THESIS

AUTOMATED DEEP LEARNING ARCHITECTURE DESIGN USING DIFFERENTIABLE ARCHITECTURE SEARCH (DARTS)

Submitted by

Kartikay Sharma

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2019

Master's Committee:

Advisor: Chuck Anderson

Ross Beveridge
Michael Kirby

ABSTRACT

AUTOMATED DEEP LEARNING ARCHITECTURE DESIGN USING DIFFERENTIABLE ARCHITECTURE SEARCH (DARTS)

Creating neural networks by hand is a slow trial-and-error based process. Designing new architectures similar to GoogleNet or FractalNets, which use repeated tree-based structures, is highly likely to be inefficient and sub-optimal because of the large number of possibilities for composing such structures. Recently, neural architecture search algorithms have been able to automate the process of architecture design and have often attained state-of-the-art performances on CIFAR-10, ImageNet and Penn Tree Bank datasets.

Even though the search time has been reduced to tens of GPU hours from tens of thousands of GPU hours, most search algorithms rely on additional controllers and hypernetworks to generate architecture encoding or predict weights for sampled architectures. These controllers and hypernetworks might require optimal structure when deployed on a new task on a new dataset. And since this is done by hand, the problem of architecture search is not really solved. Differentiable Architecture Search (DARTS) avoids this problem by using gradient descent methods.

In this work, the DARTS algorithm is studied under various conditions and search hyperparameters. DARTS is applied to CIFAR-10 to check reproducibility of the original results. It is also tested in a new setting — on the CheXpert dataset — to discover new architectures and is compared to a baseline DenseNet121 model. The architectures searched using DARTS achieve better performance on the validation set than the baseline model.

ACKNOWLEDGEMENTS

I would like to thank my parents, without their support none of this would have been possible. I'd like to thank my adviser Dr. Chuck Anderson for his patience during the period when I was exploring various topics for the thesis while he offered consistent guidance. To all members of my committee, SNA and Wayne Trzyna I'd like to extend sincere gratitude for their help in providing machines with GPUs without which I would not have been able to run a major chunk of the experiments in this thesis. Finally, I'd like to thank Elliot Forney and Graduate School for providing the LaTeX Thesis Template and Documentation.

DEDICATION

I would like to dedicate this thesis to my friend Vandesh Jajoo.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Deep Learning architectures have been successfully deployed for image classification tasks achieving state-of-the-art performance [Szegedy et al., 2014, Larsson et al., 2016, Huang et al., 2016]. However, designing architectures by hand is a tedious process and requires years of experience. For someone who is a beginner to the Deep Learning field, the architecture designs often seem unintuitive to study let alone to design them manually.

There is a growing trend in the field to use architectures that employ repeated cell based motifs. Manually picking one cell architecture out of billions of possibilities is not only highly likely to be sub-optimal due human error and biases, but also highly inefficient. The problem is even worse for domains such as time-series and text-based problems, which do not have a repository of well defined architectures as image-based tasks do. For such tasks, the architecture design is a slow trial-and-error based procedure. These problems are alleviated by architecture search algorithms.

Various architecture search algorithms, developed in the last 4-5 years, have been able to tackle large datasets such as ImageNet and Penn Tree Bank and achieved better performances, often state-of-the-art performance, than most human designed network [Zoph and Le, 2016, Zoph et al., 2017, Pham et al., 2018, Liu et al., 2017]. But, the recent field has been facing its own set of challenges. While the search cost has been reduced to tens of GPU hours from *tens of thousands* of GPU hours, most algorithms utilize a controller, hypernetwork or surrogate network to generate architecture encoding, predict performance or weights for sampled architectures. Despite their success, these controllers and hypernetworks themselves might require optimal architectures when deployed on new tasks from different domains. This optimization is done manually and thus the problem of architecture search is not really solved.

Differentiable Architecture Search (DARTS) [Liu et al., 2018] alleviates this problem. They make the search space continuous which allows it to be represented by parameters and thus be optimized by gradient descent methods while still keeping the search cost low. The experiments in

this thesis employ DARTS on CheXpert dataset, a new test domain, and study its performance on the task and its search efficiency. The experiments successfully replicated the original findings on CIFAR-10 dataset and discovered a cell in under 12 hours which achieved 97.35% test set accuracy. On CheXpert dataset, the best model obtained 0.8832 ± 0.002 mean AUC on the validation set as compared to 0.8752 by the human-designed baseline model. The results also provide evidence that the architectures learned on one dataset are transferable to other datasets.

This text is organized in the following chapters. Chapter 2 presents relevant background for the task of network architecture search. It provides an overview of some recent search algorithms and describes how they address various components of the search problem along with their performance on various datasets. Chapter 3 discusses DARTS algorithm and provides details about the subtleties in the algorithm during network search and evaluation. Chapter 4 presents DARTS's application on CIFAR-10 dataset to replicate the original study and to explore the effects of various hyperparameters on network search and evaluation. The experiments and results also demonstrate DARTS's successful application on CheXpert dataset. Finally, Chapter 5 concludes by summarizing the results of the experiments and discusses the advantages and disadvantages of the DARTS algorithm. Directions for future research on the topic and further experiments are also described in the chapter.

Chapter 2

Background

This chapter presents a very brief overview of the problem of Neural Architecture Search by breaking it down to its parts. Then it describes some recent works that have advanced the field. These works present diverse ideas and techniques that work towards solving the efficiency issues faced by the field to make using such algorithms practical.

There are three major components to the architecture search problem. The first component is the search space. It describes the structure for architectures which can be represented and searched for by the algorithm. For example, its capacity to represent architectures may range from simple fully connected feedforward networks to feedforward convolutional networks, like AlexNet [Krizhevsky et al., 2012], to convolutional networks with branching, like GoogleNet [Szegedy et al., 2014] and FractalNets [Larsson et al., 2016]. The second component is the search space exploration method. Various methods such as reinforcement learning, evolutionary methods and back propagation have been used for exploring search spaces. The final component of the problem is the evaluation of architectures in the search space. This is done to relatively rank the architectures to select the optimal one. A variety of techniques such as training the architectures from scratch, weight prediction and weight reuse have been used for this aspect of the problem. The remainder of this section describes how the three components mentioned above are tackled by different algorithms along with other challenges encountered in the task of architecture search.

Neural Architecture Search (NAS) [Zoph and Le, 2016] is based on the idea that network architectures can be encoded as a sequence of tokens. So, a controller may be trained using reinforcement learning to generate such strings. The search space includes feedforward convolutional networks and may include skip connections, pooling and batch normalization operations. An autoregressive recurrent neural network (RNN) controller is utilized to generate encoding for child architectures which are then created and trained for a fixed number of epochs. The validation set performance is used as a reward signal to update the controller using the REINFORCE al-

gorithm [Williams, 1992] so that the controller learns to generate better child architectures over time.

The searched architectures achieved near state-of-the-art performance on CIFAR-10 dataset — 4.47 % test set error with 7.1 M parameters and 3.65 % test set error with the same network but with more filters on each layer. But, the algorithm is extremely inefficient. They trained 12,800 child networks before selecting the final architecture and the total search process required 22,400 GPU hours. Again the algorithm can search for RNNs and CNNs without branching structures.

Deep Architect [Negrinho and Gordon, 2017] introduced a compositional and modular language for representing search spaces. The language is capable of representing tree-based search spaces for convolutional networks and covers architectures like AlexNet [Krizhevsky et al., 2012], VGG [Simonyan and Zisserman, 2014] and ResNet [He et al., 2015]. They used Monte Carlo Tree Search (MCTS) [Browne et al., 2012] and Sequential Model Based Optimization (SMBO) [Hutter et al., 2011] to explore and search the space. They evaluate the models by training the child architectures and rank them based on their performance on the validation set. Although the algorithm requires fewer resources than NAS, it resulted in poorer 11% error on the CIFAR-10 test set.

One of few major drawbacks of NAS [Zoph and Le, 2016] is that even with a large search space it isn't capable of representing tree-based architectures which allow computation branching. Also, exploring large search spaces is computationally expensive. So, to reduce the size of the search space in an effective way, NASNET [Zoph et al., 2017] following the ideas of inception module like structures from GoogleNet [Szegedy et al., 2014], used computation units called cells as the building blocks of the network. The goal of the search is to discover cell(s). The final architecture is composed by repeatedly stacking the learned cells. This approach solves the above mentioned issue in the original work — the search space is smaller yet it is capable of representing tree-based architectures.

During search an RNN controller, which uses proximal policy optimization [Schulman et al., 2017] for updates, is trained to sample cells. The child architecture is constructed by stacking the cells one after another. Validation set performance of the child architecture is used as a reward

signal to update the controller. Once the controller is trained, the top performing cell is used to create the final architecture and it is trained from scratch. The cells in the final architecture share the same structure but have different weights.

In addition to reduced search space, this approach also has other benefits. First, the cells found in one setting, on a particular dataset with a certain image size, generalize better to other problems — other datasets with different input image size — because the complexity of the architecture is no longer dependent on the depth and the size of input image. For example, in the paper the authors demonstrate the transferability of the cell learned on CIFAR-10 dataset. The learned cell was also able to achieve state-of-the-art performance on ImageNet and COCO datasets. Second, the search is faster due to the reduced search space. Finally, it is easy to modify the architecture design, by varying the number of cells and filters, as per the performance requirements and computation demands. But, the algorithm suffers from another major drawback, the search requires *tens of thousands* of GPU hours.

Architecture search using Reinforcement Learning with Q-Learning has been studied in Block-QNN [Zhong et al., 2017]. Their search space is restricted to convolutional networks. They were able to achieve comparable performance on CIFAR-10 as compared to original NAS but were poorer than NASNET. The algorithm also suffers from inefficient searching and requires *tens of thousands* of GPU hours for search.

Historically, architecture search using genetic methods has been studied as a sub-field of neuro-evolution [Holland, 1992,Stanley and Miikkulainen, 2002,Miller et al., 1989,Floreano et al., 2008, Yao and Liu, 1997]. Some of these methods used genetic methods for both exploring the search space and evaluating the evolved architectures, but struggled at scale. Some recent works [Xie and Yuille, 2017,Miikkulainen et al., 2017] tried tackling large tasks but weren't able to get comparable performances to the best human-designed models.

AmoebaNet [Real et al., 2018] is an image classifier based on evolutionary methods. NASNET search space, represented as a directed acyclic graph, is explored by applying mutating operations such as changing parents (incoming data such as latent activation maps) and changing operations

(eg. convolution in place of pooling) within each cell. Tournament selection is used for evaluating the sampled architectures with a key change—after each iteration the oldest architecture is discarded instead of the worst one. They claim that the concept of aging allows the algorithm to explore more search space and thus leads to good architectures. Each evolved architecture is trained for 25 epochs. They demonstrate that Evolution is faster than Reinforcement Learning in the initial stages of the search, i.e., models perform better, and may be useful in a resource constrained environment. Both search methods—reinforcement learning and evolutionary search—eventually approach the same performance. While their best network is able to achieve 3.34% test error with 3.2M parameters on the CIFAR-10 dataset, the experiments employed 450 GPUs for 7 days which, again, is *tens of thousands* of GPU hours.

The primary reason behind the enormous computation cost for these algorithms is that each sampled architecture is trained from scratch only to throw them away once the validation accuracy has been used either to update the controller or to rank them. This results in a lot of discarded learned information and a lot of computation is wasted. The following works have targeted this problem and have approached it with different and interesting perspectives.

SMASH [Brock et al., 2017] introduced One-Shot model to overcome this problem. They trained a bandit-based hypernetwork to transform an architecture's binary encoding to weight space. In other words, they used the hypernetwork to generate the weights for newly sampled architectures instead of training them and this reduces the overall search time significantly. The authors provide empirical evidence that there is a good correlation between the true validation error and SMASH validation error for 50 randomly sampled architectures.

While the search space is capable of discovering architectures like ResNet [He et al., 2015], FractalNets [Larsson et al., 2016] and DenseNets [Huang et al., 2016], it is incapable of discovering wholly new structures by itself. The performance on CIFAR-10 is poorer than NAS with 5.53% test set error with 4.6 M parameters and 4.03% error with 16M parameters.

A similar, but slightly different, approach along the same line of not training each child architecture is Progressive NAS [Liu et al., 2017]. The algorithm uses NASNET search space. They use

a surrogate model to predict the performance of sampled cells instead of training them. The cells in the search space can be imagined to be composed of simpler structures, called blocks, starting at depth zero. At depth zero, the blocks are simple operations like convolution, pooling, etc.. Blocks at level i are composed of blocks at one level below, i.e., $i - 1$.

Then, to explore and search through the search space, they construct all possible blocks at level i and use them to train a model to predict performance of blocks at level $i + 1$. The process is repeated till the cell of desired depth is found. While the correlation between predicted and actual performance of the cells does not seem to be great, the performance of the final model, *PNASNET-5*, is comparable to NASNET and AmoebaNet [Real et al., 2018] on both CIFAR-10 and ImageNet datasets.

Efficient Architecture Search [Cai et al., 2017] has addressed the problem in a yet another way. They argue that once a child network is trained, a lot of information has been learned and that these learned weights should be reused instead of wiping the slate clean and starting again. Their search space contains feedforward convolutional networks without branching and skip connections. Once a child network has been trained, they use its current structure to create new architectures using function preserving Net2Net transformations [Chen et al., 2015]. This can create a deeper or wider network by reusing the weights of the old architecture. Their approach requires 2 meta-controller actor networks which use REINFORCE [Williams, 1992] policy update. This method reduced the search time to approximately 240 GPU hours and their best network obtained 3.44% test error on CIFAR-10 dataset with 10.7M parameters and additional regularization strategies such as Cutout [DeVries and Taylor, 2017] and Dropout [Huang et al., 2016].

To overcome the shortcomings of a simple search space, which does not include branching and skip connections, the authors extend the work by introducing path-level network transformations [Cai et al., 2018a]. This allows branching to be included in the search space. The search can be initialized with different tree-cell bases such as DenseNet [Huang et al., 2016] and PyramidNet [Han et al., 2016]. To tackle the challenge of encoding the architectures with branching they use tree-structured LSTMs [Tai et al., 2015] for exploring the search space. The controller was trained

on 500 child networks for 20 epochs each. The searched models achieve 2.49% and 2.30% test error with 5.7M and 14.3M parameters, respectively. The search took approximately 200 GPU hours.

Another interesting approach to reduce search cost is taken up in Efficient Neural Architecture Search [Pham et al., 2018]. The central idea of the work is to create an over parameterized network (one-shot model) which can represent the whole search space in itself and the goal of the search is to find the optimal subgraph within this large graph. The efficiency comes from the fact that the weights are shared between all sampled subgraphs. This counter-intuitive idea that sharing weights could work for all child architectures is supported by observations in multitask learning and transfer learning which suggest that weights learned for one architecture on a particular problem are reusable in other settings without many changes. [Razavian et al., 2014, Zoph et al., 2017, Luong et al., 2015].

They define a thorough tree-based search space capable of searching for both convolutional neural networks (CNNs) and recurrent neural networks (RNNs). In the case of CNNs, there is an option to search for the full architecture (macro search) or search for cells (micro search). The search space can represent 1.6×10^{29} and 1.3^{11} unique architectures for macro and micro search, respectively, for CNNs and about 10^{15} architectures for RNNs. Their experiments show that micro search performed better on CIFAR-10 with 3.54% and 2.89% test error with 4.6M parameters for macro and micro search, respectively. ENAS is 1000 times faster than NAS.

In the next section, we'll look at a similar algorithm called Differentiable Architecture Search (DARTS) in detail. It also uses a One-Shot model as a single over-complete model but uses back propagation to discover the optimal subgraph. Due to this reason the algorithm does not require additional controllers or hypernetworks. This is crucial because it is not reasonable to assume that the hyperparameters, e.g., number of layers and units, for controllers and surrogate networks mentioned in respective works will work optimally for a new task or dataset. They require tuning themselves and so the problem becomes recursive.

Chapter 3

Methods

This chapter provides a detailed description of the search and evaluation procedure of Differentiable Architecture Search (DARTS) Algorithm [Liu et al., 2018]. During search, this method starts out with a single large network, which represents the complete search space, and utilizes gradient descent to discover the optimal subgraph. The absence of additional hypernetwork or controller makes the DARTS algorithm appealing.

3.1 DARTS

DARTS, similar to architectures like GoogleNet [Szegedy et al., 2014] and NASNET [Zoph et al., 2017], considers computation units called cells as the basic building blocks of the architecture. The final architecture is composed of the learned cell. For example, convolutional networks are constructed by repeatedly placing the cells one after the other till the desired depth is achieved.

This perspective of architecture design offers a few advantages. First, the search space is reduced significantly because the cells share structure. There are two types of cells in this algorithm, as we shall learn later, and the cells of the same type share structure in the final architecture. So the goal of the search in its essence is to find one cell of each type which performs best on the objective on the validation set. Second, the reduced search space is also efficient in terms of GPU hours. Third, cells discovered on one dataset for a particular task have been shown to be transferable to other datasets and problems. Finally, it is simple to change the capacity of the model by updating the number of cells and filters according to resource limitations.

Previous works [Brock et al., 2017,Liu et al., 2017,Cai et al., 2017,Cai et al., 2018a,Pham et al., 2018, Real et al., 2018] consider the search space to be discrete, so the architectures are treated as black-boxes during search. DARTS, on the other hand, provided a continuous representation of the architecture space, thereby allowing it to be represented by parameters. This enabled the authors to explore the search space using gradient descent. Additional hypernetworks, controllers, surrogate

models for performance predictions, etc., are not required for this approach. This viewpoint of search space exploration makes the algorithm efficient and requires orders of magnitude fewer resources while still delivering state-of-the-art performance.

The following sections describe these aspects of the algorithm: (1) the search space, (2) optimization and approximate architecture gradient, (3) deriving discrete cell architectures, (4) assembling searched architecture, and (5) summary of the complete process. Please note that the following description has been catered to simplify understanding for DARTS's application for CNNs. Please refer the original work for RNNs.

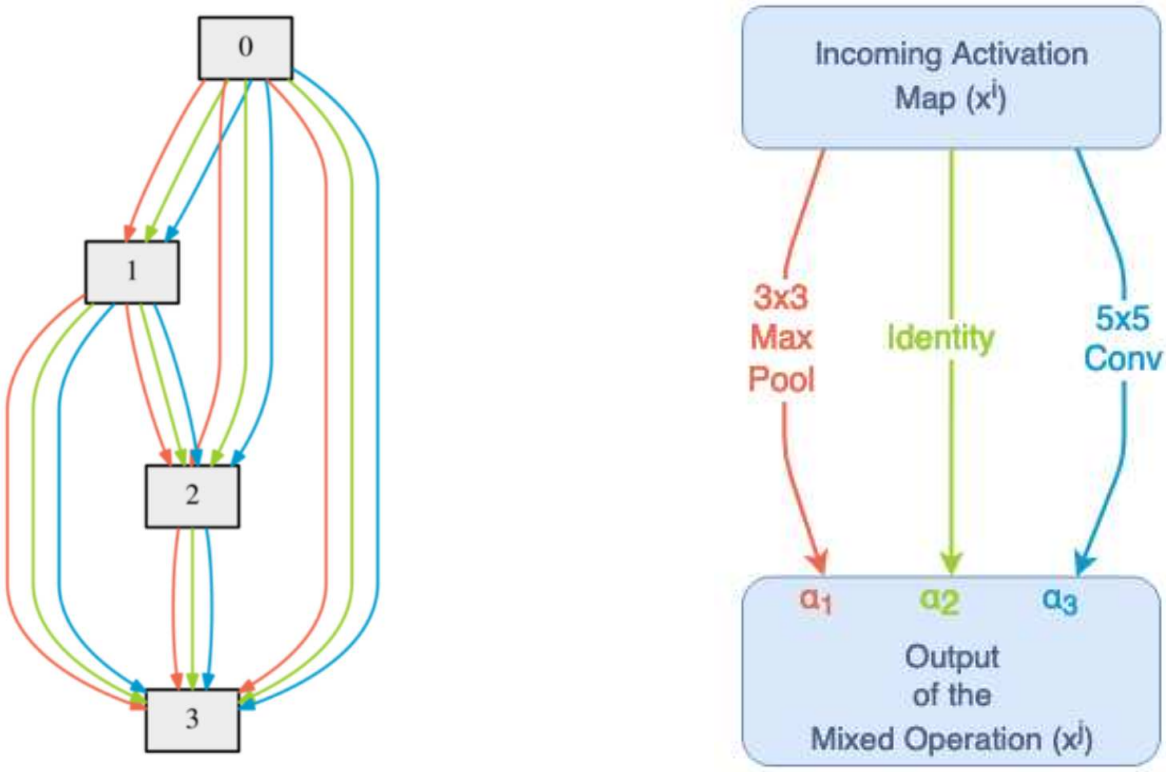
3.1.1 Search Space

Cells are an ordered sequence of B nodes connected to form a directed acyclic graph, where B is a hyperparameter. Intermediate hidden data representation, for example, feature maps, form the nodes $x^{(i)}$ of the graph. Operations such as pooling or convolution represent edges. An operation $o^{(i,j)}$ takes $x^{(i)}$ as input and computes $o^{(i,j)}(x^{(i)})$.

A user may choose various operations such as max pooling, convolution, identity, dilated convolution, etc., with varying filter sizes, if applicable, to be the set of candidate operations. At the beginning of the search, each operation of the cell is a mixed operation, represented by different colored edges between each pair of nodes in Figure 3.1a[1]. A mixed operation $\bar{o}^{(i,j)}$ is composed of all the candidate operations as visualized in Figure 3.1b. Softmax is applied over all operations to obtain the output of the mixed operation as shown in equation 3.1, where α are the parameters for the mixed operations. This relaxation of the choice of the operations makes the search space continuous and, therefore, allows α to be optimized by gradient descent to find the optimal operations.

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{exp(\alpha_o^{(i,j)})}{\sum_{\acute{o} \in \mathcal{O}} exp(\alpha_{\acute{o}}^{(i,j)})} o(x) \tag{3.1}$$

[1]Figure used from [Liu et al., 2018].



(a) Sample cell diagram with mixed operations

(b) Mixed Operation $\bar{o}^{(i,j)}$

Figure 3.1: Cells diagrams represented as a directed acyclic graph: (a) Cell Structure during search where each mixed operation contains 3 candidate operations. (b) Mixed Operation between a pair of nodes which contains 3 operations, all of which are applied to the incoming data to compute the output of the operation as shown in equation 3.1.

Each cell has two inputs, the cell at layer l takes input from previous two layers, i.e., layers $l-1$ and $l-2$. The first cell in the network uses the image on both input nodes. The outputs of intermediate nodes are calculated as shown in the equation 3.2. The outputs of all intermediate nodes are combined by applying a reduction operation, such as concatenation or addition, to get the output of the cell.

$$x^{(j)} = \sum_{i<j} \bar{o}^{(i,j)}(x^{(i)}) \tag{3.2}$$

There are two variants of the cells —*normal* and *reduction* cells. Normal cells differ by reduction cells in two ways. First, the size of the activation maps remain constant when processed by normal cells. On the other hand, the reduction cells reduce the size of the output activation maps

to half the input size. This is achieved by setting stride as two for the operations adjacent to the input nodes. Second, the number of output filters also remain constant as input filters in normal cells while they double for reduction cells.

For search, a large network is created by stacking the normal and reduction cells. This placement of normal and reduction cells is manual. The original work uses normal cells as the major chunk of the network and puts two reduction cells at 1/3rd and 2/3rd depth of the network, where depth is another hyperparameter. Additional downsampling operations may be inserted at the beginning of the network to reduce the size of the input image or feature maps.

During search, all normal cells and reduction cells share parameters for their mixed operations on edges α_{normal} and α_{reduce}, respectively. The task of searching is to discover the optimal parameters for normal and reduction cells

3.1.2 Optimization and Approximate Architecture Gradient

The goal of the architecture search is reduced to learning operations on edges which are represented as a set of continuous variables $\alpha = \{\alpha_o\}$. A vector $\alpha^{(i,j)}$, of size $|\mathcal{O}|$, parameterizes the operation mixing weights for a pair of nodes *(i, j)*. A cell is thus encoded by α. α_{normal} and α_{reduce} together represent the complete encoding of the architecture. When the search terminates, each mixed operation $\bar{o}^{(i,j)}$ gets replaced by the operation with the maximum weight in $\alpha^{(i,j)}$, i.e., $o^{(i,j)} = argmax_{o \in \mathcal{O}} \ (\alpha_o^{(i,j)})$

The task, now, is to learn the architecture weights α and the network weights w for each mixed operation simultaneously. α^* are found by minimizing the validation loss $\mathcal{L}_{val}(w^*, \alpha^*)$ and minimizing the training loss gives the network parameters w^*, i.e., $w^* = argmin_w \ \mathcal{L}_{train}(w, \alpha^*)$. It is interesting to note that both training loss ($\mathcal{L}_{train}$) and validation loss ($\mathcal{L}_{val}$) are functions of w and α.

This is a bi-level optimization problem [Colson et al., 2007]. The architecture parameters α serve as the upper-level variable and the network parameters w is the lower-level variable.

$$\min_{\alpha} \ \mathcal{L}_{val}(w^*(\alpha), \alpha) \tag{3.3}$$

$$s.t. \ \ w^*(\alpha) = argmin_w \ \mathcal{L}_{train}(w, \alpha) \tag{3.4}$$

Evaluating the inner optimization is expensive and it restricts the ability to calculate the exact gradient. To overcome this problem, the following approximation scheme has been proposed by authors:

$$\nabla_{\alpha} \ \mathcal{L}_{val}(w^*(\alpha), \alpha) \tag{3.5}$$

$$\approx \ \nabla_{\alpha} \ \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha) \tag{3.6}$$

where w represent the current state of network weights and ξ is the learning rate of inner optimization. To bypass the requirement to solve equation 3.4 completely, the architecture weights α and network parameters w are updated alternatively. First, α are updated by descending $\nabla_{\alpha} \ \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$. Then weights w are updated by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$. This alternative update is repeated till the network has been trained to convergence. When $\xi > 0$, the gradient approximation is referred to as **Second Order Approximation**.

When $\xi = 0$, $\nabla_{\alpha} \ \mathcal{L}_{val}(w, \alpha)$ gives the gradient for the architecture parameters α under the assumption that $w^*(\alpha)$ is same as w. This strategy will be referred to, in the following chapters, as **First Order Approximation**. Please check the original work [Liu et al., 2018] for details about the optimization.

3.1.3 Deriving Discrete Cell Architecture

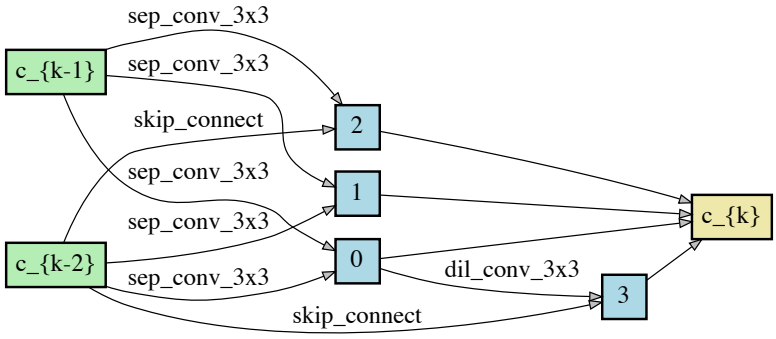
During training, the weights of the mixed operations get updated and the operations which result in lower validation set loss start to win out. Once the network training is complete, the next task is to derive discrete architecture for the cells by replacing the mixed operations with the most likely operation in cells.

This is done by keeping the top-k strongest operations from distinct previous nodes in the cell, where k is a hyperparameter. Softmax applied to the weights α_o for a mixed operation defines its strength, and is given by the following expression $\frac{exp(\alpha_o^{(i,j)})}{\sum_{ó \in \mathcal{O}} exp(\alpha_ó^{(i,j)})}$. The authors, in the original work, used k = 2 in the experiments to control for comparison with NASNET [Zoph et al., 2017], AmoebaNet [Real et al., 2018] and ProgressiveNAS [Liu et al., 2017]. All experiments in this text also use k = 2.

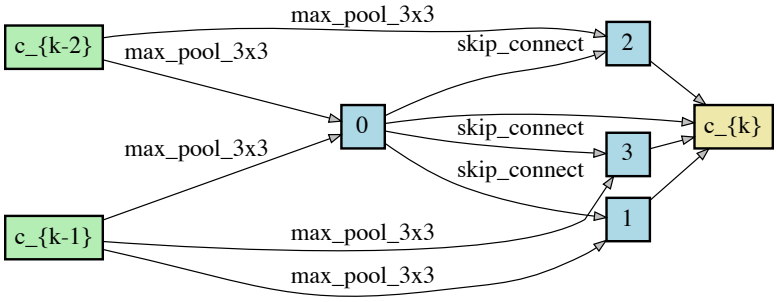
These discrete cell architectures can be represented as an encoding called a genotype. Genotypes are a list of $(operation,\ parent\_node)$ tuples for k inputs $\times$ B-1 nodes (input and intermediate) and the reduction operation, which calculates the output of the cell, for both normal and reduction cells. The cells discovered by the authors on CIFAR-10 have been visualized in the Figure 3.2. The green nodes are the input nodes for the cells and they get their values from the output of the previous two cells. The blue nodes represent the intermediate nodes, each has K=2 parents. The output of these nodes are computed by, first, applying the operations, marked on the edges, to the output of their respective parents and then adding them together. The output of the cell, the yellow node, is computed by depthwise concatenation of the outputs of the blue intermediate nodes.

3.1.4 Assembling Final Architecture From Learned Cells

Once the search has completed, the genotypes discovered from search for normal and reduction cells are used to create a new final architecture. The cells, as described earlier, are stacked manually to achieve the desired depth. Although the decision to arrange normal and reduction cells is a free setting, there has been a trend in the literature to use one reduction cell after every N normal cells. The authors insert a reduction cell at 1/3rd and 2/3rd depth of the network. This new network, once assembled, is trained on the task again with randomly initialized weights. It is easy to modify the capacity of the model by simply changing the depth of the model. Please refer to the Figure 3.4 to check the architecture design.



(a) Normal Cell



(b) Reduction Cell.

Figure 3.2: Best cells from original Study for CIFAR-10

3.1.5 Summary of the Search Process

The following set of steps are taken to discover a new architecture for a dataset. The experiments in Chapter 4 use the same steps to search for architectures on CIFAR-10 and CheXpert dataset.

- **Over-parameterized Network:** The search process starts by constructing an over parameterized network assembled by manually stacking cells, with mixed operations, of two kinds — normal and reduction cells. Two reduction cells are used and are placed at 1/3rd and 2/3rd depth of the network. Other settings are, of course, possible. The cells of same type share architecture weights α during training.

- **Search:** The goal of the search is to reduce mixed operations to the most likely operations within each cell type. Please refer to Figure 3.3 [2] for an overview. The output of the search is an encoding called a genotype that represents both normal and reduction cells.

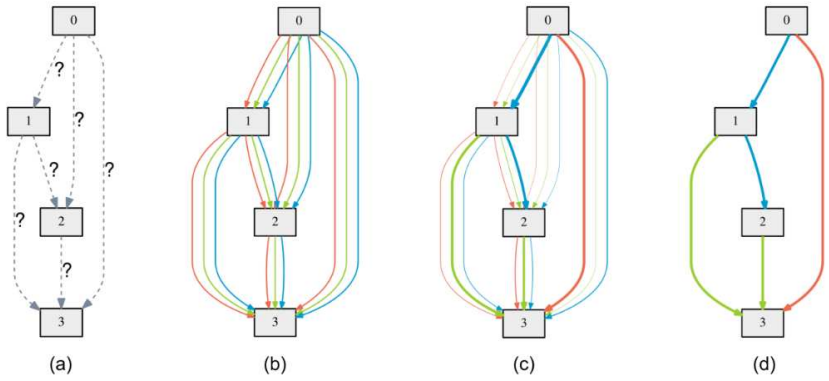
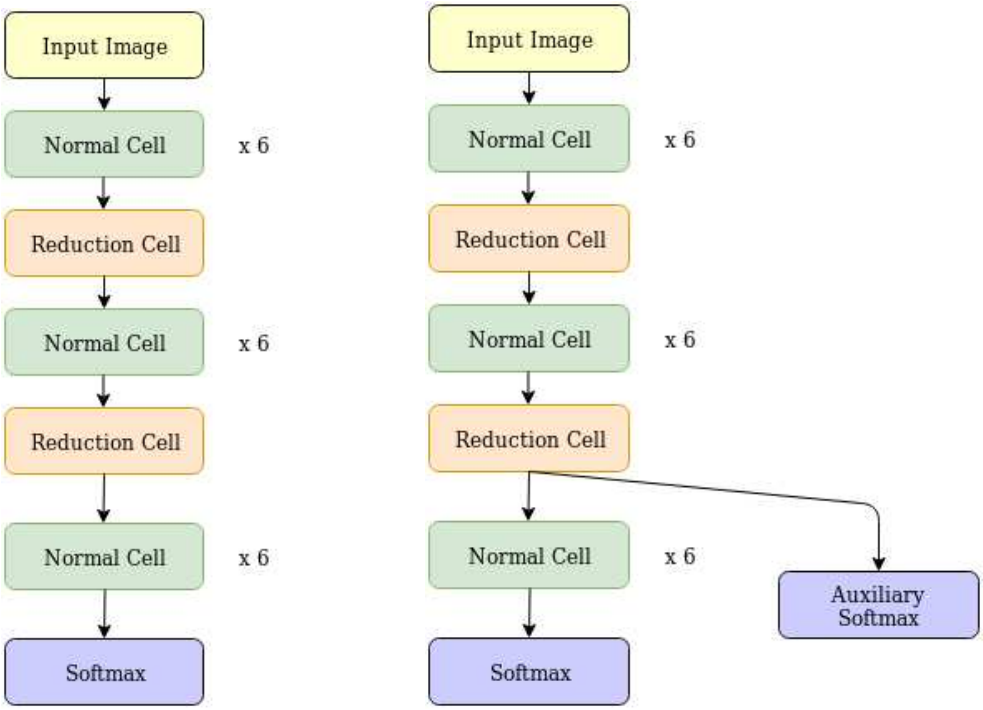


Figure 3.3: Overview of DARTS search: Initially (a) the operations between the edges are unknown. (b) The search starts with equal weights for all operations in the mixed operation. (c) During search, the optimal operations start to win out. Finally (d), the strongest operations are used to create discrete cell architectures.

- **Assemble and Train Final Architecture:** The searched genotype, which encodes both normal and reduction cell, is used to construct a new architecture of desired depth. Please note that the depth is usually increased when creating the final architecture when compared to depth of architectures during search. This is due to higher memory consumption of mixed operations during search and it forces use of smaller networks. The arrangement of cells is similar to the first step. The network is trained from scratch on the target dataset. An auxiliary classifier may be placed in the network as visualized in the Figure 3.4. This network structure used in CIFAR-10 evaluation experiments in Section 4.1.3.

[2]Figure used from [Liu et al., 2018]



(a) Network Assembly **(b)** Network with Auxiliary Classifier Layer

Figure 3.4: Network Structure for CIFAR-10 evaluation experiments (a) without and (b) with an auxiliary classifier layer.

Chapter 4

Experiments and Results

To test DARTS and its parameters in various settings, during search and evaluation, numerous experiments are performed and discussed in this chapter. The experiments in the first section are the replication of the original study on the CIFAR-10 dataset. The following section presents experiments to test DARTS in a new setting – on CheXpert dataset. The experiments are written in the Python programming language using the PyTorch deep learning framework [Paszke et al., 2017] . The original GitHub repository[3] was forked[4] to add documentation and new experiments.

4.1 CIFAR-10 Replication

The aim of this set of experiments is to test the effects of various search hyperparameters on cell performance and to check the reproducibility of the results. We start with discussing the preprocessing and augmentation applied to the dataset. Searching for new cells with different search hyperparameters is discussed in the second subsection. Finally, the genotypes discovered by search for various hyperparameters are evaluated and compared to the original study.

4.1.1 Dataset Preprocessing and Augmentation

The CIFAR-10 dataset contains 50,000 training and 10,000 test labeled images for 10 classes . The image size is 32×32 pixels. The following set of operations are applied on images before they are processed by the networks.

For training data, the images are upscaled to 40×40 by adding zero padding. Then, a patch of 32×32 pixels is randomly cropped. The next operation in the chain is random horizontal flipping which is applied to the 32×32 patches from the previous step. Additional regularization in the form of Cutout [DeVries and Taylor, 2017], which is a hyperparameter, is also used to augment

[3] https://github.com/quark0/darts

[4] https://github.com/sharmaskartik/darts_fork

training data. Finally, the images are mean centered and normalized *per channel* (not per pixel). For test data, only mean centering and normalization is applied to the images.

It is important to note that dataloaders in *pytorch* apply the transformations dynamically, i.e., only when the batch is requested. This means that the same training images, for a particular batch in subsequent epochs, are highly unlikely to be exactly the same because of the randomization in cropping and flipping operations which are applied on the fly. In other words, the network is trained with different training images each time.

4.1.2 Search

To discover new normal and reduction cells, DARTS is applied to the CIFAR-10 dataset. We begin by creating a large network that represents the complete search space. Over-parameterized cells with mixed operations as edges and $B = 7$ nodes are created and stacked repeatedly. All α's are initialized to a small random value to allow all operations to be chosen with equal probabilities, allowing more exploration in the initial phases of the training. The normal and reductions cells use the same α_{normal} and $\alpha_{reduction}$, respectively. The output of the cell is generated by applying depthwise concatenation to the outputs of all intermediate nodes. All operations are stride one, if applicable, except for the operations adjacent to the input nodes in the reduction cells, which are stride two. The following operations are used to create the mixed operations in cells:

- 3×3 and 5×5 Depthwise Separable Convolutions [Chollet, 2017]: This is a variant of standard convolution operation in which the operation is divided into two stages — depthwise convolution stage followed by a pointwise convolution stage. For example, suppose that the input image has M channels and N filters are desired of $3 \times 3 \times$ M size. The first stage has M filters of $3 \times 3 \times 1$ size, one for each image channel, and the second stage has N filters of $1 \times 1 \times$ M size. So, the overall size of the output activation maps is the same as it would be for using N filters of $3 \times 3 \times$ M size with standard convolution operation. But, depthwise separable convolutions reduces the number of multiplications and the number of weights as compared to the standard convolution.

- 3×3 and 5×5 Dilated Separable Convolutions [Yu and Koltun, 2015]: This is another variant of the standard convolution operation in which zero rows and columns are inserted in the filter. This increases the receptive field of the kernel, but keeps the number of parameters in the kernel low. For example, a 5×5 dilated kernel can be created with 9 parameters (3×3), instead of 25 parameters, by introducing zero rows and columns in the second and fourth positions.
- 3×3 Max Pooling
- 3×3 Average Pooling
- Identity (also called skip-connect)
- Zero (no connection)

Following [He et al., 2016], convolution operations are applied in ReLU-Conv-BatchNorm order. The separable convolutions are applied twice. Two settings for depth (number of cells) of the network is tested — 8 and 12. For an 8 layer network, the 3rd and 6th cells are reduction cells, while 5th and 10th cells are reduction cells for the 12 layer network. A 2-dimensional adaptive average pooling layer is put after all the cells and connects to an output linear layer. Softmax is applied to the output of the network and cross entropy loss is computed.

The 50,000 training images are split into two equal non-overlapping portions to be used as training and validation set for all search experiments. Please note that the test set is held out for this experiment and is used only for evaluation. The data augmentation and preprocessing described for the training data in Section 4.1.1 is applied to *both* training and validation data.

The network is initialized with 16 filters at the first cell and the number of filters are doubled at every reduction cell. At the same time, the activation maps are downsampled to half the size at the reduction cells. The network is trained on image classification task on the 10 classes for 50 epochs. The training set is randomly sampled with a batch size of 64. For each training batch, the validation loss, computed on a new validation batch, is used to update architecture parameters α and training loss is used to update the network parameters w.

To control for hyperparameters, in this replication study, the following settings are exactly the same as in the original work. The Adam optimizer [Kingma and Ba, 2014] is used to update the architecture parameters α's with β set to $(0.5, 0.999)$. The initial learning rate η_α is set to 3×10^{-4} and additional weight decay regularization 10^{-3} is applied. The network parameters w are updated using Stochastic Gradient Decent (SGD) with 0.9 momentum and with an initial learning rate η_w of 0.25 which is decayed to a minimum learning rate of 10^{-3} using Cosine Annealing Schedule [Loshchilov and Hutter, 2016]. Weight decay of 3×10^{-4} is used for the SGD optimizer. The authors do not discuss why two different types of optimizers, Adam and SGD, were used.

Due to resource constraints a sequential comparison was done instead of an exhaustive grid search; i.e., when one hyperparameter setting is found to work better on an experiment, then the same setting is used in all subsequent experiments. For example, once it is found that using cutout regularization makes the validation performance worse, it is not used in subsequent experiments when testing other hyperparameters. For the same reason, the results are compared on single runs and do not capture variability.

The first set of search experiments, Genotype 1 and 2, were conducted with layers = 12 and **first order gradient approximation** (please refer to Section 3.1.2) and the free variable is cutout regularization with width 16. The validation accuracy and execution time for all search experiments are summarized in Table 4.1. It can be observed that the model without cutout regularization performs better on the validation set than the one that uses it. So, as described earlier, cutout is removed from consideration for the next set of experiments.

Table 4.1: Validation accuracy and execution time for CIFAR-10 search experiments.

Genotype ID	Layers	Cutout	Gradient Approximation	Validation Accuracy	Execution Time (GPU hours)
1	12	True	First Order	86.15%	11.51
2	12	False	First Order	89.22%	10.56
3	12	False	Second Order	87.93%	71.72
4	8	False	First Order	88.50%	7.66

Next, search is performed using Genotype 3 with 12 layers, no cutout regularization and the gradient is computed using the second order method. In Table 4.1, we find that while Genotype 3 performs better than Genotype 1, its validation accuracy is poorer than Genotype 2. So, First Order gradient approximation works better than Second Order gradient approximation. Finally, with Genotype 4 as the normal and reduction cells, the last search experiment is conducted with 8 layers, no cutout and first order approximation. The validation accuracy is second best only to Genotype 2 and it does better than other two genotypes that use 12 cells during search. As noted earlier, these experiments are conducted on single runs and may vary with repeated runs.

Also in Table 4.1, in terms of time, in GPU hours[5], the search takes to finish, we find that the second order approximation method takes noticeably more time requiring almost 72 hours to finish. While Genotype 4 takes the least time to complete the search requiring 7.66 hours to finish. The best genotypes can be selected based on this information. But, to compare the relative rankings of the genotypes on validation set during search and the actual performance of their final architectures on test set, the selection is delayed until all genotypes have been evaluated on the test set. We'll look at these experiments in the next section.

4.1.3 Evaluation

All four genotypes searched in the previous section are evaluated in this section for the classification task on CIFAR-10 dataset. The aim is to check if the genotypes rank in the same order on test set accuracy during evaluation as they did on validation set during search. The complete dataset, all 50,000 training and 10,000 test images, are used for the evaluation experiments. Data preprocessing and augmentation is done exactly as described in Section 4.1.2.

For all experiments in this section the networks are created by stacking 20 cells with 36 filters at the first cell. The reduction cells are placed at 1/3rd and 2/3rd depth of the network, i.e., the 7th and 14th cells. An additional auxiliary output layer may be used. It is placed after the second

[5]on NVIDIA Tesla V100 GPUs

Table 4.2: Tuning hyperparameters on networks created using Genotype 4

Cutout (Length = 16)	Auxiliary Output Layer (Weight = 0.4)	Test Accuracy
True	True	96.99%
True	False	96.32%
False	True	96.41%
False	False	95.88%

reduction cell, i.e., after 14$^{\text{th}}$ cell in the network. The network structure is visualized in Figure 3.4. The training begins from scratch with randomly initialized weights.

The networks are trained for 600 epochs. The training set is randomly sampled with a batch size of 96. Stochastic Gradient Descent with 0.9 momentum is used to optimize the network parameters w with an initial learning rate $\eta_w = 0.025$. The learning rate is decayed to zero using a cosine annealing schedule. Additional regularization in the form of weight decay of 3×10^{-4} and drop path probability [Zoph et al., 2017] of 0.2 is used.

The above hyperparameter values are the same as in the original work. There are two additional hyperparameters — cutout regularization and auxiliary output layer — and they have one additional hyperparameter each — *cutout length* and *auxiliary weight*, respectively. *Cutout length* controls the size of the masked patch on the input image, while *auxiliary weight* controls the contribution of the loss at the auxiliary layer to the total loss. These are tuned in the experiments here using Genotype 4 (refer to Table 4.1) in a binary fashion — they are set to either *on* or *off*.

Four separate networks are trained using Genotype 4 for each combination of cutout and auxiliary and their test set accuracy is recorded. If used, the cutout length is set to 16 and auxiliary weight is 0.4. The results are presented in the Table 4.2. It can be observed that the network with both cutout and auxiliary set to true gives the highest test set accuracy of 96.99%. When only one of them is used the test set accuracy is slightly poor. It is worse when neither of them is used. So, both hyperparameters are set to true in the next set of experiments.

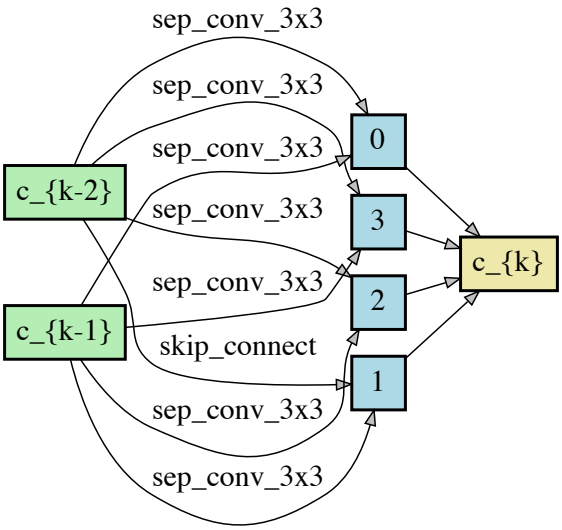
Next, to check the relative rank of all genotypes, four new networks are created using each genotype and trained from beginning. For each genotype, Table 4.3 lists the validation set accuracy during search and the test set accuracy from this experiment. We find that, although the validation accuracy of Genotype 4 is almost 2 points greater than that of Genotype 1, the test accuracy is actually slightly poor. At the same time, Genotype 2 performs best during both search and evaluation. So, due to a lack of patterns, a thorough investigation involving multiple search and validation experiments is deemed appropriate to check the correlation of validation and test set accuracy during search and evaluation, respectively. This will be considered in future work.

Table 4.3: Performance of (a) Genotypes during search (column 2) and (b) their respective networks during evaluation using cutout: True & auxiliary: True (column 3)

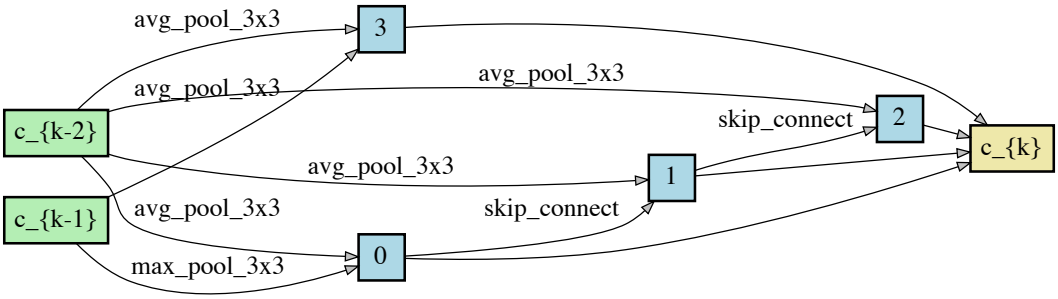
Genotype	Validation Accuracy (during search)	Test Accuracy (during evaluation)
1	86.15%	97.28%
2	89.22%	97.35%
3	87.93%	96.94%
4	88.50%	96.99%

We find that Genotype 2, with highest validation accuracy of 89.22% and test set accuracy of 97.35%, gives the best results in both cases. In fact, this seems better than the genotype found in the original study, using first order gradient approximation, which achieved 97.00% $\pm$ 0.14 test set accuracy. While on a single run, the performance of Genotype 2 seems better than the performance of the best cell from the study, the difference isn't large enough to be meaningful. The cell in the original study was found using second order gradient approximation and it gets 97.24% $\pm$ 0.09 accuracy on the test set. But, it is not reasonable to make such claims about comparison without checking the variability.

Based on its superior performance in comparison to other genotypes, Genotype 2 is selected for the experiment to check the transferability to the CheXpert dataset. In the remainder of the text,



(a) Normal Cell



(b) Reduction Cell.

Figure 4.1: Cells diagrams for Cifar Genotype (Genotype 2)

Genotype 2 is referred to as Cifar Genotype. The normal and reduction cell of Cifar Genotype are visualized in Figure 4.1. In comparison to the cells discovered in the original study, the normal cell discovered here is only slightly different. All intermediate nodes prefer inputs directly from the input nodes, while the one intermediate node in the original normal cell prefers input from a different intermediate node. For reduction cells, however, the differences in the structure are noticeable. The cell from this study prefers average pooling operation while the cell from the

original study prefers max pooling. The cell diagrams for the original study can be found in Figure 3.2.

4.2 CheXpert Dataset

CheXpert [Irvin et al., 2019] is an open dataset provided by Stanford University and it contains 224,316 one-channel images of chest radiographs from 65,240 subjects. The dataset is split into downloadable training and validation sets with 223,414 and 234 images, respectively. But the test set is withheld for an ongoing competition. Each image has 4 classes labels — 0 for confirmed negative, -1 for uncertain , 1 for confirmed positive and blank for missing label — for 14 observations. A variety of experiments such as search and validation on CheXpert using DARTS, baseline using DenseNet, transferability of Cifar Genotype, etc., are discussed in this section.

4.2.1 Task Overview

Each subject may be associated with multiple studies and each study may have multiple images in frontal or lateral view. The images are grouped by subject and study. The goal is to generate average AUC value for five of the 14 observations — Atelectasis, Cardiomegaly, Consolidation, Edema, Pleural Effusion — by generating probability of the positive class for each study (not image). This is explained in Section 4.2.3. To do so, the model outputs should represent the probabilities of the positive and negative class for the five observations. Table 4.4 shows the distribution of the class labels for the five observations. We can see that the dataset is highly skewed towards the negative class.

The presence of missing and uncertain labels increases the complexity of the task. In all experiments, the missing labels are treated as negative class. To tackle uncertain labels, a number of strategies are tried and recommended in the paper. Three of the five strategies are used in the experiments in this chapter and they are briefly discussed next. Please refer to the [Irvin et al., 2019] to learn more about them.

Table 4.4: Distribution of Positive, Uncertain and Negative labels for the five tasks

Task	Positive	Uncertain	Negative
Atelectasis	29,333	29,377	128,931
Cardiomegaly	23,002	6,597	158,042
Consolidation	12,730	23,976	150,935
Edema	48,905	11,571	127,165
Pleural Effusion	75,696	9,419	102,526

Uncertain Label Strategies

- **U-Ones**: This strategy simply replaces the uncertain class labels as positive class labels before network training. The authors found this strategy to work well for Atelectasis and Edema tasks.

- **U-MultiClass**: treats uncertain labels as independent class during the network training, i.e., the networks are trained on three classes for the image classification task. During evaluation, the three class output is converted to two class probabilities by applying softmax on the output for the positive and negative classes.

- **U-SelfTrained**: The final strategy is to, first, train a label predictor on just the positive and negative labels, while keeping all the samples with uncertain labels withheld. Then, this model can be utilized to classify the withheld images with uncertain labels to either positive or negative class. The central intuition behind this strategy is that it doesn't assume anything about the uncertain labels and is expected to capture any hidden information. Because this technique can reduce the number of uncertain labels for images which get classified with high certainty and thereby improving the probability of better results on the main task, it is tested in the next section.

Dataset Preprocessing and Augmentation

The following set of operations are applied dynamically before each batch is processed by the networks. For training images, the input images are resized to 390 $\times$ 390 pixels. Then, a random

cropping operation of size 320 $\times$ 320 pixels is applied to the resized images followed by random flipping. The final operation in the data preprocessing chain is mean centering and normalization which is applied on the channel.

For test set images, the resizing operation is similar as before, i.e., 390 $\times$ 390 pixels. The next operation is center cropping of size 320 $\times$ 320 pixels instead of random cropping. Finally, the images are mean centered and normalized as before.

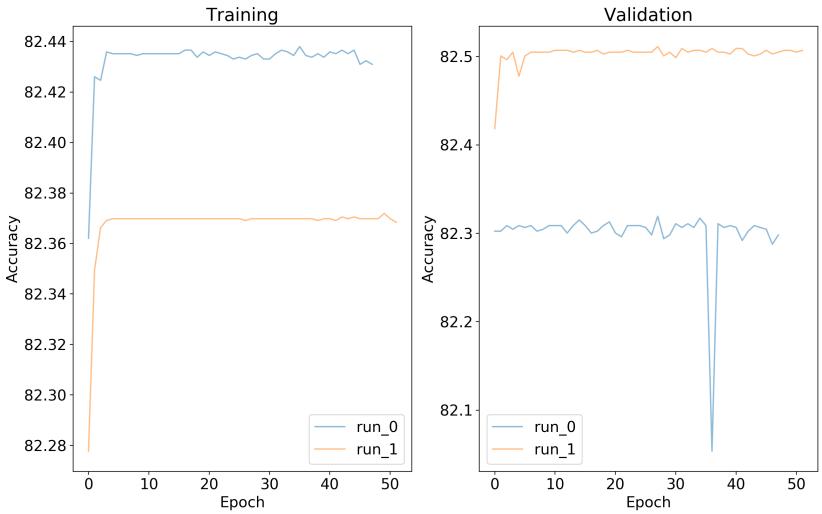
4.2.2 Uncertain Label Prediction

The experiments in this section train label predictors to test *U-SelfTrained* strategy described in Section 4.2.1. The experiment was conducted for Atelectasis and Cardiomegaly tasks. For each task, the training set images for only positive and negative labels of respective tasks are used. The original training set is split into two partitions and 75% of the images are randomly selected to be the training set and remaining 25% are used as validation set.

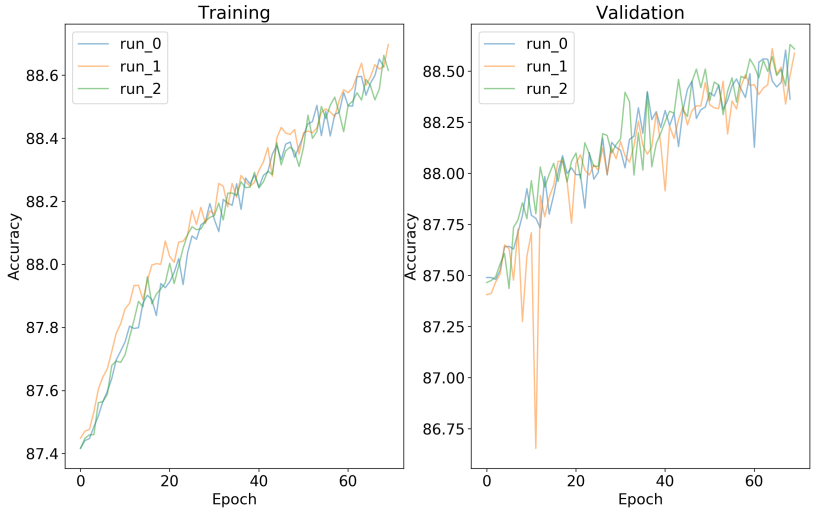
A number of convolutional networks were tested in the original study [Irvin et al., 2019] and the authors achieved highest average AUC for the 5 tasks using DenseNet121. So, because DenseNet121 is known to work well with the dataset, it is used for these and the baseline experiments. For each task, Atelectasis and Cardiomegaly, separate networks are trained on binary classification task on positive and negative class. The experiment is repeated two and three times for Atelectasis and Cardiomegaly, respectively, with different random seeds. Networks are checkpointed after every epoch.

Training images are randomly sampled with a batch size of 32 . The Adam optimizer is used with $\beta = (0.9, 0.999)$. The training starts with an initial learning rate of 10^{-4} and is annealed down to zero using cosine schedule. Weight decay of 3×10^{-4} is used for regularization. The results are presented in the Figure 4.2

For Atelectasis, we can observe that the validation accuracy remains constant throughout both runs. For Cardiomegaly, there is about 1% increase in validation accuracy after 70 epochs. Al-



(a) Atelectasis



(b) Cardiomegaly

Figure 4.2: Training Plots for Uncertain Label Predictors for (a) Atelectasis and (b) Cardiomegaly tasks. Validation accuracy for Atelectasis remains constant and there is only a small improvement in Cardiomegaly.

though the experiments were scheduled to run for 100 epochs, judging by the poor performance of the models the training was halted midway.

I think that there are a couple of reasons behind the poor performance. First, about 81% images, for Atelectasis, are labeled negative. This is very close to the validation accuracy of about 82.4%. So it seems that, due to the skew in the dataset, the network has learned to memorize and predict

the negative class. Another reason, which was discovered later, could be bad hyperparameters, especially the learning rate schedule. Cosine annealing rate decays the learning late too slowly for the dataset. The hyperparameter tuning is discussed in Section 4.2.6.

Because of the poor performance of the label predictor, *U-MultiLabel* strategy is deemed unfit to replace uncertain labels with the predicted positive or negative labels. For this reason, experiments planned for other tasks were also canceled. So, as recommended by the original work, **U-Ones** strategy is used for Atelectasis and Edema while **U-MultiClass** strategy is used for Cardiomegaly and Pleural Effusion for all subsequent experiments on the five tasks. Even though U-SelfTrained is the recommended strategy for Consolidation, due to the failure of this experiment U-Multiclass is used for Consolidation as well.

4.2.3 Baseline using DenseNet121

Five separate DenseNet121 models are trained for 30 epochs from scratch on each of the five tasks on image classification task. The training and validation dataset are augmented and preprocessed as described in Section 4.2.1. Training set is randomly sampled with a batch size of 32, the largest batch that can fit into the GPU [6].

Weights of the network are updated using Adam optimizer with $\beta = (0.9, 0.999)$. The training starts with an initial learning rate of 10^{-4} and is decayed by a factor of 0.83 after each epoch. Cutout regularization is used with cutout length 50. In addition, weight decay 3×10^{-4} is also used to regularize the model.

At the end of each epoch, AUC values are calculated for each task. AUC value is computed for binary classifiers and probabilities of the positive class are required to do so. For Cardiomegaly, Consolidation and Pleural Effusion, which use **U-MultiClass** strategy, one additional step is required to compute the AUC value. The three class probabilities output by the network for each sample are converted to a two class probability by applying softmax to the output of the positive and negative classes.

[6] Titan X GPUs are used for the experiment.

Table 4.5: Baseline experiment: AUC values for each task using DenseNet

Task	AUC
Atelectasis	0.8117
Cardiomegaly	0.8213
Consolidation	0.8816
Edema	0.9263
Pleural Effusion	0.9352
Mean	0.8752

The tasks requires the model to generate AUC values for each study rather than for each sample. This is achieved by, first, grouping the AUC values for all samples in each unique study. For example, if a study contains three images, then the AUC values, for all five tasks, for these three images are collected together, a tensor of 3×5 size. Then, mean value is computed for each task and this is considered as the final AUC value for that task for the current study.

Network check-pointing at the end of each epoch allows reuse of the weights that produce the best AUC values. The model generates 0.8752 mean AUC value. AUC values for each task are listed in the Table 4.5 and these serve as the baseline for experiments using DARTS architectures. The same experiment, but using networks discovered from DARTS, are discussed in Section 4.2.6. A detailed comparison of all evaluation experiments on CheXpert, including DARTS models, is described in Section 4.2.7.

4.2.4 Structural Changes in DARTS Models

Due to resource constraints, a single DARTS network is trained on all five tasks simultaneously instead of training five separate models. This decision reduced the training time to $1/5^{\text{th}}$ the time it takes to train five models and thus allowed a more thorough exploration of the hyperparameter space and evaluating various DARTS genotypes. Since the dataset contains images that are labeled positive for multiple tasks a new challenge is encountered.

The softmax output with cross-entropy loss function maximizes the output of the positive class while decreasing the outputs of the remaining classes. This is fine when the classification problem involves only a single task. But, cross-entropy loss with a single softmax layer isn't designed to be compatible with five tasks with multiple positive class labels. To overcome this challenge the single output layer is replaced with five layers for each task. Each task has a corresponding output layer and by working within the task the output layers don't interfere with each other. There may be interference through shared hidden layers though.

Atelectasis and Edema use *U-ones* strategy where all uncertain labels are replaced by positive labels. So, because there are only two labels, the output layer has two components. For Cardiomegaly, Consolidation and Pleural Effusion *U-MultiClass* is used, i.e., there are three classes. This means that the output layers for these tasks also have three components each. To generate the output of the network the output of all five layers are concatenated to create a list of 13 values. If an auxiliary output layer is used, it has the exact same output layer structure as just described. The changes are presented in Figure 4.3.

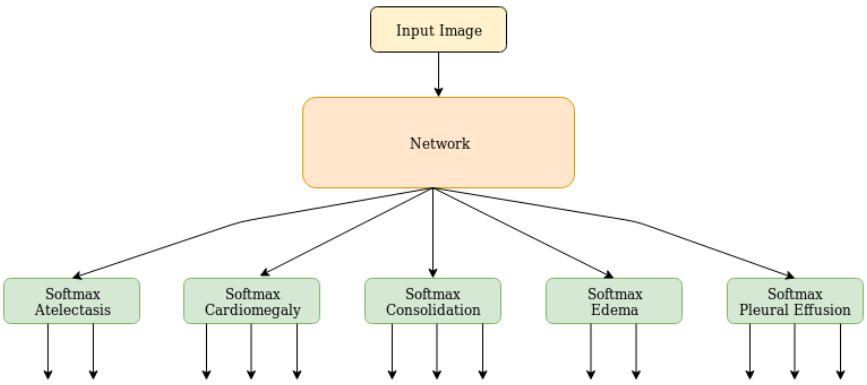


Figure 4.3: Output Layer for DARTS models on CheXpert dataset.

4.2.5 Cell Search using DARTS

This section applies DARTS to discover a new genotype — normal and reduction cells — on CheXpert dataset. A network is created with eight over-parameterized cells with mixed operations,

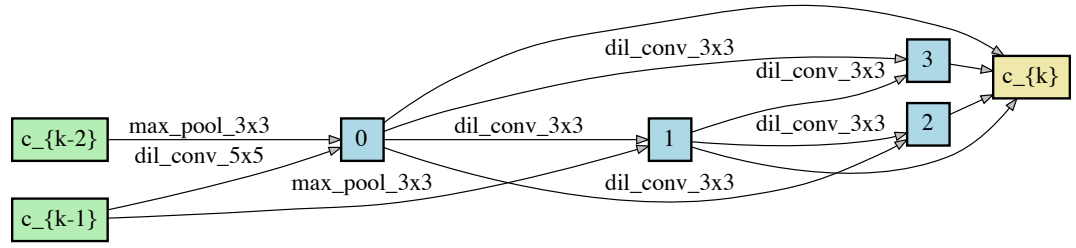
parameterized by α's and initialized to same small random value, and $B = 7$ nodes. α_{normal} and $\alpha_{reduction}$ are shared between the normal and reductions cells, respectively. Depthwise concatenation is applied to the outputs of all intermediate nodes to attain the output of the cell. Cells in this search also use the same set of operations as described in Section 4.1.2 and with same behavior.

The network is initialized with 16 filters. Cell placement is similar to the networks created in CIFAR-10 search with reduction cells placed at 1/3rd and 2/3rd depths, i.e., the 3rd and 6th cells are reduction cells, while the rest are normal cells. However, this architecture uses five output layers for CheXpert dataset instead of a single output layer (Figure 4.3). Cross entropy losses are computed separately for each task and summed up to calculate total loss.

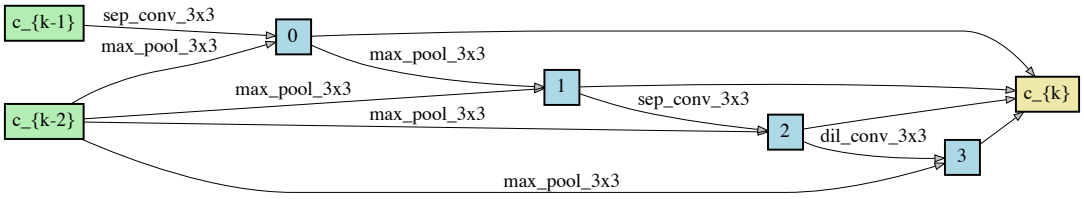
The training set is split into two equal non-overlapping random partitions to serve as training and validation partitions for the search experiment. With a batch size[7] of 25, the augmented and preprocessed training set images are used to train the network for 50 epochs. While the data augmentation operations are the same as described in Section 4.2.1, the rescaling size is changed to 64×64 from 390×390 pixels and the cropping size is changed to 56×56 from 320×320 pixels. This change is forced because larger images cause out-of-memory exception on GPU because of mixed operations during search. Please note that this change in rescaling and cropping size is for the search experiment only and not for evaluation experiments.

For each training batch, first, the loss computed on a new validation set batch is used to update the architecture parameters α. For this purpose, Adam optimizer is used with initial learning rate 3×10^{-4}, β = $(0.9, 0.999)$ and weight decay 10^{-3}. Then, a different Adam optimizer is used to optimize model parameters w with same β's as before and weight decay of 3×10^{-4}. This update is computed on loss on the current training batch. Cosine annealing learning rate schedule is initialized with learning rate of 0.025 and it decays the learning rate to 10^{-3} over the course of 50 epochs. Finally, the best setting for cutout regularization and architecture gradient approximation found during search on CIFAR-10 (Section 4.1.2) are used here. They are set to *off* and first order approximation, respectively.

[7]Set to largest value that fits in NVIDIA Titan X GPU.



(a) Normal Cell



(b) Reduction Cell.

Figure 4.4: Cells for genotype discovered on CheXpert

The search took about 123 GPU hours to finish. Figure 4.4 displays the cell structures for normal and reduction cells for this search. This genotype is named CheXpert Genotype for future references. There are striking differences in the cells structures when CheXpert Genotype is compared to the cells of Cifar Genotype. The normal cell of Cifar Genotype consists only of separable convolution and skip-connect operations, whereas this normal cell contains max pooling and dilated convolution operations. The differences in the reduction cell are even more striking. Recall that the reduction cells from the original study and Cifar Genotype consist mostly of max pooling and average pooling operations, respectively. But, the reduction cell for CheXpert Genotype contains max pooling, dilated and separable convolutions. The cell diagrams for the original study and Cifar Genotype can be found in Figure 3.2 and Figure 4.1, respectively. CheXpert Genotype is evaluated in the next section on the image classification task.

4.2.6 Genotype Evaluation

This section describes the architectural details of networks created for the CheXpert dataset, hyperparameter tuning experiments and evaluation of networks created using CheXpert and Cifar Genotypes. The aim of the experiments is to achieve maximum average AUC value for the five tasks.

A new architecture, with 18 layers, is created for each evaluation experiment. Following original work for ImageNet experiments, which also has a large input image size, downsampling operations (RELU-Conv-BatchNorm), with stride set to two, are applied to the input image three times. This scales the input image down to $1/8^{th}$ the original size. Then, the cells are stacked repeatedly till the desired depth is achieved. All the cells except 7^{th} and 14^{th} are normal cells, while these two are reduction cells. Auxiliary classifier layer is placed just after the second reduction cell. A 2-dimensional average pooling layer follows the cells which is then connected to the output layer. Note that both normal and auxiliary classifier layers have five separate outputs for each task.

The network is initialized with 64 filters at the first cell and they double at every reduction cells. At the same time activation maps' sizes are halved. To calculate AUC on the validation set, network outputs are processed in same fashion as in Section 4.2.3. The Stochastic Gradient Descent optimizer [Robbins and Monro, 1951] is employed to update the network weights with momentum set to 0.9. With a batch size of 32, all 223,414 augmented training set images are randomly sampled for network training. At the end of each epoch, AUC values are calculated for each unique study and subject combination in the 234 validation set samples (Section 4.2.3). This is done for all five tasks and the mean AUC is computed and recorded.

While the next section discusses hyperparameter tuning in detail, some preliminary experiments revealed that 320×320 input image size works better than 224×224. Also, average validation AUC for networks trained with Cutout regularization and Auxiliary layer was found to be better than the models that were trained without them. This observation is consistent with genotype evaluation on CIFAR-10 dataset, in Section 4.2.2, which revealed that the network performed best when both cutout regularization and auxiliary classifier layer were utilized.

Hyperparameter Tuning

When the search experiments were being implemented and executed, the hyperparameter tuning experiments were started with networks created using Cifar Genotype for better resource utilization. To control for network architectures, all hyperparameter tuning experiments were conducted on Cifar Genotype. Once the best value for a hyperparameter has been identified, it is used for all subsequent experiments for other hyperparameters. Please note that the experiments were conducted once and, for some hyperparameter values, the difference in validation set accuracy may be too small to be significant. The value that results in highest performance is picked to be the optimal one. These are discussed below and their optimized values are summarized at the end of the section.

1. Learning Rate, Schedule And Epochs: Learning rate and schedule were the most important hyperparameters for CheXpert dataset. It was discovered that the first setting that the experiments were started with, cosine annealing schedule with initial learning rate 10^{-4} for 70 epochs, decayed the learning rate too slowly and the networks were overtraining. On the other hand, decaying learning rate by constant multiplicative factor γ after each epoch improved the mean validation AUC. This, as mentioned in Section 4.2.2, could be one of the reasons why uncertain label predictors failed to improve their validation set accuracy.

The training and validation plots also indicated that the first 10-15 epochs are very important during training and that the learning rate and weight decay play a very significant role. Experiments performed for different values for γ are presented in the Table 4.6. The number of epochs were reduced to 30 because with learning becoming too small the mean AUC approaches its asymptotic value.

Table 4.6: Optimization results for initial learning rate η_0 and decay factor γ with step size = 1.

η_0	10^{-4}	10^{-4}	$\mathbf{10^{-4}}$	10^{-4}	10^{-4}	10^{-5}	10^{-5}
γ	0.77	0.79	**0.83**	0.85	0.87	0.89	0.91
Mean AUC	0.8777	0.8807	**0.8808**	0.8754	0.8793	0.8738	0.8701

2. Auxiliary Weight controls the contribution of the loss at auxiliary classifier layer to the total loss, i.e., $Loss_{total} = Loss_{primary} + a_w \times Loss_{auxiliary}$. The results are listed in Table 4.7.

Table 4.7: Results for optimization experiments for Auxiliary Weight a_w.

a_w	0.3	**0.4**	0.5	0.6	0.7
Mean AUC	0.8718	**0.8808**	0.8790	0.8738	0.8735

3. DropPath Probability Regularization: When used, the paths in the network are randomly dropped with a probability schedule that increases linearly from zero to p at the beginning of each epoch during training. Please refer to Table 4.8 for results.

Table 4.8: Results for optimization experiments for DropPath Probability.

Probability p	**0**	0.1	0.2	0.3	0.4
Mean AUC	**0.8865**	0.8801	0.8791	0.8804	0.8831

4. Weight Decay (λ) is a multiplicative factor applied to the network weights of a model while computing the loss. It acts as a penalty on weights and prevents them from getting too big. The results are presented in Table 4.9

Table 4.9: Results for optimization experiments for Weight Decay

λ	10^{-2}	10^{-3}	$\mathbf{10^{-4}}$	10^{-5}	10^{-6}
Mean AUC	0.8043	0.8672	**0.8808**	0.8734	0.8724

5. Cutout Length: Cutout regularization is a data augmentation technique in which a square patch of input image, placed randomly, is masked to zero. Cutout length sets the size of the masked portion in the image. The results are listed in Table 4.10.

Table 4.10: Results for optimization experiments for Cutout Length.

Cutout Length	40	**50**	60	70	80
Mean AUC	0.8749	**0.8865**	0.8853	0.8832	0.8818

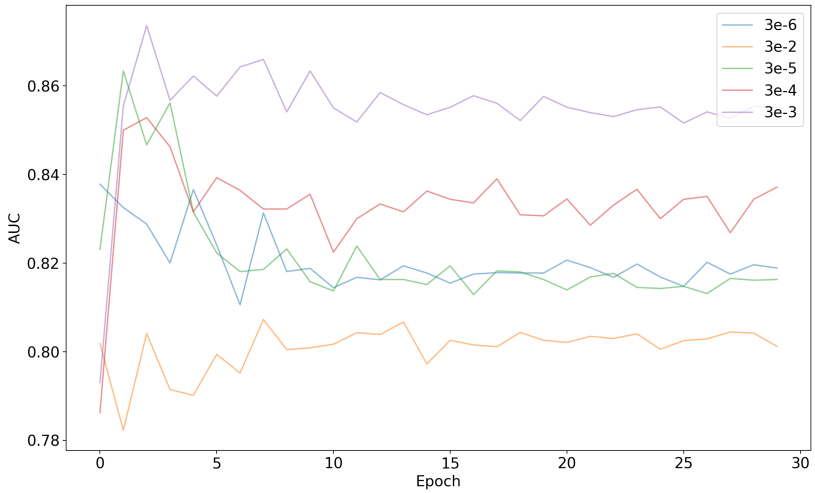
Figure 4.5 visualizes the validation AUCs for Atelectasis and Edema for weight decay optimization experiment. It is interesting to note that while $\lambda = 10^{-3}$ performed best for Atelectasis, it is no where near as good on Edema. Same is true for $\lambda = 10^{-6}$ which performs best on Edema but does poorly on Atelectasis. This trend is also observed for other hyperparameters. This suggests that five separate models for each task with respectively tuned hyperparameters might be a better approach to achieve the highest mean AUC. This experiment will be considered in future work.

Optimized values for the hyperparameters:

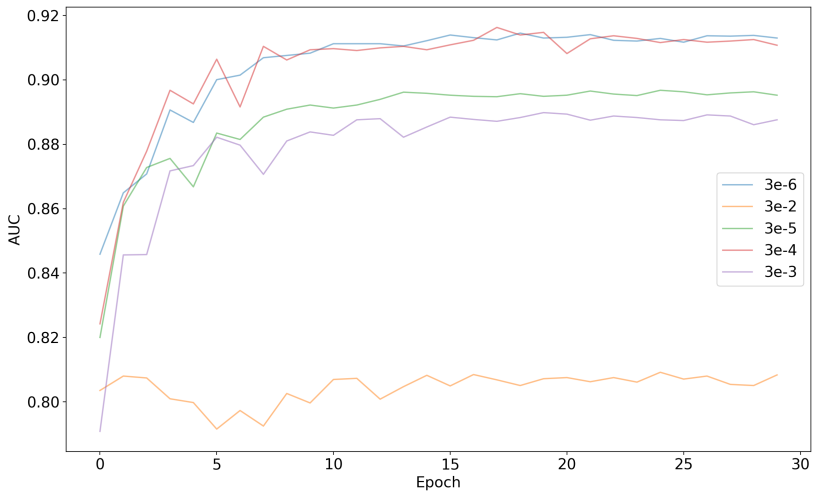
- Learning Rate=10^{-4}, Schedule=StepLR with γ=0.83 and step size=1 and #Epochs=30
- Auxiliary Weight = 0.4
- DropPath Probability = 0
- Weight Decay = 10^{-4}
- Cutout Length = 50

Evaluation of CheXpert Genotype

CheXpert Genotype's (Section 4.2.5) performance, in terms of mean AUC for five tasks computed on the validation set, is tested by creating a new network using the genotype and is trained three times from scratch with random seeds. The dataset preprocessing, network structural ar-



(a) Atelectasis



(b) Edema

Figure 4.5: Validation AUCs for (a) Atelectasis and (b) Edema tasks for weight decay tuning experiment.

rangements and augmentations and hyperparameters are used as described in Sections 4.2.1 and 4.2.6. The genotype achieves 0.8764 mean AUC with 0.003 standard deviation over three runs.

Transferability of Cifar Genotype

One feature and advantage of DARTS's search space is that the cells learned on one dataset are transferable to other datasets. To test this, a network, created using Cifar Genotype, is trained three

times starting with random weights and random seed each time. This network produces 0.8832 $\pm$ 0.002 mean AUC for the five tasks over three runs. The networks created using Cifar and CheXpert Genotype are compared in the next section.

4.2.7 Comparison with Baseline

This section presents a comparison of the models created using CheXpert and Cifar Genotypes with respect to the baseline. Please note that both DARTS models are controlled for data preprocessing and augmentation, network assembly and hyperparameters. The models of each genotype are trained from scratch with random initial weights and random seed. For the baseline DenseNet121 model, however, there are a couple of differences. The baseline model does not use the auxiliary classifier layer and was trained only once.

Table 4.11: CheXpert: Comparison of genotypes vs DenseNet121

	DenseNet121 (Baseline)	Cifar Genotype	CheXpert Genotype
Atelectasis	0.8117	0.8270 $\pm$ 0.007	0.8263 $\pm$ 0.008
Cardiomegaly	0.8213	0.8297 $\pm$ 0.011	0.8012 $\pm$ 0.007
Consolidation	0.8816	0.9134 $\pm$ 0.001	0.9143 $\pm$ 0.003
Edema	0.9263	0.9144 $\pm$ 0.007	0.9154 $\pm$ 0.004
Pleural Effusion	0.9352	0.9317 $\pm$ 0.002	0.9247 $\pm$ 0.003
Mean AUC	0.8752	**0.8832 $\pm$ 0.002**	0.8764 $\pm$ 0.003

The validation set AUC for the five tasks are presented in Table 4.11. Both DARTS models perform better on Atelectasis and Consolidation in comparison to DenseNet. On the other hand, DenseNet generates better AUCs for Edema and Pleural Effusion. On Cardiomegaly, DenseNet has similar performance in comparison to Cifar Genotype. When both genotypes are compared against each other, the Cifar Genotype wins out over all but only because of its better performance on Cardiomegaly and Pleural Effusion. The AUC on the other three tasks are roughly the same.

Please note that a t-test, which would require 15-25 repetition on each model, was not performed due to time and resource constraints.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The experiments on CIFAR-10 dataset successfully reproduced the results of the original study. Search time on the dataset using DARTS ranged from 10.5 to 72 GPU hours depending on which gradient approximation method was used. The best discovered genotype (Cifar Genotype) obtained 97.35% test set accuracy as compared to 97.24 $\pm$ 0.09 % from the original study . On CheXpert, the search took about 123 GPU hours to discover CheXpert Genotype which achieved 0.8764 $\pm$ 0.003 mean AUC on the validation set. In comparison, networks created using Cifar Genotype got 0.8832 $\pm$ 0.002 which successfully demonstrates that genotypes learned on one dataset are transferable to other datasets. Both genotypes perform similar, if not better, than the baseline DenseNet121 models.

While there are numerous benefits of DARTS algorithm, it is not free from certain caveats. These are discussed below.

5.1.1 Advantages of DARTS

- **Reduced Search Cost:** The search on CIFAR-10 took 10.5 and 72 GPU hours[8] using first and Second order gradient approximation, respectively. The original work discovered cells in 1.5 and 4 GPU days[9]. This difference in search cost can be attributed to larger memory size and faster computation processing of Tesla V100 which allowed the batch size to be twice as big for the experiments. This is a more than three order of magnitude reduction in search cost in comparison to original NAS [Zoph and Le, 2016].

[8] on NVIDIA Tesla V100 GPUs.

[9] on NVIDIA GTX 1080Ti GPUs.

- **Performance:** In addition to results described in this document, the original paper achieved $2.76 \pm 0.09\%$ test error on CIFAR-10 with 3.3M parameters which is better than most hand designed architectures. They also attained 55.7 test perplexity, then state-of-the-art, on Penn Treebank (PTB).

- **Optimization using back propagation**: The search space is made to be continuous by relaxing the choice of operations. This allows the architecture parameters to be optimized using gradient descent. This avoids the need for controllers to generate architecture encoding or hypernetworks to generate weights or predict validation set performance for child networks. A major drawback of using controllers and hypernetworks is that they are network architectures themselves and require optimal architecture design to search good architectures on new datasets from different domains. The problem of finding architectures thus becomes recursively unsolvable using algorithms that rely on controllers and hypernetworks.

- **Flexibility of Micro Search**: Cell based architectures have a neat advantage — to meet performance and resource requirements, architectures of different capacity can be created by simply varying the number of cells and filters in the network.

- **Transferability of Learned Cells**: Another advantage of cell based architectures is that the cells searched for a task on one dataset are transferable to other tasks on different datasets. The results in this work are also in line with this observation: the cells learned on CIFAR-10 produced best results on CheXpert.

- **Less Reliance on Proxy Metrics**: Search algorithms that treat the search space to be discrete often reduce the number of epochs to 20-25 while training sampled architectures to reduce the search execution time. DARTS however allows the search networks enough time during training that the validation accuracy starts to asymptote. But, DARTS still relies on a few proxy metrics. They are discussed next.

5.1.2 Drawbacks of DARTS

- **Memory Explosion**: During search, all operations in the mixed operations are applied on the activation maps in each cell. This increases the memory footprint and causes GPU out-of-memory exceptions when large images are used, as discussed in Section 4.2.5. Even though 320×320 image size was discovered to work best for CheXpert, the images size had to be reduced to 56×56 to avoid running out of memory.

  The same issue also prevents the algorithm from training deeper networks during search. While the final architectures for evaluation are composed of 20 and 18 cells for CIFAR-10 and CheXpert, respectively, the networks during search contain only 12 and 8 cells. This means that DARTS's application on large datasets is likely to be sub-optimal due to reduced image sizes or training smaller networks during training. A more recent work than DARTS, Proxyless NAS [Cai et al., 2018b], resolves the memory explosion issue. They introduce binarization of the mixed operation, which means that only one operation, from the set of all candidate operations, is applied at the run time. This reduces the memory footprint by an order of magnitude and thus allows search to be executed on larger networks and datasets. DARTS, on the other hand, applies all candidate operations on each mixed operation in the network.

- **Manual Effort during Search and Training** A downside of micro search is that the network assembly for search and evaluation — downsampling operations at the beginning of the network to reduce activation map size, placement and quantity of normal and reduction cells, operations following the cells, etc., — are done manually. While the results are competitive this leads to a couple of problems. First, by allowing human biases to enter the architecture design, all possibilities for optimizing the performance are not searched for by the algorithm. Second, for a novice deep learning practitioner, the process might seem quite complex and hard to understand. This might discourage them from using such algorithms.

- For architecture search, the algorithm consists of a lot of hyperparameters such as the number of nodes in a cell (N), the number of operations retained, for each intermediate node in a cell, to select a discrete cell architecture (K), the number of cells in the network (depth) and the gradient approximation method (first order or second order). The study does not provide any guide on how to tune these parameters and their effects on the results. Having a wide range of parameters that result in high variability on search performance, as described in Table 4.1, and with no guide, makes the use of the algorithm tedious.

5.1.3 Drawbacks of the Original Study

- In the original study, the Stochastic Gradient Descent optimizer is used to optimize the network parameters w and the Adam optimizer is used for updating the architecture parameters α, during search on CIFAR-10 dataset. The study does not explain why this particular setup was used and does not provide a comparative analysis on both optimizers. So, the optimizers become yet another hyperparameter during search.

- The original study does not provide the measure of correlation of the performance on the validation set during search and the performance of the searched genotype on the test set during evaluation. So, ranking the genotypes requires multiple search and evaluation runs, for each search hyperparameter, to select the best genotype with confidence. This is a lot of computation.

5.2 Future Work

While DARTs offers numerous advantages as detailed in the previous section, most architecture search algorithms, including DARTS, lag behind in some key areas. First, the search spaces in all works described in the Chapter 2 are limited to discovering networks that are based on ideas developed manually, such as tree-based or fractal based structures. They are not capable of searching for wholly new architecture styles.

Second, the hyperparameters are not searched. Hyperparameter tuning is a time and resource consuming process. The tuning experiments in this work took thrice more computations resources

than the search and evaluation combined. While there are packages available for hyperparameter tuning, crucial hyperparameters, such as learning rate and weight decay which can heavily influence network performance, should be a part of the search algorithm. New algorithms should be developed in the future to resolve these problems.

In terms of experiments, to study the correlation of validation accuracy during search and test accuracy during evaluation for a genotype and its final architecture, respectively, the experiments in Section 4.1.2 can be repeated multiple times. Second, because hyperparameter tuning experiments on CheXpert revealed that each task has its own optimal hyperparameters, five separate models may be trained for Cifar and CheXpert Genotype instead of using a single model (Section 4.2.6). This should improve the mean AUC on the validation set. Third, the uncertain label prediction experiments, in Section 4.2.2, can be retried with the optimal hyperparameters for the five tasks to check if the results improve. Fourth, ProxylessNAS [Cai et al., 2018b] should allow use of the full image size and a larger network during search, due to its memory efficiency. So, the algorithm could be deployed on CheXpert dataset to check its performance relative to DARTS. Finally, because DARTS is capable of designing RNNs, it may be applied and studied on text or time-series datasets.

Bibliography

[Brock et al., 2017] Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: One-Shot Model Architecture Search through HyperNetworks. *arXiv e-prints*, page arXiv:1708.05344.

[Browne et al., 2012] Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Liebana, D. P., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1–43.

[Cai et al., 2017] Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. (2017). Efficient Architecture Search by Network Transformation. *arXiv e-prints*, page arXiv:1707.04873.

[Cai et al., 2018a] Cai, H., Yang, J., Zhang, W., Han, S., and Yu, Y. (2018a). Path-Level Network Transformation for Efficient Architecture Search. *arXiv e-prints*, page arXiv:1806.02639.

[Cai et al., 2018b] Cai, H., Zhu, L., and Han, S. (2018b). ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. *arXiv e-prints*, page arXiv:1812.00332.

[Chen et al., 2015] Chen, T., Goodfellow, I., and Shlens, J. (2015). Net2net: Accelerating learning via knowledge transfer. *arXiv e-prints*, page arXiv:1511.05641.

[Chollet, 2017] Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807.

[Colson et al., 2007] Colson, B., Marcotte, P., and Savard, G. (2007). An overview of bilevel optimization. *Annals of Operations Research*, 153:235–256.

[DeVries and Taylor, 2017] DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *arXiv e-prints*, page arXiv:1708.04552.

[Floreano et al., 2008] Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62.

[Han et al., 2016] Han, D., Kim, J., and Kim, J. (2016). Deep pyramidal residual networks. *arXiv e-prints*, page arXiv:1610.02915.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *arXiv e-prints*, page arXiv:1512.03385.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Identity mappings in deep residual networks. *arXiv e-prints*, page arXiv:1603.05027.

[Holland, 1992] Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

[Huang et al., 2016] Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2016). Densely connected convolutional networks. *arXiv e-prints*, page arXiv:1608.06993.

[Hutter et al., 2011] Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2011). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg. Springer-Verlag.

[Irvin et al., 2019] Irvin, J., Rajpurkar, P., Ko, M., Yu, Y., Ciurea-Ilcus, S., Chute, C., Marklund, H., Haghgoo, B., Ball, R., Shpanskaya, K., Seekins, J., Mong, D. A., Halabi, S. S., Sandberg, J. K., Jones, R., Larson, D. B., Langlotz, C. P., Patel, B. N., Lungren, M. P., and Ng, A. Y. (2019). Chexpert: A large chest radiograph dataset with uncertainty labels and expert comparison.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv e-prints*, page arXiv:1412.6980.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA. Curran Associates Inc.

[Larsson et al., 2016] Larsson, G., Maire, M., and Shakhnarovich, G. (2016). Fractalnet: Ultra-deep neural networks without residuals. *arXiv e-prints*, page arXiv:1605.07648.

[Liu et al., 2017] Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2017). Progressive neural architecture search. *arXiv e-prints*, page arXiv:1712.00559.

[Liu et al., 2018] Liu, H., Simonyan, K., and Yang, Y. (2018). Darts: Differentiable architecture search. *arXiv e-prints*, page arXiv:1806.09055.

[Loshchilov and Hutter, 2016] Loshchilov, I. and Hutter, F. (2016). Sgdr: Stochastic gradient descent with warm restarts. *arXiv e-prints*, page arXiv:1608.03983.

[Luong et al., 2015] Luong, M.-T., Le, Q. V., Sutskever, I., Vinyals, O., and Kaiser, L. (2015). Multi-task sequence to sequence learning. *arXiv e-prints*, page arXiv:1511.06114.

[Miikkulainen et al., 2017] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Hodjat, B. (2017). Evolving deep neural networks. *arXiv e-prints*, page arXiv:1703.00548.

[Miller et al., 1989] Miller, G. F., Todd, P. M., and Hegde, S. U. (1989). Designing neural networks using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 379–384, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Negrinho and Gordon, 2017] Negrinho, R. and Gordon, G. (2017). Deeparchitect: Automatically designing and training deep architectures. *arXiv e-prints*, page arXiv:1704.08792.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[Pham et al., 2018] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *arXiv e-prints*, page arXiv:1802.03268.

[Razavian et al., 2014] Razavian, A. S., Azizpour, H., Sullivan, J., and Carlsson, S. (2014). Cnn features off-the-shelf: an astounding baseline for recognition. *arXiv e-prints*, page arXiv:1403.6382.

[Real et al., 2018] Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized evolution for image classifier architecture search. *arXiv e-prints*, page arXiv:1802.01548.

[Robbins and Monro, 1951] Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *ArXiv*, page arXiv:1707.06347.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv e-prints*, page arXiv:1409.1556.

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

[Szegedy et al., 2014] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *arXiv e-prints*, page arXiv:1409.4842.

[Tai et al., 2015] Tai, K. S., Socher, R., and Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual*

Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 1556–1566, Beijing, China. Association for Computational Linguistics.

[Williams, 1992] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256.

[Xie and Yuille, 2017] Xie, L. and Yuille, A. (2017). Genetic cnn. *arXiv e-prints*, page arXiv:1703.01513.

[Yao and Liu, 1997] Yao, X. and Liu, Y. (1997). A new evolutionary system for evolving artificial neural networks. *Trans. Neur. Netw.*, 8(3):694–713.

[Yu and Koltun, 2015] Yu, F. and Koltun, V. (2015). Multi-scale context aggregation by dilated convolutions. *arXiv e-prints*, page arXiv:1511.07122.

[Zhong et al., 2017] Zhong, Z., Yan, J., Wu, W., Shao, J., and Liu, C.-L. (2017). Practical block-wise neural network architecture generation. *arXiv e-prints*, page arXiv:1708.05552.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural Architecture Search with Reinforcement Learning. *arXiv e-prints*, page arXiv:1611.01578.

[Zoph et al., 2017] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *arXiv e-prints*, page arXiv:1707.07012.