OPTIMIZING NEURAL NETWORK STRUCTURES: FASTER SPEED, SMALLER SIZE, LESS TUNING

by

Zhe Li

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

Aug 2018

Thesis Supervisor: Professor Tianbao Yang

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

Zhe Li

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the Aug 2018 graduation.

Thesis Committee:

Tianbao Yang, Thesis Supervisor

Qihang Lin

Suely Oliveira

Kasturi R. Varadajan

Padmini Srinivasan

ABSTRACT

Deep neural networks have achieved tremendous success in many domains (e.g., computer vision [48, 79, 19], speech recognition [33, 14], natural language processing [14, 11], games [78, 77]), however, there are still many challenges in deep learning comunity such as how to speed up training large deep neural networks, how to compress large nerual networks for mobile/embed device without performance loss, how to automatically design the optimal network structures for a certain task, and how to further design the optimal networks with improved performance and certain model size with reduced computation cost.

To speed up training large neural networks, we propose to use multinomial sampling for dropout, i.e., sampling features or neurons according to a multinomial distribution with different probabilities for different features/neurons. To exhibit the optimal dropout probabilities, we analyze the shallow learning with multinomial dropout and establish the risk bound for stochastic optimization. By minimizing a sampling dependent factor in the risk bound, we obtain a distribution-dependent dropout with sampling probabilities dependent on the second order statistics of the data distribution. To tackle the issue of evolving distribution of neurons in deep learning, we propose an efficient adaptive dropout (named evolutional dropout) that computes the sampling probabilities on-the-fly from a mini-batch of examples.

To compress large neural network structures, we propose a simple yet powerful method for compressing the size of deep Convolutional Neural Networks (CNNs) based on parameter binarization. The striking difference from most previous work on parameter binarization/quantization lies at different treatments of 1×1 convolutions and $k \times k$ convolutions ($k > 1$), where we only binarize $k \times k$ convolutions into binary patterns. By doing this, we show that previous deep CNNs such as GoogLeNet and Inception-type Nets can be compressed dramatically with marginal drop in performance. Second, in light of the different functionalities of 1×1 (data projection/transformation) and $k \times k$ convolutions (pattern extraction), we propose a new block structure codenamed the pattern residual block that adds transformed feature maps generated by 1×1 convolutions to the pattern feature maps generated by $k \times k$ convolutions, based on which we design a small network with ~ 1 million parameters. Combining with our parameter binarization, we achieve better performance on ImageNet than using similar sized networks including recently released Google MobileNets.

To automatically design neural networks, we study how to design a genetic programming approach for optimizing the structure of a CNN for a given task under limited computational resources yet without imposing strong restrictions on the search space. To reduce the computational costs, we propose two general strategies that are observed to be helpful: (i) aggressively selecting strongest individuals for survival and reproduction, and killing weaker individuals at a very early age; (ii) increasing mutation frequency to encourage diversity and faster evolution. The combined strategy with additional optimization techniques allows us to explore a large search space but with affordable computational costs.

To further design the optimal networks with improved performance and certain model size under reduced computation cost, we propose an ecologically inspired genetic approach for neural network structure search , that includes two types of succession: primary and secondary succession as well as accelerated extinction. Specifically, we first use primary succession to rapidly evolve a community of poor initialized neural network structures into a more diverse community, followed by a secondary succession stage for fine-grained searching based on the networks from the primary succession. Acceleted extinction is applied in both stages to reduce computational cost.

PUBLIC ABSTRACT

Deep neural networks have achieved tremendous success in many domains (e.g., computer vision [48, 79, 19], speech recognition [33, 14], natural language processing [14, 11], games [78, 77]), however, there are still many challenges in deep learning comunity such as how to speed up training large deep neural networks, how to compress the large nerual networks for mobile/embed device without performance loss and how to automatically design the optimal network structures for a certain task.

To speed up training process of large neural network, we propose to use multinomial sampling for dropout, i.e., sampling features or neurons according to a multinomial distribution with different probabilities for different features/neurons. Further we propose an efficient adaptive dropout (named evolutional dropout) that computes the sampling probabilities on-the-fly from a mini-batch of examples to tackle the issue of evolving distribution of neurons in deep learning. To compress large neural network structures, we propose a simple yet powerful method for compressing the size of deep CNNs based on parameter binarization, and design a new block structure codenamed the pattern residual block that adds transformed feature maps generated by 1×1 convolutions to the pattern feature maps generated by $k \times k$ convolutions, based on which we design a small network with ~ 1 million parameters. To automatically design neural networks, we study how to design a genetic programming approach for optimizing the structure of a CNN for a given task under limited computational resources yet without imposing strong restrictions on the search space. To reduce the computational costs, we propose two general strategies that are observed to be helpful: (i) aggressively selecting strongest individuals for survival and reproduction, and killing weaker individuals at a very early age; (ii) increasing mutation frequency to encourage diversity and faster evolution. To further design the optimal networks with improved performance and certain model size with reduced computation cost, we propose an ecologically inspired genetic approach for neural network structure search, that includes two types of succession: primary and secondary succession as well as accelerated extinction.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Deep neural networks have achieved tremendous success in many domains (e.g., computer vision (image classification [48, 79, 86, 29], object detection [20, 75, 19, 72], segmentation [58, 28], video analysis [97, 100, 18], human pose estimation [88]), speech recognition [33, 14], natural language processing [14, 11], games [78, 77]). However, there are still many challenges in deep learning community such as how to speed up traning process of deep neural network, how to compress large neural network structure with similar performance for mobile/embedded application, how to automatically design neural network structure and how to online update deep neural network models among many others.

1.1 Faster Speed for Training Neural networks

Training deep neural networks is known to be time-consuming task. For example, training ResNet [29] on ImageNet [15] takes days to weeks under recent modern deep learning framekworks such as Caffe [42], tensorflow [1]. Speeding up training deep neural networks becomes crucial. By zooming in some successful neural network structures, it is usural case that those successful neural network structures [48] are stacked with convolutional layers, pooling layers, fully connected layers, dropout layers between those fully connected layers and softmax layer in the end. We speed up training deep neural networks by the improved dropout proposed in [53]. Dropout has been widely used to avoid overfitting of deep neural networks with a large number of parameters [49, 82], which usually identically and independently at random samples neurons and sets their outputs to be zeros. Extensive experiments [36] have shown that dropout can help obtain the state-of-the-art performance on a range of benchmark data sets. Recently, dropout has also been found to improve the performance of logistic regression and other single-layer models for natural language tasks such as document classification and named entity recognition [93].

Instead of identically and independently at random zeroing out features or neurons, we propose to use multinomial sampling for dropout, i.e., sampling features or neurons according to a multinomial distribution with different probabilities for different features/neurons. Intuitively, it makes more sense to use non-uniform multinomial sampling than identical and independent sampling for different features/neurons. For example, in shallow learning if input features are centered, we can drop out features with small variance more frequently or completely allowing the training to focus on more important features and consequentially enabling faster convergence. To justify the multinomial sampling for dropout and reveal the optimal sampling probabilities, we conduct a rigorous analysis on the risk bound of shallow learning by stochastic optimization with multinomial dropout, and demonstrate that a distribution-dependent dropout leads to a smaller expected risk (i.e., faster convergence and smaller generalization error).

1.2 Smaller Size Neural Networks for Mobile/Embeded Application

Due to the need in mobile/embedded applications, there is a new trend of going smaller while retaining the performance of large and deep CNNs. While Inception

Nets and ResNets have tried to reduce the model size by reducing the size of convolution kernels, using 1×1 convolutions and trimming the fully connected layers, they are still too large to meet the demanding requirement for mobile and embedded devices (e.g., FPGA). For example, ResNet-101 has 200MB and GoogLeNet has 50MB. However, FPGAs often have less than 10MB of on-chip memory and no off-chip memory or storage [40]. To further reduce the model size, various compressing techniques have been introduced to deep CNNs, including parameter quantization, binarization, sharing, pruning, hashing, Huffman coding, etc [8, 13, 26, 27, 10, 39, 12, 70, 68, 104, 54]. There also emerge few studies recently attempting to design small and compact networks, including the SqueezeNets [40] and the MobileNets [37]. Nevertheless, the performance drop of smaller networks is still a critical concern for many designs. For example, the authors of [40] have designed an extremely small network with less than 0.5MB and achieved 57.5% top-1 accuracy on ImageNet, which is considerably less than state-of-the-art results of deep CNNs (e.g., 68.65% of GoogLeNet according to our implementation).

We address this concern by proposing several new techniques in the two aforementioned directions for reducing the model size. First, we consider parameter binarization - a simple and effective method for reducing the model size. While many previous works try to quantize or binarize all weights in deep CNNs, we propose a novel treatment of 1×1 kernels and $k \times k$ kernels (e.g., $k = 3, 5$). In particular, we only binarize $k \times k$ convolutional kernels (with $k > 1$). This design is motivated by the difference between 1×1 convolutions and $k \times k$ convolutions and the communitys prior knowledge about them. Unlike $k \times k$ convolutions that explicitly extract features in a spatial manner, 1×1 convolutions serve as data projection and transformation. In this sense, 1×1 convolutions need to preserve the information as much as possible and $k \times k$ convolutions is only required to extract abstract patterns from images. In addition, many works in computer vision have used binary convolutions to extracted features from images [94, 83], while sparse projection has been reported with performance drop compared with dense projection [96]. The different treatments of 1×1 and $k \times k$ kernels also has several benefits in terms of computation: (i) 1×1 convolutions using floating points is cheaper and simpler than $k \times k$ convolutions; (ii) this splitting is very suitable for FPGAs where logic blocks can efficiently handle the binarized convolutions and DSP units can handle the 1×1 convolutions. Second, we propose a simple new design of small networks by stacking up several layers of a novel module, which is built on a new block codenamed pattern residual block. The idea of the pattern residual block is to add transformed feature maps generated by 1×1 convolutions to the pattern feature maps generated by $k \times k$ convolutions, which resembles but generalizes the skip connection in ResNets. The new pattern residual block is well suited to the design of small networks for increasing the model capacity and more importantly to the binarized pattern networks for offsetting the effect of pattern binarization. Using 5.2MB, our designed small network (termed as **SEP-Net**) achieves 65.8% top-1 accuracy, beating that of the SqueezeNet (4.8MB, 60.4%) and the MobileNet (5.2MB, 63.7%) with simlar sizes. Leveraging our pattern binarization, we reduce our model size to 4.2MB while maintaining 63.7% top-1 accuracy. By further quantizing 1×1 filters using 8 bits, we achieve 63.5% top-1 accuracy with a model size 1.3MB.

1.3 Less Tuning to Design Neural Networks

Deep Convolutional Neural Networks (CNN) have achieved tremendous success for many computer vision tasks. However, a hand-crafted network structure tailored to one task may perform poorly on another task. Therefore, it usually requires extensive amount of human efforts to design an appropriate network structure for a certain task. Recently there is a trend to automatically optimizing neural network structures from the predefined neural network structure searching space.

Optimizing the network structures involves two fundamental issues: how to define the search space of network structures; and how to design an efficient algorithm to search a good network structure in the search space. A great challenge in solving these issues lies in how to balance the trade-off between the size of search space and the computational cost of the search algorithm. Earlier works based on *neuro-evolution* for automatically discovering network structures usually impose strong restrictions on the search space of the network structures due to limited computational power and scarcity of data [84].

There emerged revived interests in optimizing network structures (especially deep convolutional neural networks) using genetic/evolutionary algorithms [17, 95, 71]. However, the dilemma of computational costs and search space trade-off has pushed these works into two ends. At one end, one has to restrict the search space by imposing strong constraints on the network structures. For example, in [95] a network is composed of a fixed number of stages and each stage is composed of a fixed number of nodes representing convolutional operations. In [17], Dufourq and Bassett restricted mutation operations to adding, deleting and replacing a randomly selected layer in a network with a predetermined maximum number of layers (e.g., 7 is used in their experiments). As a result, their evolved networks share a single path structure in contrast to a multiple-path structures as in residual networks [29] and Inception [86]. On the other end, Real *et al* [71] investigated large-scale evolution of networks operating at unprecedented scale by using a large amount of computing resources (e.g., spending over 10 days on 250 GPUs), which verifies neuro-evolution can achieve competitive performance as hand-crafted models built on many years of human experience. However, such an brute-force approach is not affordable for general users who have limited computational resources.

We focus on optimizing deep CNN structures for image classification due to the availability of existing results for comparison and its popularity in computer vision. Similar to [71], we also study evolution-based algorithms, which search CNNs in a large search space that is defined by a set of mutations. Nevertheless, the difference from previous works [17, 95, 71] is that our main focus is to tackle an important and challenging question for optimizing neural network structures, i.e., how to maximize the exploration in the search space under limited computational resources [1]. Instead of imposing strong restrictions on the search space, we propose new effective strategies to reduce the computational costs. We use an aggressive

[1]computational resources include not only the hardware but also the computing time.

method to select strong individuals for survival and reproduction. In particular, among a set of individuals (i.e., population) only a small number of fittest individuals that are sufficiently different from each other are selected for producing the next generation. This strategy avoids wasting time on training weaker individuals that may eventually be eliminated in a later stage. However, a potential issue caused by this strategy is that the diversity of population decreases, which is very important for genetic programming. To remedy this issue, we propose to (i) increase the number of possible mutations; (ii) make clones of the selected fittest individuals to undergo different mutations. Additional techniques are also investigated to speed up the search process and to shorten the training time of each individual during evolution.

1.4 Designing the Improved Neural Networks

Even though there are emerging research works [71, 95, 107, 108, 103, 3] on automatically searching neural network structures for the image recognition tasks as well as our research work briefly mentioned in 1.3 , the existing works suffer from either one of the following issues: prohibitive computational cost or unsatisfied performance compared with hand-crafted network structures. As mentioned before in [71], it costs more than 256 hours on 250 GPU for searching neural network structures, which can not be affordable by general users. In [95], the final learned network structure by their genetic approach achieves about 77% test accuracy on CIFAR-10, even though better performance as 92.9% could be obtained after fine-tuning certain parameters and modifying some structures on the discovered network. In our previous work, we firstly aim to achieve the better performance with the reduced computational cost by the proposed aggressive selection strategy in genetic approach and more mutations operation to increase diversity which is decreased by the proposed selection strategy. We reduce computational cost dramatically from more than $65,356$ GPU hours (GPUH) to few hundreds GPUH. However, our approach still suffers performance sacrifice, for example, 90.5% test accuracy compared to 94.6% test accuracy from [71] on CIFAR-10 dataset. How to automatically design neural networks with improved performance and certain model size under the limited computation cost is still one of the most challenging task remaining in deep learning community.

Along our previous research line, in this work we futher study the genetic approach to achieve the better test performance compared to [71] or competitive performance to hand-crafted network structures [29] under limited computation cost as the previous work and without the pre-designed architectures introduced by human [57]. Inspired from primary, secondary succession from ecological system [73], we enforce a poor initialized community of neural network structures to rapidly evolve to a community containing network structures with dramatically improved performance. After the first stage of primary succession, we perform the fine-grained search for better networks in a community during the secondary succession stage.

CHAPTER 2

RELATED WORK

In this chapter, we review previous works related to training deep neural network structure and dropout in Sec 2.1, compressing larger neural network structures 2.2, automatically designing neural network structures 2.3.

2.1 Training Neural Networks and Dropout

Stochastic gradient descent with back-propagation has been used a lot in optimizing deep neural networks. There exists several variants optimization algorithms for training deep neural networks such as AdaDelta [99], Adaptive Gradient (AdaGrad) [16], Adam [45], Nestrov's Accelerated Gradient [64] and RMPprop [87] and among others. However, it is notorious for its slow convergence especially for deep learning. Recently, there emerge a battery of studies trying to accelearte the optimization of deep learning [85, 66, 102, 41, 45], which tackle the problem from different perspectives. In [85], the authors show that stochastic gradient descent with momentum can be successfully utilized to train deep and recurrent neural network when using a well-designed random initialization and a particular type of slowly increasing schedule for momentum parameters. Further, they show that curvature issues in deep and recurrent network training can be circuvmented by well-tuned momentum methods. Path-stochastic gradiet descent (Path-SGD) proposed in [66] is an approximated steepest descent method with respect to a path-wise regularizer related to max-norm regularization, since the authors argue for a geometry invariant to rescaling of weights that does not affect the output of the network. In [102], the authors focus on optimizing deep learning in the parallel computing environment under communication constraints and proposed EASGD algorithm in which concurrent process (local worker) and parameter server (master) cooperate together to update the weights of deep neural network. The communication and coordination among those concurrent process is based on an elastic force linking to parameters each concurrent process compute with a center variable stored by parameter server (master). Both synchronous and asynchronous variants of EASGD are given. Among them, we notice that the developed evolutional dropout for deep learning achieves similar effect as batch normalization [41] addressing the internal covariate shift issue (i.e., evolving distributions of internal hidden units). For other training deep neural networks techniques such as layerwise pretraining [34, 6, 5], Contrasitive Divergence [32], conjugate Gradient [35, 67], the second order method BFGS [67] and Hessian-free optimization [59], interested readers are refered to the corresponding reference.

Dropout is a simple yet effective technique to prevent overfitting in training deep neural networks [82]. It has received much attention recently from researchers to study its practical and theoretical properties. Notably, [90, 4] have analyzed the dropout from a theoretical viewpoint and found that dropout is equivalent to a data-dependent regularizer. The most simple form of dropout is to multiply hidden units by i.i.d Bernoulli noise. Several recent works also found that using other types of noise works as well as Bernoulli noise (e.g., Gaussian noise), which could lead to a better approximation of the marginalized loss [92, 46]. Some works tried to optimize the hyper-parameters that define the noise level in a Bayesian framework [105, 46]. [23] used the same noise across a batch of examples in order to speed up the computation. The adaptive dropout proposed in [2] overlays a binary belief network over a neural netowrk, incurring more computational overhead to dropout because one has to train the additional binary belief network. In constrast, other studies focus on shallow learning with dropout noise [89, 31, 9]. It has been applied to improve the performance of logistic regression, support vector machine and other single-layer models for natural language tasks such as document classification, named entity recognition, etc. We proposes a new dropout with noise sampled according to distribution-dependent sampling probabilities. To the best of our knowledge, this is the first work that rigorously studies this type of dropout with theoretical analysis of the risk bound. It is demonstrated that the new dropout can improve the speed of convergence.

2.2 Compressing Neural Networks

In this section, we review some related work on designing modern network structures and techniques for reducing neural network model size. Since introduced in [55], the 1×1 convolutions have been extensively used in modern networks such as Inception Nets and ResNets, which can reduce the number of parameters comparing with large convolutional kernels. In these network designs, the 1×1 convolutions mainly serve as data projection for reducing the channels of feature maps. Inception Nets also concatenate the 1×1 convolved feature maps and the $k \times k$ convolved feature maps in the inception modules. In this paper, we will innovatively leverage the power of 1×1 convolutions to improve the performance of binarized networks.

In addition, we will utilize the data transformation capability of 1×1 convolutions to design a new residual block.

Recently, there emerge intensive studies on compressing the size of CNNs. Various techniques have been introduced to CNNs to either reduce the number of parameters or reduce the size of parameter representations. These include weight pruning [25, 27], weight binarization [39, 12, 70], weight ternarization [68, 98, 60, 52, 56], weight quantization [13, 8] and designing small and compact networks [55, 40]. There are several differences between the proposed weight binarization and previous work on weight binarization [39]. First, unlike previous work that binarize all weights, we only binarize the weights of $k \times k$ filters ($k > 1$). Second, our focus is not to reduce the computational costs of training by binarization but instead to reduce the costs for deploying the model, which is clearly different from some previous works focusing on training, e.g., BinaryConnect [12], Binarized Neural Networks [39], XNOR-Nets [70]. Our approach is to directly binarize fully trained deep CNNs and fine-tune the 1×1 filters. The benefit of this two-step approach is that it will not suffer from difficulties of training binarized networks and therefore enjoy less performance drop.

The design of small and compact networks is the focus of several recent works. Xu et al. [44] exploited the local binary convolutional operators in deep CNNs. They utilize traditional local binary operators in place of $k \times k$ ($k > 1$) convolutions. The difference from our work is that the binary convolutional filters in their work are randomly generated and fixed during training of the networks. In addition, the performance of their networks on large scale ImageNet data is not shown. The SqueezeNet explored several strategies to reduce the number of parameters including (i) replacing $k \times k$ convolutions ($k > 1$) by 1×1 convolutions; (ii) decreasing the number of input channels to 3×3 filters; and (iii) postponing the down sampling to late layers in the network [40]. They designed Fire module that consists of 1×1 convolutional layers to squeeze the size of feature maps and an expand layer that has a mix of 1x1 and 3x3 convolution filters. The MobileNets approximate the standard $k \times k$ ($k > 1$) convolutions by depth-wise convolutions and 1×1 convolutions, and also introduce two hyper-parameters to balance between latency and accuracy [37]. The small pattern networks explored in this work share some similarities to these previous small networks in that much computation burden will shift to the 1×1 convolutions but also bear some subtle differences. Most importantly, the present work has achieved the best performance on ImageNet data with the same network size among the MobileNets and SqueezeNets.

2.3 Designing Neural Networks Automatically

The widely used neural network structures such as AlexNet [48], NIN (Network In Network) [55], VGG-Net [79], Inception Network [86], and ResNet [29] are handcrafted or modified from others, even though automatically designing deep neural network structures could date back to work [74], in which they tried to aumatically find struture and weight of model. Due to this tremendous success among different areas such computer vision, speech recognition and game by exploring deep neural network, there exist abundant studies on using genetic or evolutionary algorithms for discovering neural networks structures before the re-emergence of deep learning [63, 84, 24] in 2012. These algorithms are also known as *neuro-evolution*. Most of these works are restricted to feedforward neural networks of a few layers. However, many techniques in these earlier works are also useful for optimizing large convolutional neural networks. In designing a neuro-evolution algorithm, several fundamental questions need to be answered, including (i) how to encode an individual; (ii) what are the allowed mutations; (iii) how to select individuals for reproduction; (iv) what is the fitness function. Existing works may differ from each other on how to address these questions, which are also related to a fundamental issue in neuro-evolution (and also in other meta-heuristic optimization algorithms): how to balance between the size of search space and the computational costs. In the following discussion, we will highlight how to address these fundamental questions and how to balance the trade-off.

In [95], the authors developed a genetic algorithm using a fixed-length binary string to encode the network structure. The search space consists of all networks with a fixed number of stages, where each stage is composed of a fixed number of nodes representing convolutional operations. Mutations are easily operated by randomly flipping each bit in the string representation, which correspond to adding, deleting and changing connections of nodes within each stage. By restricting the number of stages (e.g., 3) and the number of nodes in each stage, their computational cost is controlled under a manageable level. The selection of individuals for reproduction is done by a Russian roulette process, which selects individuals based on a non-uniform distribution whose probabilities are proportional to the fitness of the individuals, i.e., individuals with higher fitness score will be selected with higher probability.

The focus of [71] is to scale up neuro-evolution to take advantage of the tremendous computational resources Google LLC has. A total of 250 GPUs are used in their experiments. They used a graph to encode an individual, and defined seven mutations to change the structure of networks [1], and used a standard binary tournament selection method [21] for selecting individuals for reproduction.

The fitness function of the above works purely depends on the performance of individual models on a validation data, which are trained by back-propagation. The evolutionary techniques used in [17] is similar to that in [71] except that their mutations are restricted to adding, deleting and replacing one of six predefined layers, which include two-dimension convolution, one-dimension convolution, fully connected, dropout, one-, and two-dimension max pooling. As a result, their algorithm cannot discover multiple-path networks, which are prevalent in modern deep learning community. A common feature of these works is that they use a combined strategy that lets the structure evolve but optimizes the weights of each individual by back-propagation, which is also adopted in the present work.

Another fundamental issue in genetic/evolutionary algorithm is the diversity of the population. A traditional approach for encouraging diversity of population is fitness sharing, where the fitness of each individual is scaled based on its proximity to others. It means that originally good solutions in densely populated regions will be given a lower fitness value than comparably good solutions in sparsely populated

[1]they also used several other mutations that do not affect the structure of the network.

regions. All the three recent works [95, 71, 17] did not use any type of fitness sharing to encourage diversity. In [71], the authors simply use a very large population size (i.e., 1000) to increase the diversity. A key difference between our work and these previous work lies in the selection process and the number of mutation operations. The proposed solution takes both the limitations of computational resources and diversity of the population into account. As a result, even though we do not impose any strong restriction on the search space, we can use less computational costs to achieve competitive if not better prediction performance than [95, 17, 71].

Other related works on automatically discovering network structures include reinforcement learning [3, 107] based approaches and Bayesian optimization [80, 7, 61] based approaches. We refer the readers to [71] for more discussion and references.

2.4 Designing the Improved Neural Networks Automatically

For the purpose of brevity, we skip some related genetic works [95, 71, 17] for searching the optimal neural networks which have been reviewed in Sec. 2.3. Since this work is in the line of our research work mentioned in Sec. 2.3, we will highlight the difference between this work and the previous one. In our previous reserach work, we aim to achieve better performance from the automatically learned network structure with limited computation cost in the course of evolution, which is not brought up previously. Different from restricting the search space to reduce computational cost [95, 17], they propose the aggressive selection strategy to eliminate the weak neural network structures in the early stage. However, this aggressive selection strategy may decrease the diversity which is the nature of genetic approach to improve performance. In order to remedy this issue, they define more mutation operations such as add_fully_connected or add_pooling. Finally, they reduce computation cost dramatically to 72 GPUH on CIFAR-10. However, there is still performance loss in their approach. For example, on CIFAR-10 dataset, the test accuracy of the found network is about 4% lower than [71]. In order to designing the improved neural networks, we propose the ecologically inspired genetic approach for neural network structure search by evolving the networks through rapid succession, and explore the mimicry and gene duplication along the evolution, inspired from ecological concepts.

CHAPTER 3

SPEEDING UP TRAINING NEURAL NETWORKS

In this chapter, we discuss the improved dropout to speed up training nerual network structures. Some successful neural network strcutures [48] are stacked with convolutional layers, pooling layers followed by fully connected layers and dropout layers between those fully connected layers. In our work we focus on dropout layers and study how to accelerate training for those neural network structures. In existing dropout layers neurons/features are uniformly at random dropped to tackle overfitting issue. We study whether uniformly at random dropping out neurons or features are optimal in terms of convergence rate. We start this chapter by presenting the existing dropout technique. Following that, we discuss the improved dropout for shallow and deep learning and analyze it in terms of convergence rate. At the end, we present empirical results to justify the improved dropout.

3.1 Dropout

In this section, we present some preliminaries, including the framework of risk minimization in machine learning and learning with dropout noise. We also introduce the multinomial dropout, which allows us to construct a distribution-dependent dropout as revealed in the next section.

Let $(\mathbf{x}, y)$ denote a feature vector and a label, where $\mathbf{x} \in \mathbb{R}^d$ and $y \in \mathcal{Y}$. Denote by $\mathcal{P}$ the joint distribution of $(\mathbf{x}, y)$ and denote by $\mathcal{D}$ the marginal distribution of $\mathbf{x}$. The goal of risk minimization is to learn a prediction function $f(\mathbf{x})$ that minimizes the expected loss, i.e., $\min_{f\in\mathcal{H}} \mathrm{E}_{\mathcal{P}}[\ell(f(\mathbf{x}), y)]$, where $\ell(z, y)$ is a loss function (e.g., the logistic loss) that measures the inconsistency between z and y and $\mathcal{H}$ is a class of prediction functions. In deep learning, the prediction function $f(\mathbf{x})$ is determined by a deep neural network. In shallow learning, one might be interested in learning a linear model $f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$. In the following presentation, the analysis will focus on the risk minimization of a linear model, i.e.,

$$\min_{\mathbf{w}\in\mathbb{R}^d} \mathcal{L}(\mathbf{w}) \triangleq \mathrm{E}_{\mathcal{P}}[\ell(\mathbf{w}^\top\mathbf{x}, y)] \tag{3.1}$$

we are interested in learning with dropout, i.e., the feature vector $\mathbf{x}$ is corrupted by a dropout noise. In particular, let $\boldsymbol{\epsilon} \sim \mathcal{M}$ denote a dropout noise vector of dimension d, and the corrupted feature vector is given by $\widehat{\mathbf{x}} = \mathbf{x}\circ\boldsymbol{\epsilon}$, where the operator $\circ$ represents the element-wise multiplication. Let $\widehat{\mathcal{P}}$ denote the joint distribution of the new data $(\widehat{\mathbf{x}}, y)$ and $\widehat{\mathcal{D}}$ denote the marginal distribution of $\widehat{\mathbf{x}}$. With the corrupted data, the risk minimization becomes

$$\min_{\mathbf{w}\in\mathbb{R}^d} \widehat{\mathcal{L}}(\mathbf{w}) \triangleq \mathrm{E}_{\widehat{\mathcal{P}}}[\ell(\mathbf{w}^\top(\mathbf{x}\circ\boldsymbol{\epsilon}), y)] \tag{3.2}$$

In standard dropout [36, 90], the entries of the noise vector $\boldsymbol{\epsilon}$ are sampled independently according to $\Pr(\epsilon_j = 0) = \delta$ and $\Pr(\epsilon_j = \frac{1}{1-\delta}) = 1-\delta$, i.e., features are dropped with a probability δ and scaled by $\frac{1}{1-\delta}$ with a probability $1-\delta$. We can also write $\epsilon_j = \frac{b_j}{1-\delta}$, where $b_j \in \{0, 1\}, j \in [d]$ are i.i.d Bernoulli random variables with $\Pr(b_j = 1) = 1-\delta$. The scaling factor $\frac{1}{1-\delta}$ is added to ensure that $\mathrm{E}_{\boldsymbol{\epsilon}}[\widehat{\mathbf{x}}] = \mathbf{x}$. It is obvious that using the standard dropout different features will have equal probabilities to be dropped out or to be selected independently. However, in practice some features could be more informative than the others for learning purpose. Therefore, it makes more sense to assign different sampling probabilities for different features and make the features compete with each other.

3.2 Improved Dropout

We introduce the following multinomial dropout.

Definition 1 ***(Multinomial Dropout)*** *A multinomial dropout is defined as* $\widehat{\mathbf{x}} = \mathbf{x} \circ \boldsymbol{\epsilon}$*, where* $\epsilon_i = \frac{m_i}{kp_i}, i \in [d]$ *and* $\{m_1, \dots, m_d\}$ *follow a multinomial distribution* $Mult(p_1, \dots, p_d; k)$ *with* $\sum_{i=1}^{d} p_i = 1$ *and* $p_i \geq 0$*.*

Remark: The multinomial dropout allows us to use non-uniform sampling probabilities $p_1, \dots, p_d$ for different features. The value of m_i is the number of times that the i-th feature is selected in k independent trials of selection. In each trial, the probability that the i-th feature is selected is given by p_i. As in the standard dropout, the normalization by kp_i is to ensure that $\mathrm{E}_{\boldsymbol{\epsilon}}[\widehat{\mathbf{x}}] = \mathbf{x}$. The parameter k plays the same role as the parameter $1-\delta$ in standard dropout, which controls the number of features to be dropped. In particular, the expected total number of the kept features using multinomial dropout is k and that using standard dropout is $d(1-\delta)$. In the sequel, to make fair comparison between the two dropouts, we let $k = d(1-\delta)$. In this case, when a uniform distribution $p_i = 1/d$ is used in multinomial dropout to which we refer as *uniform dropout*, then $\epsilon_i = \frac{m_i}{1-\delta}$, which acts similarly to the standard dropout using i.i.d Bernoulli random variables. Note that another choice to make the sampling probabilities different is still using i.i.d Bernoulli random variables but with different probabilities for different features. However, multinomial dropout is more suitable because (i) it is easy to control the level of dropout by varying the value of k; (ii) it gives rise to natural competition among features because of the constraint $\sum_i p_i = 1$; (iii) it allows us to minimize the sampling dependent risk bound for obtaining a better distribution than uniform sampling.

Dropout is a data-dependent regularizer Dropout as a regularizer has been studied in [90, 4] for logistic regression, which is stated in the following proposition for ease of discussion later.

Proposition 1 *If $\ell(z, y) = \log(1 + \exp(-yz))$, then*

$$\mathrm{E}_{\widehat{\mathcal{P}}}[\ell(\mathbf{w}^\top \widehat{\mathbf{x}}, y)] = \mathrm{E}_{\mathcal{P}}[\ell(\mathbf{w}^\top \mathbf{x}, y)] + R_{\mathcal{D},\mathcal{M}}(\mathbf{w}) \tag{3.3}$$

where $\mathcal{M}$ denotes the distribution of $\boldsymbol{\epsilon}$ and $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}) = \mathrm{E}_{\mathcal{D},\mathcal{M}}\left[\log \frac{\exp(\mathbf{w}^\top \frac{\mathbf{x}\circ\boldsymbol{\epsilon}}{2}) + \exp(-\mathbf{w}^\top \frac{\mathbf{x}\circ\boldsymbol{\epsilon}}{2})}{\exp(\mathbf{w}^\top \mathbf{x}/2) + \exp(-\mathbf{w}^\top \mathbf{x}/2)}\right]$.

Remark: It is notable that $R_{\mathcal{D},\mathcal{M}} \geq 0$ due to the Jensen inequality. Using the second order Taylor expansion, [90] showed that the following approximation of $R_{\mathcal{D},\mathcal{M}}(\mathbf{w})$ is easy to manipulate and understand:

$$\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}) = \frac{\mathrm{E}_{\mathcal{D}}[q(\mathbf{w}^\top \mathbf{x})(1 - q(\mathbf{w}^\top \mathbf{x}))\mathbf{w}^\top C_{\mathcal{M}}(\mathbf{x} \circ \boldsymbol{\epsilon})\mathbf{w}]}{2} \tag{3.4}$$

where $q(\mathbf{w}^\top \mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^\top \mathbf{x}/2)}$, and $C_{\mathcal{M}}$ denotes the covariance matrix in terms of $\boldsymbol{\epsilon}$. In particular, if $\boldsymbol{\epsilon}$ is the standard dropout noise, then $C_{\mathcal{M}}[\mathbf{x} \circ \boldsymbol{\epsilon}] = diag(x_1^2 \delta/(1 - \delta), \dots, x_d^2 \delta/(1 - \delta))$, where $diag(s_1, \dots, s_n)$ denotes a $d \times d$ diagonal matrix with the i-th entry equal to s_i. If $\boldsymbol{\epsilon}$ is the multinomial dropout noise in Definition 1, we have

$$C_{\mathcal{M}}[\mathbf{x} \circ \boldsymbol{\epsilon}] = \frac{1}{k} diag(x_i^2/p_i) - \frac{1}{k}\mathbf{x}\mathbf{x}^\top \tag{3.5}$$

3.2.1 Learning with Multinomial Dropout

In this section, we analyze a stochastic optimization approach for minimizing the dropout loss in (3.2). Assume the sampling probabilities are known. We first obtain a risk bound of learning with multinomial dropout for stochastic optimization. Then we try to minimize the factors in the risk bound that depend on the sampling probabilities. We would like to emphasize that our goal here is not to show that using dropout would render a smaller risk than without using dropout, but rather focus on the impact of different sampling probabilities on the risk. Let the initial solution be $\mathbf{w}_1$. At the iteration t, we sample $(\mathbf{x}_t, y_t) \sim \mathcal{P}$ and $\boldsymbol{\epsilon}_t \sim \mathcal{M}$ as in Definition 1 and then update the model by

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla \ell(\mathbf{w}_t^\top (\mathbf{x}_t \circ \boldsymbol{\epsilon}_t), y_t) \tag{3.6}$$

where $\nabla \ell$ denotes the (sub)gradient in terms of $\mathbf{w}_t$ and η_t is a step size. Suppose we run the stochastic optimization by n steps (i.e., using n examples) and compute the final solution as $\widehat{\mathbf{w}}_n = \frac{1}{n}\sum_{t=1}^n \mathbf{w}_t$.

We note that another approach of learning with dropout is to minimize the empirical risk by marginalizing out the dropout noise, i.e., replacing the true expectations $\mathrm{E}_{\mathcal{P}}$ and $\mathrm{E}_{\mathcal{D}}$ in (3.3) with empirical expectations over a set of samples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ denoted by $\mathrm{E}_{\mathcal{P}_n}$ and $\mathrm{E}_{\mathcal{D}_n}$. Since the data dependent regularizer $R_{\mathcal{D}_n,\mathcal{M}}(\mathbf{w})$ is difficult to compute, one usually uses an approximation $\widehat{R}_{\mathcal{D}_n,\mathcal{M}}(\mathbf{w})$ (e.g., as in (3.4)) in place of $R_{\mathcal{D}_n,\mathcal{M}}(\mathbf{w})$. However, the resulting problem is a non-convex optimization, which together with the approximation error would make the risk analysis much more involved. In contrast, the update in (3.6) can be considered as a stochastic gradient descent update for solving the convex optimization problem in (3.2), allowing us to establish the risk bound based on previous results of stochastic gradient descent for risk minimization [76, 81]. Nonetheless, this restriction does not lose the generality. Indeed, stochastic optimization is usually employed for solving empirical loss minimization in big data and deep learning.

3.2.2 Distribution Dependent Dropout

Next, we consider the sampling dependent factors in the risk bounds. From Theorem 1, we can see that there are two terms that depend on the sampling probabilities, i.e., B^2 - the upper bound of $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$, and $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) - R_{\mathcal{D},\mathcal{M}}(\widehat{\mathbf{w}}_n) \leq R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$. We note that the second term also depends on $\mathbf{w}_*$ and $\widehat{\mathbf{w}}_n$, which is more difficult to optimize. We first try to minimize $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$ and present the discussion on minimizing $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$ later. From Theorem 1, we can see that minimizing $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$ would lead to not only a smaller risk (given the same number of total examples, smaller $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$ gives a smaller risk bound) but also a faster convergence (with the same number of iterations, smaller $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$ gives a smaller optimization error).

The proofs of Proposition 2, 3, 4 are included in Appendix. The following proposition simplifies the expectation $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$.

Proposition 2 *Let $\boldsymbol{\epsilon}$ follow the distribution $\mathcal{M}$ defined in Definition 1. Then*

$$\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2] = \frac{1}{k}\sum_{i=1}^{d} \frac{1}{p_i}\mathrm{E}_{\mathcal{D}}[x_i^2] + \frac{k-1}{k}\sum_{i=1}^{d}\mathrm{E}_{\mathcal{D}}[x_i^2] \tag{3.7}$$

Given the expression of $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2]$ in Proposition 2, we can minimize it over $\mathbf{p}$, leading to the following result.

Proposition 3 *The solution to* $\mathbf{p}_* = \arg\min_{\mathbf{p}\geq 0, \mathbf{p}^\top \mathbf{1}=1} \mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x}\circ\boldsymbol{\epsilon}\|_2^2]$ *is given by*

$$p_i^* = \frac{\sqrt{\mathrm{E}_{\mathcal{D}}[x_i^2]}}{\sum_{j=1}^d \sqrt{\mathrm{E}_{\mathcal{D}}[x_j^2]}}, i = 1, \ldots, d \tag{3.8}$$

Next, we examine $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$. Since direct manipulation on $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$ is difficult, we try to minimize the second order Taylor expansion $\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$ for logistic loss. The following theorem establishes an upper bound of $\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$.

Proposition 4 *Let* $\boldsymbol{\epsilon}$ *follow the distribution* $\mathcal{M}$ *defined in Definition 1. We have* $\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) \leq \frac{1}{8k}\|\mathbf{w}_*\|_2^2 \left(\sum_{i=1}^d \frac{\mathrm{E}_{\mathcal{D}}[\mathbf{x}_i^2]}{p_i} - \mathrm{E}_{\mathcal{D}}[\|\mathbf{x}\|_2^2]\right)$

Remark: By minimizing the relaxed upper bound in Proposition 4, we obtain the same sampling probabilities as in (3.8). We note that a tighter upper bound can be established, however, which will yield sampling probabilities dependent on the unknown $\mathbf{w}_*$.

In summary, using the probabilities in (3.8), we can reduce both $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x}\circ\boldsymbol{\epsilon}\|_2^2]$ and $R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$ in the risk bound, leading to a faster convergence and a smaller generalization error. In practice, we can use empirical second-order statistics to compute the probabilities, i.e.,

$$p_i = \frac{\sqrt{\frac{1}{n}\sum_{j=1}^n [[\mathbf{x}_j]_i^2]}}{\sum_{i'=1}^d \sqrt{\frac{1}{n}\sum_{j=1}^n [[\mathbf{x}_j]_{i'}^2]}} \tag{3.9}$$

where $[\mathbf{x}_j]_i$ denotes the i-th feature of the j-th example, which gives us a data-dependent dropout. We state it formally in the following definition.

Definition 2 *(**Data-dependent Dropout**) Given a set of training examples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, a data-dependent dropout is defined as $\widehat{\mathbf{x}} = \mathbf{x} \circ \boldsymbol{\epsilon}$, where $\epsilon_i = \frac{m_i}{kp_i}, i \in [d]$ and $\{m_1, \ldots, m_d\}$ follow a multinomial distribution $Mult(p_1, \ldots, p_d; k)$ with p_i given by (3.9).*

Remark: Note that if the data is normalized such that each feature has zero mean and unit variance (i.e., according to Z-normliazation), the data-dependent dropout reduces to uniform dropout. It implies that the data-dependent dropout achieves similar effect as Z-normalization plus uniform dropout. In this sense, our theoretical analysis also explains why Z-normalization usually speeds up the training [69].

3.2.3 Evolutional Dropout for Deep Learning

Next, we discuss how to implement the distribution-dependent dropout for deep learning. In training deep neural networks, the dropout is usually added to the intermediate layers (e.g., fully connected layers and convolutional layers). Let $\mathbf{x}^l = (x_1^l, \ldots, x_d^l)$ denote the outputs of the l-th layer (with the index of data omitted). Adding dropout to this layer is equivalent to multiplying $\mathbf{x}^l$ by a dropout noise vector $\boldsymbol{\epsilon}^l$, i.e., feeding $\widehat{\mathbf{x}}^l = \mathbf{x}^l \circ \boldsymbol{\epsilon}^l$ as the input to the next layer. Inspired by the data-dependent dropout, we can generate $\boldsymbol{\epsilon}^l$ according to a distribution given in Definition 1 with sampling probabilities p_i^l computed from $\{\mathbf{x}_1^l, \ldots, \mathbf{x}_n^l\}$ similar to that (3.9). However, deep learning is usually trained with big data and a deep neural network is optimized by mini-batch stochastic gradient descent. Therefore, at each iteration it would be too expensive to afford the computation to pass through all examples. To address this issue, we propose to use a mini-batch of examples to

```
Evolutional Dropout for Deep Learning

Input: a batch of outputs of a layer: $X^l = (\mathbf{x}_1^l, \ldots, \mathbf{x}_m^l)$
       and dropout level parameter $k \in [0, d]$

Output: $\widehat{X}^l = X^l \circ \Sigma^l$

 Compute sampling probabilities by (3.10)
 For $j = 1, \ldots, m$
   Sample $\mathbf{m}_j^l \sim Mult(p_1^l, \ldots, p_d^l; k)$
   Construct $\boldsymbol{\epsilon}_j^l = \frac{\mathbf{m}_j^l}{k\mathbf{p}^l} \in \mathbb{R}^d$, where $\mathbf{p}^l = (p_1^l, \ldots, p_d^l)^\top$
 Let $\Sigma^l = (\boldsymbol{\epsilon}_1^l, \ldots, \boldsymbol{\epsilon}_m^l)$ and compute $\widehat{X}^l = X^l \circ \Sigma^l$
```

Figure 3.1. Evolutional Dropout applied to a layer over a mini-batch

calculate the second-order statistics similar to what was done in batch normalization. Let $X^l = (\mathbf{x}_1^l, \ldots, \mathbf{x}_m^l)$ denote the outputs of the l-th layer for a mini-batch of m examples. Then we can calculate the probabilities for dropout by

$$p_i^l = \frac{\sqrt{\frac{1}{m}\sum_{j=1}^m [[\mathbf{x}_j^l]_i^2]}}{\sum_{i'=1}^d \sqrt{\frac{1}{m}\sum_{j=1}^m [[\mathbf{x}_j^l]_{i'}^2]}}, i = 1, \ldots, d \tag{3.10}$$

which define the evolutional dropout named as such because the probabilities p_i^l will also evolve as the the distribution of the layer's outputs evolve. We describe the evolutional dropout as applied to a layer of a deep neural network in Figure 3.1.

3.3 Theoretical Analysis

The following theorem establishes a risk bound of $\widehat{\mathbf{w}}_n$ in expectation.

Theorem 1 *Let $\mathcal{L}(\mathbf{w})$ be the expected risk of $\mathbf{w}$ defined in (3.1). Assume $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2] \leq B^2$ and $\ell(z, y)$ is G-Lipschitz continuous. For any $\|\mathbf{w}_*\|_2 \leq r$, by appropriately* *choosing η, we can have*

$$\mathrm{E}[\mathcal{L}(\widehat{\mathbf{w}}_n) + R_{\mathcal{D},\mathcal{M}}(\widehat{\mathbf{w}}_n)] \leq \mathcal{L}(\mathbf{w}_*) + R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) + \frac{GBr}{\sqrt{n}}$$

where $\mathrm{E}[\cdot]$ *is taking expectation over the randomness in* $(\mathbf{x}_t, y_t, \boldsymbol{\epsilon}_t), t = 1, \ldots, n$.

Remark: In the above theorem, we can choose $\mathbf{w}_*$ to be the best model that minimizes the expected risk in (3.1). Since $R_{\mathcal{D},M}(\mathbf{w}) \geq 0$, the upper bound in the theorem above is also the upper bound of the risk of $\widehat{\mathbf{w}}_n$, i.e., $\mathcal{L}(\widehat{\mathbf{w}}_n)$, in expectation. The proof of the above theorem follows the standard analysis of stochastic gradient descent.

Proof: The update given by $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\nabla\ell(\mathbf{w}_t^\top(\mathbf{x}_t \circ \boldsymbol{\epsilon}_t), y_t)$ can be considered as the stochastic gradient descent (SGD) update of the following problem

$$\min_{\mathbf{w}}\{\widehat{\mathcal{L}}(\mathbf{w}) \triangleq \mathrm{E}_{\widehat{\mathcal{P}}}[\ell(\mathbf{w}^\top(\mathbf{x} \circ \boldsymbol{\epsilon}), y)]\}$$

Define $\mathbf{g}_t$ as $\mathbf{g}_t = \nabla\ell(\mathbf{w}_t^\top(\mathbf{x}_t \circ \boldsymbol{\epsilon}_t), y_t) = \ell'(\mathbf{w}_t^\top(\mathbf{x}_t \circ \boldsymbol{\epsilon}_t), y_t)\mathbf{x}_t \circ \boldsymbol{\epsilon}_t$, where $\ell'(z, y)$ denotes the derivative in terms of z. Since the loss function is G-Lipschitz continuous, therefore $\|\mathbf{g}_t\|_2 \leq G\|\mathbf{x}_t \circ \boldsymbol{\epsilon}_t\|_2$. According to the analysis of SGD [106], we have the following lemma.

Lemma 1 *Let* $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta\mathbf{g}_t$ *and* $\mathbf{w}_1 = 0$. *Then for any* $\|\mathbf{w}_*\|_2 \leq r$ *we have*

$$\sum_{t=1}^{n} \mathbf{g}_t^\top(\mathbf{w}_t - \mathbf{w}_*) \leq \frac{r^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{n} \|\mathbf{g}_t\|_2^2 \tag{3.11}$$

By taking expectation on both sides over the randomness in $(\mathbf{x}_t, y_t, \boldsymbol{\epsilon}_t)$ and noting the bound on $\|\mathbf{g}_t\|_2$, we have

$$\mathrm{E}_{[n]}\left[\sum_{t=1}^{n} \mathbf{g}_t^\top(\mathbf{w}_t - \mathbf{w}_*)\right] \leq \frac{r^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{n} G^2\mathrm{E}_{[n]}[\|\mathbf{x}_t \circ \boldsymbol{\epsilon}_t\|_2^2]$$

where $\mathrm{E}_{[t]}$ denote the expectation over $(\mathbf{x}_i, y_i, \boldsymbol{\epsilon}_i), i = 1, \ldots, t$. Let $\mathrm{E}_t[\cdot]$ denote the expectation over $(\mathbf{x}_t, y_t, \boldsymbol{\epsilon}_t)$ with $(\mathbf{x}_i, y_i, \boldsymbol{\epsilon}_i), i = 1, \ldots, t-1$ given. Then we have

$$\sum_{t=1}^{n} \mathrm{E}_{[t]}[\mathbf{g}_t^\top(\mathbf{w}_t - \mathbf{w}_*)] \leq \frac{r^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{n} G^2 \mathrm{E}_t[\|\mathbf{x}_t \circ \boldsymbol{\epsilon}_t\|_2^2]$$

Since

$$\mathrm{E}_{[t]}[\mathbf{g}_t^\top(\mathbf{w}_t - \mathbf{w}_*)] = \mathrm{E}_{[t-1]}[\mathrm{E}_t[\mathbf{g}_t]^\top(\mathbf{w}_t - \mathbf{w}_*)] = \mathrm{E}_{[t-1]}[\nabla\widehat{\mathcal{L}}(\mathbf{w}_t)^\top(\mathbf{w}_t - \mathbf{w}_*)] \geq \mathrm{E}_{[t-1]}[\widehat{\mathcal{L}}(\mathbf{w}_t) - \widehat{\mathcal{L}}(\mathbf{w}_*)]$$

As a result

$$\mathrm{E}_{[n]}\left[\sum_{t=1}^{n}(\widehat{\mathcal{L}}(\mathbf{w}_t) - \widehat{\mathcal{L}}(\mathbf{w}_*))\right] \leq \frac{r^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{n} G^2 \mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x}_t \circ \boldsymbol{\epsilon}_t\|_2^2] \leq \frac{r^2}{2\eta} + \frac{\eta}{2}G^2B^2n \quad (3.12)$$

where the last inequality follows the assumed upper bound of $\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x}_t \circ \boldsymbol{\epsilon}_t\|_2^2]$. Following the definition of $\widehat{\mathbf{w}}_n$ and the convexity of $\mathcal{L}(\mathbf{w})$ we have

$$\mathrm{E}_{[n]}[\widehat{\mathcal{L}}(\widehat{\mathbf{w}}_n) - \widehat{\mathcal{L}}(\mathbf{w}_*)] \leq \mathrm{E}_{[n]}\left[\frac{1}{n}\sum_{t=1}^{n}(\widehat{\mathcal{L}}(\mathbf{w}_t) - \widehat{\mathcal{L}}(\mathbf{w}_*))\right] \leq \frac{r^2}{2\eta n} + \frac{\eta}{2}G^2B^2$$

By minimizing the upper bound in terms of η, we have $\mathrm{E}_{[n]}[\widehat{\mathcal{L}}(\widehat{\mathbf{w}}_n) - \widehat{\mathcal{L}}(\mathbf{w}_*)] \leq \frac{GBr}{\sqrt{n}}$. According to Proposition 1 in the paper $\widehat{\mathcal{L}}(\mathbf{w}) = \mathcal{L}(\mathbf{w}) + R_{\mathcal{D},\mathcal{M}}(\mathbf{w})$, therefore

$$\mathrm{E}_{[n]}[\mathcal{L}(\widehat{\mathbf{w}}_n) + R_{\mathcal{D},\mathcal{M}}(\widehat{\mathbf{w}}_n)] \leq \mathcal{L}(\mathbf{w}_*) + R_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) + \frac{GBr}{\sqrt{n}}$$

Proof of Lemma 1: We have the following:

$$\frac{1}{2}\|\mathbf{w}_{t+1} - \mathbf{w}_*\|_2^2 = \frac{1}{2}\|\mathbf{w}_t - \eta\mathbf{g}_t - \mathbf{w}_*\|_2^2 = \frac{1}{2}\|\mathbf{w}_t - \mathbf{w}_*\|_2^2 + \frac{\eta^2}{2}\|\mathbf{g}_t\|_2^2 - \eta(\mathbf{w}_t - \mathbf{w}_*)^\top\mathbf{g}_t$$

Then

$$(\mathbf{w}_t - \mathbf{w}_*)^\top\mathbf{g}_t \leq \frac{1}{2\eta}\|\mathbf{w}_t - \mathbf{w}_*\|_2^2 - \frac{1}{2\eta}\|\mathbf{w}_{t+1} - \mathbf{w}_*\|_2^2 + \frac{\eta}{2}\|\mathbf{g}_t\|_2^2$$

By summing the above inequality over $t = 1, \ldots, n$, we obtain

$$\sum_{t=1}^{n} \mathbf{g}_t^\top (\mathbf{w}_t - \mathbf{w}_*) \leq \frac{\|\mathbf{w}_* - \mathbf{w}_1\|_2^2}{2\eta} + \frac{\eta}{2} \sum_{t=1}^{n} \|\mathbf{g}_t\|_2^2$$

By noting that $\mathbf{w}_1 = 0$ and $\|\mathbf{w}_*\|_2 \leq r$, we obtain the inequality in Lemma 1.

3.4 Experimental results

In the section, we present some experimental results to justify the proposed dropouts. For the sake of clarity, we divided the experiments into three parts. In the first part, we compare the performance of the data-dependent dropout (**d-dropout)** to the standard dropout (**s-dropout**) for logistic regression. In the second part, we compare the performance of evolutional dropout (**e-dropout**) to the standard dropout for training deep convolutional neural networks.

3.4.1 On Shallow Learning

We implement the presented stochastic optimization algorithm. To evaluate the performance of data-dependent dropout for shallow learning, we use the three data sets: real-sim, news20 and RCV1[1]. In this experiment, we use a fixed step size and tune the step size in $[0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001]$ and report the best results in terms of convergence speed on the training data for both standard dropout and data-dependent dropout. The left three panels in Figure **??** show the obtained results on these three data sets. In each figure, we plot both the training error and the testing error. We can see that both the training and testing errors using the proposed data-dependent dropout decrease much faster than using the standard

[1]`https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

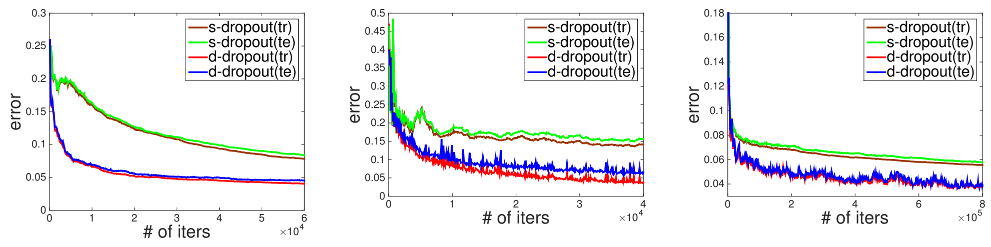

Figure 3.2. data-dependent dropout vs. standard dropout on three data sets (real-sim, news20, RCV1) for logistic regression;

dropout and also a smaller testing error is achieved by using the data-dependent dropout.

3.4.2 On Deep Learning

We conduct experiments on four benchmark data sets for comparing e-dropout and s-dropout: MNIST [50], SVHN [65], CIFAR-10 and CIFAR-100 [47]. We use the same or similar network structure as in the literatures for the four data sets. In general, the networks consist of convolution layers, pooling layers, locally connected layers, fully connected layers, softmax layers and a cost layer. For the detailed neural network structures and their parameters, please refer to the supplementary materials. The dropout is added to some fully connected layers or locally connected layers. The rectified linear activation function is used for all neurons. All the experiments are conducted using the cuda-convnet library [2]. The training procedure is similar to [49] using mini-batch SGD with momentum (0.9). The size of mini-batch is fixed to 128. The weights are initialized based on the Gaussian distribution with mean zero and standard deviation 0.01. The learning rate (i.e., step size) is decreased after a

[2]`https://code.google.com/archive/p/cuda-convnet/`

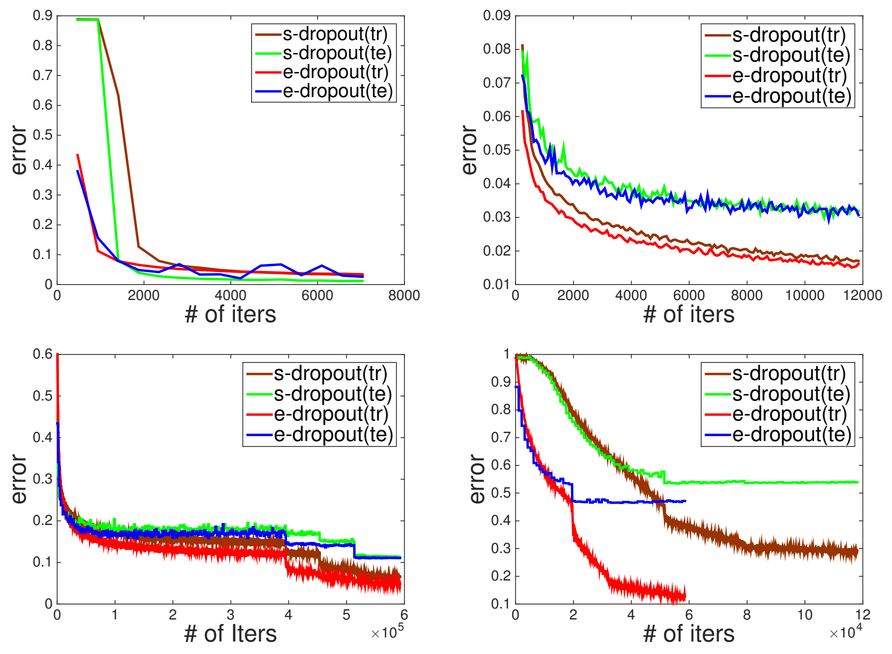

Figure 3.3. Evolutional dropout vs. standard dropout on four benchmark datasets for deep learning (best seen in color).

number of epochs similar to what was done in previous works [49]. We tune the initial learning rates for s-dropout and e-dropout separately from $0.001, 0.005, 0.01, 0.1$ and report the best result on each data set that yields the fastest convergence.

3.5 The Proofs of Proposition 2, 3, 4

In this section, we provide the proofs of Proposition 2, 3, 4 and neural network structures used when we conducted experiments on MNIST, SVHN, CIFAR-10, and CIFAR-100 dataset in chapter 3.

3.5.1 Proof of Proposition 2

We have

$$\mathrm{E}_{\widehat{\mathcal{D}}}\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2 = \mathrm{E}_{\mathcal{D}}\left[\sum_{i=1}^{d} \frac{x_i^2}{k^2 p_i^2}\mathrm{E}[m_i^2]\right]$$

Since $\{m_1, \ldots, m_d\}$ follows a multinomial distribution $Mult(p_1, \ldots, p_d; k)$, we have

$$\mathrm{E}[m_i^2] = var(m_i) + (\mathrm{E}[m_i])^2 = kp_i(1-p_i) + k^2 p_i^2$$

The result in the Proposition follows by combining the above two equations.

3.5.2 Proof of Proposition 3

Note that only the first term in the R.H.S of Eqn. (7) depends on p_i. Thus,

$$\mathbf{p}_* = \arg \min_{\mathbf{p} \geq 0, \mathbf{p}^\top \mathbf{1} = 1} \sum_{i=1}^{d} \frac{\mathrm{E}_{\mathcal{D}}[x_i^2]}{p_i}$$

The result then follows the KKT conditions.

3.5.3 Proof of Proposition 4

We prove the first upper bound first. From Eqn. (4) in the paper, we have

$$\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) \leq \frac{1}{8} \mathrm{E}_{\mathcal{D}}[\mathbf{w}_*^\top C_{\mathcal{M}}(\mathbf{x} \circ \epsilon) \mathbf{w}_*]$$

where we use the fact $\sqrt{ab} \leq \frac{a+b}{2}$ for $a, b \geq 0$. Using Eqn. (5) in the paper, we have

$$\mathrm{E}_{\mathcal{D}}[\mathbf{w}_*^\top C_{\mathcal{M}}(\mathbf{x} \circ \epsilon)\mathbf{w}_*] = \mathrm{E}_{\mathcal{D}}\left[\mathbf{w}_*^\top \left(\frac{1}{k} diag(x_i^2/p_i) - \frac{1}{k}\mathbf{x}\mathbf{x}^\top\right)\mathbf{w}_*\right] = \frac{1}{k}\mathrm{E}_{\mathcal{D}}\left[\sum_{i=1}^{d} \frac{w_{*i}^2 x_i^2}{p_i} - (\mathbf{w}_*^\top \mathbf{x})^2\right]$$

This gives a tight bound of $\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*)$, i.e.,

$$\widehat{R}_{\mathcal{D},\mathcal{M}}(\mathbf{w}_*) \leq \frac{1}{8k}\left\{\sum_{i=1}^{d} \frac{w_{*i}^2 \mathrm{E}_{\mathcal{D}}[\mathbf{x}_i^2]}{p_i} - \mathrm{E}_{\mathcal{D}}(\mathbf{w}_*^\top \mathbf{x})^2\right\}$$

By minimizing the above upper bound over p_i, we obtain following probabilities

$$p_i^* = \frac{\sqrt{w_{*i}^2 \mathrm{E}_{\mathcal{D}}[x_i^2]}}{\sum_{j=1}^{d} \sqrt{w_{*i}^2 \mathrm{E}_{\mathcal{D}}[x_j^2]}} \tag{3.13}$$

which depend on unknown $\mathbf{w}_*$. We address this issue, we derive a relaxed upper bound. We note that

$$C_{\mathcal{M}}(\mathbf{x} \circ \epsilon) = \mathrm{E}_{\mathcal{M}}[(\mathbf{x} \circ \boldsymbol{\epsilon} - \mathbf{x})(\mathbf{x} \circ \boldsymbol{\epsilon} - \mathbf{x})^\top]$$
$$\leq (\mathrm{E}_{\mathcal{M}} \|\mathbf{x} \circ \boldsymbol{\epsilon} - \mathbf{x}\|_2^2) \cdot I_d = \left(\mathrm{E}_{\mathcal{M}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2] - \|\mathbf{x}\|_2^2\right) I_d$$

where I_d denotes the identity matrix of dimension d. Thus

$$\mathrm{E}_{\mathcal{D}}[\mathbf{w}_*^\top C_{\mathcal{M}}(\mathbf{x} \circ \epsilon)\mathbf{w}_*] \leq \|\mathbf{w}_*\|_2^2 \left(\mathrm{E}_{\widehat{\mathcal{D}}}[\|\mathbf{x} \circ \boldsymbol{\epsilon}\|_2^2] - \mathrm{E}_{\mathcal{D}}[\|\mathbf{x}\|_2^2]\right)$$

By noting the result in Proposition 2 in the paper, we have

$$\mathrm{E}_{\mathcal{D}}[\mathbf{w}_*^\top C_{\mathcal{M}}(\mathbf{x} \circ \epsilon)\mathbf{w}_*] \leq \frac{1}{k}\|\mathbf{w}_*\|_2^2 \left(\sum_{i=1}^{d} \frac{\mathrm{E}_{\mathcal{D}}[\mathbf{x}_i^2]}{p_i} - \mathrm{E}_{\mathcal{D}}[\|\mathbf{x}\|_2^2]\right)$$

which proves the upper bound in Proposition 4.

CHAPTER 4

COMPRESSING NEURAL NETWORK

In this chapter, we disucss the proposed techniques to compress neural network without large performance loss. Recent succussful neural networks that achieved better performance becomes more complex and large, making those structures not applicable to mobile/embeded device. We propose several techniques to compress large neural network at same time without large performance loss. Based on the proposed techniques, we design a new neural network structure called SEP-Nets, which achives better performance compared with recently released mobilenet by Google. In Sec 4.1 we discuss the proposed technique pattern binarization. Following that, we discuss three ingredients used in the proposed SEP-Nets: SEP-Module, Residual Pattern Block and Group-wised Convolution in Sec 4.2. In Sec 4.3, we present the designed network structures SEP-Nets. We justify the efficiency and effectiveness of the proposed techniques and the designed network structure in Sec 4.4.

4.1 The proposed Technique

Since fully connected layers have been removed in modern deep CNNs (including Inception Nets, ResNets), here we only consider parameterized convolutional layers. We adopt the following simple procedure to obtain a compressed network from any successful network structures including our designed SEP-Net as described later:

- Step 1: train a full neural network such as GoogLeNet, ResNet and SEP-Net from scratch.

- Step 2: binarize $k \times k$ $(k > 1)$ convolutional filters in the well-trained neural network model.
- Step 3: fine-tune the scaling factors of all binarized $k \times k$ filters and the floating point representations of all 1×1 filters by back-propagation on the same dataset.

The different treatments of 1×1 filters and $k \times k$ filters is motivated by their complementary roles in CNNs. $k \times k$ filters serve as spatial pattern extraction from an input image/feature map, while 1×1 filters mainly serve as data projection and transformation. To justify our choice, we also quantitatively analyze the effect of binarizing 1×1 filters and $k \times k$ filters from the viewpoint of quantization error. In particular, if we let W denote an $c \times k \times k$ convolutional filter. The binarization seeks to approximate it by αB, where B is a binary filter with entries from $\{1, -1\}$ and α is a scaling factor. From the viewpoint of minimizing the quantization error, α, B can be sought by solving the following problem:

$$\min_{\alpha \in \mathbb{R}, B \in \{1,-1\}^{c \times k \times k}} E(W, B, \alpha) \triangleq \|W - \alpha B\|_F^2 \tag{4.1}$$

The optimal solutions have been studied in [40]. Actually, the optimal B^* can be found by thresholding, i.e., $B^*_{i,j,l} = 1$ if $W_{i,j,l} \geq 0$ and $B^*_{i,j,l} = -1$ if $W_{i,j,l} < 0$. This binarization procedure is illustrated in Figure 4.1. The optimal α_* can be computed by $\alpha_* = \frac{\sum_{i,j,l} |W_{i,j,l}|}{c \times k \times k}$. To quantitatively understand the effect of binarizing 1×1 filters and $k \times k$ filters, we first train a fully GoogLeNet [86], which is composed of 1×1, 3×3 and 5×5 convolutions filters. Then we compute the quantization error for all filters, and obtain averaged quantization error for 1×1, 3×3 and 5×5 filters,

Table 4.1. Averaged Quantization Error of Different Sized Filters from GoogLeNet.

1×1	3×3	5×5
0.0462	0.0029	0.0056

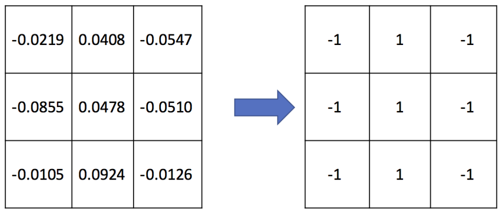

Figure 4.1. A trained 3×3 filter from GoogLeNet (Left), and its binarized version (Right)

respectively. The result is reported in Table 4.1, from which we can clearly see that that quantization error for 1×1 filters is by a order of magnitude larger than that of 3×3 and 5×5 filters. This justifies our choice of binarizing $k \times k$ filters ($k > 1$) while retaining 1×1 filters. We present more results on prediction performance by binarizing 1×1, 3×3 and 5×5 filters.

4.2 Three ingredients in SEP-Nets

In this section, we discuss three ingredients used in our proposed SEP-Nets.

4.2.1 SEP-Module

Pattern binarization is an effective method for compressing the size of a large and deep CNN. We can apply this technique to reducing the size of previous deep CNNs (e.g., GoogLeNets, Inception Nets and ResNets). However, due to that the original sizes of these deep CNNs are very large, the resulting pattern networks from these deep CNNs may not be small enough for deployment in mobile and embedded applications. To address this issue, we propose a new design of a small and effective network (**SEP-Net**).

4.2.2 Residual Pattern Block

Pattern Residual Block. We first describe the building block of our design - a novel block codenamed pattern residual block (PRB). As shown in Figure 4.2(a), the PRB consists of 1×1 convolutions and $k \times k$ convolutions (in particular $k = 3$), which are executed in parallel and their feature maps are added together. In particular, if we let x denote an input, the output of this building block can be expressed as $O(x) = C_{k\times k}(x) + C_{1\times 1}(x)$, where $C_{k\times k}$ denotes a $k \times k$ convolution ($k > 1$) and $C_{1\times 1}$ represents 1×1 convolution. If we consider x a vector, $C_{1\times 1}(x)$ is equivalent to Ax, where A is a linear mapping. Since this is a generalization of identity mapping, we find it particularly useful in building pattern networks. Especially when the pattern block ($3 * 3$ convolutions) is binarized, the additive 1×1 convolutions is able to offset the change incurred by the binarization, which acts the residual between fully $3 * 3$ filtered maps and binarized $3 * 3$ filtered maps. One might notice that a similar block structure has been explored in [30], named as conv-shortcut. However, we would like to emphasize that the key difference is that the k*k convolutions in our block are intended to be binary. In [30], the so-called "conv-shortcut" is not as effective as the original shortcut connection in the full parameter setting. However, our experiments show that for binary pattern k*k convolutions, adding 1*1 filtered maps to binarized k*k filter maps can greatly improve the performance.

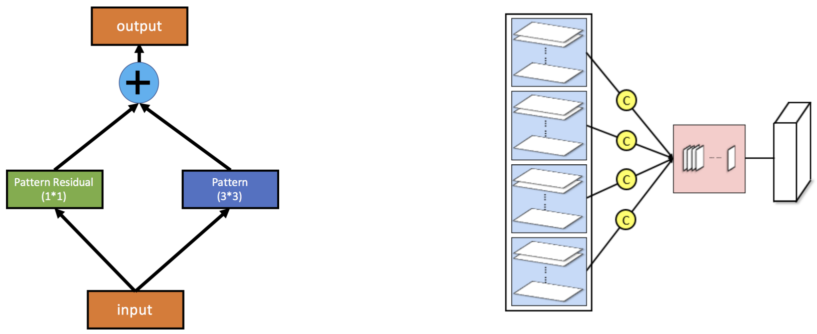

Figure 4.2. Left: Pattern Residual Block; Right: Illustration of Group-wise convolution

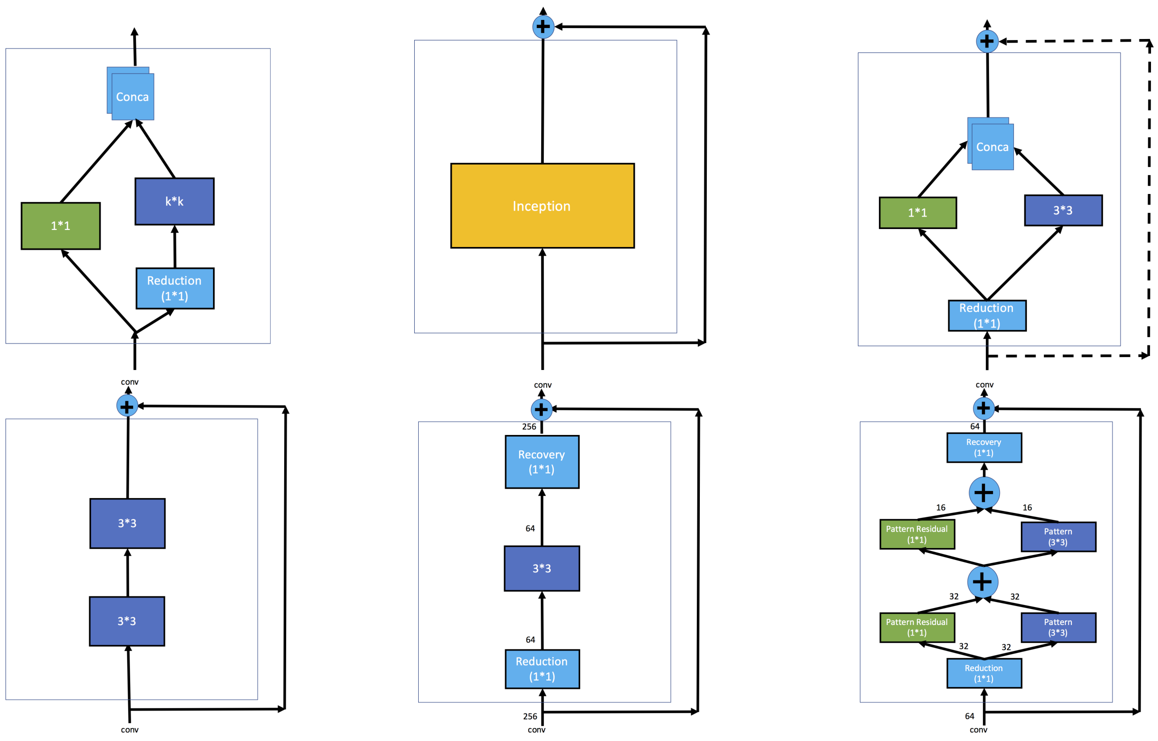

Figure 4.3. Different Network Modules (dashed line represents either the identity mapping or the transformed mapping by $1 * 1$ convolutions. The outside solid lines represent the identity mapping. The numbers in ResNet "bottleneck" module and SEP-Net module represent the number of output channels for illustration purpose.

4.2.3 Group-wised Convolution

To further reduce the model size, we adopt group convolution [48] in our architecture. In particular, for each convolution (including both 1×1 convolution and 3×3 convolution), we split the input feature maps into N groups and apply the corresponding convolution with a smaller number of channels to each group. The resulting feature maps in each group will be concatenated together. This simple design can effectively reduce the number of parameters by a factor of N. To see this, we can think about an example that maps an input feature map with a channels to a feature map with b channels. Using a single convolution $k \times k$, the size of the filter is $k * k * a * b$. If we use group convolution with N groups, the size of each group filter is $k * k * \frac{a}{N} \times \frac{b}{N}$. As a result, the total size of all group filters is $k * k * a * b/N$. In our implementation, we utilize group convolutions with a factor of 4 for all 1×1 and 3×3 kernels, thus reducing number of parameters by 4 times compared to that without using group convolutions. In Figure 4.2(b), we show 4 group convolutions that are applied to all 1×1 and 3×3 convolutions. It is notable that if we set the number of groups equal to the number of input channels, it degenerates to depth-wise convolutions as used in Google's MobileNets [37].

4.3 The proposed SEP-Nets Structures

Built on the PRB, we design a new module for our SEP-Net, which is shown in Figure 4.3(f). Our SEP-Net module consits of a dimension reduction layer (1×1 convolutions), 2 PRB blocks with different output channles, and a dimension recovery layer (1×1 convolutions). The last recovery layer enables us to add the skip connection

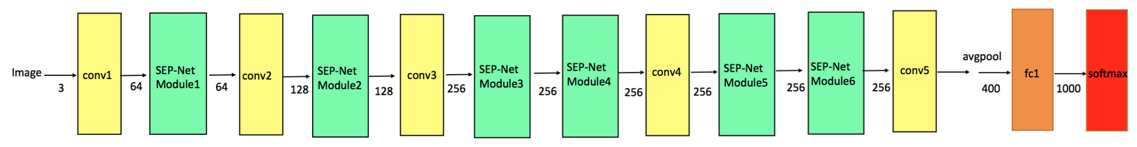

Figure 4.4. The architecture of our experimented SEP-Nets (the numbers illustrate the number of input/output channels of each block for one SEP-Net with 1.7M parameters)

as in ResNets, which is helpful for building up more layers. We also compare with other module designs of different networks in Figure 4.3. In particular, comparing with the Fire module of the SqueezeNet, the SEP-Net module does not use the filter concatenation as introduced in Inception Net. Comparing with the ResNet bottleneck module, we replace the 3×3 convolutional layers with the PRB blocks. Finally, we plot the architecture of our experimented SEP-Nets in Figure 4.4. The detailed information of our experimented SEP-Nets (filter size, number of output channels, pad and stride of convolutional layers) is delayed to the supplement.

4.4 Experimental Results

In this section, we first present experimental results on CIFAR-10 [47] and ImageNet dataset [15] to justify that pattern binarization could reduce the effective number of parameters dramatically and fine-tuning other parameters of the binarized network with fixed binarized pattern could achieve comparable performance to that of the original neural network models. Then, we show the designed **SEP-Net** structure could achieve better or comparable performance on ImageNet than using similar sized networks such as recently released Google MobileNets. We conduct all experiments using Caffe [43] open sourced library.

4.4.1 CIFAR-10

We first conduct experiments on the CIFAR-10 dataset [47], which has 50,000 training images and 10,000 test images. Each image belongs to one of 10 classes and has RGB format with 32x32 size in the original data set. The data is preprocessed by Global Contrast Normalization (GCN) and ZCA whitening [22] and also padded by 4 pixels on each side of image. In the training phase, 32x32 crop is randomly sampled from the padded image while in the test phase we only test on the original image. We start the learning rate from 0.1 and divide by 10 at iteration 32k and 48k and the maximum number of iteration is 64K. The momentum is 0.9 and the weight decay is 0.0001. We train on one GPU using mini-batch SGD with a batch size 256. We report the test accuracy from the original paper and neural network models trained by us from scratch. Note that the minor difference of performance between our results and that reported in the original paper might be due to data augmentation. We apply pattern binarization to several recent successful network structures: ResNet-20, ResNet-32, ResNet-44, ResNet-56 [29]. We first binarize 3x3 kernels in all convolutional layers of the obtained **Full** Model. For presentation purpose, we term the model after pattern binarization without fine-tuning as (**BiPattern** model). Then we do fine-tuning other parameters of **BiPattern** model fixing binarized pattern on the original Cifar10 data set to obtain **Refined** model. We report test accuracy from the original paper denoted by **Ref**, and of the **Full** model, the **BiPattern** model and the **Refined** model in the Table 4.2. We can see from Table 4.2 that the test accuracy

Table 4.2. Comparison on the-state-of-art models, **Full** model, **BiPattern** model, **Refined** model on **Cifar10** Dataset

Model	Acc	**Ref**	**Full**	**BiPattern**	**Refined**
ResNet-20	Top-1	0.9125	0.9118	0.1546	0.8649
	Top-5	-	0.9974	0.5104	0.9941
ResNet-32	Top-1	0.9249	0.9276	0.2634	0.9021
	Top-5	-	0.9972	0.6932	0.9962
ResNet-44	Top-1	0.9283	0.9283	0.4825	0.9145
	Top-5	-	0.9982	0.8765	0.9965
ResNet-56	Top-1	0.9303	0.9375	0.5382	0.9302
	Top-5	-	0.9977	0.9574	0.9971

of **Refined** models are very close to that of Full models, which justify that 3x3 filters could be replaced with binary pattern extractions without sacrificing performance.

In Table 4.3 we compare the effective number of parameters between full ResNet-20, ResNet-32, ResNet-44, ResNet-56 models and their corresponding binarized pattern networks. The effective number of parameters of the pattern network is referred to as the number of parameters that use floating point representations. In particular, we use one number to represent the binarized 3×3 or 5×5 kernels by the scaling factor. From Table 4.3 we can see that the number of parameters is reduced dramatically. For example the number of parameters in corresponding pattern network is reduced 86% compared to original ResNet-56 model.

4.4.2 ImageNet

We also carry out experiments on the ImageNet 2012 classification data set [15], which has 1.28 millions of training images and 50k validation images. Each image

Table 4.3. Comparison of the number of parameters between **Full** networks and **Pattern** networks. We use one number to represent a binarized 3×3 or 5×5 kernel.

Model	**Full Network**	**Pattern Network**
ResNet-20	292K	55K
ResNet-32	487K	78K
ResNet-44	682K	100K
ResNet-56	876K	123K

belongs to one of 1000 classes. We apply the same pattern binarization procedure used to ResNet on CIFAR-10 experiment to GoogLeNet [86] and our customized Inception-net (denoted as C-InceptionNet) that removes all computationally expensive 5×5 convolutional kernels . For training GoogLeNet, we adopt the learning strategy from Caffe website and the learning rate follows a polynomial decay with the initial learning rate being 0.01, momentum term 0.9 and weight decay 0.0002. We train GoogLeNet with a maximum number of iterations 600K on one GPU and a batch size 128. For training C-InceptionNet, we set initial learning rate as 0.1, divide learning rate 10 time after every 24 epochs, and train in total 90 epochs. At the end, we compare test accuracy of the **Full** model, the **BiPattern** model, and the **Refined** model corresponding to GoogLeNet and C-InceptionNet.

To illustrate the effect of binarizing different size filters, we report several results on GoogLeNet. We first binarize 3×3 filters fixing 5×5 filters. Alternatively, we binarize 5×5 filters meanwhile fixing 3×3 filters. For comparison, we also conduct experiment by a) binarizing both 3×3 and 5×5 filters; b) binarizing only 1×1 kernels. For C-InceptionNet, we only show the result of binarizing all 3 filters. We present comparison results including test performance on **Refined** model using

Table 4.4. Comparison on the-state-of-art models, **Full** model, **BiPattern** model, **Refined** model on **ImageNet** Dataset, including test accuracy from **Refined** model with multicrop. **C-InceptionNet**: customized Inception-Net removing all 5×5 convolutions.

Model	Acc	**Ref**	**Full**	**BiPattern**	**Refined**	**Multicrop**
GoogLeNet				1x1 pattern:		
				0.0013	0.6117	0.636
				0.0075	0.8395	0.856
				2-8 3x3 pattern:		
				0.3706	0.6797	0.6893
	Top-1	-	0.6865	0.6290	0.8827	0.8898
	Top-5	0.8993	0.8891	5x5 pattern:		
				0.5141	0.6917	0.6984
				0.7619	0.8904	0.8965
				3x3 & 5x5 pattern:		
				0.1428	0.6694	0.6812
				0.31738	0.8763	0.8844
C-InceptionNet	Top-1		0.648	0.0476	0.6400	0.6521
	Top-5		0.863	0.1464	0.8550	0.8626

multicrop [48] in Table 4.4, from which we could see: i) for C-InceptionNet, the performance of the **Refined** model is competitive to the full model, just 0.8% less than the full model on Top-1 accuracy; ii) for GoogLeNet, the performance of the **Refined** models of binarizing $k \times k$ filters ($k > 1$) is significantly competitive to the full model; iii) binarizing 1×1 kernels suffers from more performance loss compared to binarizing $k \times k$ kernels ($k > 1$), which justifies that our choice of binarizing $k \times k$ kernels ($k > 1$) while retaining 1×1 kernels. All the performance numbers are based on a single center crop when performing testing except for the last column, which is included for future reference.

In Table 4.5, we compare the number of parameters of the full GoogLeNet and C-InceptionNet and their corresponding pattern networks. We can see that the

Table 4.5. Comparison of number of parameters between **Full** networks and **Pattern** networks. We use one number to represent a 3×3 or 5×5 kernel, resulting in parameter reduction.

Model	**Full Network**	**Pattern Network**	
GoogLeNet	6.99M	3×3	4.43M
		5×5	6.43M
		3×3 and 5×5	3.87M
Customized-InceptionNet	5.10M		2.43M

pattern networks have dramatically less number of parameters that that of the full models. For example through binarizing 3×3 and 5×5 filters, we reduce the number of parameters by 44.6% compared to that of the full GoogLeNet.

4.4.3 Comparison with The-State-Of-Art

Finally, we compare the proposed **SEP-Net** with the SqueezeNet and the recently released MobileNet in terms of model size and classification accuracy. We present the performance of several variations of our SEP-Net architectures: SEP-Net-R, SEP-Net-B, SEP-Net-BQ. Here SEP-Net-R is our SEP-Net with raw filters. SEP-Net-B denotes SEP-Net with pattern binarization. SEP-Net-BQ further quantizes all other parameters to 8 bits.

As shown in Table 4.6, we have trained two extremely small SEP-Nets suited for mobile/embedded devices. One model has 1.3M parameters, while the other has 1.7M parameters. The two SEP-Nets share the same neural network structure as Figure 4.4. The difference between the two SEP-Nets is: (1) in the SEP-Net with 1.7M parameters the last convolution layer uses a factor of 4 for group-wise convolution while the SEP-Net with 1.3M parameters uses a factor of 16 for the last

Table 4.6. Comparison between the designed **SEP-Net** and the MobileNet and the SqueezeNet in terms of model size and Top1 accuracy. **SEP-Net-R**: SEP-Net with raw valued weights. **SEP-Net-B**: SEP-Net with pattern binarization. **SEP-Net-BQ**: SEP-Net with pattern binarization and other weights quantized using linear quantization with 8 bits.

Model	Parameter number	Size (bytes)	Top-1 Acc
MobileNet	1.3M	5.2MB	0.637
	2.6M	10.4MB	0.684
SEP-Net-R	1.3M (small)	5.2MB	0.658
	1.7M (large)	6.7MB	0.667
SqueezeNet [40]	1.2M	4.8MB	0.604
MobileNet [37]	1.3M	5.2MB	0.637
SEP-Net-R (Small)	**1.3M**	**5.2MB**	**0.658**
SEP-Net-B (Small)	1.1M	4.2MB	0.637
SEP-Net-BQ (Small)	1.1M	1.3MB	0.635

layer group-wise convolution; (2) the output dimension of the last convolutional layers of the SEP-Net with 1.7M parameters is 400 while that of the SEP-Net with 1.3M parameters is 512. They produce 65.8% and 66.7% top 1 classification accuracy on the ImageNet dataset, respectively. Our small model outperforms MobileNet's equivalent with the same model size. Our larger model increases number of parameters by 0.3M, boosting the performance by 2%.

For easy comparison to other small CNNs, we also present the memory size of different models. Our **SEP-Net-B** reduces the model size to 4.2MB with slightly decreased accuracy of 63.7% that equals to the performance of the MobileNet with 5.2MB. Our **SEP-Net-BQ** further reduces the storage or memory cost to 1.3MB while maintaining roughly the same performance. It indicates that our extremely compact model also works with standard compression techniques.

CHAPTER 5

DESIGNING NEURAL NETWORK STRUCTURES AUTOMATICALLY

In this chapter we discuss designing neural network structure automatically by exploring aggressive genetic approach since designing appropriate neural netowrk structures requires extensive amount of human effort for a centain task. In Sec 5.1 we study genetic approach for designing neural network structures. In Sec 5.2 we discuss the proposed aggressive selection and multiple cloning. Following that we present all defined mutations operation in Sec 5.3. Training strategy and mutation operation sampling are given in Sec 5.4 and Sec 5.5. Finally, we report some experimental results of the proposed aggressive genetic programming approach for optimizing convolutional neural network structures in Sec 5.6.

5.1 Genetic approach

The proposed algorithm follows the standard flow of neuro-evolution, i.e., population initialization, individual selection, reproduction, mutation/cross-over, and fitness score evaluation. The individuals in the initial population are simple neural network structures with only one global pooling layer or one fully connected layer. We use an acyclic graph to encode an individual with each node in the graph representing a basic operation or connection including *convolution, pooling, fully connected, concatenation and skip.* Those operations are standard in the neural network literature. Please refer to Figure 5.2 for examples of individuals represented by a graph. We also use the prediction performance on a validation data as fitness score.

However, different from [95, 71, 17], we are not only exploring how to implement a genetic/evolutionary algorithm for optimizing deep convolutional neural networks, but also exploring how to reduce the computational costs under the framework of neuro-evolution without imposing strong restriction on the search space. Next, we will present our strategies for reducing the computational costs.

5.2 Aggressive selection and multiple cloning

A potential issue in traditional selection strategies (e.g., tournament selection or sampling-based selection) is that weak individuals might survive for a long period. While this feature is helpful to increase the diversity of the population, however it may waste a lot of time to train these weak individuals that will eventually be eliminated. We propose to eliminate these weak individuals at their very early age, and use other approaches to increase the diversity of the population. The algorithmic description of the proposed aggressive selection is presented in Algorithm 5.1, and an illustration of the proposed selection process is presented in Figure 5.1. In particular, we greedily select the top k individuals from a population of individuals $\mathcal{P}_{t-1}$ based on their fitness scores. To encourage the diversity, we also make sure the distance between the selected top individuals exceeds a certain threshold. The distance between two individuals is computed by comparing the nodes of two graph structures from input layer to output layer. If the nodes are represented by an alphabet denoting their operation or connection types, the distance is simply the hamming distance. It is notable that the distance between individuals is also considered by fitness sharing in

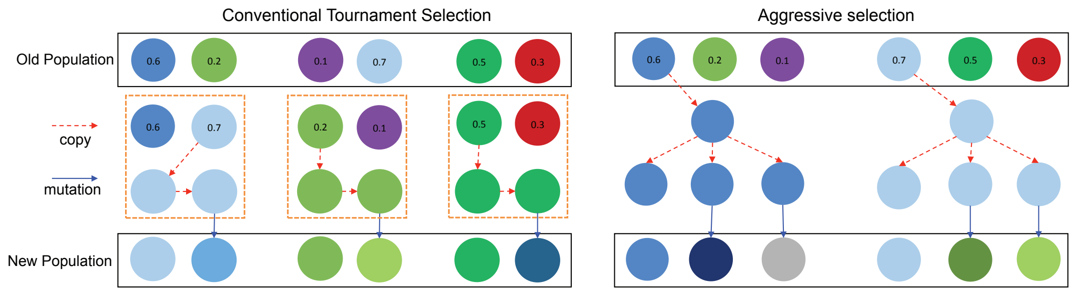

Figure 5.1. Comparison between the proposed aggressive selection and mutation strategy (right) vs conventional tournament selection and mutation strategy (left). Each colored ball denotes an individual, the number within each ball denotes its fitness score, red dashed arrows denote a copy and green solid arrows denote mutations.

previous studies [21] to encourage diversity.

Multiple Cloning. With the aggressive selection strategy described above, we can eliminate many weak individuals at their early ages. However, the small number of retained individuals will reduce the size of the next population and thus restrict diversity. To address this issue, we will resort to cloning, i.e., making multiple copies of the selected individuals to undergo different mutations for generating the next population. For comparison, in traditional tournament selection as illustrated in Figure 5.1, a weak individual might be selected and each survived individual only undergoes one mutation, which is the strategy adopted in [71, 17]. In conventional sampling-based selection and mutation, each individual has a certain probability to be retained and the retained individual has a certain probability to be mutated. This is the strategy adopted in [95], where the mutation probability is set to a small value (e.g. 0.05). We can see that the proposed selection and mutation strategy is more

Algorithm 5.1 Aggressive Selection of top-k individuals

```
1: **Input**: a population of individuals ranked according to their fitness score from
   large to small, $\mathcal{P}_{t-1} = \{i_1, \ldots, i_N\}$. A target number of individuals $k$ and a
   distance threshold $d$.
2: Initialize an empty set $\mathcal{P}_t$
3: for $j = 1, 2, \ldots, N$ do
4:     choose the next individual $i_j$ in $\mathcal{P}_{t-1}$
5:     if the distance between $i_j$ and individuals in $\mathcal{P}_t$ exceeds a certain threshold $d$
       then
6:         add $i_j$ into $\mathcal{P}_t$
7:     end if
8:     if the size of $\mathcal{P}_t$ is equal to $k$ then
9:         return $\mathcal{P}_t$
10:    end if
11: end for
```

aggressive than the existing works in that only a small number of strong individuals are retained and each survived individual reproduces themselves to undergo more mutations for potential growth in the fitness.

5.3 Mutation operations

To complement our proposed aggressive selection and mutation, we increase the number of possible mutation operations compared to the existing works [71, 17, 95]. We define 15 different types of mutation operations as shown in Table 5.1, which almost doubles the amount considered in [71]. Note that three mutation operations (`reset_weight`, `continue_training`, `alter_learning_rate`) appeared in [71] are not included in Table 5.1 since those operations do not change the structure of a neural network. Next, we provide more details regarding each mutation operation. In the following, we discuss some implementation details of each mutation operation. We are going to focus on the `add` operations. For all `removal` operations, we randomly select and remove one of existing layers of the chosen type. If no such layer exists, no operation will be applied.

- `add_convolution`: Firstly, we randomly select the position to add a convolution layer. Then we insert a convolution layer with channel number 32, stride 1, filter size 3×3, and number of padding pixel 1. For simplicity, those values are chosen to ensure the input and output of the feature map dimensions do not change after the convolution. Note that even though we use a predefined set of channel number, stride and filter size, those values could be altered later through `alter_channel_number`, `alter_stride`, `alter_filter_size` mutation operations, which we will discuss shortly. In this work, a convolutional layer is by default followed by batch normalization [41] with Relu [48] activation unit.

- `alter_channel_number, alter_stride, alter_filter_size`: These three types of mutation operations are to reset the hyper-parameters in a convolution layer. We randomly choose a new value for the corresponding hyper-parameter from a predefined list, i.e., $\{8, 16, 32, 48, 64, 96, 128\}$ for channel numbers, $\{1 \times 1, 3 \times 3, 5 \times 5\}$ for filter sizes, and $\{1, 2\}$ for strides.

- `add_skip`: A skip layer, illustrated in Figure 5.2, is to implement the skip connection introduced in residual networks [29]. Since a skip layer requires its two bottom layers to share the same feature map dimension and channel number, we first find out all pairs of layers which could potentially be bottom layers of the skip layer. Then, a skip connection is added on top of a randomly selected pair from all possible pairs.

- `add_concatenate`: Similar to skip layer, a concatenate layer requires two bottom layers to share the same feature map dimension, but they could have different channel numbers. Thus, the `add_concatenate` mutation follows a similar procedure as `add_skip` mutation.

- `add_pooling`: Here, we restrict the insertion of a pooling layer such that it can only take place right after a convolution layer. For simplicity, we limit the pooling strategy to be max pooling and kernel size to 2×2 with stride 2. This predefined pooling configuration could be relaxed in future work.

- `add_fully_connected`: For this operation, we limit its position to be the last layer or immediately following another fully connected layer. The output dimension of this inserted layer is uniformly chosen from the following set $\{50, 100, 150, 200\}$.

- `add_dropout`: For this operation, we limit its position to be immediately after a fully connected layer. For simplicity, we set dropout ratio to 0.5.

Note that applying some mutation operations (e.g., `alter_stride`, `add_pooling`) may result in inconsistency of feature map dimensions. In this situation, we will adopt the following strategies to address the above issue: i) adding additional padding pixel; ii) adding a 1×1 convolution layer to adjust channel numbers. If it still results in an invalid network structure, we simply apply a mutation again.

In Figure 5.2, we show an example of how a neural network structure evolves to new ones after undergoing some mutation operations: `add_convolution, add_concatenate,`

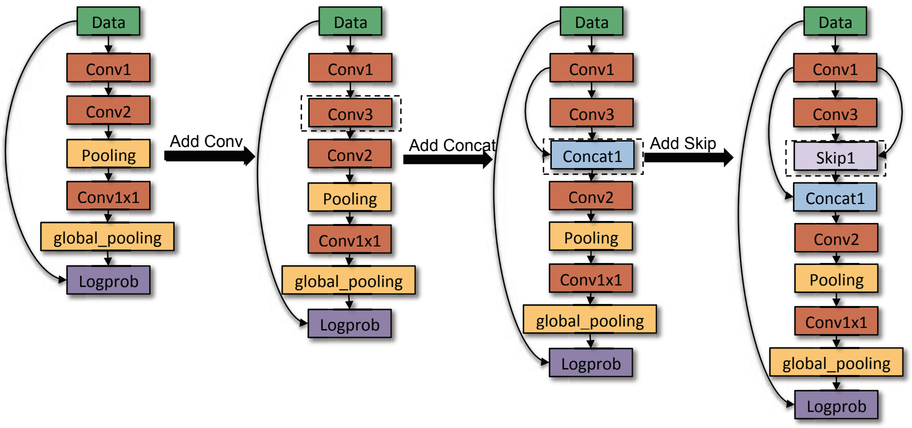

Figure 5.2. Example of how a neural network structure mutates to new ones after undergoing different mutation operations: `add_convolution`, `add_concatenate` and `add_skip`. The dotted squares mark added convolution layer "Conv3", concatenate layer "Concat1" and skip layer "Skip1".

`add_skip`. We use the dotted square to mark the new layers. Clearly, we could see that as network evolves, we can explore the diverse neural network structures and obtain neural network structure with potential better performance.

5.4 Training strategy

To evaluate the fitness score of each individual, a standard approach is to use existing optimization algorithms off-the-shelf to learn the weight parameters. Training a CNN may take tens of thousands gradient descent iterations to achieve a good local minimum. However, we observe that "deep" training (i.e., setting a very stringent condition for stopping the training process) is not necessary during the evolution since our goal is to have a ranked list of individuals for selection. Therefore, a rough estimate of the prediction performance for each individual is sufficient for driving the evolution.

Mutations	[71]	Ours
`add_convolution`	✓	✓
`remove_convolution`	✓	✓
`alter_channel_number`	✓	✓
`alter_filter_size`	✓	✓
`alter_stride`	✓	✓
`add_dropout`	-	✓
`remove_dropout`	-	✓
`add_pooling`	-	✓
`remove_pooling`	-	✓
`add_skip`	✓	✓
`remove_skip`	✓	✓
`add_concatenate`	-	✓
`remove_concatenate`	-	✓
`add_fully_connected`	-	✓
`remove_fully_connected`	-	✓

Table 5.1. The allowed mutation operations in our work and in [71]; ✓represents that mutation operation is defined while - represents not available

To reduce the training time of each individual during the evolution, we explore a different learning rate decay strategy to train a deep neural network. There are two popular strategies for decaying the learning rate. One method is an inverse learning rate decay strategy [42], where the learning rate η_t at the t-th iteration is set to

$$\eta_t = \eta_0 * (1 + \gamma * t)^{-\alpha} \tag{5.1}$$

where η_0 is the initial step size and γ, α are the hyper-parameters. Another popular method is a multi-stage strategy [48], where the learning rate is reduced by a fixed factor (e.g., 10) after a large number of iterations. We use a mixture of both strategies. We divide our training process into three stages with a maximum number of 20000 iterations: with the first stage being the first 10000 iterations, from 10000 to 15000 iterations as the second stage, and the last 5000 iterations as the final stage. Within each stage, we use the inverse learning rate strategy. The learning rate is reduced by a fixed factor after each stage. This strategy avoids running a large number of iterations with the same step size without improving the prediction performance much at each stage, and also quickly gives a rough estimate of the prediction performance without spending long time at the tail of the learning curve that has little improvement on the prediction performance. Finally, after the neuro-evolution process terminates with a good structure, we switch to existing optimization algorithms for deep training.

5.5 Mutation operation sampling

To speed up the evolution process, we also use non-uniform sampling probabilities for choosing a mutation operation. Using uniform probabilities to choose a mutation operation will waste lots of time training weak individuals that are mutated by removing convolution, skip, concatenation from their parents in the early stage of evolution process. To avoid this issue, we explicitly set the sampling probabilities of `add_convolution`, `add_skip`, `add_concatenate`, `alter_ stride` `alter_filter_size`, and `alter_channel_number` two times larger than that of other mutation operations at the earlier stage of the evolution process.

5.6 Experimental Results

Datasets and Preprocessing: We conduct experiments on four benchmark datasets: MNIST [50], SVHN [65], CIFAR-10 [47] and CIFAR-100 [47]. MNIST dataset contains $60,000$ training images and $10,000$ test image where each gray-scale image contains one of the 10 digits, 0 to 9. CIFAR-10 dataset [47] has 50,000 training images and

10,000 test images. It contains 10 classes and each RGB image has a size of 32×32. The data is preprocessed by applying a Global Contrast Normalization (GCN) and ZCA whitening [22] and each side is padded with four pixels. In the training phase, a 32×32 patch is randomly cropped from the padded image while in the test phase the original images are used. CIFAR-100 dataset is similar to CIFAR-10 dataset but has 100 classes in total. SVHN is a street view house number dataset which contains about $73,257$ training images and $26,032$ test images.

Experiment configuration: In our experiment, the population size is set to be 10. Given a population of 10 individuals (which are clones of top k individuals in intermediate generations), we let each individual undergo a mutation, and then select top k individuals from the 10 mutated individuals and the original 10 individuals. A new population will be created by making equal number of clones of the selected individuals to reach the population size 10. It is worth mentioning that even though we use such a small population size, our performance is competitive and even better than [17, 95], in which the population size is set to 100 and 20, respectively. It is expected that using a larger population size will further increase our performance according to [71]. We use mini-batch Stochastic Gradient Descent (SGD) to train each individual neural network for a maximum of 20,000 iterations with a momentum 0.9. The mini-batch size is fixed to be 128. The weight decay is set to be 0.0005. The learning rate strategy is described in subsection 5.4. The initial learning rates for the three stage are set to be $10^{-1}, 10^{-3}, 10^{-5}$, respectively. The parameters in (5.1) are

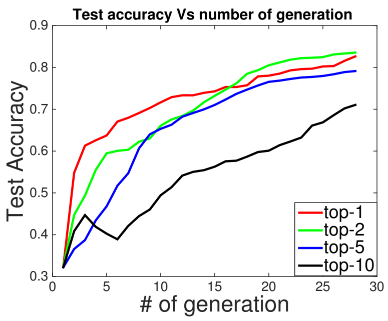

Figure 5.3. The test accuracy of the best individual among the selected top k individuals vs the number of generations on CIFAR-10 dataset, where different curves correspond to aggressive selection with different values of k.

set to $\gamma = 0.001$ and $\alpha = 0.75$. The distance threshold in aggressive selection is set to be 1. In our experiments, one evolution process is always run on one GPU.

5.6.1 The effect of aggressive selection

Here, we present the evidence that the proposed aggressive selection strategy can dramatically speed up the evolution process. The following experiments are conducted in the CIFAR-10 dataset. In Figure 5.3, we plot the evolved network performance under four different values of $k = 1, 2, 5, 10$ used in our aggressive selection strategy. The smaller the k value, the more aggressive the selection strategy. For each experiment setting, we plot the test accuracy of the best individual among the selected top k individuals from each generation. We observe that aggressive selection with smaller values of k (e.g., 1 and 2) evolves faster than non-aggressive selection using larger values of k (e.g., $5, 10$). Due to limitation of space, we include more results in supplementary materials to justify the proposed aggressive selection.

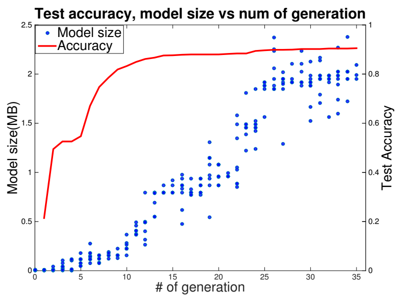

Figure 5.4. The evolution of model size and test accuracy of the best individual in **AG-Evolution** algorithm on CIFAR-10.

5.6.2 Comparison with existing methods

In this section, we compare the performance of the proposed aggressive genetic programming approach with existing genetic approaches on benchmark datasets MNIST, SVHN, CIFAR-10, CIFAR-100. We refer to the genetic approaches presented in [17, 95, 71] as **EDEN**, **Genetic-CNN** and **LS-Evolution**, respectively. For our method, we report the result using aggressive selection with $k = 1$, referred to **AG-Evolution**. For reference and comparison, we also include state-of-the-art results based on hand-crafted neural network structures (referred them as **SOTA**) as well as the results using non-aggressive selection (i.e., by setting $k = 10$ in our framework), which is referred to as **NA-Evolution**.

For **NA-Evolution** and **AG-Evolution**, we terminate the evolution process when the performance on the validation data saturates on all datasets except on the CIFAR-100 dataset, in which we terminate the process earlier. It is possible that by continuing the evolution process, the performance might be further improved. We would also like to emphasize that state-of-the-art results could be attributed to not only a good network structure but also some other factors (e.g., using a good pooling function [51]), which are not considered in current genetic approaches. For **Genetic-CNN**, we do not compare with the results in their Table 3 for SVHN, CIFAR-10 and CIFAR-100. The reason is that their results in Table 3 are not directly achieved by the evolution process. They re-trained the networks by using large number of filters, which are not exactly the networks found by their genetic algorithm.

Table 5.2, 5.3, 5.4, 5.5 show our results on different datasets. It is notable that on some datasets the results of **EDEN**, **Genetic-CNN** and **LS-Evolution** are missing, which is because they are not reported in their original papers or because of the reason menioned before. For each method, we report both the test accuracy and the computational cost measured by the total number of used GPU hours (GPUH) if available. From the results, we have the following observations.

- First, the performance of the discovered neural network structures by our genetic approach **AG-Evolution** on MNIST and SVHN datasets is very close to the state-of-the-art results.

- Second, compared with **EDEN** [17], **AG-Evolution** achieves better performance on MNIST and CIFAR-10, and compared with **Genetic-CNN** [95], **AG-Evolution** can find a much better neural network on CIFAR-10 (0.9052 vs 0.7706 for test accuracy) with much less time (72GPH vs 408GPUH).

- Third, compared with the **LS-Evolution** [71] on CIFAR-10 data, which achieves a test accuracy of 0.9180 with 17971 GPUH, our **AG-Evolution** achieves a sim-

Approach	**Test Acc**	**Comp Cost**
SOTA [91]	0.9979	–
Genetic-CNN [95]	0.9966	48 GPUH
EDEN [17]	0.9840	–
AG-Evolution	0.9969	35 GPUH

Table 5.2. Comparison of test accuracy and computational cost on MNIST dataset. **NA-Evolution** is not run on this dataset due to that **AG-Evolution** almost achieves the same performance as the state-of-the-art.

ilar performance of 0.9052 with much less time, i.e., 72GPUH. It is expected that by continuing our evolution process, we might achieve similar test accuracy to 0.9460 but with less amount of time.

- Finally, **AG-Evolution** uses a much shorter time to find network structures achieving almost similar performance to **NA-Evolution** with a much longer time on SVHN, CIFAR-10 and CIFAR-100 datasets, which further verifies the benefit of the proposed aggressive selection.

In Figure 5.5, we plot the test accuracy of the best individual in each generation versus the number of generations on the four datasets for **AG-Evolution**, from which we could see the proposed genetic approach gradually improves the performance of neural network structures. We also report the evolution of model size of individuals on CIFAR-10 dataset in Figure 5.4.

Approach	**Test Acc**	**Comp Cost**
SOTA [51]	0.9831	–
NA-Evolution	0.9620	552 GPUH
AG-Evolution	0.9541	60 GPUH

Table 5.3. Comparison of test accuracy and computational cost on SVHN dataset.

Approach	**Test Acc**	**Comp Cost**
SOTA [38]	0.9650	-
LS-Evolution [71]	0.9460	65,536 GPUH
LS-Evolution [71]	0.9180	17,971 GPUH
Genetic-CNN [95]	0.7706	408 GPUH
EDEN [17]	0.7450	–
NA-Evolution	0.9037	552 GPUH
AG-Evolution	0.9052	72 GPUH

Table 5.4. Comparison of test accuracy and computational cost on CIFAR-10 dataset.

Approach	**Test Acc**	**Comp Cost**
SOTA [38]	0.8280	-
LS-Evolution [71]	0.7700	>65,536 GPUH
NA-Evolution	0.6560	552 GPUH
AG-Evolution	0.6686	136 GPUH

Table 5.5. Comparison of test accuracy and computational cost on CIFAR-100 dataset.

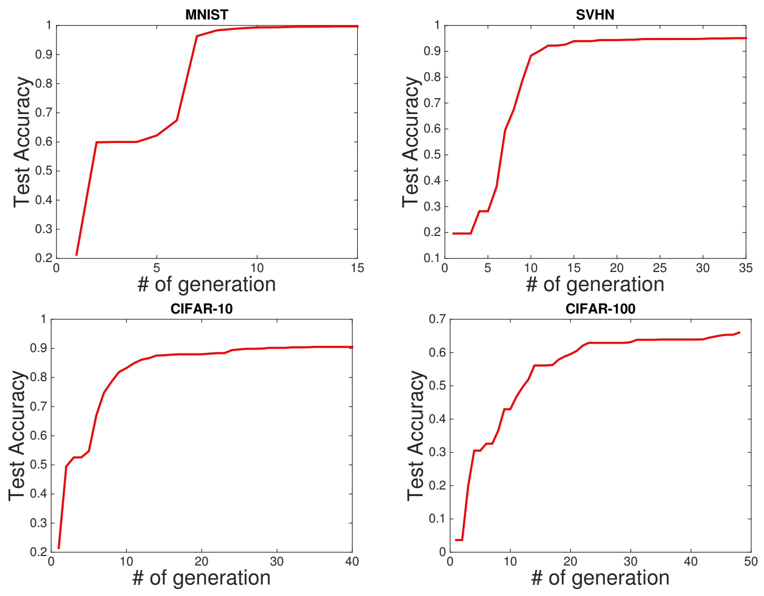

Figure 5.5. Test Accuracy vs Generation Number on MNIST, SVHN, CIFAR-10, CIFAR-100 dataset

CHAPTER 6

DESIGNING THE IMPROVED NEURAL NETWORKS

In this chapter, we propose an ecologically inspired genetic approach for neural network structure search, that includes two types of succession: primary and secondary succession as well as accelerated extinction and gene duplication. Specifically, we first use primary succession to rapidly evolve a community of poor initialized neural network structures into a more diverse community, followed by a secondary succession stage for fine-grained searching based on the networks from the primary succession. Extinction is applied in both stages to reduce computational cost. Extensive experimental results show that our proposed approach can achieve the similar or better performance compared to the existing genetic approaches with dramatically reduced computation cost. For example, the network discovered by our approach on CIFAR-100 dataset achieves 78.1% test accuracy under 120 GPU hours, compared to 77.0% test accuracy in more than 65, 536 GPU hours in [71].

6.1 Introduction

Recently, there are emerging research works [71, 95, 107, 108, 103, 3] on automatically searching neural network structures for the image recognition tasks. In this paper, we focus on optimizing the evolution-based algorithms [95, 62, 57, 84] for network structures searching as the existing works suffer from either one of the following issues: prohibitive computational cost or unsatisfied performance compared with hand-crafted network structures. In [71], it costs more than 256 hours on 250 GPU for searching neural network structures, which can not be affordable by general users. In [95], the final learned network structure by their genetic approach achieves about 77% test accuracy on CIFAR-10, even though better performance as 92.9% could be obtained after fine-tuning certain parameters and modifying some structures on the discovered network. In previous chapter, we firstly aim to achieve the better performance with the reduced computational cost by the proposed aggressive selection strategy in genetic approach and more mutations operation to increase diversity which is decreased by the proposed selection strategy. In our previous work, we reduce computational cost dramatically from more than $65,356$ GPU hours (GPUH) to few hundreds GPUH. However, our approach still suffers performance sacrifice, for example, 90.5% test accuracy compared to 94.6% test accuracy from [71] on CIFAR-10 dataset.

Along our previous research line, in this chapter, we study the genetic approach to achieve the better test performance compared to [71] or competitive performance to hand-crafted network structures [29] under limited computation cost as the previous chapter and without the pre-designed architectures introduced by human [57]. Inspired from primary, secondary succession from ecological system [73], we enforce a poor initialized community of neural network structures to rapidly evolve to a community containing network structures with dramatically improved performance. After the first stage of primary succession, we perform the fine-grained search for better networks in a community during the secondary succession stage. In addition, we also introduce the gene duplication to further utilize the novel block of layers that appeared in the discovered network structure.

6.2 The proposed method

Our genetic approach for searching the optimal neural network structures follows the standard procedures: i) initialize population in the first generation with simple network structures; ii) evaluate the *fitness* score of each neural network structure (fitness score is the measurement defined by users for their purpose such as validation accuracy, number of parameters in network structure, number of FLOP in inference stage, and so on); iii) apply a selection strategy to decide the survived network structures based on the fitness scores; iv) apply mutation operations on the survived *parent* network structures to create the *children* networks for next generation. The last three steps are repeated until the convergence of the fitness score. Note that in our genetic approach, the *individual* is denoted by an acyclic graph with each node representing a certain layer such as *convolution*, *pooling* and *concatenation* layer. A *children* network can be generated from a *parent* network through a mutation procedure. A *population* includes a fixed number of networks in each generation, which is set as 10 in our experiments. For details of how to use genetic approach to search neural network structures, we refer the readers to the previous chapter. In the following, we discuss our approach which applies the ecological concepts of succession, extinction and gene duplication to the genetic approach for an accelerated search of neural network structures.

6.2.1 Evolution under Primary and Secondary Succession

Our inspiration comes from the fact that in an ecological system, the community is dominated by diversified fast-growing individuals during the primary succession, while in the secondary succession, the community is dominated by more competitive individuals [73]. Therefore, we treat all the networks during each generation of the evolution process as a *community* and treat the individual networks in each generation as a population, instead of focusing on evolving a single network [71].

With this treatment, we propose a two-stage *rapid succession* for accelerated evolution, analogous to the ecological succession.

The proposed rapid succession includes a primary succession, where it starts with a community consists of a group of poorly initialized individuals which only contains one global pooling layer, and a secondary succession which starts after the primary succession. In the primary succession, a large search space is explored to allow the community grow at a fast speed, and a relatively small search space is used in the secondary succession for fine-grained search.

To depict the exploration space at each generation, we define *mutation step-size* m as the maximum mutation iterations between the parent and children. The actual mutation step for each child is uniformly chosen from $[1, m]$. In the primary succession, in order to have diversified fast-growing individuals, a large mutation step-size is used in each generation so the mutated children could be significantly different from each other and from their parent. Since we only go through the training procedure after finishing the entire mutation steps, the computation cost for each generation will not increase with the larger step-size. In the secondary succession, we adopt a relative small mutation step-size to perform a fine-grained search for network structures.

Each mutation step is randomly selected from the nine mutations operations, for completeness, we list those as the followings:

- `add_concatenation`: A concatenation layer is randomly inserted into the network where two bottom layers share the same size of feature maps.
- `add_pooling`: A pooling layer is randomly inserted into the network with kernel size as 2×2 and stride as 2.
- `remove_convolution`,`remove_concatenation`,`remove_pooling`: The three operations randomly remove a convolutional layer, a concatenation layer and a pooling layer, respectively.
- `alter_channel_number`, `alter_stride`, `alter_filter_size`: The three operations modify the hyper-parameters in the convolutional layer. The number of channels is randomly selected from a list of $\{16, 32, 48, 64, 96\}$; the stride is randomly selected from a list of $\{1, 2\}$; and the filter size is randomly selected from $\{1 \times 1, 3 \times 3\}$.

6.2.2 Speeding up Extinction

During the succession, we employ the idea from previous chapter that only the best individual in the previous generation will survive. However, instead of evaluating the population in each generation after all the training iterations, it is more efficient to extinguish the individuals that may possibly fail at early iterations, especially during the primary succession where the diversity in the population leads to various performances. Based on the assumption that a better network should have better fitness score at earlier training stages, we design our extinction algorithm as follows.

To facilitate the presentation, we denote n as the number of population in each generation, T_1 and T_2 as the landmark iterations, f_{g,i,T_1} and f_{g,i,T_2} as fitness score (validation accuracy used in our work) of the i^{th} network in the g^{th} generation after training T_1 and T_2 iteration, v_{g,T_1} and v_{g,T_2} as threshold to eliminate the weak networks at T_1 and T_2 iterations in the g^{th} generation. In the g^{th} generation, we have fitness scores for all networks $\mathcal{F}_{g,T_1} = \{f_{g,i,T_1}, i = 1, \cdots, n\}$ and $\mathcal{F}_{g,T_2} = \{f_{g,i,T_2}, i = 1, \cdots, \hat{n}\}$ after training T_1 and T_2 iterations, respectively. Note that $\hat{n}$ can be less than n since the weak networks are eliminated after T_1 iterations. The thresholds v_{g,T_1} and v_{g,T_2} are updated at g^{th} iteration as

$$v_{g,T_1} = \max\Big(S(\mathcal{F}_{g,T_1})_p,\ v_{g-1,T_1}\Big) \tag{6.1}$$

and

$$v_{g,T_2} = \max\Big(S(\mathcal{F}_{g,T_2})_q,\ v_{g-1,T_2}\Big) \tag{6.2}$$

where $S(.)$ is a sorting operator in decreasing order on a list of values and the subscribes p and q represents p^{th} and q^{th} value after the sorting operation, p and q are the hyper-parameters.

For each generation, we perform the following steps until the convergence of the fitness: (i) train the population for T_1 iterations, extinguish the individuals with fitness less than v_{g,T_1}; (ii) train the remaining population for T_2 iterations, and

Algorithm 6.1 Algorithm for Extinction

```
1: **Input:** $T_1$, $T_2$, $v_{0,T_1}$, $v_{0,T_2}$, $p$, $q$
2: **for** $g = 1 \cdots , G$ **do**
3:     Obtain $\mathcal{F}_{g,T_1} = \{f_{g,i,T_1}, i = 1, ..., n\}, n = 10$ by training all individuals for $T_1$ iterations
4:     Update $v_{g,T_1}$ based on Eq. 6.1
5:     Extinguish the individuals with fitness value less than $v_{g,T-1}$
6:     Obtain $\mathcal{F}_{g,T_2} = \{f_{g,i,T_1}, i = 1, ..., \hat{n}\}$ by training the remain individuals for $T_2$ iterations
7:     Update $v_{g,T_2}$ based on Eq. 6.2
8:     Extinguish the individuals with fitness value less than $v_{g,T_2}$
9:     Train the remain individuals for $T_3$ iterations and select the best one as parent
10: **end for**
```

distinguish the population with fitness less than v_{g,T_2}; (iii) the survived individuals are further trained till convergence and the best one is chosen as the parent for next generation. The details for the extinction algorithm are described in Algorithm 6.1.

6.2.3 Gene Duplication

During the primary succession, the rapid changing of network architectures leads to the novel beneficial structures decoded in DNA [71] that are not shown in the previous hand-designed networks. To further leverage the automatic discovered structures, we propose an additional mutation operation named *duplication* to simulate the process of gene duplication since it has been proved as an important mechanism for obtaining new genes and could lead to evolutionary innovation [101]. In our implementation, we treat the encoded DNA as a combination of blocks where each block includes the layers with the same size of feature maps. As shown in Figure 6.1, the optimal structure discovered from the rapid succession could mutate into different networks by combining the blocks in several ways through the duplication.

We duplicate the entire block instead of single layer because the block contains the beneficial structures discovered automatically while simple layer copying is already an operation in the succession.

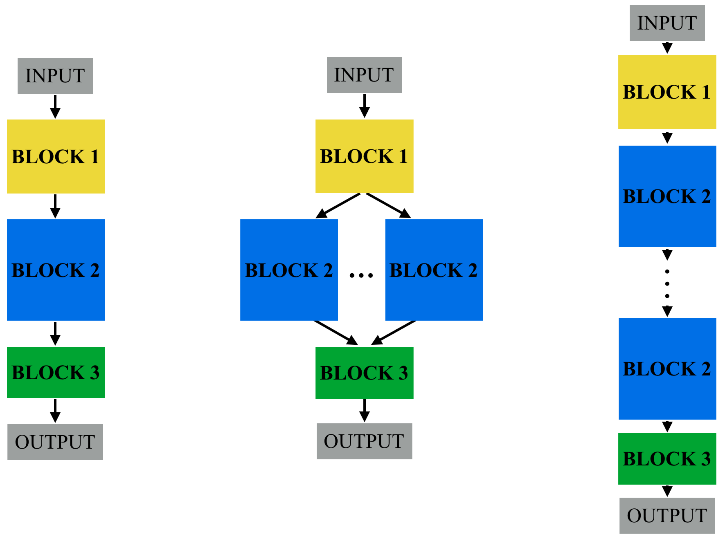

Figure 6.1. Example of duplication. The image on the left shows the structure discovered after the rapid succession, where each block includes a number of layers with the same size of feature maps. The image in the middle and right are two examples of the duplication that the Block 2 undergoes different combination to create new architectures.

6.3 Experimental Results and Analysis

In this section, we report the experimental results of using EIGEN for structure search of neural networks. We firstly describe the experiment setup including datasets prepossessing and training strategy in Subsection 6.3.1 and show the results. Following that, we analyze the experimental results of rapid succession and accelerated extinction.

6.3.1 Experiment Setup and Results

Datasets. The experiments are conducted on two benchmark datasets including CIFAR-10 [47] and CIFAR-100 [47]. The CIFAR-10 dataset contains 10 classes with 50, 000 training images and 10, 000 test images. The images has the size of 32×32. The data augmentation is applied by a Global Contrast Normalization (GCN) and ZCA whitening [22]. The CIFAR-100 dataset is similar to CIFAR-10 except it includes 100 classes.

Training Strategy and Details. During the training process, we use mini-batch Stochastic Gradient Descent (SGD) to train each individual network with the batch size as 128, momentum as 0.9, and weight-decay as 0.0005. Each network is trained for a maximum of 25, 000 iterations. The initial learning rate is 0.1 and is set as 0.01 and 0.001 at 15, 000 iterations and 20, 000 iterations, respectively. The parameters in Algorithm 6.1 are set to $T_1 = 5,000$, $T_2 = 15,000$, $T_3 = 5,000$, $p = 5$, and $q = 2$. The fitness score is validation accuracy from validation set. The primary succession ends when the fitness score saturates and then the secondary succession starts. The entire evolution procedure is terminated until the fitness score converges. Training is conducted with *TensorFlow* [1].

We directly adopt the hyper-parameters developed on CIFAR-10 dataset to CIFAR-100 dataset. The experiments are run on a machine that has one Intel Xeon E5-2680 v4 2.40GHz CPU and one Nvidia Tesla P100 GPU.

Approach	PARAMS.	CIFAR-10	CIFAR-100	Comp Cost
EDEN [17]	0.2 M	74.5%	-	-
Genetic CNN [95]	-	92.9%	71.0%	408 GPUH
LS-Evolution [71]	5.4 M	94.6%	-	65,536 GPUH
LS-Evolution [71]	40.4 M	-	77.0%	>= 65,536 GPUH
AG-Evolution†	-	90.5%	-	72 GPUH
AG-Evolution†	-	-	66.9%	136 GPUH
EIGEN	1.2 M	**93.7**%	-	48 GPUH
EIGEN-D	2.6 M	**94.6**%	-	48 GPUH
EIGEN	6.1 M	-	**76.9**%	120 GPUH
EIGEN-D	11.6 M	-	**78.1**%	120 GPUH

Table 6.1. Comparison with hand-designed architectures and automatically discovered architectures using genetic algorithms. The CIFAR-10 and CIFAR-100 columns indicate the test accuracy achieved on data-augmented CIFAR-10 and CIFAR-100 datasets, respectively. The PARAMS. column indicates the number of parameters in the discovered network. †: approach in the prevous chapter

6.3.2 Comparison Results

The experimental results in Table 6.1 show that the proposed approach are competitive with hand designed networks. We refer our approach without and with gene duplication as **EIGEN** and **EIGEN-D**, respectively. Compared with the evolution-based algorithms, we can achieve the best results with the minimum computational cost. For example, we obtain similar results on the two benchmark datasets compared to [71], but our approach is 1,000 times faster. Also, the number of parameters of the networks found by our approach on the two datasets are more than two times smaller than LS-Evolution [71].

6.3.3 Effect of Gene Duplication

After the rapid succession, the duplication operation is applied to leverage the automatically discovered structures. To analyze the effect of gene duplication, we denote the approach with duplication as **EIGEN-D** and show the results on CIFAR-10 and CIFAR-100 in Table 6.2. Although more parameters are induced in the networks by duplication, the beneficial structures contained in the block can actually contribute to the network performance through duplication.

Method	CIFAR-10 (PARAMS.)	CIFAR-100 (PARAMS.)
EIGEN	93.7% (1.2 M)	76.9% (6.1 M)
EIGEN-D	94.6% (2.6 M)	78.1% (11.8 M)

Table 6.2. Analysis of the gene duplication operation on CIFAR-10 and CIFAR-100. The performance on the two datasets is improved with more parameters on the networks discovered from gene duplication.

CHAPTER 7
CONCLUSION AND FUTURE WORK

In this thesis, we have proposed a distribution-dependent dropout for both shallow learning and deep learning. Theoretically, we proved that the new dropout achieves a smaller risk and faster convergence. Based on the distribution-dependent dropout, we developed an efficient evolutional dropout for training deep neural networks that adapts the sampling probabilities to the evolving distributions of layers' outputs. Experimental results on various data sets verified that the proposed dropouts can dramatically improve the convergence and also reduce the testing error. This is our first contribution.

In this thesis, we view neural network operations as 1×1 data transformation and $k \times k$ abstract pattern extraction. By converting $k \times k$ convolution kernels into binary patterns, we significantly reduced the model size as well as computational cost of modern neural network architectures such as InceptionNets and ResNets, without significantly sacrificing the network performances. Our binarization approach is extremely simple compared to previous literatures. We further proposed a small network architecture containing pattern residual blocks, which utilize binarized patterns to extract features and 1×1 transformation to compute pattern residuals. The resulting concise neural network is small and effective compared to recent advances in compact neural network design. The effectiveness of our approach is demonstrated intensively on the CIFAR-10 dataset and the ImageNet dataset. We hope our investigation will inspire the community for advanced architecture design from a pattern point of view. This is our second contribution.

The third contribution of this thesis is that in this thesis, we have developed an aggresive genetic programing approach to optimize the structure of convolutional neural networks under limited computational resources without imposing strong restrictions on the search space. Our study shows that it is possible to achieve promising result using the proposed aggressive genetic programming approach in a reasonable amount of time. We expect the proposed strategies can be also useful for optimizing other types of neural networks (e.g., recurrent neural networks), which will be left as future work. We will also explore whether better performance than the state-of-the-art on CIFAR-10, CIFAR-100 and large-scale ImageNet datasets can be achieved by our genetic programming approach by running more generations and introducing more mutation operations.

Lastly, in this thesis we propose an ecologically inspired genetic approach for searching neural network architectures automatically with the two types of succession: primary and second succession, and gene duplication. The rapid succession could evolve the community of networks into an optimal status under the limited computational resources. With the help of gene duplication, the performance of the found network could be boosted without sacrificing any computational cost. The experimental results show that the proposed approach can achieve competitive results on CIFAR-10 and CIFAR-100 under dramatically reduced computational cost compared with other genetic-based algorithms.

The following can be considered as the future research works related to this thesis:

- Reducing computional cost during inferencing stage when designing and applying deep neural networks.
- More exploration to improve the efficiency when searching neural networks through genetic approach.

CHAPTER 8

APPENDIX

8.1 Speeding up Training Neural Networks

In this section, we provide neural network structures used when we conducted experiments on MNIST, SVHN, CIFAR-10, and CIFAR-100 dataset in chapter 3.

8.1.1 Neural Network Structures

In this section we present the neural network structures and the number of filters, filter size, padding and stride parameters for MNIST, SVHN, CIFAR-10 and CIFAR-100, respectively. Note that in Table 8.2, Table 8.3 and Table 8.4, the rnorm layer is the local response normalization layer and the local layer is the locally-connected layer with unshared weights.

8.1.1.1 MNIST

We used the similar neural network structure to [91]: two convolution layers, two fully connected layers, a softmax layer and a cost layer at the end. The dropout is added to the first fully connected layer. Tables 8.1 presents the neural network structures and the number of filters, filter size, padding and stride parameters for MNIST.

8.1.1.2 SVHN

The neural network structure used for this data set is from [91], including 2 convolutional layers, 2 max pooling layers, 2 local response layers, 2 fully connected layers, a softmax layer and a cost layer with one dropout layer. Tables 8.2 presents

Table 8.1. The Neural Network Structure for MNIST

Layer Type	Input Size	#Filters	Filter size	Padding/Stride	Output Size
conv1	$28 \times 28 \times 1$	32	4×4	0/1	$21 \times 21 \times 32$
pool1(max)	$21 \times 21 \times 32$		2×2	0/2	$11 \times 11 \times 32$
conv2	$11 \times 11 \times 32$	64	5×5	0/1	$7 \times 7 \times 64$
pool2(max)	$7 \times 7 \times 64$		3×3	0/3	$3 \times 3 \times 64$
fc1	$3 \times 3 \times 64$				150
dropout	150				150
fc2	150				10
softmax	10				10
cost	10				1

the neural network structures and the number of filters, filter size, padding and stride parameters used for SVHN data set.

8.1.1.3 CIFAR-10

The neural network structure is adopted from [91], which consists two convolutional layer, two pooling layers, two local normalization response layers, 2 locally connected layers, two fully connected layers and a softmax and a cost layer. Table 8.3 presents the detail neural network structure and the number of filters, filter size, padding and stride parameters used.

8.1.1.4 CIFAR-100

The network structure for this data set is similar to the neural network structure in [47], which consists of 2 convolution layers, 2 max pooling layers, 2 local response normalization layers, 2 locally connected layers, 3 fully connected layers, and a softmax and a cost layer. Table 8.4 presents the neural network structures and the number of filters, filter size, padding and stride parameters used for CIFAR-100

Table 8.2. The Neural Network Structure for SVHN

Layer Type	Input Size	#Filters	Filter Size	Padding/Stride	Output Size
conv1	$28 \times 28 \times 3$	64	5×5	0/1	$24 \times 24 \times 64$
pool1(max)	$24 \times 24 \times 64$		3×3	0/2	$12 \times 12 \times 64$
rnorm1	$12 \times 12 \times 64$				$12 \times 12 \times 64$
conv2	$12 \times 12 \times 64$	64	5×5	2/1	$12 \times 12 \times 64$
rnorm2	$12 \times 12 \times 64$				$12 \times 12 \times 64$
pool2(max)	$12 \times 12 \times 64$		3×3	0/2	$6 \times 6 \times 64$
local3	$6 \times 6 \times 64$	64	3×3	1/1	$6 \times 6 \times 64$
local4	$6 \times 6 \times 64$	32	3×3	1/1	$6 \times 6 \times 32$
dropout	1152				1152
fc1	1152				512
fc10	512				10
softmax	10				10
cost	10				1

data set.

8.1.1.5 The Neural Network Structure used for BN

Tables 8.5 and 8.6 present the network structures of different methods in subsection 5.3 in the paper. The layer pool(ave) in Table 8.5 and Table 8.6 represents the average pooling layer.

8.2 Compressing Neural Network

In this section, we present the whole neural network structures for the designed **SEP-Net** with 1.7M parameters in Table 8.7 and **SEP-Net** with less parameters in Table 8.8 in chapter 4.

8.3 Designing Neural Network Structures Automatically

In this section, we provide the comparison on the proposed aggressive selection strategy with other existing selection strategies such as **Tournmanet**, **Sampling**

Table 8.3. The Neural Network Structure for CIFAR-10

Layer Type	Input Size	#Filters	Filter Size	Padding/Stride	Output Size
conv1	$24 \times 24 \times 3$	64	5×5	2/1	$24 \times 24 \times 64$
pool1(max)	$24 \times 24 \times 64$		3×3	0/2	$12 \times 12 \times 64$
rnorm1	$12 \times 12 \times 64$				$12 \times 12 \times 64$
conv2	$12 \times 12 \times 64$	64	5×5	2/1	$12 \times 12 \times 64$
rnorm2	$12 \times 12 \times 64$				$12 \times 12 \times 64$
pool2(max)	$12 \times 12 \times 64$		3×3	0/2	$6 \times 6 \times 64$
local3	$6 \times 6 \times 64$	64	3×3	1/1	$6 \times 6 \times 64$
local4	$6 \times 6 \times 64$	32	3×3	1/1	$6 \times 6 \times 32$
dropout	1152				1152
fc1	1152				128
fc10	128				10
softmax	10				10
cost	10				1

Uniformly and **Sampling by Fitness**. In Figure 8.1, we plot the test performance of the best individual in one generation by using those different selection strategies. From Figure 8.1, we can observe that aggressive selection evolves faster than other strageties dramatically.

Table 8.4. The Neural Network Structure for CIFAR-100

Layer Type	Input Size	#Filters	Filter Size	Padding/Stride	Output Size
conv1	$32 \times 32 \times 3$	64	5×5	2/1	$32 \times 32 \times 64$
pool1(max)	$32 \times 32 \times 64$		3×3	0/2	$16 \times 16 \times 64$
rnorm1	$16 \times 16 \times 64$				$16 \times 16 \times 64$
conv2	$16 \times 16 \times 64$	64	5×5	2/1	$16 \times 16 \times 64$
rnorm2	$16 \times 16 \times 64$				$16 \times 16 \times 64$
pool2(max)	$16 \times 16 \times 64$		3×3	0/2	$8 \times 8 \times 64$
local3	$8 \times 8 \times 64$	64	3×3	1/1	$8 \times 8 \times 64$
local4	$8 \times 8 \times 64$	32	3×3	1/1	$8 \times 8 \times 32$
fc1	2048				128
dropout	128				128
fc2	128				128
fc100	128				100
softmax	100				100
cost	100				1

Table 8.5. Layers of networks for the experiment comparing with BN on CIFAR-10

Layer Type	noBN-noDropout	BN	e-dropout
Layer 1	conv1	conv1	conv1
Layer 2	pool1(max)	pool(max)	pool1(max)
Layer 3	N/A	bn1	N/A
Layer 4	conv2	conv2	conv2
Layer 5	N/A	bn2	N/A
Layer 6	pool2(ave)	pool2(ave)	pool2(ave)
Layer 7	conv3	conv3	conv3
Layer 8	N/A	bn3	e-dropout
Layer 9	pool3(ave)	pool3(ave)	pool3(ave)
Layer 10	fc1	fc1	fc1
Layer 11	softmax	softmax	softmax

Table 8.6. Sizes in networks for the experiment comparing with BN on CIFAR-10

Layer Type	Input size	#Filters	Filter size	Padding/Stride	Output size
conv1	$32 \times 32 \times 3$	32	5×5	2/1	$32 \times 32 \times 32$
pool1(max)	$32 \times 32 \times 32$		3×3	0/2	$16 \times 16 \times 32$
conv2	$16 \times 16 \times 32$	32	5×5	2/1	$16 \times 16 \times 32$
pool2(ave)	$16 \times 16 \times 32$		3×3	0/2	$8 \times 8 \times 32$
conv3	$8 \times 8 \times 32$	64	5×5	2/1	$8 \times 8 \times 64$
pool3(ave)	$8 \times 8 \times 64$		3×3	0/2	$4 \times 4 \times 64$
fc1	$4 \times 4 \times 64$				10
softmax	10				10
cost	10				1

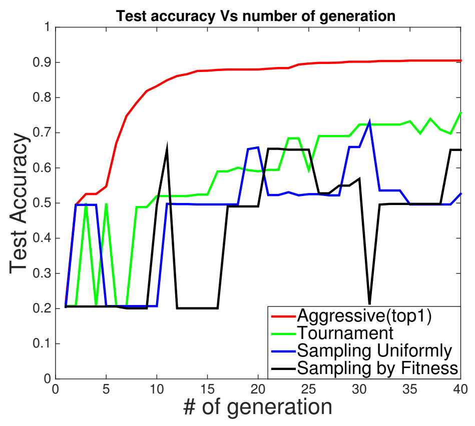

Figure 8.1. The test accuracy of the best individual among the selected individuals by using aggressive, tournament, sampling uniformly and sampling by fitness selection strategies vs the number of generations on CIFAR-10 dataset.

Table 8.7. The Neural Network Structure for the Designed **SEP-Net** with 1.7M parameters

Layer Type	#Channel	Kernel size	Pad/Stride	#Group
conv1_base	64	5×5	1/2	1
st/sep-module1_svd1_base	32	1×1	0/1	1
st/sep-module1_slice1_1x1_0_base	32	1×1	0/1	4
st/sep-module1_slice1_3x3_0_base	32	3×3	1/1	4
st/sep-module1_slice2_1x1_0_base	16	1×1	0/1	4
st/sep-module1_slice2_3x3_0_base	16	3×3	1/1	4
st/sep-module1_svd2_base	64	1×1	0/1	1
conv2_base	128	3×3	1/2	1
st/sep-module2_svd1_base	64	1×1	0/1	1
st/sep-module2_slice1_1x1_0_base	64	1×1	0/1	4
st/sep-module2_slice1_3x3_0_base	64	3×3	1/1	4
st/sep-module2_slice2_1x1_0_base	32	1×1	0/1	4
st/sep-module2_slice2_3x3_0_base	32	3×3	1/1	4
st/sep-module2_svd2_base	128	1×1	0/1	1
conv3_base	256	3×3	1/2	4
st/sep-module3_svd1_base	128	1×1	0/1	1
st/sep-module3_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module3_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module3_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module3_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module3_svd2_base	256	1×1	0/1	1
st/sep-module4_svd1_base	128	1×1	0/1	1
st/sep-module4_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module4_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module4_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module4_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module4_svd2_base	256	1×1	0/1	1
conv4_base	256	3×3	1/2	1
st/sep-module5_svd1_base	128	1×1	0/1	4
st/sep-module5_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module5_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module5_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module5_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module5_svd2_base	256	1×1	0/1	1
st/sep-module6_svd1_base	128	1×1	0/1	1
st/sep-module6_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module6_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module6_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module6_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module6_svd2_base	256	1×1	0/1	1
conv5_base	400	3×3	1/2	4

Table 8.8. The Neural Network Structure for the Designed **SEP-Net** with 1.3M parameters

Layer Type	#Channel	Kernel size	Pad/Stride	#Group
conv1_base	64	5×5	1/2	1
st/sep-module1_svd1_base	32	1×1	0/1	1
st/sep-module1_slice1_1x1_0_base	32	1×1	0/1	4
st/sep-module1_slice1_3x3_0_base	32	3×3	1/1	4
st/sep-module1_slice2_1x1_0_base	16	1×1	0/1	4
st/sep-module1_slice2_3x3_0_base	16	3×3	1/1	4
st/sep-module1_svd2_base	64	1×1	0/1	1
conv2_base	128	3×3	1/2	1
st/sep-module2_svd1_base	64	1×1	0/1	1
st/sep-module2_slice1_1x1_0_base	64	1×1	0/1	4
st/sep-module2_slice1_3x3_0_base	64	3×3	1/1	4
st/sep-module2_slice2_1x1_0_base	32	1×1	0/1	4
st/sep-module2_slice2_3x3_0_base	32	3×3	1/1	4
st/sep-module2_svd2_base	128	1×1	0/1	1
conv3_base	256	3×3	1/2	4
st/sep-module3_svd1_base	128	1×1	0/1	1
st/sep-module3_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module3_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module3_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module3_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module3_svd2_base	256	1×1	0/1	1
st/sep-module4_svd1_base	128	1×1	0/1	1
st/sep-module4_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module4_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module4_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module4_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module4_svd2_base	256	1×1	0/1	1
conv4_base	256	3×3	1/2	4
st/sep-module5_svd1_base	128	1×1	0/1	1
st/sep-module5_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module5_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module5_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module5_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module5_svd2_base	256	1×1	0/1	1
st/sep-module6_svd1_base	128	1×1	0/1	1
st/sep-module6_slice1_1x1_0_base	128	1×1	0/1	4
st/sep-module6_slice1_3x3_0_base	128	3×3	1/1	4
st/sep-module6_slice2_1x1_0_base	64	1×1	0/1	4
st/sep-module6_slice2_3x3_0_base	64	3×3	1/1	4
st/sep-module6_svd2_base	256	1×1	0/1	1
conv5_base	512	3×3	1/2	16

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Jimmy Ba and Brendan Frey. Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092, 2013.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[4] Pierre Baldi and Peter J Sadowski. Understanding dropout. In *Advances in Neural Information Processing Systems*, pages 2814–2822, 2013.

[5] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[6] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[7] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International Conference on Machine Learning*, pages 115–123, 2013.

[8] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. *arXiv preprint arXiv:1702.00953*, 2017.

[9] Ning Chen, Jun Zhu, Jianfei Chen, and Bo Zhang. Dropout training for support vector machines. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 1752–1759, 2014.

[10] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *ICML*, 2015.

[11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[14] George E Dahl, Dong Yu, Li Deng, and Alex Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42, 2012.

[15] Jia Deng, W. Dong, R. Socher, Lijia Li, Kai Li, and Li Fei-Fei. Imagenet: a large-scale hierachical image database. In *CVPR*, 2009.

[16] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[17] Emmanuel Dufourq and Bruce A Bassett. Eden: Evolutionary deep networks for efficient machine learning. *arXiv preprint arXiv:1709.09161*, 2017.

[18] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alex G Hauptmann. Devnet: A deep event network for multimedia event detection and evidence recounting. In *CVPR*, 2015.

[19] Ross Girshick. Fast r-cnn. In *ICCV*, 2015.

[20] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[21] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[22] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[23] Benjamin Graham, Jeremy Reizenstein, and Leigh Robinson. Efficient batch-wise dropout training using submatrices. *CoRR*, abs/1502.02478, 2015.

[24] Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 318–325. Morgan Kaufmann Publishers Inc., 1993.

[25] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[26] Song Han, Huizi Mao, and William J Dally. A deep neural network compression pipeline: Pruning, quantization, huffman encoding. *arXiv preprint arXiv:1510.00149*, 10, 2015.

[27] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

[28] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. *arXiv preprint arXiv:1703.06870*, 2017.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[31] David P. Helmbold and Philip M. Long. On the inductive bias of dropout. *CoRR*, abs/1412.4736, 2014.

[32] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9(1):926, 2010.

[33] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[34] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[35] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[36] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[37] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[38] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

[39] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

[40] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[41] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[43] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[44] Felix Juefei-Xu, Vishnu Naresh Boddeti, and Marios Savvides. Local binary convolutional neural networks. *arXiv preprint arXiv:1608.06049*, 2016.

[45] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[46] Diederik P. Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *CoRR*, abs/1506.02557, 2015.

[47] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[48] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[50] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[51] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *Artificial Intelligence and Statistics*, pages 464–472, 2016.

[52] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[53] Zhe Li, Boqing Gong, and Tianbao Yang. Improved dropout for shallow and deep learning. In *Advances in Neural Information Processing Systems*, pages 2523–2531, 2016.

[54] Darryl D Lin and Sachin S Talathi. Overcoming challenges in fixed point training of deep convolutional networks. *arXiv preprint arXiv:1607.02241*, 2016.

[55] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[56] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[57] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[58] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *CVPR*, 2015.

[59] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[60] Naveen Mellempudi, Abhisek Kundu, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary neural networks with fine-grained quantization. *arXiv preprint arXiv:1705.01462*, 2017.

[61] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pages 58–65, 2016.

[62] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[63] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989.

[64] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[65] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

[66] Behnam Neyshabur, Ruslan R Salakhutdinov, and Nati Srebro. Path-sgd: Path-normalized optimization in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2413–2421, 2015.

[67] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272, 2011.

[68] Joachim Ott, Zhouhan Lin, Ying Zhang, Shih-Chii Liu, and Yoshua Bengio. Recurrent neural networks with limited numerical precision. *arXiv preprint arXiv:1608.06902*, 2016.

[69] Marc'Aurelio Ranzato, Alex Krizhevsky, and Geoffrey E. Hinton. Factored 3-way restricted boltzmann machines for modeling natural images. In *AISTATS*, pages 621–628, 2010.

[70] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*. Springer, 2016.

[71] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[72] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *NIPS*, 2015.

[73] Sarda Sahney and Michael J Benton. Recovery from the most profound mass extinction of all time. *Proceedings of the Royal Society of London B: Biological Sciences*, 275(1636):759–765, 2008.

[74] J David Schaffer, Darrell Whitley, and Larry J Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37. IEEE, 1992.

[75] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[76] Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, and Karthik Sridharan. Stochastic convex optimization. In *The 22nd Conference on Learning Theory (COLT)*, 2009.

[77] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[78] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[79] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[80] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[81] Nathan Srebro, Karthik Sridharan, and Ambuj Tewari. Smoothness, low noise and fast rates. In *Advances in Neural Information Processing Systems 23 (NIPS)*, pages 2199–2207, 2010.

[82] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[83] Pierre-Luc St-Charles, Guillaume-Alexandre Bilodeau, and Robert Bergevin. Fast image gradients using binary feature convolutions. In *CVPR*, 2016.

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

[85] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1139–1147, 2013.

[86] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[87] T Tieleman and G Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. Technical report, Technical report, 2012. 31.

[88] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *CVPR*, 2014.

[89] Stefan Wager, William Fithian, Sida Wang, and Percy S Liang. Altitude training: Strong bounds for single-layer dropout. In *Advances in Neural Information Processing Systems*, pages 100–108, 2014.

[90] Stefan Wager, Sida Wang, and Percy S Liang. Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems*, pages 351–359, 2013.

[91] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1058–1066, 2013.

[92] Sida Wang and Christopher Manning. Fast dropout training. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 118–126, 2013.

[93] Sida I Wang, Mengqiu Wang, Stefan Wager, Percy Liang, and Christopher D Manning. Feature noising for log-linear structured prediction. In *EMNLP*, pages 1170–1179, 2013.

[94] Jacob Whitehill and Christian W Omlin. Haar features for facs au recognition. In *Automatic Face and Gesture Recognition, 2006. FGR 2006. 7th International Conference on*, pages 5–pp. IEEE, 2006.

[95] Lingxi Xie and Alan Yuille. Genetic cnn. arxiv preprint. *arXiv preprint arXiv:1703.01513*, 2017.

[96] Yi Xu, Haiqin Yang, Lijun Zhang, and Tianbao Yang. Efficient non-oblivious randomized reduction for risk minimization with improved excess risk guarantee. *arXiv preprint arXiv:1612.01663*, 2016.

[97] Zhongwen Xu, Yi Yang, and Alex G Hauptmann. A discriminative cnn video representation for event detection. In *CVPR*, 2015.

[98] Penghang Yin, Shuai Zhang, Jack Xin, and Yingyong Qi. Training ternary neural networks with exact proximal operator. *arXiv preprint arXiv:1612.06052*, 2016.

[99] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[100] Shengxin Zha, Florian Luisier, Walter Andrews, Nitish Srivastava, and Ruslan Salakhutdinov. Exploiting image-trained cnn architectures for unconstrained video classification. *arXiv preprint arXiv:1503.04144*, 2015.

[101] Jianzhi Zhang. Evolution by gene duplication: an update. *Trends in ecology & evolution*, 18(6):292–298, 2003.

[102] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. *arXiv preprint arXiv:1412.6651*, 2014.

[103] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 2017.

[104] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[105] Jingwei Zhuo, Jun Zhu, and Bo Zhang. Adaptive dropout rates for learning with corrupted features. In *IJCAI*, pages 4126–4133, 2015.

[106] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 928–936, 2003.

[107] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[108] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.