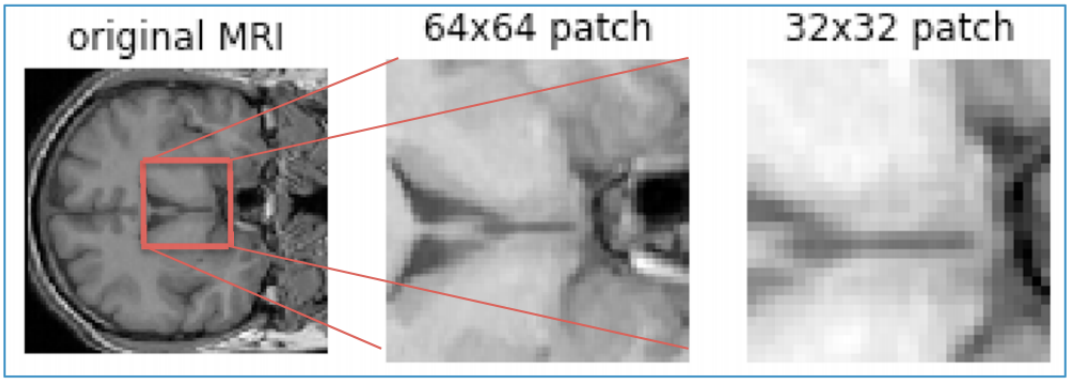


Figure 2. MRI with 64 and 32 size patches.

2.1. Growth Rate. An important feature in DeseNet is the use of growth rate which describes the rate at which the size of each layers inside the dense block grows. The growth rate regulates how much information is passed at each layer to the future layers. The colored connection in Figure 3 shows how the information is passes between the layers in dense block. For e.g., a growth rate $k = 12$ implies that a filter size of 12 is used for each layer and therefore the output size of each layer will be $12 + n * 12$ wher n is the layer number. Experiments have shown that a small value of k is sufficient to efficiently transfer the information between the layers [1]. In our approach, we set the growth rate at 10.

2.2. Bottlenecks. It was shown in [19,20] that a 1×1 convolution can be introduced as bottleneck layer before each 3×3 convolution to reduce the number of input feature-maps, and thus to improve computational efficiency. In our approach, we used bottlenecks before convolutional and pooling layers

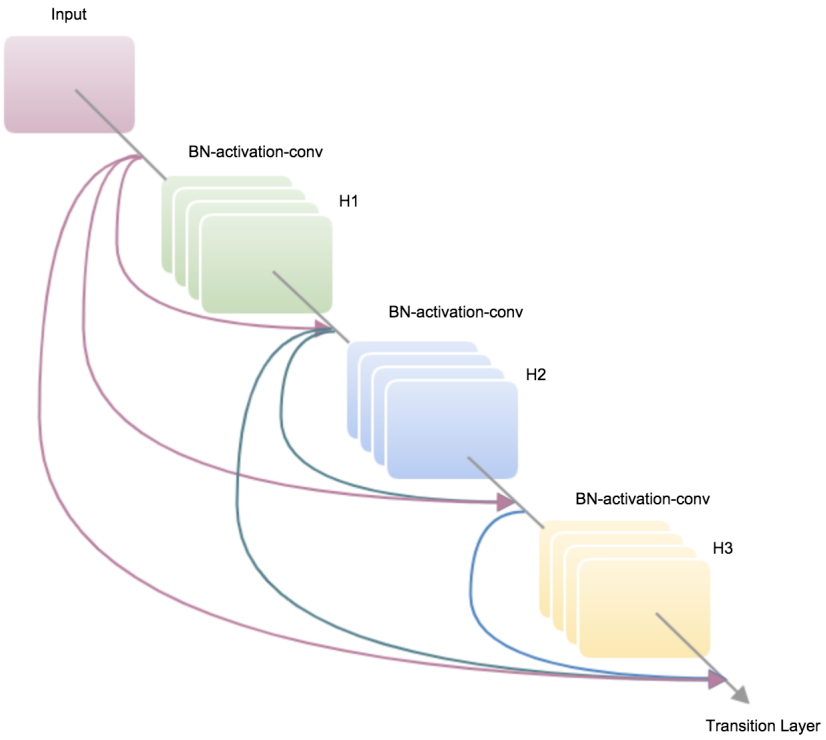


Figure 3. Dense Block.

2.3. Compression. The compression feature reduces the number of feature maps at the transition layer and also to improve computational efficiency. Since we used only two layers inside each dense block, there is no advantage to use compression for reducing feature maps. However, we observed that our architecture performs better when the compression is set at $\theta = 0.5$

2.4. Data. The IBSR (http://www.cma.mgh.harvard.edu/ibsr) dataset consists of 18 manually segment MR images of brain. Figure 4 shows an MRI of brain with the corresponding segmentation. The goal of our architecture is to perform segmentation identical to the manual approach. The quality of our segmentation is measured using the mean dice coefficient over the anatomical regions. All computations were run in-memory using a single NVIDIA Tesla K40 GPU with 12GB memory.

3. EXPERIMENTS

We started our experiment by normalizing the given data and generating the concentric patches for training. The outer patches were of dimensions 64x64 and the inner patches were of dimension 32x32. We trained our architecture for 200 epochs with a batch size of 32 using SGD optimization. We also use l2 regularization with dropout to prevent overfitting. The testing was done on the test images using dice coefficient. In order to get the best architecture from training, we used early stopping to the error rate on validation data with a patience limit of 10 epochs.

Figure 4 shows the results for few frames of a single patient. The results show that our architecture was able to accurately segment majority of the brain tissue. The borders show some unwanted segmentation which will be removed in future work.

The results in Table 1 show the values obtained on validation data. Here it is observed that using compression slightly improves the accuracy of the model which is in accordance with the DenseNet architecture. We evaluated our model on 5 test MRIs and achieved a dice coefficient of 0.673. Note that we achieved this level of accuracy with only 0.4M parameters which is a significantly low number for CNN architectures.

Table 1. Table showing the measurment type for each KPP

Layer	Configuration
Input (64 x 64 and 32 x 32)	2
Dense block (1) x 2	$\begin{bmatrix} 1 & 1 \\ 3 & 3 \end{bmatrix} x2$
Transition layer (1) x 2	1x1 conv and 2x2 pool
Dense block (2) x 2	$\begin{bmatrix} 1 & 1 \\ 3 & 3 \end{bmatrix} x2$
Transition layer (2) x 2	1x1 conv and 2x2 pool
Merge layer	-
Dense block (3) x 1	$\begin{bmatrix} 1 & 1 \\ 3 & 3 \end{bmatrix} x2$
Transition layer (3) x 1	1x1 conv and 2x2 pool
Classification layer	4D Fully Connected

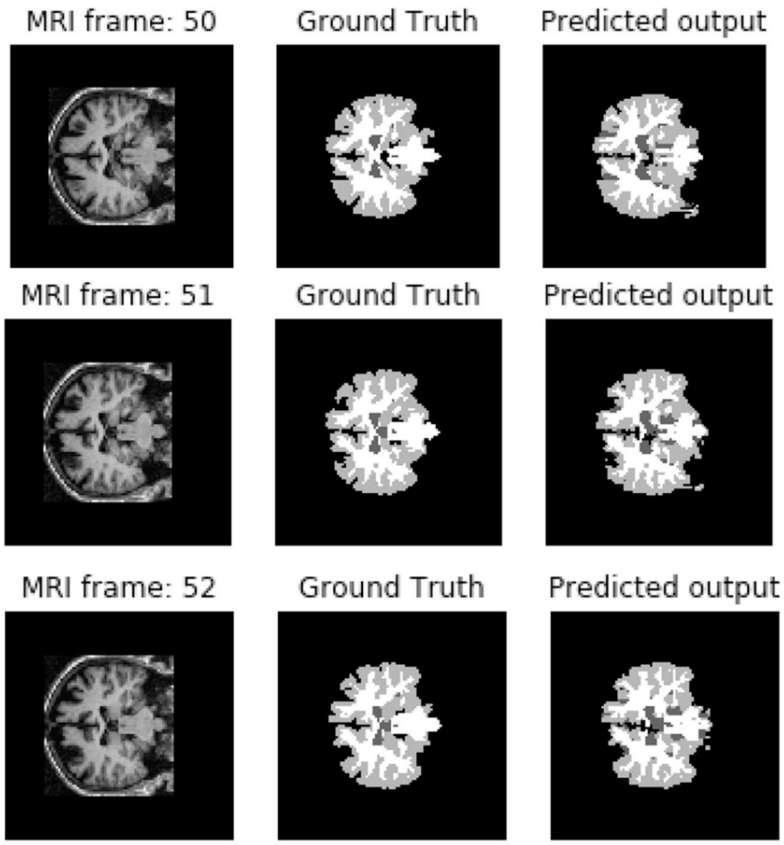


Figure 4. Dense Block.

4. CONCLUSION

In this section, we have performed the MRI brain segmentation using the DenseNet architecture. The results show that our approach can accurately perform the segmentation. We plan to improve the accuracy of this approach and use it for Parkinson's MRI segmentation in the future work.

REFERENCES

[1] S. Agarwal, L. E. Pape, C. H. Dagli, N. K. Ergin, D. Enke, A. Gosavi, R. Qin, D. Konur, R. Wang, and R. D. Gottapu. Flexible and intelligent learning architectures for sos (fila-sos): Architectural evolution in systems-of-systems. *Procedia Computer Science*, 44:76–85, 2015.

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[3] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[4] E. Curry. System of systems information interoperability using a linked dataspace. In *System of Systems Engineering (SoSE), 2012 7th International Conference on*, pages 101–106. IEEE, 2012.

[5] C. H. Dagli, D. L. Enke, N. Ergin, D. Konur, R. Qin, A. Gosavi, R. Wang, I. Pape, S. Agarwal, R. D. Gottapu, et al. Flexible and intelligent learning architectures for sos (fila-sos). *Center for Systems and Software Engineering, University of Southern California IN*, 20, 2014.

[6] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2012.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[10] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[11] S. Hochreiter, A. S. Younger, and P. R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

[12] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[13] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision*, pages 646–661. Springer, 2016.

[14] E. Kremers, P. Viejo, O. Barambones, and J. G. de Durana. A complex systems modelling approach for decentralised simulation of electrical microgrids. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 302–311. IEEE, 2010.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply-supervised nets. In *Artificial intelligence and statistics*, pages 562–570, 2015.

[19] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[20] A.-r. Mohamed, G. E. Dahl, G. Hinton, et al. Acoustic modeling using deep belief networks. *IEEE Trans. Audio, Speech & Language Processing*, 20(1):14–22, 2012.

[21] J. M. Parker. Applying a system of systems approach for improved transportation. *SAPI EN. S. Surveys and Perspectives Integrating Environment and Society*, (3.2), 2010.

[22] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[23] J. K. Pugh and K. O. Stanley. Evolving multimodal controllers with hyperneat. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 735–742. ACM, 2013.

[24] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[26] S. Saxena and J. Verbeek. Convolutional neural fabrics. In *Advances in Neural Information Processing Systems*, pages 4053–4061, 2016.

[27] T. Schaul and J. Schmidhuber. Metalearning. *Scholarpedia*, 5(6):4650, 2010.

[28] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[30] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[31] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.

[32] S. Zagoruyko and N. Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[33] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[34] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[35] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2(6), 2017.

VITA

Ram Deepak Gottapu was born in India in 1988. He received a Bachelor of Science degree in Electronics and Communications Engineering from Anil Neerukonda Institute of Science and Technology (Andhra Pradesh, India) in 2010. He also received his Master's of Science degree in Systems Engineering from Missouri University of Science and Technology in December 2015. He began his PhD study in January 2016 in the Department of Engineering Management and Systems Engineering at Missouri University of Science and Technology. His research interests includeed deep learning architecture modeling, computer vision and natural language processing. He received his Doctor of Philosophy in Systems Engineering from Missouri University of Science and Technology in May 2020.