ABSTRACT

Title of dissertation: TOWARD FAST AND EFFICIENT REPRESENTATION LEARNING

Hao Li
Doctor of Philosophy, 2018

Dissertation directed by: Prof. Hanan Samet and Prof. Tom Goldstein
Department of Computer Science

The success of deep learning and convolutional neural networks in many fields is accompanied by a significant increase in the computation cost. With the increasing model complexity and pervasive usage of deep neural networks, there is a surge of interest in fast and efficient model training and inference on both cloud and embedded devices. Meanwhile, understanding the reasons for trainability and generalization is fundamental for its further development. This dissertation explores approaches for fast and efficient representation learning with a better understanding of the trainability and generalization. In particular, we ask following questions and provide our solutions: 1) How to reduce the computation cost for fast inference? 2) How to train low-precision models on resources-constrained devices? 3) What does the loss surface looks like for neural nets and how it affects generalization?

To reduce the computation cost for fast inference, we propose to prune filters from CNNs that are identified as having a small effect on the prediction accuracy. By removing filters with small norms together with their connected feature maps,

the computation cost can be reduced accordingly without using special software or hardware. We show that simple filter pruning approach can reduce the inference cost while regaining close to the original accuracy by retraining the networks.

To further reduce the inference cost, quantizing model parameters with low-precision representations has shown significant speedup, especially for edge devices that have limited computing resources, memory capacity, and power consumption. To enable on-device learning on lower-power systems, removing the dependency of full-precision model during training is the key challenge. We study various quantized training methods with the goal of understanding the differences in behavior, and reasons for success or failure. We address the issue of why algorithms that maintain floating-point representations work so well, while fully quantized training methods stall before training is complete. We show that training algorithms that exploit high-precision representations have an important greedy search phase that purely quantized training methods lack, which explains the difficulty of training using low-precision arithmetic.

Finally, we explore the structure of neural loss functions, and the effect of loss landscapes on generalization, using a range of visualization methods. We introduce a simple filter normalization method that helps us visualize loss function curvature, and make meaningful side-by-side comparisons between loss functions. The sharpness of minimizers correlates well with generalization error when this visualization is used. Then, using a variety of visualizations, we explore how training hyper-parameters affect the shape of minimizers, and how network architecture affects the loss landscape.

TOWARDS FAST AND EFFICIENT REPRESENTATION LEARNING

by

Hao Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Hanan Samet, Chair/Advisor
Professor Thomas Goldstein, Co-Chair/Co-Advisor
Professor David Jacobs
Professor Larry Davis
Professor Rama Chellappa

© Copyright by
Hao Li
2018

Dedication

To my father, Yuxiang Li.

Acknowledgments

I would like to thank my advisor, Professor Hanan Samet for giving me the opportunity to pursue my Ph.D. at University of Maryland. We met at SIGIR'11 in Beijing and that is the start of this magic journey. Hanan always encouraged me to explore widely and to define my own research problems rather than following others. His enthusiastic of building working system and doing long-term impactful projects greatly motivated me. He has always made himself available for help and advice.

My thanks also go to my co-advisor, Professor Tom Goldstein. I met Tom in 2015 and we both had strong interest in the intersection of machine learning and high performance computing. Tom provided extraordinary theoretical ideas and computational expertise, without which this thesis would have been a distant dream. I learned a lot from him about good practices and skills on writing, programming, teaching and presentation. It has been a great fortune to work with and learn from such an extraordinary individual.

I sincerely thank Professor David Mount, Professor David Jacobs, Professor Larry Davis and Professor Rama Chellappa for serving on my proposal and thesis committee and for providing valuable feedback and suggestions about my work. It has been a great pleasure to collaborate with Professor Gavin Taylor at United Naval Academy and Prof Christoper Studer at Cornell University, who provide their insightful ideas and feedback. I would also thank my collaborators at NEC Labs America, Hans Peter Graf, Igor Durdanovic and Asim Kadav, for their generous

support and insightful discussions.

I would thank my labmates and friends, Zheng Xu, Soham De, Sohil Sigh, Ronny Huang, Shangfu Peng, Hong Wei, Ang Li, and Jin Sun. It was a pleasure to working with you all.

I would like to acknowledge the help and support from the staff members from Computer Science Department, Prof. Jeff Foster, Jeniffer Story, Tom Hurst, Jodie Gray, Sharron McElroy and Janice Perrone. I would like to thank UMIACS staff members for their support of hosting my self-supported servers and GPU workstations.

Finally, I owe my deepest thanks to my parents and sister who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them. Thanks to my wife and my daughter, they are the best gifts to me during this journey.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

A good feature representation is essential for the success of many domains, including computer vision, speech recognition, natural language processing and many other fields. Representation learning is about learning representations of the data that make it easier to extract useful information when building classifiers or other predictors [Bengio et al., 2013a, LeCun et al., 2015], which is a fundamental problem for many applications.

Visual feature representation is an active research area as it is the most essential part for many computer vision tasks, such as object recognition, detection and tracking. Before the success of deep learning, computer vision researchers used to design low-level image features for specific tasks, e.g., Local Binary Pattern (LBP) [Ahonen et al., 2006] for face recognition and Histogram of Oriented Gradients (HOG) [Dalal and Triggs, 2005] for pedestrian detection. For the object recognition task, mid-level representation were developed based on low level features, such as Bag of Visual Words model [Fei-Fei and Perona, 2005] based on SIFT descriptors [Lowe, 2004]. However, limited success have been made with this representation. Since the break through of AlexNet [Krizhevsky et al., 2012] on the ImageNet competition (ILSVRC) [Russakovsky et al., 2015], deep learning and

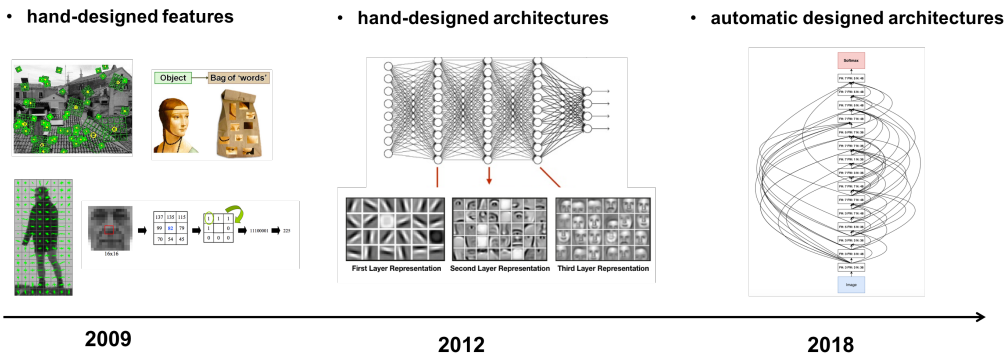


Figure 1.1: The evolution of visual feature representations from handcrafted features to hand-designed architectures and then automatically designed architectures.

Convolutional Neural Networks (CNNs) [LeCun et al., 1990, 1998] have regained attention, got quickly developed and widely used in computer vision. A lot of architectures were proposed since then [He et al., 2016, Huang et al., 2017, Simonyan and Zisserman, 2015, Szegedy et al., 2015a,b]. CNNs have now become the de facto standard model for many computer vision applications. One reason for the success is that their hierarchical, layered structure may allow them to capture the geometric regularities [Basri and Jacobs, 2016]. On the hand, the choice of convolution and pooling layers is motivated by the desire of invariance to image translation, scaling, and other small deformations [Azulay and Weiss, 2018].

In the past several years, we have witnessed the transition from human-crafted feature representations to the automatically learned features from manually-designed neural networks (Figure 1.1). The advantage over handcrafted features is that deep learning models can automatically learn complex and hierarchical feature representations from data. The features and the classifier are learned in an end-to-end way with supervised learning.

1.1 Enabling Factors and Challenges

The key enabling factors for recent quick development of CNNs can be summarized in four-folds:

- *Large amount of training data.* From MINST to CIFAR-10 [Krizhevsky, 2009] and ImageNet [Russakovsky et al., 2015], the size of benchmark dataset has expanded remarkably. The ILSVRC has led to significant advancements in exploring various architectural choices in CNNs [He et al., 2016, Huang et al., 2017, Krizhevsky et al., 2012, Simonyan and Zisserman, 2015, Szegedy et al., 2015a].

- *High-capacity models.* As shown in Figure 1.2, the general trend for CNNs since the past few years is that the networks have grown deeper and wider, with an overall increase in the number of parameters and convolution operations.

- *Faster hardware.* The advance in hardware accelerators like GPUs greatly accelerated the training speed and enabled more architectures to be explored in a short time.

- *Advances in algorithms.* Deep neural networks were believed to be difficult to train due to the problem of gradient vanishing or explosion Glorot and Bengio [2010]. There is no theoretical guarantees for optimizing high-dimension non-linear problems, but stochastic gradient descent methods were found working pretty well in practice. Training algorithms have shifted from layer-wise pre-training to end-to-end supervised training. To provide better regularization

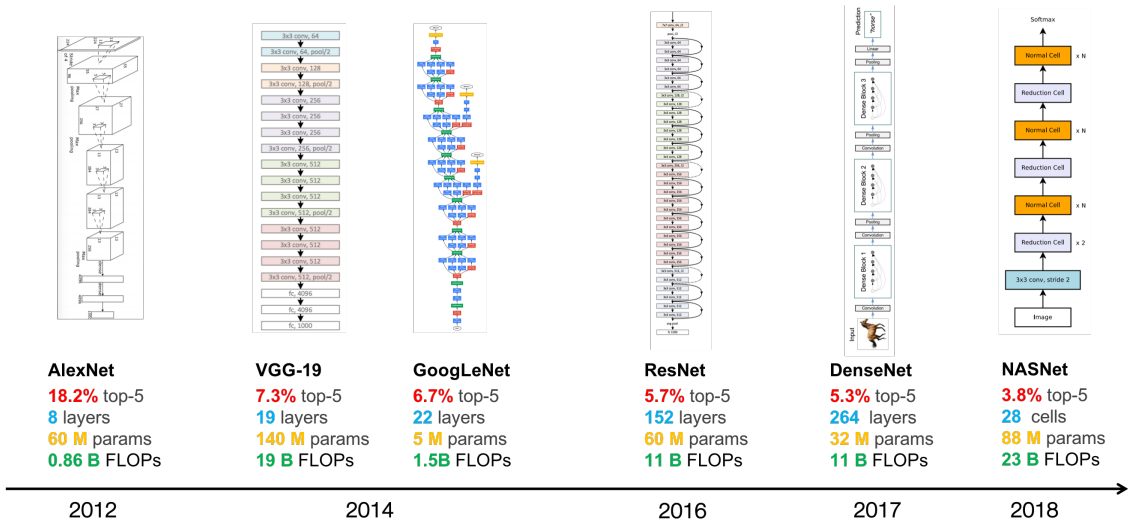


Figure 1.2: The evolution of CNNs for ImageNet. While the classification accuracy keeps increasing, the architecture grows deeper and wider with more parameters and computation costs.

and facilitate the training process, new modules such as ReLU Krizhevsky et al. [2012], Dropout [Srivastava et al., 2014] and Batch Normalization [Ioffe and Szegedy, 2015] have been proposed and widely used. Better parameter initialization strategies were also designed to make training easier [He et al., 2015].

Despite the practical success of deep learning, there are accompanied challenges with each of the enabling factors:

- *Low data efficiency.* Humans are able to learn from very few labeled data. However, training deep neural networks requires large-scale dataset, which makes it hard to work for scenarios with limited training samples.

- *High computation cost for inference and training.* Modern CNNs often have high capacity with large training and inference costs. For example, VGG-

19 [Simonyan and Zisserman, 2015], the winner of ILSVRC'14, contains 144 million parameters and takes 19.6 billion *floating point operations (FLOPs)* for each image. The high capacity networks have significant computation costs, especially when used with embedded or mobile devices where computational resources or power supply may be limited. With the rapidly increasing model complexity, there is a surge of interest in fast and efficient model training and inference, which is critical for real-world machine learning applications. For example, for web services that provide image search and image classification APIs that operate on a time budget often serving billions of images per day, benefit significantly from lower inference times.

- *High power consumption.* The adoption of powerful hardware like GPUs results in high power consumption, which could begin expensive for massive usage in datacenters. Moreover, the demands of deploying CNNs on platforms with limited computation power are increasing.

- *Lack of deep understanding.* There is a lack of theoretical understanding about the reasons for the good generalization ability of deep neural networks. Though designing neural network architectures eliminates the need of crafting low-level feature extractors, there is no clear guidance for the architecture design. Discovering the optimal net architecture for a given task remains a hard problem [Saxena and Verbeek, 2016], which still requires human expertise with time-consuming trial-and-error. On the other hand, the optimization process is still a black box and it is not clear how trainable a model is, such as why

stochastic gradient descent (SGD) results in solutions that generalize well [Le et al., 2011].

1.2 Problems to Solve

In this dissertation, we focus on problems related with the last three challenges, i.e., we are interested in developing fast and efficient CNNs with better understanding about the reasons for its trainability and generalization. In particular, we try to answer following questions:

1.2.1 How to reduce the model complexity for fast inference?

Recent efforts toward reducing the overheads involve pruning and compressing the weights of various layers without hurting accuracy. However, model compression techniques reduces a significant number of parameters from the fully connected (FC) layers but may not adequately reduce the computation costs in the convolutional layers due to irregular sparsity without support of special libraries or hardware.

1.2.2 How to train quantized models on low-power devices?

Because of the high memory requirements and computational complexity, deep neural networks (DNNs) are usually designed and trained using powerful hardware, which are expensive and less accessible. There are increasing demands of training and deploying neural networks directly on embedded devices with limited resources, such as mobile phones, drones and self-driving cars. Such low-power systems are

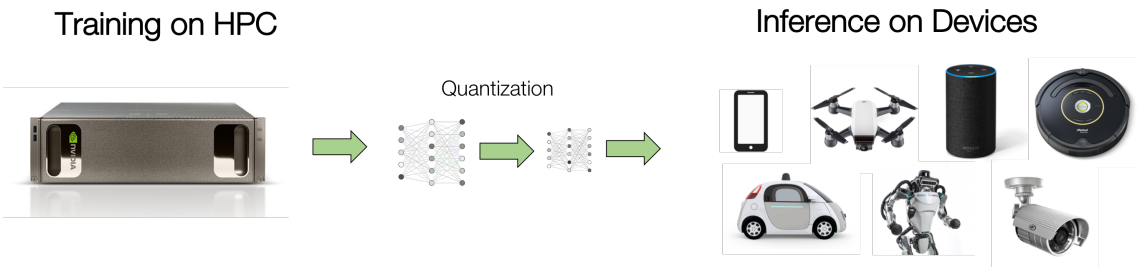


Figure 1.3: The current practice of deploying a quantized model, which is training the high-precision model on the HPC and deploy the low-precision model on the low-power devices for inference. The quantized model is used only for inference and cannot be updated on device.

memory and power limited, and in some cases lack basic support for floating-point arithmetic.

Quantization of neural networks has shown competitive accuracy with significant less computing resources compared to full-precision models. Fixed-point compute units are faster and consume less hardware resources and power than floating-point units. It has been shown that the arithmetic operations of deep networks can be encoded down to 8-bit fixed-point without significant performance deterioration [Gupta et al., 2015, Lin et al., 2016a].

Currently deep neural networks are deployed on low-power devices by first training a full-precision model using powerful hardware, and then deriving a corresponding low-precision model for efficient inference on such systems (Figure 1.3). However, training models directly with coarsely quantized weights is a key step towards learning on embedded platforms that have limited computing resources, memory capacity, and power consumption. Recent work have studied methods for training quantized networks, but these studies have mostly been empirical and used high-power devices while keeping full-precision weights while training low-precision

networks [Courbariaux et al., 2015, 2016, Hubara et al., 2016, Rastegari et al., 2016, Zhou et al., 2016b].

1.2.3 What does the loss surface looks like?

Training neural networks requires minimizing a high-dimensional non-convex loss function – a task that is hard in theory, but sometimes easy in practice. Neural network training relies on our ability to find "good" minimizers of highly non-convex loss functions. We already know that certain network architecture designs (e.g., skip connections) produce loss functions that train easier, and well-chosen training parameters (batch size, learning rate, optimizer) produce minimizers that generalize better. However, the reasons for these differences, and their effect on the underlying loss landscape, is not well understood.

1.3 Contributions and Outline

To answer these questions, we propose our solutions in the following sections. The outline of the thesis can be visualized in Figure 1.4.

We summarize our solutions and contributions as follows:

Pruning Filters for Efficient ConvNets. In Chapter 3, we focus on reducing the inference cost of CNNs. We present an acceleration and compression method for CNNs, where we prune filters that are identified as having a small effect on the prediction accuracy. By removing whole filters in the network together with their connected feature maps, the computation costs are reduced accordingly. In contrast

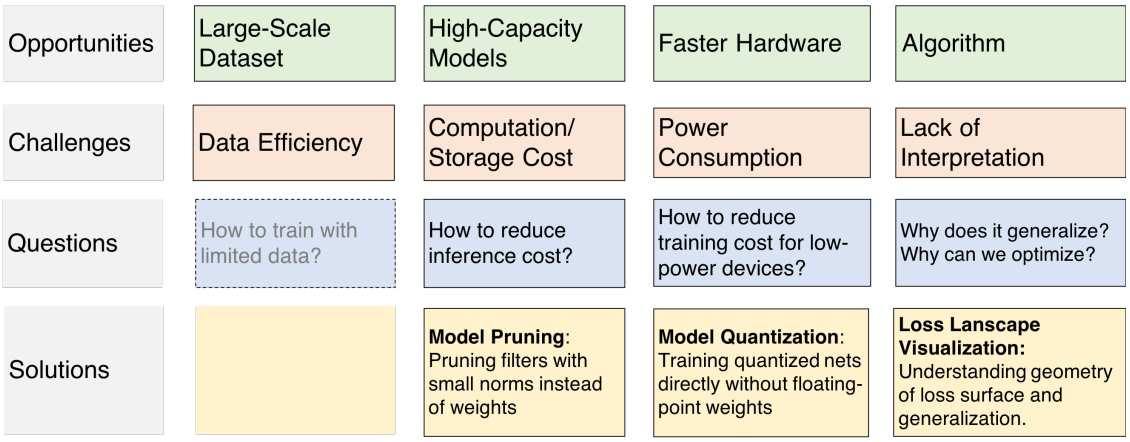


Figure 1.4: The structure of the thesis. Each row represents the enable factors for the success, the challenges we face, the questions we ask and the solutions we provide.

to pruning weights, this approach does not result in sparse connectivity patterns. Hence, it does not need the support of sparse convolution libraries and can work with existing efficient BLAS libraries for dense matrix multiplications. We show that filter pruning techniques can reduce inference costs for VGG-16 by up to 34% and ResNet-110 by up to 38% on CIFAR-10 while regaining close to the original accuracy by retraining the networks.

A Deeper Understanding of Training Quantized Nets. In Chapter 4, we explore optimization algorithms for training discrete neural networks on low-power devices. We study quantized training methods from a theoretical perspective, with the goal of understanding the differences in behavior, and reasons for success or failure, of various methods. We address the issue of why algorithms that maintain floating-point representations work so well, while fully quantized training methods stall before training is complete. We show that the long-term behavior of full-precision training has an important annealing property that is needed for non-convex optimization, while classical rounding methods lack this property.

Visualizing the Loss Landscape of Neural Nets. In Chapter 5, we explore the structure of neural loss functions, and the effect of loss landscapes on generalization, using a range of visualization methods. We introduce a simple "filter normalization" method that helps us visualize loss function curvature, and make meaningful side-by-side comparisons between loss functions. Then, using a variety of visualizations, we explore how network architecture affects the loss landscape, and how training parameters affect the shape of minimizers.

Finally, we summarize the thesis and discuss future work in Chapter 6.

Most content described in this thesis has appeared in previous publications [Li et al., 2017a,b,c]. Specifically, Chapter 3, Chapter 4 and Chapter 5 are based on following works, respectively:

- Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, Hans Peter Graf. *Pruning Filters for Efficient ConvNets. 5th International Conference on Learning Representations (ICLR)*, 2017

- Hao Li*, Soham De*, Zheng Xu, Christoph Studer, Hanan Samet, Tom Goldstein. *Training Quantized Nets: A Deeper Understanding. The 32nd Annual Conference on Neural Information Processing Systems (NIPS)*, 2017 (* denotes equal contribution)

- Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein. *Visualizing the Loss Landscape of Neural Nets.* arXiv preprint, 2017

Chapter 2: Background

In this chapter, we state the basics of training neural networks for the image classification task. We also introduce the common notations that will be used in the next few chapters.

2.1 Deep Neural Networks

Deep Neural networks is essentially a function mapping from the input domain to the output domain. Let $w \in \mathbb{R}^d$ denotes the parameters (weights) of the deep neural network, where d is the dimension of w. Given the input tensor (e.g., image) x_i, the DNN produces its predicted label $\hat{y}$ through multiple layer transformation. In a *feedforward neural network*, each layer maps its input x_i to output via the linear transformation $a_i = w_i x_i + b_i$, where w_i is the weights for the ith layer and b_i is the bias. The output neuron is followed by a non-linear activation function $h(\cdot)$ to the outputs, i.e., $x_{i+1} = h(a_i)$, which is used as the input to the next layer. Currently the most widely used activation function is rectified linear unit (ReLU) [Krizhevsky et al., 2012] where $h(a) = \max(0, a)$.

Different with feedforward neural nets, convolutional neural networks (CNNs) build local connections instead of fully connections. For each neuron in the out-

put channel, it is generated from a subset of its input data with the same set of weights. The weights for generating neurons in an output channel is called a *filter* and different channels have different filters. Each layer i accepts inputs x_i and generates the response corresponding to filter j with convolutional transformation $a_{i,j} = w_{i,j} * x_i + b_{i,j}$, where $w_{i,j}$ represent the weights of j filter at layer i, and $*$ denotes the *convolution* operation. In Chapter 3, we also refer to filter $w_{i,j}$ with the notation $\mathcal{F}_{i,j}$.

2.2 Loss Function

The difference between the prediction and the true label y_i can be measured by *training error*. a better choice is to define a *loss function* $\ell(x_i, y_i; w)$ to measures the inconsistency between the predicted value with parameters w and the label of a data sample. For the multi-class classification problem, usually the *cross-entropy loss* is used as the loss function.

Neural networks are trained on a corpus of input vectors $\{x_i\}$ and accompanying labels $\{y_i\}$ by minimizing an overall loss of the form

$$\mathcal{L}(w) = \frac{1}{N} \sum_{i=1}^{m} \ell(x_i, y_i; w), \tag{2.1}$$

where N is the number of data samples. Due to the nonlinearity of DNNs, the loss function $\mathcal{L}$ is also a non-convex function.

The notation of $\ell(x_i, y_i; w)$ can be simplified by $f_i(w) : \mathbb{R}^d \to \mathbb{R}$ and the

empirical risk minimization problems can be wrote as:

$$\min_{w \in \mathcal{W}} \mathcal{L}(w) := \frac{1}{m} \sum_{i=1}^{m} f_i(w), \tag{2.2}$$

After training, the performance is measured by the *test error* on the test dataset. The performance gap between test error and training error is usually called the *generalization gap*.

2.3 Optimization

The standard method for training neural networks is *stochastic gradient descent (SGD)*. In each iteration, it selects a subset training examples from $\{x_1, x_2, \ldots, x_N\}$, and then computes their averaged gradients $\nabla \tilde{f}(w^t)$. For DNNs, the gradients with respect to w are calculated via *back propagation*. At iteration $t+1$, the weights are updated in the negative gradient direction:

$$w^{t+1} = w^t - \alpha_t \nabla \tilde{f}(w^t), \tag{2.3}$$

for some learning rate α_t.

Chapter 3: Pruning Filters for Efficient ConvNets

CNNs with large capacity usually have significant redundancy among different filters and feature channels. In this chapter, we will focus on reducing the computation cost of well-trained CNNs by pruning filters.

There has been a significant amount of work on reducing the storage costs by model compression [Han et al., 2015, Hassibi and Stork, 1993, Le Cun et al., 1989, Mariet and Sra, 2016, Srinivas and Babu, 2015]. Han et al. [2015, 2016b] report impressive compression rates on AlexNet [Krizhevsky et al., 2012] and VGGNet [Simonyan and Zisserman, 2015] by pruning weights with small magnitudes and then retraining without hurting the overall accuracy. However, pruning parameters does not necessarily reduce the computation time since the majority of the parameters removed are from the fully connected layers where the computation cost is low, e.g., the fully connected layers of VGG-16 occupy 90% of the total parameters but only contribute less than 1% of the overall FLOPs. In addition, recent development on CNNs have yielded deep architectures with more efficient design [He and Sun, 2015, He et al., 2016, Szegedy et al., 2015a,b], in which fully connected layers are often replaced with average pooling layers [Lin et al., 2013], which reduces the number of parameters significantly. The computation cost is also reduced by downsampling

the image at an early stage to reduce the size of feature maps [He and Sun, 2015]. Nevertheless, as the networks continue to become deeper, the computation costs of convolutional layers continue to dominate.

Though convolutional layers can be compressed and accelerated by pruning weights Han et al. [2015], Iandola et al. [2016], it requires sparse BLAS libraries or even specialized hardware to achieve actual speedup [Han et al., 2016a]. Modern libraries that provide sparse operations for CNNs often yield limited speedup [Liu et al., 2015, Szegedy et al., 2015a] and maintaining sparse data structures also creates an additional storage overhead which can be significant for low-precision weights.

Compared to pruning weights across the network, filter pruning is a naturally structured way of pruning without introducing sparsity and therefore does not require using sparse libraries or any specialized hardware. The number of pruned filters correlates directly with acceleration by reducing the number of matrix multiplications, which is easy to tune for a target speedup. Instead of layer-wise iterative fine-tuning (retraining), we adopt a *one-shot* pruning and retraining strategy to save retraining time for pruning filters across multiple layers, which is critical for pruning very deep networks. We observe that even for ResNets [He et al., 2016], which have significantly fewer parameters and inference costs than AlexNet or VGGNet, still have about 30% of FLOPs reduction without sacrificing too much accuracy. We conduct sensitivity analysis for convolutional layers in ResNets that improves the understanding of ResNets.

3.1 Related Work

Le Cun et al. [1989] introduces Optimal Brain Damage, which prunes weights with a theoretically justified saliency measure. Later, Hassibi and Stork [1993] propose Optimal Brain Surgeon to remove unimportant weights determined by the second-order derivative information. Mariet and Sra [2016] reduce the network redundancy by identifying a subset of diverse neurons that does not require retraining. However, this method only operates on the fully-connected layers and introduces sparse connections.

To reduce the computation costs of the convolutional layers, past works have proposed to approximate convolutional operations by representing the weight matrix as a low rank product of two smaller matrices without changing the original number of filters [Denil et al., 2013, Ioannou et al., 2016, Jaderberg et al., 2014, Tai et al., 2016, Zhang et al., 2015]. Other approaches to reduce the convolutional overheads include using FFT based convolutions [Mathieu et al., 2013] and fast convolution using the Winograd algorithm [Lavin and Gray, 2016]. Additionally, quantization can be used to reduce the model size and lower the computation overheads [Courbariaux and Bengio, 2016, Han et al., 2016b, Rastegari et al., 2016]. Our method can be used in addition to these techniques to reduce computation costs without incurring additional overheads.

Several work have studied removing redundant feature maps from a well trained network. Anwar et al. [2015b] introduce a three-level pruning of weights and locate the pruning candidates using particle filtering, which selects the best combination

from a number of random generated masks. Polyak and Wolf [2015] detect the less frequently activated feature maps with sample input data for face detection applications. We choose to analyze the filter weights and prune filters with their corresponding feature maps using a simple magnitude based measure, without examining possible combinations. We also introduce network-wide holistic approaches to prune filters for simple and complex convolutional network architectures.

There is also a growing interest in training compact CNNs with sparse constraints. Lebedev and Lempitsky [2016] leverage group-sparsity on the convolutional filters to achieve structured brain damage, i.e., prune the entries of the convolution kernel in a group-wise fashion. Zhou et al. [2016a] add group-sparse regularization on neurons during training to learn compact CNNs with reduced filters. Wen et al. [2016] add structured sparsity regularizer on each layer to reduce trivial filters, channels or even layers. In the filter-level pruning, all above work use $\ell_{2,1}$-norm as a regularizer. Similar to the above work, we use ℓ_1-norm to select unimportant filters and physically prune them. Our fine-tuning process is the same as the conventional training procedure, without introducing additional regularization. Our approach does not introduce extra layer-wise meta-parameters for the regularizer except for the percentage of filters to be pruned, which is directly related to the desired speedup. By employing stage-wise pruning, we can set a single pruning rate for all layers in one stage.

3.2 Pruning Filters and Feature Maps

Let n_i denote the number of input channels for the ith convolutional layer and h_i/w_i be the height/width of the input feature maps. The convolutional layer transforms the input feature maps $\mathbf{x}_i \in \mathbb{R}^{n_i \times h_i \times w_i}$ into the output feature maps $\mathbf{x}_{i+1} \in \mathbb{R}^{n_{i+1} \times h_{i+1} \times w_{i+1}}$, which are used as input feature maps for the next convolutional layer. This is achieved by applying n_{i+1} 3D filters $\mathcal{F}_{i,j} \in \mathbb{R}^{n_i \times k \times k}$ on the n_i input channels, in which one filter generates one feature map. Each filter is composed by n_i 2D kernels $\mathcal{K} \in \mathbb{R}^{k \times k}$ (e.g., 3×3). All the filters, together, constitute the kernel matrix $\mathcal{F}_i \in \mathbb{R}^{n_i \times n_{i+1} \times k \times k}$. The number of operations of the convolutional layer is $n_{i+1} n_i k^2 h_{i+1} w_{i+1}$.

As shown in Figure 3.1, when a filter $\mathcal{F}_{i,j}$ is pruned, its corresponding feature map $\mathbf{x}_{i+1,j}$ is removed, which reduces $n_i k^2 h_{i+1} w_{i+1}$ operations. The kernels that apply on the removed feature maps from the filters of the next convolutional layer are also removed, which saves an additional $n_{i+2} k^2 h_{i+2} w_{i+2}$ operations. Pruning m filters of layer i will reduce m/n_{i+1} of the computation cost for both layers i and $i+1$.

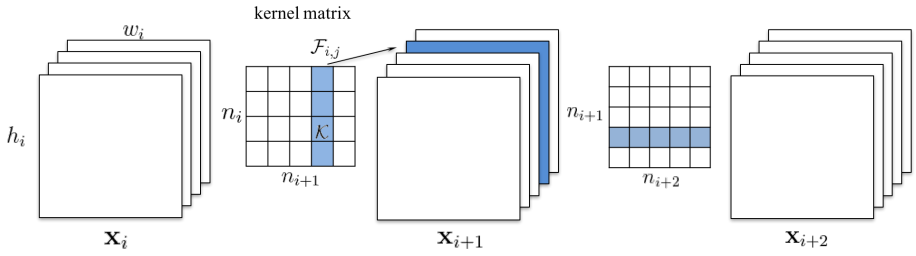


Figure 3.1: Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer.

3.2.1 Determining Which Filters to Prune within a Single Layer

Our method prunes the less useful filters from a well-trained model for computational efficiency while minimizing the accuracy drop. We measure the relative importance of a filter in each layer by calculating the sum of its absolute weights $\sum |\mathcal{F}_{i,j}|$, i.e., its ℓ_1-norm $\|\mathcal{F}_{i,j}\|_1$. Since the number of input channels, n_i, is the same across filters, $\sum |\mathcal{F}_{i,j}|$ also represents the average magnitude of its kernel weights. This value gives an expectation of the magnitude of the output feature map. *Filters with smaller kernel weights tend to produce feature maps with weak activations as compared to the other filters in that layer.* Figure 3.2 illustrates the distribution of filters' absolute weights sum for each convolutional layer in a VGG-16 network trained on the CIFAR-10 dataset, where the distribution varies significantly across layers. We find that pruning the *smallest* filters works better in comparison with pruning the same number of *random* or *largest* filters (Section 3.3.4). Compared to other criteria for activation-based feature map pruning (Section 3.3.5), we find that the ℓ_1-norm is a good criterion for data-free filter selection.

The procedure of pruning m filters from the ith convolutional layer is as follows:

1. For each filter $\mathcal{F}_{i,j}$, calculate its absolute weights sum $\sum_{l=1}^{n_i} \sum |\mathcal{K}_l|$.

2. Sort the filters by $\|\mathcal{F}_{i,j}\|_1$.

3. Prune m filters with the smallest sum values and their corresponding feature maps. The kernels in the next convolutional layer corresponding to the pruned feature maps are also removed.

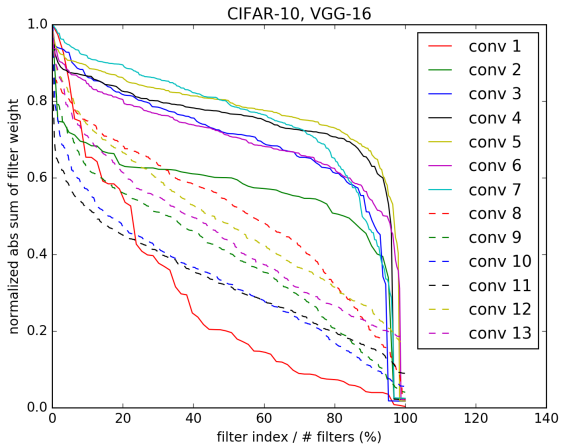
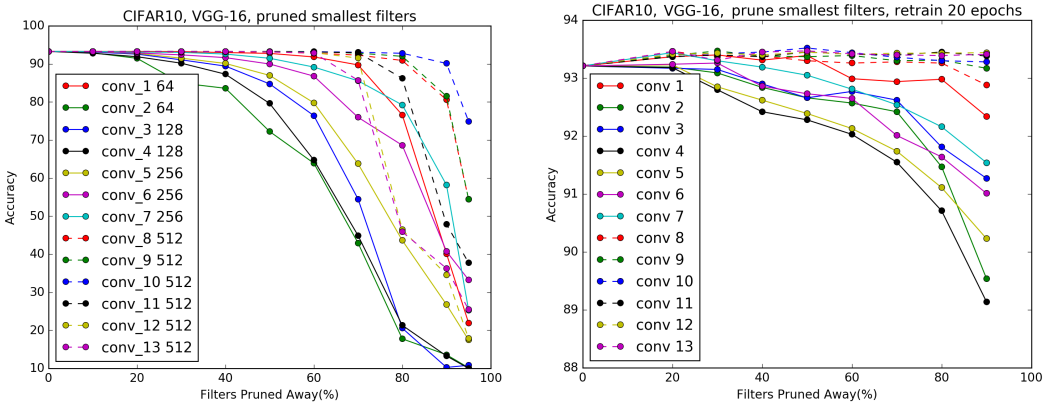


Figure 3.2: Filters are ranked by $\|\mathcal{F}_{i,j}\|_1$.Sorting filters by absolute weights sum for each layer of VGG-16 on CIFAR-10. The x-axis is the filter index divided by the total number of filters. The y-axis is the filter weight sum divided by the max sum value among filters in that layer.

4. A new kernel matrix is created for both the ith and $i+1$th layers, and the remaining kernel weights are copied to the new model.

Relationship to pruning weights Pruning filters with low absolute weights sum is similar to pruning low magnitude weights [Han et al., 2015]. Magnitude-based weight pruning may prune away whole filters when all the kernel weights of a filter are lower than a given threshold. However, it requires a careful tuning of the threshold and it is difficult to predict the exact number of filters that will eventually be pruned. Furthermore, it generates sparse convolutional kernels which can be hard to accelerate given the lack of efficient sparse libraries, especially for the case of low-sparsity.

Relationship to group-sparse regularization on filters Recent work Wen et al. [2016], Zhou et al. [2016a] apply group-sparse regularization ($\sum_{j=1}^{n_i} \|\mathcal{F}_{i,j}\|_2$ or $\ell_{2,1}$-norm) on



(a) Prune the smallest filters (b) Prune and retrain

Figure 3.3: (a) Pruning filters with the lowest absolute weights sum and their corresponding test accuracies on CIFAR-10. (b) Prune and retrain for each single layer of VGG-16 on CIFAR-10. Some layers are sensitive and it can be harder to recover accuracy after pruning them.

convolutional filters, which also favor to zero-out filters with small l_2-norms, i.e. $\mathcal{F}_{i,j} = \mathbf{0}$. In practice, we do not observe a noticeable difference between the ℓ_2-norm and the ℓ_1-norm for filter selection, as the important filters tend to have large values for both measures (Appendix A.1). Zeroing out weights of multiple filters during training has a similar effect to pruning filters with the strategy of iterative pruning and retraining as introduced in Section 3.2.4.

3.2.2 Determining Single Layer's Sensitivity to Pruning

To understand the sensitivity of each layer, we prune each layer independently and evaluate the resulting pruned network's accuracy on the validation set. Figure 3.3(a) shows that layers that maintain their accuracy as filters are pruned away correspond to layers with larger slopes in Figure 3.2. On the contrary, layers with relatively flat slopes are more sensitive to pruning. We empirically determine the

number of filters to prune for each layer based on their sensitivity to pruning. For deep networks such as VGG-16 or ResNets, we observe that layers in the same stage (with the same feature map size) have a similar sensitivity to pruning. To avoid introducing layer-wise meta-parameters, we use the same pruning ratio for all layers in the same stage. For layers that are sensitive to pruning, we prune a smaller percentage of these layers or completely skip pruning them.

3.2.3 Pruning Filters across Multiple Layers

We now discuss how to prune filters across the network. Previous work prunes the weights on a layer by layer basis, followed by iteratively retraining and compensating for any loss of accuracy [Han et al., 2015]. However, understanding how to prune filters of multiple layers at once can be useful: 1) For deep networks, pruning and retraining on a layer by layer basis can be extremely time-consuming 2) Pruning layers across the network gives a holistic view of the robustness of the network resulting in a smaller network 3) For complex networks, a holistic approach may be necessary. For example, for the ResNet, pruning the identity feature maps or the second layer of each residual block results in additional pruning of other layers.

To prune filters across multiple layers, we consider two strategies for layer-wise filter selection:

- *Independent pruning* determines which filters should be pruned at each layer independent of other layers.

- *Greedy pruning* accounts for the filters that have been removed in the previous

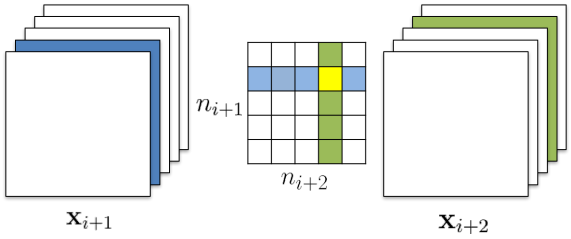


Figure 3.4: Pruning filters across consecutive layers. The independent pruning strategy calculates the filter sum (columns marked in green) without considering feature maps removed in previous layer (shown in blue), so the kernel weights marked in yellow are still included. The greedy pruning strategy does not count kernels for the already pruned feature maps. Both approaches result in a $(n_{i+1} - 1) \times (n_{i+2} - 1)$ kernel matrix.

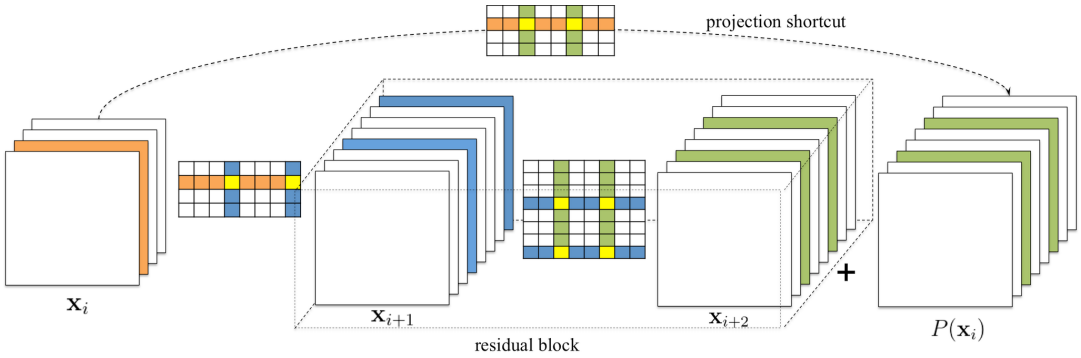


Figure 3.5: Pruning residual blocks with the projection shortcut. The filters to be pruned for the second layer of the residual block (marked as green) are determined by the pruning result of the shortcut projection. The first layer of the residual block can be pruned without restrictions.

layers. This strategy does not consider the kernels for the previously pruned feature maps while calculating the sum of absolute weights.

Figure 3.4 illustrates the difference between two approaches in calculating the sum of absolute weights. The greedy approach, though not globally optimal, is holistic and results in pruned networks with higher accuracy especially when many filters are pruned.

For simpler CNNs like VGGNet or AlexNet, we can easily prune any of the filters in any convolutional layer. However, for complex network architectures such

as ResNets, pruning filters may not be straightforward. The architecture of ResNet imposes restrictions and the filters need to be pruned carefully. We show the filter pruning for residual blocks with projection mapping in Figure 3.5. Here, the filters of the first layer in the residual block can be arbitrarily pruned, as it does not change the number of output feature maps of the block. However, the correspondence between the output feature maps of the second convolutional layer and the identity feature maps makes it difficult to prune. Hence, to prune the second convolutional layer of the residual block, the corresponding projected feature maps must also be pruned. Since the identical feature maps are more important than the added residual maps, the feature maps to be pruned should be determined by the pruning results of the shortcut layer. To determine which identity feature maps are to be pruned, we use the same selection criterion based on the filters of the shortcut convolutional layers (with 1×1 kernels). The second layer of the residual block is pruned with the same filter index as selected by the pruning of the shortcut layer.

3.2.4 Retraining Pruned Networks to Regain Accuracy

After pruning the filters, the performance degradation should be compensated by retraining the network. There are two strategies to prune the filters across multiple layers:

- *Prune once and retrain*: Prune filters of multiple layers at once and retrain them until the original accuracy is restored.

- *Prune and retrain iteratively*: Prune filters layer by layer or filter by filter and

then retrain iteratively. The model is retrained before pruning the next layer for the weights to adapt to the changes from the pruning process.

We find that for the layers that are resilient to pruning, the prune and retrain once strategy can be used to prune away significant portions of the network and any loss in accuracy can be regained by retraining for a short period of time (less than the original training time). However, when some filters from the sensitive layers are pruned away or large portions of the networks are pruned away, it may not be possible to recover the original accuracy. Iterative pruning and retraining may yield better results, but the iterative process requires many more epochs especially for very deep networks.

3.3 Experiments

We prune two types of networks: simple CNNs (VGG-16 on CIFAR-10) and residual networks (ResNet-56/110 on CIFAR-10 and ResNet-34 on ImageNet). Unlike AlexNet or VGGNet (on ImageNet) that are often used to demonstrate model compression, both VGGNet (on CIFAR-10) and ResNets have fewer parameters in the fully connected layers. Hence, pruning a large percentage of parameters from these networks is challenging. We implement our filter pruning method in Torch7 [Collobert et al., 2011]. When filters are pruned, a new model with fewer filters is created and the remaining parameters of the modified layers as well as the unaffected layers are copied into the new model. Furthermore, if a convolutional layer is pruned, the weights of the subsequent batch normalization layer are also

Table 3.1: Overall results. The best test/validation accuracy during the retraining process is reported. Training a pruned model from scratch performs worse than retraining a pruned model, which may indicate the difficulty of training a network with a small capacity.

Model	Error(%)	FLOP	Pruned %	Parameters	Pruned %
VGG-16	6.75	3.13×10^8		1.5×10^7	
VGG-16-pruned-A	**6.60**	2.06×10^8	34.2%	5.4×10^6	64.0%
VGG-16-pruned-A scratch-train	6.88				
ResNet-56	6.96	1.25×10^8		8.5×10^5	
ResNet-56-pruned-A	6.90	1.12×10^8	10.4%	7.7×10^5	9.4%
ResNet-56-pruned-B	**6.94**	9.09×10^7	27.6%	7.3×10^5	13.7%
ResNet-56-pruned-B scratch-train	8.69				
ResNet-110	6.47	2.53×10^8		1.72×10^6	
ResNet-110-pruned-A	**6.45**	2.13×10^8	15.9%	1.68×10^6	2.3%
ResNet-110-pruned-B	6.70	1.55×10^8	38.6%	1.16×10^6	32.4%
ResNet-110-pruned-B scratch-train	7.06				
ResNet-34	26.77	3.64×10^9		2.16×10^7	
ResNet-34-pruned-A	27.44	3.08×10^9	15.5%	1.99×10^7	7.6%
ResNet-34-pruned-B	27.83	2.76×10^9	24.2%	1.93×10^7	10.8%
ResNet-34-pruned-C	27.52	3.37×10^9	7.5%	2.01×10^7	7.2%

removed. To get the baseline accuracies for each network, we train each model from scratch and follow the same pre-processing and hyper-parameters as ResNet [He et al., 2016]. For retraining, we use a constant learning rate 0.001 and retrain 40 epochs for CIFAR-10 and 20 epochs for ImageNet, which represents one-fourth of the original training epochs. Past work has reported up to 3× original training times to retrain pruned networks [Han et al., 2015].

3.3.1 VGG-16 on CIFAR-10

VGG-16 [Simonyan and Zisserman, 2015] is a high-capacity network originally designed for the ImageNet dataset. Recently, Zagoruyko [2015] applies a slightly modified version of the model on CIFAR-10 and achieves state of the art results. As shown in Table 3.2, VGG-16 on CIFAR-10 consists of 13 convolutional layers

Table 3.2: VGG-16 on CIFAR-10 and the pruned model. The last two columns show the number of feature maps and the reduced percentage of FLOP from the pruned model.

layer type	$w_i \times h_i$	#Maps	FLOP	#Params	#Maps	FLOP%
Conv_1	32×32	64	1.8E+06	1.7E+03	32	50%
Conv_2	32×32	64	3.8E+07	3.7E+04	64	50%
Conv_3	16×16	128	1.9E+07	7.4E+04	128	0%
Conv_4	16×16	128	3.8E+07	1.5E+05	128	0%
Conv_5	8×8	256	1.9E+07	2.9E+05	256	0%
Conv_6	8×8	256	3.8E+07	5.9E+05	256	0%
Conv_7	8×8	256	3.8E+07	5.9E+05	256	0%
Conv_8	4×4	512	1.9E+07	1.2E+06	256	50%
Conv_9	4×4	512	3.8E+07	2.4E+06	256	75%
Conv_10	4×4	512	3.8E+07	2.4E+06	256	75%
Conv_11	2×2	512	9.4E+06	2.4E+06	256	75%
Conv_12	2×2	512	9.4E+06	2.4E+06	256	75%
Conv_13	2×2	512	9.4E+06	2.4E+06	256	75%
Linear	1	512	2.6E+05	2.6E+05	512	50%
Linear	1	10	5.1E+03	5.1E+03	10	0%
Total			3.1E+08	1.5E+07		34%

and 2 fully connected layers, in which the fully connected layers do not occupy large portions of parameters due to the small input size and less hidden units. We use the model described in Zagoruyko [2015] but add Batch Normalization [Ioffe and Szegedy, 2015] layer after each convolutional layer and the first linear layer, without using Dropout [Srivastava et al., 2014]. Note that when the last convolutional layer is pruned, the input to the linear layer is changed and the connections are also removed.

As shown in Figure 3.3(a), each of the convolutional layers with 512 feature maps can drop at least 60% of filters without affecting the accuracy. Figure 3.3(b) shows that with retraining, almost 90% of the filters of these layers can be safely removed. One possible explanation is that these filters operate on 4×4 or 2×2 feature maps, which may have no meaningful spatial connections in such small

Figure 3.6: Visualization of filters in the first convolutional layer of VGG-16 trained on CIFAR-10. Filters are ranked by ℓ_1-norm.

dimensions. For instance, ResNets for CIFAR-10 do not perform any convolutions for feature maps below 8×8 dimensions. Unlike previous work [Han et al., 2015, Zeiler and Fergus, 2014], we observe that the first layer is robust to pruning as compared to the next few layers. This is possible for a simple dataset like CIFAR-10, on which the model does not learn as much useful filters as on ImageNet (as shown in Figure. 3.6). Even when 80% of the filters from the first layer are pruned, the number of remaining filters (12) is still larger than the number of raw input channels. However, when removing 80% filters from the second layer, the layer corresponds to a 64 to 12 mapping, which may lose significant information from previous layers, thereby hurting the accuracy. With 50% of the filters being pruned in layer 1 and from 8 to 13, we achieve 34% FLOP reduction for the same accuracy.

3.3.2 ResNet-56/110 on CIFAR-10

ResNets for CIFAR-10 have three stages of residual blocks for feature maps with sizes of 32×32, 16×16 and 8×8. Each stage has the same number of residual blocks. When the number of feature maps increases, the shortcut layer provides an identity mapping with an additional zero padding for the increased dimensions.

Since there is no projection mapping for choosing the identity feature maps, we only consider pruning the first layer of the residual block. As shown in Figure 3.7, most of the layers are robust to pruning. For ResNet-110, pruning some single layers without retraining even improves the performance. In addition, we find that layers that are sensitive to pruning (layers 20, 38 and 54 for ResNet-56, layer 36, 38 and 74 for ResNet-110) lie at the residual blocks close to the layers where the number of feature maps changes, e.g., the first and the last residual blocks for each stage. We believe this happens because the precise residual errors are necessary for the newly added empty feature maps.

The retraining performance can be improved by skipping these sensitive layers. As shown in Table 3.1, ResNet-56-pruned-A improves the performance by pruning 10% filters while skipping the sensitive layers 16, 20, 38 and 54. In addition, we find that deeper layers are more sensitive to pruning than layers in the earlier stages of the network. Hence, we use a different pruning rate for each stage. We use p_i to denote the pruning rate for layers in the ith stage. ResNet-56-pruned-B skips more layers (16, 18, 20, 34, 38, 54) and prunes layers with p_1=60%, p_2=30% and p_3=10%. For ResNet-110, the first pruned model gets a slightly better result with p_1=50% and layer 36 skipped. ResNet-110-pruned-B skips layers 36, 38, 74 and prunes with p_1=50%, p_2=40% and p_3=30%. When there are more than two residual blocks at each stage, the middle residual blocks may be redundant and can be easily pruned. This might explain why ResNet-110 is easier to prune than ResNet-56.

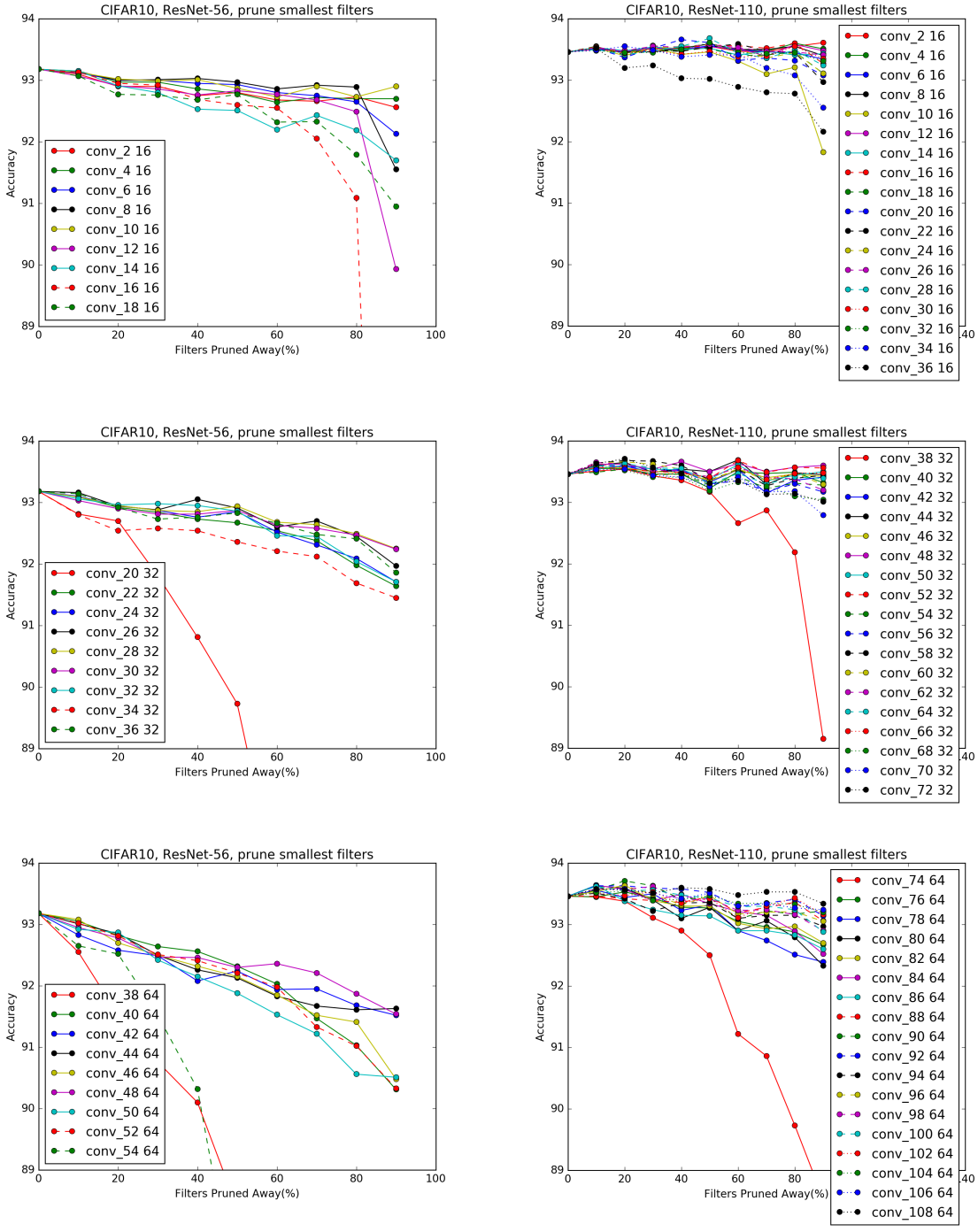
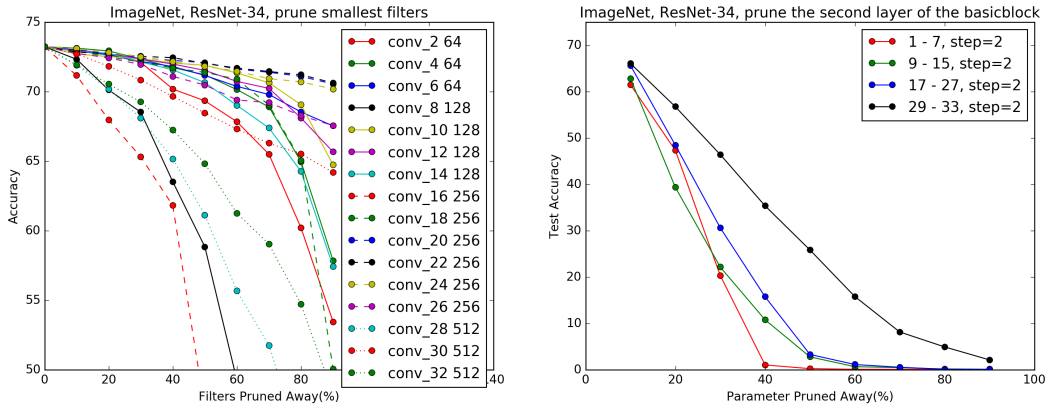


Figure 3.7: Sensitivity to pruning for the first layer of each residual block of ResNet-56 (left column) and ResNet-110 (right column). The three rows correspond to three stages of ResNets with different input sizes (32×32, 16×16, 8×8)

3.3.3 ResNet-34 on ILSVRC2012

ResNets for ImageNet have four stages of residual blocks for feature maps with sizes of 56×56, 28×28, 14×14 and 7×7. ResNet-34 uses the projection shortcut when the feature maps are down-sampled. We first prune the first layer of each residual block. Figure 3.8 shows the sensitivity of the first layer of each residual block. Similar to ResNet-56/110, the first and the last residual blocks of each stage are more sensitive to pruning than the intermediate blocks (i.e., layers 2, 8, 14, 16, 26, 28, 30, 32). We skip those layers and prune the remaining layers at each stage equally. In Table 3.1 we compare two configurations of pruning percentages for the first three stages: (A) p_1=30%, p_2=30%, p_3=30%; (B) p_1=50%, p_2=60%, p_3=40%. Option-B provides 24% FLOP reduction with about 1% loss in accuracy. As seen in the pruning results for ResNet-50/110, we can predict that ResNet-34 is relatively more difficult to prune as compared to deeper ResNets.

We also prune the identity shortcuts and the second convolutional layer of the residual blocks. As these layers have the same number of filters, they are pruned equally. As shown in Figure 3.8(b), these layers are more sensitive to pruning than the first layers. With retraining, ResNet-34-pruned-C prunes the third stage with p_3=20% and results in 7.5% FLOP reduction with 0.75% loss in accuracy. Therefore, pruning the first layer of the residual block is more effective at reducing the overall FLOPs than pruning the second layer. This finding also correlates with the bottleneck block design for deeper ResNets, which first reduces the dimension of input feature maps for the residual layer and then increases the dimension to match



(a) Pruning the first layer of residual blocks (b) Pruning the second layer of residual blocks

Figure 3.8: Sensitivity to pruning for the residual blocks of ResNet-34.

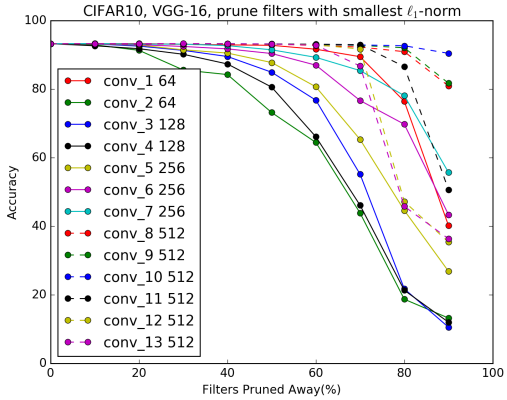
the identity mapping.

3.3.4 Pruning Random Filters and Largest Filters

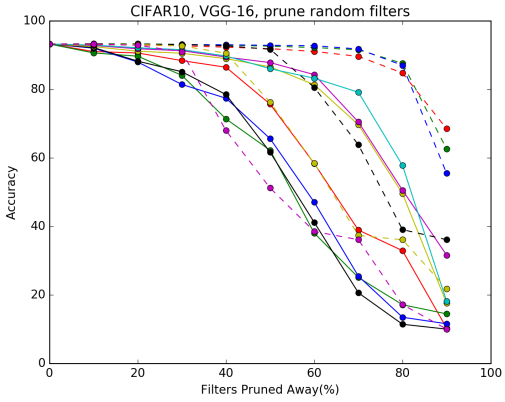
We compare our approach with pruning random filters and largest filters. As shown in Figure 3.9, pruning the smallest filters outperforms pruning random filters for most of the layers at different pruning ratios. For example, smallest filter pruning has better accuracy than random filter pruning for all layers with the pruning ratio of 90%. The accuracy of pruning filters with the largest ℓ_1-norms drops quickly as the pruning ratio increases, which indicates the importance of filters with larger ℓ_1-norms.

3.3.5 Activation-based Feature Map Pruning

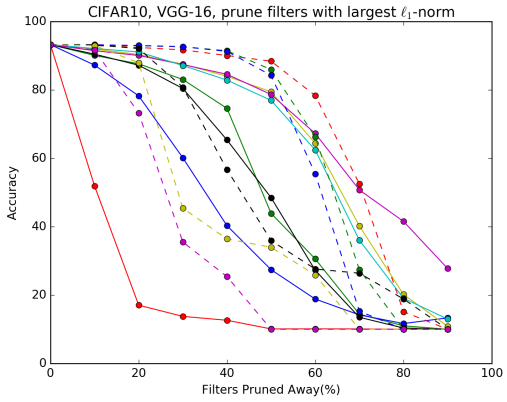
The activation-based feature map pruning method removes the feature maps with weak activation patterns and their corresponding filters and kernels [Polyak and Wolf, 2015], which needs sample data as input to determine which feature



(a) Pruning smallest filters



(b) Pruning random filters



(c) Pruning largest filters

Figure 3.9: Comparison of three pruning methods for VGG-16 on CIFAR-10: pruning the smallest filters, pruning random filters and pruning the largest filters. In random filter pruning, the order of filters to be pruned is randomly permuted.

maps to prune. A feature map $\mathbf{x}_{i+1,j} \in \mathbb{R}^{w_{i+1} \times h_{i+1}}$ is generated by applying filter $\mathcal{F}_{i,j} \in \mathbb{R}^{n_i \times k \times k}$ to feature maps of previous layer $\mathbf{x}_i \in \mathbb{R}^{n_i \times w_i \times h_i}$, i.e., $\mathbf{x}_{i+1,j} = \mathcal{F}_{i,j} * \mathbf{x}_i$. Given N randomly selected images $\{\mathbf{x}_1^n\}_{n=1}^N$ from the training set, the statistics of each feature map can be estimated with one epoch forward pass of the N sampled data. Note that we calculate statistics on the feature maps generated from the convolution operations before batch normalization or non-linear activation. We compare our ℓ_1-norm based filter pruning with feature map pruning using the

following criteria:

$$\sigma_{\texttt{mean-mean}}(\mathbf{x}_{i,j}) = \frac{1}{N}\sum_{n=1}^{N} \texttt{mean}(\mathbf{x}_{i,j}^{n}) \tag{3.1}$$

$$\sigma_{\texttt{mean-std}}(\mathbf{x}_{i,j}) = \frac{1}{N}\sum_{n=1}^{N} \texttt{std}(\mathbf{x}_{i,j}^{n}) \tag{3.2}$$

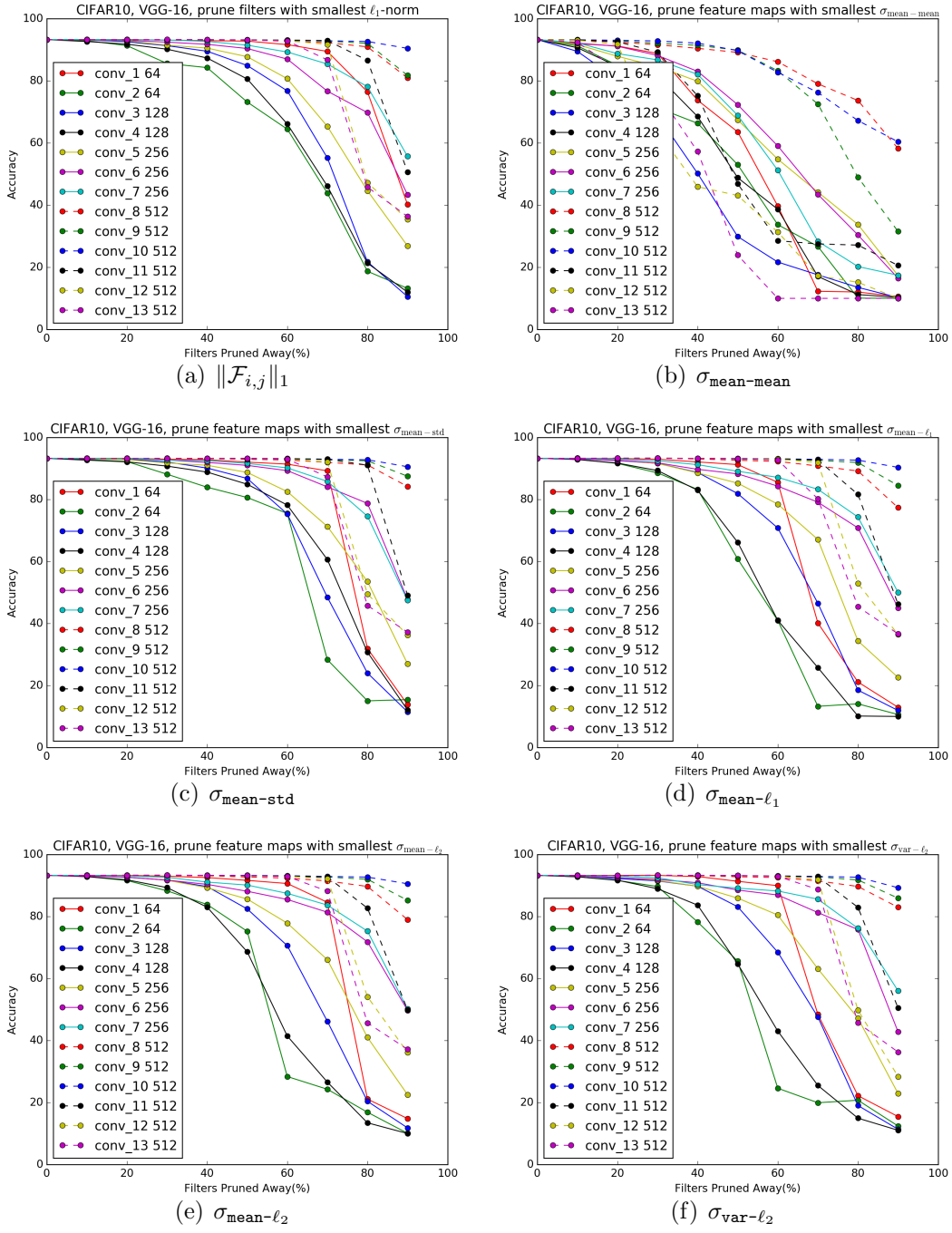
$$\sigma_{\texttt{mean-}\ell_1}(\mathbf{x}_{i,j}) = \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_{i,j}^{n}\|_1 \tag{3.3}$$

$$\sigma_{\texttt{mean-}\ell_2}(\mathbf{x}_{i,j}) = \frac{1}{N}\sum_{n=1}^{N} \|\mathbf{x}_{i,j}^{n}\|_2 \tag{3.4}$$

$$\sigma_{\texttt{var-}\ell_2}(\mathbf{x}_{i,j}) = \texttt{var}(\{\|\mathbf{x}_{i,j}^{n}\|_2\}_{n=1}^{N}) \tag{3.5}$$

where `mean`, `std` and `var` are standard statistics (average, standard deviation and variance) of the input. Here, $\sigma_{\texttt{var-}\ell_2}$ is the *contribution variance of channel* criterion proposed by Polyak and Wolf [2015], which is motivated by the intuition that an unimportant feature map has almost similar outputs for the whole training data and acts like an additional bias.

The estimation of the criteria becomes more accurate when more sample data is used. Here we use the whole training set ($N = 50,000$ for CIFAR-10) to compute the statistics. The performance of feature map pruning with above criteria for each layer is shown in Figure 3.10. Smallest filter pruning outperforms feature map pruning with the criteria $\sigma_{\texttt{mean-mean}}$, $\sigma_{\texttt{mean-}\ell_1}$, $\sigma_{\texttt{mean-}\ell_2}$ and $\sigma_{\texttt{var-}\ell_2}$. The $\sigma_{\texttt{mean-std}}$ criterion has better or similar performance to ℓ_1-norm up to pruning ratio of 60%. However, its performance drops quickly after that especially for layers of `conv_1`, `conv_2` and



(a) $\|\mathcal{F}_{i,j}\|_1$ (b) $\sigma_{\texttt{mean-mean}}$

(c) $\sigma_{\texttt{mean-std}}$ (d) $\sigma_{\texttt{mean-}\ell_1}$

(e) $\sigma_{\texttt{mean-}\ell_2}$ (f) $\sigma_{\texttt{var-}\ell_2}$

Figure 3.10: Comparison between filter pruning (a) and activation-based feature map pruning (b-f) for VGG-16 on CIFAR-10.

`conv_3`. We find ℓ_1-norm is a good heuristic for filter selection considering that it is data free.

3.4 Summary

This chapter presented a method to prune filters with relatively low weight magnitudes to produce CNNs with reduced computation costs without introducing irregular sparsity. It achieves about 30% reduction in FLOPs for VGG-Net (on CIFAR-10) and deep ResNets without significant loss in the original accuracy. Instead of pruning with specific layer-wise hayperparameters and time-consuming iterative retraining, we use the one-shot pruning and retraining strategy for simplicity and ease of implementation. By performing lesion studies on very deep CNNs, we identify layers that are robust or sensitive to pruning, which can be useful for further understanding and improving the architectures.

Chapter 4: Training Quantized Nets: A Deeper Understanding

To make DNNs efficient on embedded systems, many researchers have focused on training them with *coarsely quantized* weights. For example, weights may be constrained to take on integer/binary values, or may be represented using low-precision fixed-point numbers (8 bits or less). Quantized DNNs offer the potential of superior memory and computation efficiency, while achieving performance that is competitive with state-of-the-art full-precision DNNs. Quantized weights can dramatically reduce memory size and access bandwidth, increase power efficiency, exploit hardware-friendly bitwise operations, and accelerate inference throughput [Courbariaux et al., 2016, Marchesi et al., 1993, Rastegari et al., 2016].

Despite these benefits, handling low-precision weights is difficult and motivates interest in new training methods. When the learning rate is small, stochastic gradient methods make small updates to weight parameters. Discretization of weights after each training iteration "rounds off" these small updates and causes training to stagnate [Courbariaux et al., 2016]. Thus, the naive approach of quantizing weights using a rounding procedure yields poor results when weights are represented using a small number of bits. As a result, successful methods for training neural nets with discrete values usually maintain full-precision weights so that tiny changes can

be accumulated [Courbariaux et al., 2015, 2016, Rastegari et al., 2016]. Other approaches include classical *stochastic rounding* methods [Gupta et al., 2015]. While some of these schemes seem to work in practice, results in this area are largely experimental, and little work has been devoted to explaining the excellent performance of some methods, the poor performance of others, and the important differences in behavior between these methods.

Removing the dependence of real-valued weights is important for training on resource-constrained hardware and simplifying the hardware implementation. In this chapter, we study quantized training methods from a theoretical perspective, with the goal of understanding the differences in behavior, and reasons for success or failure, of various methods. In particular, we present a convergence analysis showing that both the classical stochastic rounding (SR) methods [Gupta et al., 2015] as well as newer and more powerful methods like BinaryConnect (BC) [Courbariaux et al., 2015] are capable of solving convex discrete problems up to a level of accuracy that depends on the quantization level. We then address the issue of why algorithms that maintain floating-point representations, like BC, work so well, while fully quantized training methods like SR stall before training is complete. We show that the long-term behavior of BC has an important annealing property that is needed for non-convex optimization, while classical rounding methods lack this property.

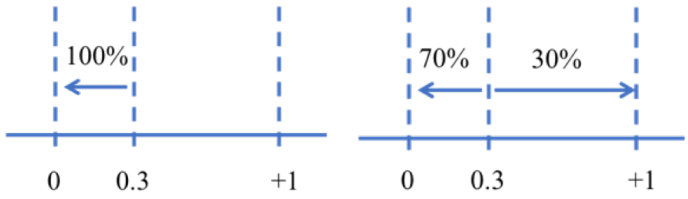
4.1 Related Work

Previous work on obtaining a quantized CNN can be divided into two categories: 1) qunatizing a pre-trained model and 2) raining a quantized model from scratch. We focus on approaches that belong to the second category, as they can be used for accelerating both training and inference under constrained resources.

Qunatizing a pre-trained model with or without retraining The goal of quantizing a pre-trained network is to minimize the difference between the quantized model and the high-precision model. Most approaches adopt quantization with retraining [Anwar et al., 2015a, Courbariaux et al., 2015, Hwang and Sung, 2014, Rastegari et al., 2016]. Converting a pre-trained network into fixed point representation without retraining has been explored in Lin et al. [2016a], which use an optimization strategy based on signal-to-quantization-noise-ratio and identify the optimal bit-width allocation for each layer. Anwar et al. [2015a] analyzed the quantization sensitivity of the network for each layer and then manually decide the quantization bit-widths. Rastegari et al. [2016] propose to use adaptive scale for each filter to better approximate the real-valued models. The extreme scenario of quantization is binarization, in which only 1-bit (2 states) is available for the weights. With binary weights $\{-1, 1\}$, the multiplication operations can be replaced by addition and subtraction operations [Courbariaux et al., 2015, 2016, Hubara et al., 2016, Kim and Smaragdis, 2015, Rastegari et al., 2016]. A relaxation of the binary representation is the ternary representation, which allows weights to be zero and enables higher capacity of ex-

pression [Hwang and Sung, 2014, Li et al., 2016, Lin et al., 2016b]. e.g., [Li et al., 2016] show that ternary weight quantization into levels $\{-1, 0, 1\}$ only have slight accuracy loss even on large dataset. Quantizing both weights and activations into a binary or a power-of-two representation is more attractive , as the multiplications in forward and backward propagation can be replaced by XNOR or binary shifts [Courbariaux et al., 2016, Hubara et al., 2016, Lin et al., 2016b, Miyashita et al., 2016, Rastegari et al., 2016]. However, quantizating activations requires processing with non-differentiable quantization functions. To optimize a neural network with non-differentiable quantization functions, a *straight through estimator*(STE) is widely used for back propagation [Bengio et al., 2013b].

Training a quantized model from scratch Previous work on training a quantized network usually require keeping sufficient resolution for weights and gradients, in which binarized weights are used in forward propagation and backward propagation while the real-valued weights are kept for accumulating updates [Courbariaux et al., 2015, 2016, Hubara et al., 2016, Rastegari et al., 2016, Zhou et al., 2016b]. However, a floating point unit may not be available for certain low power devices, e.g., the biology-inspired spiking neural networks is naturally binary and can be computationally efficient with specific hardware [Esser et al., 2015]. One of the widely used solutions for training low-bits fixed point networks is *stochastic rounding*, which solves the problem of tiny gradients for low bit-width weights. Training neural networks with stochastic rounding was studied in the early 90's [Höhfeld and Fahlman, 1992] and was recently revisited for training low-precision networks [Gupta et al.,



(a) Deterministic rounding (b) Stochastic rounding

Figure 4.1: Given a random number 0.3 and quantization level $\Delta = 1$, deterministic rounding (Q_d) always produces 0 while stochastic rounding (Q_s) will produce 0 with 70% chance and 1 with 30% chance.

2015]. Gupta et al. [2015] trains networks using 16-bit fixed-point representation with stochastic rounding and delivers results nearly identical as 32-bit floating-point computations. However, the accumulation result is still stored with 48-bit to avoid loss of precision. Zhou et al. [2016b] successfully train a CNN with 1-bit weights and 2-bits activations using 6-bit gradients, but they also require keeping full-precision weights.

4.2 Training Quantized Neural Nets

To train neural networks using a low-precision representation of the weights, a quantization function $Q(\cdot)$ is required that converts a real-valued number w into a quantized version $\hat{w} = Q(w)$. We use the same notation for quantizing vectors, where we assume Q acts on each dimension of the vector. Different quantized optimization routines can be defined by selecting different quantizers, and also by selecting when quantization happens during optimization. The common options are:

Deterministic Rounding (R) A basic *uniform* or deterministic quantization function converts a floating point value to the closest quantized value as:

$$Q_d(w) = \text{sign}(w) \cdot \Delta \cdot \left\lfloor \frac{|w|}{\Delta} + \frac{1}{2} \right\rfloor \tag{4.1}$$

where Δ denotes the quantization step or resolution, i.e., the smallest positive number that is representable. One exception to this definition is when we consider binary weights, where all weights are constrained to have two values $w \in \{-1, 1\}$ and uniform rounding becomes $Q_d(w) = \text{sign}(w)$.

The deterministic rounding SGD maintains quantized weights with updates of the form:

$$\text{Deterministic Rounding:} \quad w_b^{t+1} = Q_d\big(w_b^t - \alpha_t \nabla \tilde{f}(w_b^t)\big), \tag{4.2}$$

where w_b denotes the low-precision weights, which are quantized using Q_d immediately after applying the gradient descent update. If gradient updates are significantly smaller than the quantization step, this method loses gradient information and weights may never be modified from their starting values.

Stochastic Rounding (SR) The quantization function for *stochastic rounding* is defined as:

$$Q_s(w) = \Delta \cdot \begin{cases} \lfloor \frac{w}{\Delta} \rfloor + 1 & \text{for } p \leq \frac{w}{\Delta} - \lfloor \frac{w}{\Delta} \rfloor, \\ \lfloor \frac{w}{\Delta} \rfloor & \text{otherwise,} \end{cases} \tag{4.3}$$

where $p \in [0, 1]$ is produced by a uniform random number generator. This operator is non-deterministic, and rounds its argument up with probability $w/\Delta - \lfloor w/\Delta \rfloor$, and down otherwise. This quantizer satisfies the important property $\mathbb{E}[Q_s(w)] = w$. Similar to the deterministic rounding method, the SR optimization method also maintains quantized weights with updates of the form:

$$\text{Stochastic Rounding:} \quad w_b^{t+1} = Q_s\big(w_b^t - \alpha_t \nabla \tilde{f}(w_b^t)\big). \tag{4.4}$$

BinaryConnect (BC) The BinaryConnect algorithm [Courbariaux et al., 2015] accumulates gradient updates using a full-precision buffer w_r, and quantizes weights only just before gradient computations. BinaryConnect uses updates of the form:

$$\text{BinaryConnect:} \quad w_r^{t+1} = w_r^t - \alpha_t \nabla \tilde{f}\big(Q(w_r^t)\big). \tag{4.5}$$

Either stochastic rounding Q_s or deterministic rounding Q_d can be used for quantizing the weights w_r, but in practice, Q_d is the common choice. The original BinaryConnect paper constrains the weights to be $\{-1, 1\}$, which can be generalized to $\{-\Delta, \Delta\}$. A more recent method, Binary-Weights-Net (BWN) [Rastegari et al., 2016], allows different filters to have different scales for quantization, which often results in better performance on large datasets.

The above algorithms can be summarized in Algorithm 1. In the rest of this chapter, we use Q to denote both Q_s and Q_d unless the situation requires this to be distinguished. We also drop the subscripts on w_r and w_b, and simply write w.

Algorithm 1: Algorithms for training DNNs with quantized weights

Input: neural network f, mini-batch data $\{\mathbf{x}^t, \mathbf{y}^t\}_{t=0}^{T-1}$, learning rate α^t
Output: quantized weights w_b
$w_r^0 = w_b^0 = Q_d(\texttt{rand})$
for $t = 0$ **to** $T-1$ **do**
  forward and back propagation to get gradient $\nabla \tilde{f}(w_b^t)$.
  if BC-SGD **then**
    $w_r^{t+1} = w_r^t - \alpha^t \nabla \tilde{f}(w_b^t)$
    $w_b^{t+1} = Q_d(w_r^{t+1})$
  else if R-SGD **then**
    $w_b^{t+1} = Q_d(w_b^t - \alpha^t \nabla \tilde{f}(w_b^t))$
  else if SR-SGD **then**
    $w_b^{t+1} = Q_s(w_b^t - \alpha^t \nabla \tilde{f}(w_b^t))$
  end if
end for

4.3 Convergence Analysis

We now present convergence guarantees for the Stochastic Rounding (SR) and BinaryConnect (BC) algorithms, with updates of the form (4.4) and (4.5), respectively. For the purposes of deriving theoretical guarantees, we assume each f_i in (2.2) is differentiable and convex, and the domain $\mathcal{W}$ is convex and has dimension d. We consider both the case where the loss function L is μ-strongly convex: $\langle \nabla \mathcal{L}(w'), w - w' \rangle \leq \mathcal{L}(w) - \mathcal{L}(w') - \frac{\mu}{2}\|w - w'\|^2$, as well as where L is weakly convex. We also assume the (stochastic) gradients are bounded: $\mathbb{E}\|\nabla \tilde{f}(w^t)\|^2 \leq G^2$. Some results below also assume the domain of the problem is finite. In this case, the rounding algorithm clips values that leave the domain. For example, in the binary case, rounding returns bounded values in $\{-1, 1\}$.

4.3.1 Convergence of Stochastic Rounding (SR)

We can rewrite the update rule (4.4) as:

$$w^{t+1} = w^t - \alpha_t \nabla \tilde{f}(w^t) + r^t,$$

where $r^t = Q_s(w^t - \alpha_t \nabla \tilde{f}(w^t)) - w^t + \alpha_t \nabla \tilde{f}(w^t)$ denotes the quantization error on the t-th iteration. We want to bound this error in expectation. To this end, we present the following lemma.

Lemma 1. *The stochastic rounding error r^t on each iteration can be bounded, in expectation, as:*

$$\mathbb{E}\left\|r^t\right\|^2 \leq \sqrt{d}\Delta\alpha_t G,$$

where d denotes the dimension of w.

Proofs for all theoretical results are presented in the Appendix B. From Lemma 1, we see that the rounding error per step decreases as the learning rate α_t decreases. This is intuitive since the probability of an entry in w^{t+1} differing from w^t is small when the gradient update is small relative to Δ. Using the above lemma, we now present convergence rate results for Stochastic Rounding (SR) in both the strongly-convex case and the non-strongly convex case. Our error estimates are ergodic, i.e., they are in terms of $\bar{w}^T = \frac{1}{T}\sum_{t=1}^{T} w^t$, the average of the iterates.

Theorem 1. *Assume that $\mathcal{L}$ is μ-strongly convex and the learning rates are given*

by $\alpha_t = \frac{1}{\mu(t+1)}$. *Consider the SR algorithm with updates of the form* (4.4). *Then, we have:*

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{(1+\log(T+1))G^2}{2\mu T} + \frac{\sqrt{d}\Delta G}{2},$$

where $w^\star = \arg\min_w \mathcal{L}(w)$.

Theorem 2. *Assume the domain has finite diameter* D, *and learning rates are given by* $\alpha_t = \frac{c}{\sqrt{t}}$, *for a constant* c. *Consider the SR algorithm with updates of the form* (4.4). *Then, we have:*

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{1}{c\sqrt{T}}D^2 + \frac{\sqrt{T+1}}{2T}cG^2 + \frac{\sqrt{d}\Delta G}{2}.$$

We see that in both cases, SR converges until it reaches an "accuracy floor." As the quantization becomes more fine grained, our theory predicts that the accuracy of SR approaches that of high-precision floating point at a rate linear in Δ. This extra term caused by the discretization is unavoidable since this method maintains quantized weights.

4.3.2 Convergence of Binary Connect (BC)

When analyzing the BC algorithm, we assume that the Hessian satisfies the Lipschitz bound: $\|\nabla^2 f_i(x) - \nabla^2 f_i(y)\| \leq L_2\|x-y\|$ for some $L_2 \geq 0$. While this is a slightly non-standard assumption, we will see that it enables us to gain better insights into the behavior of the algorithm.

The results here hold for both stochastic and uniform rounding. In this case, the quantization error r does not approach 0 as in SR-SGD. Nonetheless, the effect of this rounding error diminishes with shrinking α_t because α_t multiplies the gradient update, and thus implicitly the rounding error as well.

Theorem 3. *Assume $\mathcal{L}$ is L-Lipschitz smooth, the domain has finite diameter D, and learning rates are given by $\alpha_t = \frac{c}{\sqrt{t}}$. Consider the BC-SGD algorithm with updates of the form (4.5). Then, we have:*

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{1}{2c\sqrt{T}} D^2 + \frac{\sqrt{T+1}}{2T} cG^2 + \sqrt{d}\Delta LD.$$

As with SR, BC can only converge up to an error floor. So far this looks a lot like the convergence guarantees for SR. However, things change when we assume strong convexity and bounded Hessian.

Theorem 4. *Assume that $\mathcal{L}$ is μ-strongly convex and the learning rates are given by $\alpha_t = \frac{1}{\mu(t+1)}$. Consider the BC algorithm with updates of the form (4.5). Then we have:*

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{(1 + \log(T+1))G^2}{2\mu T} + \frac{DL_2\sqrt{d}\Delta}{2}.$$

Now, the error floor is determined by both Δ and L_2. For a quadratic least-squares problem, the gradient of $\mathcal{L}$ is linear and the Hessian is constant. Thus, $L_2 = 0$ and we get the following corollary.

Corollary 1. *Assume that $\mathcal{L}$ is quadratic and the learning rates are given by $\alpha_t =$*

$\frac{1}{\mu(t+1)}$. *The BC algorithm with updates of the form* (4.5) *yields*

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{(1+\log(T+1))G^2}{2\mu T}.$$

We see that the real-valued weights accumulated in BC can converge to the *true minimizer* of quadratic losses. Furthermore, this suggests that, when the function behaves like a quadratic on the distance scale Δ, one would expect BC to perform fundamentally better than SR. While this may seem like a restrictive condition, there is evidence that even non-convex neural networks become well approximated as a quadratic in the later stages of optimization within a neighborhood of a local minimum [Martens and Grosse, 2015].

Note that our convergence results on BC are for w_r instead of w_b, and these measures of convergence are not directly comparable. It is not possible to bound w_b when BC is used, as the values of w_b may not converge in the usual sense (e.g., in the +/-1 binary case w_r might converge to 0, in which case arbitrarily small perturbations to w_r might send w_b to +1 or -1).

4.4 What About Non-Convex Problems?

The global convergence results presented for convex problems show that, in general, both the SR and BC algorithms converge to within $\mathcal{O}(\Delta)$ accuracy of the minimizer (in expected value). However, these results do not explain the large differences between these methods when applied to non-convex neural nets. We study how the long-term behavior of SR differs from BC. Typical (continuous-valued) SGD

methods have an important exploration-exploitation tradeoff. When the learning rate is large, the algorithm explores by moving quickly between states. Exploitation happens when the learning rate is small. In this case, noise averaging causes the algorithm to more greedily pursue local minimizers with lower loss values. Thus, the distribution of iterates produced by the algorithm becomes increasingly concentrated near minimizers as the learning rate vanishes (see, e.g., the large-deviation estimates in [Lan et al., 2012]). BC maintains this property as well.

In this section, we show that the SR method lacks this important tradeoff: as the stepsize gets small and the algorithm slows down, the quality of the iterates produced by the algorithm does *not* improve, and the algorithm does *not* become progressively more likely to produce low-loss iterates. This behavior is illustrated in Figures 4.2 and 4.4. Note that this section makes no convexity assumptions, and the proposed theoretical results are directly applicable to neural networks.

To understand this problem conceptually, consider the simple case of a one-variable optimization problem starting at $x^0 = 0$ with $\Delta = 1$ (Figure 4.2). On each iteration, the algorithm computes a stochastic approximation $\nabla \tilde{f}$ of the gradient by sampling from a distribution, which we call p. This gradient is then multiplied by the stepsize to get $\alpha \nabla \tilde{f}$. The probability of moving to the right (or left) is then roughly proportional to the magnitude of $\alpha \nabla \tilde{f}$. Note the random variable $\alpha \nabla \tilde{f}$ has distribution $p_\alpha(z) = \alpha^{-1} p(z/\alpha)$.

Now, suppose that α is small enough that we can neglect the tails of $p_\alpha(z)$ that lie outside the interval $[-1, 1]$. The probability of transitioning from $x^0 = 0$ to

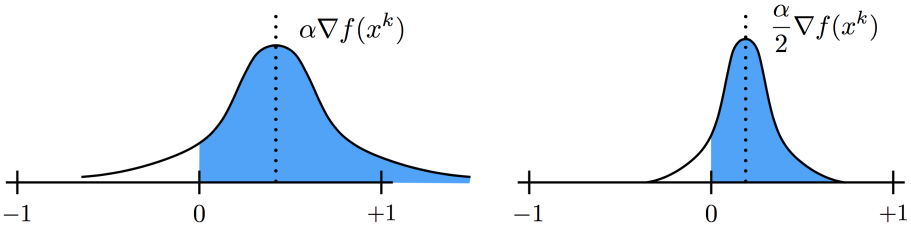


Figure 4.2: The SR method starts at some location x (in this case 0), adds a perturbation to x, and then rounds. As the learning rate α gets smaller, the distribution of the perturbation gets "squished" near the origin, making the algorithm less likely to move. The "squishing" effect is the same for the part of the distribution lying to the left and to the right of x, and so it does not effect the *relative* probability of moving left or right.

$x^1 = 1$ using stochastic rounding, denoted by $T_\alpha(0, 1)$, is then

$$T_\alpha(0,1) \approx \int_0^1 z p_\alpha(z) dz = \frac{1}{\alpha}\int_0^1 z p(z/\alpha)\, dz = \alpha \int_0^{1/\alpha} p(x) x\, dx \approx \alpha \int_0^\infty p(x) x\, dx,$$

where the first approximation is because we neglected the unlikely case that $\alpha\nabla\tilde{f} > 1$, and the second approximation appears because we added a small tail probability to the estimate. These approximations get more accurate for small α. We see that, assuming the tails of p are "light" enough, we have $T_\alpha(0,1) \sim \alpha \int_0^\infty p(x) x\, dx$ as $\alpha \to 0$. Similarly, $T_\alpha(0,-1) \sim \alpha \int_{-\infty}^0 p(x) x\, dx$ as $\alpha \to 0$.

What does this observation mean for the behavior of SR? First of all, the probability of leaving x^0 on an iteration is

$$T_\alpha(0,-1) + T_\alpha(0,1) \approx \alpha \left[\int_0^\infty p(x) x\, dx + \int_{-\infty}^0 p(x) x\, dx \right],$$

which vanishes for small α. This means the algorithm slows down as the learning rate drops off, which is not surprising. However, the *conditional* probability of ending up

at $x^1 = 1$ given that the algorithm *did* leave x^0 is

$$T_\alpha(0, 1|x^1 \neq x^0) \approx \frac{T_\alpha(0,1)}{T_\alpha(0,-1) + T_\alpha(0,1)} = \frac{\int_0^\infty p(x)x\,dx}{\int_{-\infty}^0 p(x)x\,dx + \int_0^\infty p(x)x\,dx}, \tag{4.6}$$

which does not depend on α. In other words, provided α is small, SR, on average, makes the same decisions/transitions with learning rate α as it does with learning rate $\alpha/10$; it just takes 10 times longer to make those decisions when $\alpha/10$ is used. In this situation, there is no exploitation benefit in decreasing α.

4.4.1 Toy Problem

To gain more intuition about the effect of shrinking the learning rate in SR vs BC, consider the following simple 1-dimensional non-convex problem:

$$\min_w \mathcal{L}(w) := \begin{cases} w^2 + 2, & \text{if } w < 1, \\ (w - 2.5)^2 + 0.75, & \text{if } 1 \leq w < 3.5, \\ (w - 4.75)^2 + 0.19, & \text{if } w \geq 3.5. \end{cases} \tag{4.7}$$

Figure 4.3 shows a plot of this loss function.

To visualize the distribution of iterates, we initialize at $w = 4.0$, and run SR and BC for 10^6 iterations using a quantization resolution of 0.5. Figure 4.4 shows the distribution of the quantized weight parameters w over the iterations when optimized with SR and BC for different learning rates α. As we shift from $\alpha = 1$ to $\alpha = 0.001$, the distribution of BC iterates transitions from a wide/explorative distribution to

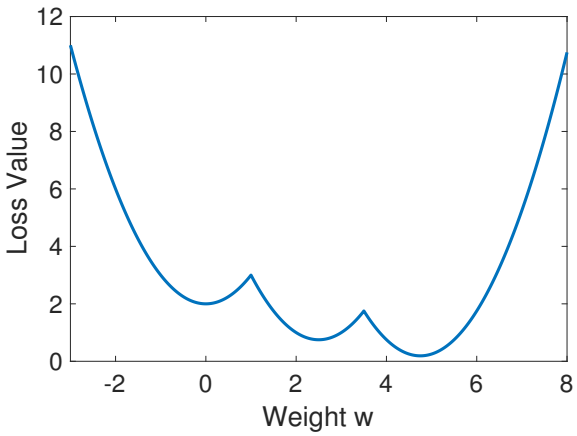
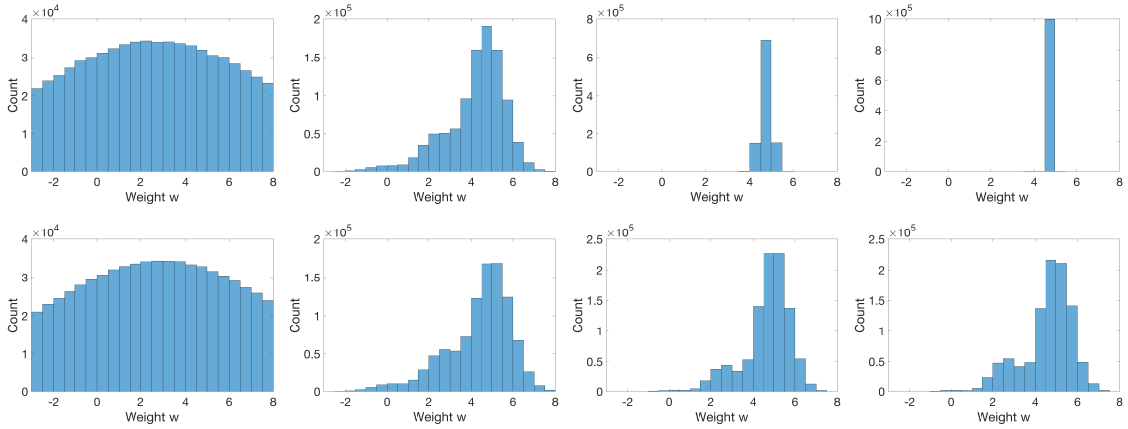


Figure 4.3: The objective function for the toy problem of Eq.(4.7).

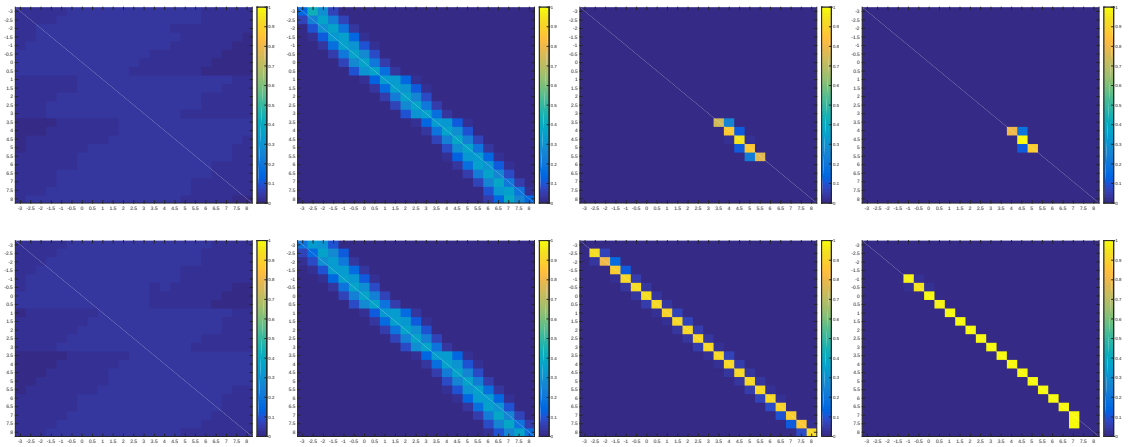


(a) $\alpha = 1.0$ (b) $\alpha = 0.1$ (c) $\alpha = 0.01$ (d) $\alpha = 0.001$

Figure 4.4: Effect of shrinking the learning rate in SR vs BC on a toy problem. Histograms plot the distribution of the quantized weights over 10^6 iterations. The top row of plots corresponds to BC, while the bottom row is SR, for different learning rates α. As the learning rate α shrinks, the BC distribution concentrates on a minimizer, while the SR distribution stagnates.

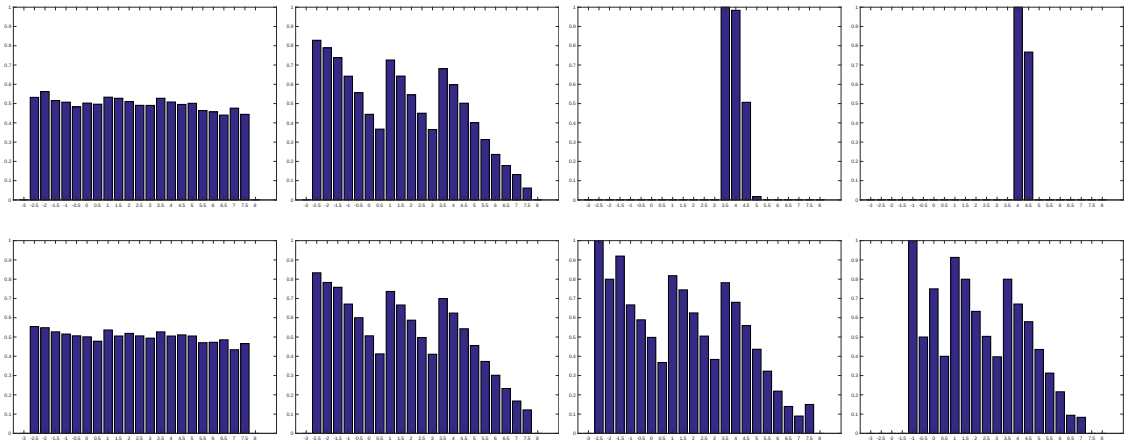
a narrow distribution in which iterates aggressively concentrate on the minimizer. In contrast, the distribution produced by SR concentrates only slightly and then stagnates; the iterates are spread widely even when the learning rate is small.

We can calculate the exact state transition matrix $T_\alpha(x, y)$ given the limit of the state space. Figure 4.5 shows the transition probability matrix for the toy problem. We can see that as the learning rate shrinks, the transition probabilities of



(a) $\alpha = 1.0$ (b) $\alpha = 0.1$ (c) $\alpha = 0.01$ (d) $\alpha = 0.001$

Figure 4.5: The transition probability matrix $T_\alpha(x, y)$ for the toy problem. The top row of plots corresponds to BC, while the bottom row is SR, for different learning rates α. Each axis shows the possible states (exact discrete weights).



(a) $\alpha = 1.0$ (b) $\alpha = 0.1$ (c) $\alpha = 0.01$ (d) $\alpha = 0.001$

Figure 4.6: The conditional transition probability $T_\alpha(x, x{+}1)/(T_\alpha(x, x{-}1){+}T_\alpha(x, x{+}1))$ for the toy problem. The top row of plots corresponds to BC, while the bottom row is SR, for different learning rates α. The the x axis is the weight space and the y axis is the probability ranging from 0 to 1.

jumping from one state to the other decreases while the probabilities of staying at the same state (the diagonal entries) increases. On the other hand, BC has non-zero transition probabilities around the minimum while SR spans nearly all states even with very small learning rate.

We can also show the conditional transition probability $T_\alpha(x, x+1)/(T_\alpha(x, x-1) + T_\alpha(x, x+1))$. Figure 4.2 and Eq 4.6 can be verified by Figure 4.6, which shows the relative probability of jumping to the next right state comparing to jumping to adjacent states in both directions. We can see that as the learning rate shrinks, the conditional probability distribution does not change for SR, while it improves for BC (the state in the left of the local minimum has better chance of jumping right).

4.4.2 Asymptotic Analysis of Stochastic Rounding

The above argument is intuitive, but also informal. To make these statements rigorous, we interpret the SR method as a Markov chain. On each iteration, SR starts at some state (iterate) x, and moves to a new state y with some transition probability $T_\alpha(x, y)$ that depends only on x and the learning rate α. For fixed α, this is clearly a Markov process with transition matrix[1] $T_\alpha(x, y)$.

The long-term behavior of this Markov process is determined by the *stationary distribution* of $T_\alpha(x, y)$. We show below that for small α, the stationary distribution of $T_\alpha(x, y)$ is nearly invariant to α, and thus decreasing α below some threshold has virtually no effect on the long term behavior of the method. This happens because, as α shrinks, the relative transition probabilities remain the same (conditioned on the fact that the parameters change), even though the absolute probabilities decrease. In this case, there is no exploitation benefit to decreasing α.

Theorem 5. *Let $p_{x,k}$ denote the probability distribution of the kth entry in $\nabla \tilde{f}(x)$,*

[1] Our analysis below does not require the state space to be finite, so $T_\alpha(x, y)$ may be a linear operator rather than a matrix. Nonetheless, we use the term "matrix" as it is standard.

the stochastic gradient estimate at x. Assume there is a constant C_1 such that for all x, k, and ν we have $\int_\nu^\infty p_{x,k}(z)\,dz \le \frac{C_1}{\nu^2}$, and some C_2 such that both $\int_0^{C_2} p_{x,k}(z)\,dz > 0$ and $\int_{-C_2}^0 p_{x,k}(z)\,dz > 0$. Define the matrix

$$\tilde{U}(x,y) = \begin{cases} \int_0^\infty p_{x,k}(z)\frac{z}{\Delta}\,dz, \text{ if } x \text{ and } y \text{ differ only at coordinate } k, \text{ and } y_k = x_k + \Delta \\ \int_{-\infty}^0 p_{x,k}(z)\frac{z}{\Delta}\,dz, \text{ if } x \text{ and } y \text{ differ only at coordinate } k, \text{ and } y_k = x_k - \Delta \\ 0, \text{ otherwise,} \end{cases}$$

and the associated markov chain transition matrix

$$\tilde{T}_{\alpha_0} = I - \alpha_0 \cdot \mathrm{diag}(\mathbf{1}^T \tilde{U}) + \alpha_0 \tilde{U}, \tag{4.8}$$

where α_0 is the largest constant that makes $\tilde{T}_{\alpha_0}$ non-negative. Suppose $\tilde{T}_\alpha$ has a stationary distribution, denoted $\tilde{\pi}$. Then, for sufficiently small α, T_α has a stationary distribution π_α, and

$$\lim_{\alpha \to 0} \pi_\alpha = \tilde{\pi}.$$

Furthermore, this limiting distribution satisfies $\tilde{\pi}(x) > 0$ for any state x, and is thus not concentrated on local minimizers of f.

While the long term stationary behavior of SR is relatively insensitive to α, the convergence speed of the algorithm is not. To measure this, we consider the *mixing time* of the Markov chain. Let π_α denote the stationary distribution of a Markov chain. We say that the ϵ-mixing time of the chain is M_ϵ if M_ϵ is the smallest integer

such that [Levin et al., 2009]

$$|\mathbb{P}(x^{M_\epsilon} \in A|x^0) - \pi(A)| \leq \epsilon, \quad \text{for all } x^0 \text{ and all subsets of states } A \subseteq X. \tag{4.9}$$

We show below that the mixing time of the Markov chain gets large for small α, which means exploration slows down, even though no exploitation gain is being realized.

Theorem 6. *Let $p_{x,k}$ satisfy the assumptions of Theorem 5. Choose some ϵ sufficiently small that there exists a* proper *subset of states $A \subset X$ with stationary probability $\pi_\alpha(A)$ greater than ϵ. Let $M_\epsilon(\alpha)$ denote the ϵ-mixing time of the chain with learning rate α. Then,*

$$\lim_{\alpha \to 0} M_\epsilon(\alpha) = \infty.$$

4.5 Experiments

To explore the implications of the above theory, we train VGGNet [Simonyan and Zisserman, 2015] and ResNets [He et al., 2016] with binarized weights on image classification problems. On CIFAR-10, we train ResNet-56, wide ResNet-56 (WRN-56-2, with 2X more filters than ResNet-56), VGG-9, and the high capacity VGG-BC network [Courbariaux et al., 2015]. We also train ResNet-56 on CIFAR-100, and ResNet-18 on ImageNet [Russakovsky et al., 2015].

VGG-9 on CIFAR-10 consists of seven convolutional layers and two fully connected layers, with Batch Normalization [Ioffe and Szegedy, 2015] after each convo-

lutional layer and the first linear layer VGG-BC is a high-capacity network originally used in [Courbariaux et al., 2015]. We use the same architecture as Courbariaux et al. [2015] except using softmax and cross-entropy loss instead of SVM and squared hinge loss, respectively. ResNets-56 has 55 convolutional layers and one linear layer, and contains three stages of residual blocks where each stage has the same number of residual blocks (The details of VGG-9 and VGG-BC are presented in Table B.1 and Table B.2 in Appendix B.8). Similarly, ResNets-18 for ImageNet has the same description as in [He et al., 2016]. We also create a wide ResNet-56 (WRN-56-2) that doubles the number of filters in each residual block as [Zagoruyko and Komodakis, 2016].

We use Adam [Kingma and Ba, 2015] as our baseline optimizer as we found it to frequently give better results than well-tuned SGD (an observation that is consistent with previous papers on quantized models [Courbariaux et al., 2015, 2016, Gupta et al., 2015, Marchesi et al., 1993, Rastegari et al., 2016], and we train with the three quantized algorithms mentioned in Section 4.2, i.e., R-ADAM, SR-ADAM and BC-ADAM. All methods start with the same binary weight initialization of random $\pm$1s. The image pre-processing and data augmentation procedures are the same as He et al. [2016]. Following [Rastegari et al., 2016], we only quantize the weights in the convolutional layers, but not linear layers, during training (See Appendix B.8 for a discussion of this issue, and a detailed description of experiments).

We set the initial learning rate to 0.01 and decrease the learning rate by a factor of 10 at epochs 82 and 122 for CIFAR-10 and CIFAR-100. For ImageNet experiments, we train the model for 90 epochs and decrease the learning rate at

Table 4.1: Test error after training with binarized initial weights. The deafult batch size is 128 and learning rate is 0.01. Big SR-ADAM uses batch size 512 for WSN-56-2 and 1024 for other models.

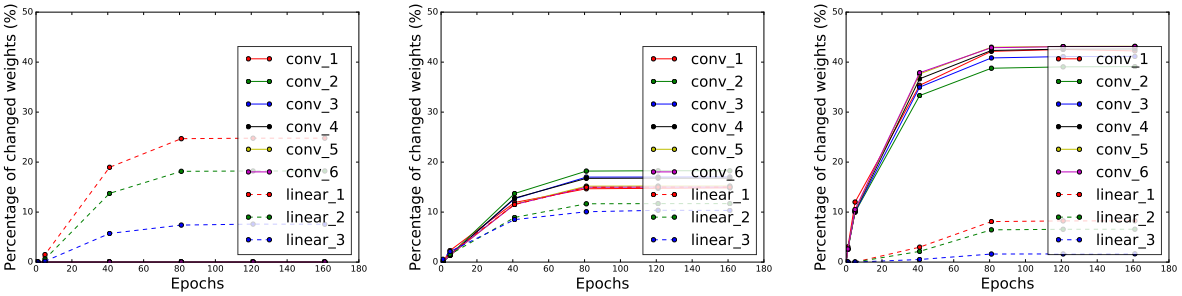
	CIFAR-10				CIFAR-100	ImageNet
	VGG-9	VGG-BC	ResNet-56	WSN-56-2	ResNet-56	ResNet-18
ADAM	7.97	7.12	8.10	6.62	33.98	36.04
R-ADAM	23.99	21.88	33.56	27.90	68.39	91.07
BC-ADAM	10.36	8.21	8.83	7.17	35.34	45.29
SR-ADAM	23.33	20.56	26.49	21.58	58.06	88.86
Big SR-ADAM	16.95	16.77	19.84	16.04	50.79	77.68

epochs 30 and 60. See Appendix B.8 for additional experiments.

The overall results are summarized in Table 4.1. The binary model trained by BC-ADAM has comparable performance to the full-precision model trained by ADAM. Note that we explores several tricks that makes BC-ADAM converges faster and summarize in the Appendix B.10 and B.11. SR-ADAM outperforms R-ADAM, which verifies the effectiveness of Stochastic Rounding. There is a 6% to 10% performance gap between SR-ADAM and BC-ADAM across all models and datasets. This is consistent with our theoretical results in Section 4.3 and 4.4, and keeping track of the real-valued weights to quantize the binary weights in BC-ADAM seems to really help empirically.

4.5.1 Exploration vs Exploitation Tradeoffs

Section 4.4 discusses the exploration/exploitation tradeoff of continuous-valued SGD methods and predicts that fully discrete methods like SR are unable to enter an exploitative phase. To test this effect, we plot the percentage of changed weights (signs different from the initialization) as a function of the training epochs (Figures



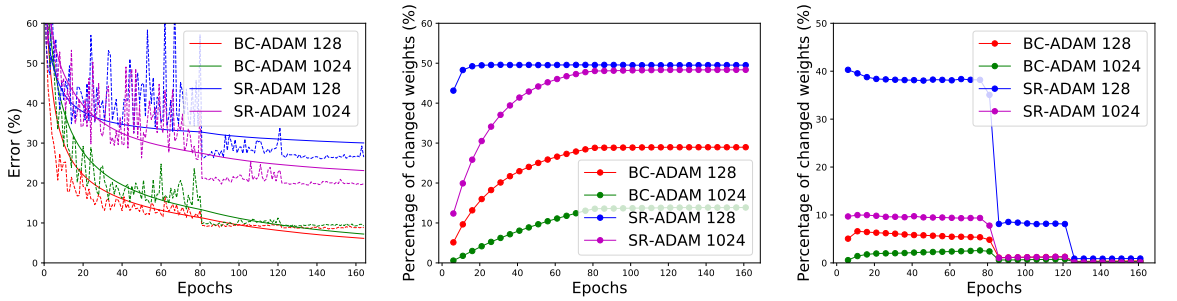
(a) R-ADAM (b) BC-ADAM (c) SR-ADAM

Figure 4.7: Percentage of weight changes during training of VGG-BC on CIFAR-10. The x-axis is the number of epochs and the y-axis is the percentage of weights changes comparing to the initial weights.

4.7 and 4.8). SR-ADAM explores aggressively; it changes more weights in the Conv layers than both R-ADAM and BC-ADAM, and keeps changing weights until nearly 40% of the weights differ from their starting values (in a binary model, randomly re-assigning weights would result in 50% change). The BC method never changes more than 20% of the weights (Fig 4.7(b)), indicating that it stays near a local minimizer and explores less. Interestingly, we see that the weights of the conv layers were not changed at all by R-ADAM; when the tails of the stochastic gradient distribution are light, this method is ineffective.

4.5.2 Effect of Batch Size

We saw in Section 4.4 that SR is unable to exploit local minima because, for small learning rates, shrinking the learning rate does not produce additional bias towards moving downhill. This was illustrated in Figure 4.2. If this is truly the cause of the problem, then our theory predicts that we can improve the performance of SR for low-precision training by increasing the batch size. This shrinks the variance of



(a) BC-ADAM vs SR-ADAM (b) Weight changes since beginning (c) Weight changes every 5 epochs

Figure 4.8: Effect of batch size on SR-ADAM when tested with ResNet-56 on CIFAR-10. (a) Test error vs epoch. Test error is reported with dashed lines, train error with solid lines. (b) Percentage of weight changes since initialization. (c) Percentage of weight changes per every 5 epochs.

the gradient distribution in Figure 4.2 without changing the mean and concentrates more of the gradient distribution towards downhill directions, making the algorithm more greedy. Our theory predicts that this should help mitigate the problem of SR being unable to exploit local minima by decreasing the learning rate.

To verify this, we tried different batch sizes for SR including 128, 256, 512 and 1024, and found that the larger the batch size, the better the performance of SR. Figure 4.8(a) illustrates the effect of a batch size of 1024 for BC and SR methods. We find that the BC method, like classical SGD, performs best with a small batch size. However, a large batch size is essential for the SR method to perform well. Figure 4.8(b) shows the percentage of weights changed by SR and BC during training. We see that the large batch methods change the weights less aggressively than the small batch methods, indicating less exploration. Figure 4.8(c) shows the percentage of weights changed during each 5 epochs of training. It is clear that small-batch SR changes weights much more frequently than using a big batch.

This property of big batch training clearly benefits SR; we see in Figure 4.8(a) and Table 4.1 that big batch training improved performance over SR-ADAM consistently.

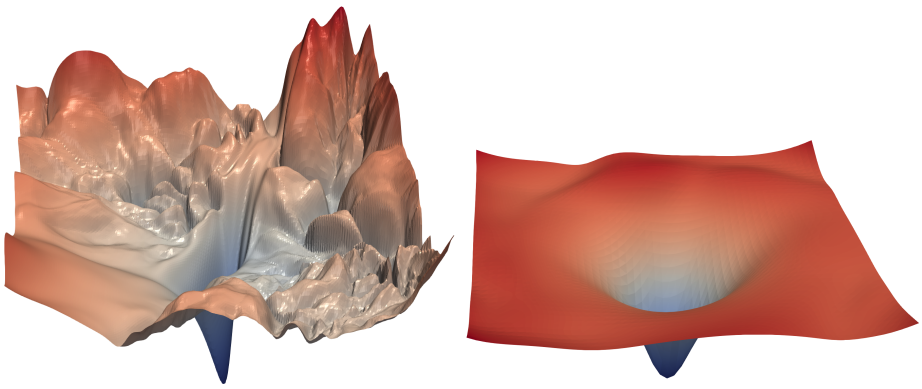
In addition to providing a means of improving fixed-point training, this suggests that recently proposed methods using big batches [Goyal et al., 2017] may be able to exploit lower levels of precision to further accelerate training.

4.6 Summary

The training of quantized neural networks is essential for deploying machine learning models on embedded and ubiquitous devices. In this chapter, we provided a theoretical analysis to better understand the BinaryConnect (BC) and Stochastic Rounding (SR) methods for training binary neural networks. We proved convergence results for BC and SR methods that provide convergence rates and predict an accuracy bound that depends on the coarseness of discretization. For general non-convex problems, we proved that SR differs from conventional stochastic methods in that it is unable to exploit greedy local search. Experiments confirm these findings, and show that the mathematical properties of SR are indeed observable in practice.

Chapter 5: Visualizing the Loss Landscape of Neural Nets

Training deep neural networks requires minimizing a high-dimensional non-convex loss function – a task that is hard in theory, but sometimes easy in practice. Despite the NP-hardness of training general neural loss functions [Blum and Rivest, 1989], simple gradient methods often find global minimizers (parameter configurations with zero or near-zero training loss), even when data and labels are randomized before training [Zhang et al., 2017]. However, this good behavior is not universal; the trainability of neural nets is highly dependent on network architecture design choices, the choice of optimizer, variable initialization, and a variety of other considerations. Unfortunately, the effect of each of these choices on the structure of the



(a) without skip connections (b) with skip connections

Figure 5.1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures. Note that the vertical axis is logarithmic to show dynamic range.

underlying loss surface is unclear. Because of the prohibitive cost of loss function evaluations (which requires looping over all the data points in the training set), studies in this field have remained predominantly theoretical.

Visualizations have the potential to help us answer several important questions about why neural networks work. In particular, why are we able to minimize highly non-convex neural loss functions? And why do the resulting minima generalize? To clarify these questions, a number of authors have used theoretical tools to peer into the structure of loss functions without resorting to expensive computations. Unfortunately, some of these theoretical studies have arrived at conflicting conclusions, or require assumptions that are difficult to validate for practical neural network implementations. It is therefore desirable to visualize network loss function topography. However, this landscape is difficult to visualize because it lives in a high dimensional space. Furthermore, exact evaluation of a neural loss function requires a full pass through the training data, which makes visualization a computationally expensive task.

Our goal is to use high-resolution visualizations to provide an empirical characterization of neural loss functions, and to explore how different network architecture choices affect the loss landscape. We use high-resolution visualizations to provide an empirical characterization of neural loss functions, and explore how different network architecture choices affect the loss landscape. Furthermore, we explore how the non-convex structure of neural loss functions relates to their trainability, and how the geometry of neural minimizers (i.e., their sharpness/flatness, and their surrounding landscape), affects their generalization properties.

To do this in a meaningful way, we propose a simple "filter normalization" scheme that enables us to do side-by-side comparisons of different minima found during neural networks training. We then use visualizations to explore sharpness/flatness of minimizers found by different methods, as well as the effect of network architecture choices (use of skip connections, number of filters, network depth) on the loss landscape. Our goal is to understand how differences in loss function geometry affect the generalization of neural nets.

In this chapter, we study methods for producing meaningful loss function visualizations. Then, using these visualization methods, we explore how loss landscape geometry affects generalization error and trainability. More specifically, we address the following issues:

- We reveal faults in a number of visualization methods for loss functions, and show that simple visualization strategies fail to accurately capture the local geometry (sharpness or flatness) of loss function minimizers.

- We present a simple visualization method based on "filter normalization." The sharpness of minimizers correlates well with generalization error when this normalization is used, even when making comparisons across disparate network architectures and training methods. This enables side-by-side comparisons of different minimizers.

- We observe that, when networks become sufficiently deep, neural loss landscapes quickly transition from being nearly convex to being highly chaotic. This transition from convex to chaotic behavior coincides with a dramatic

drop in generalization error, and ultimately to a lack of trainability.

- We show that skip connections promote flat minimizers and prevent the transition to chaotic behavior, which helps explain why skip connections are necessary for training extremely deep networks.

- We study the visualization of SGD optimization trajectories. We explain the difficulties that arise when visualizing these trajectories, and show that optimization trajectories lie in an extremely low dimensional space. This low dimensionality can be explained by the presence of large nearly convex regions in the loss landscape, such as those observed in our 2-dimensional visualizations.

5.1 Related Work

Because of the difficulty of visualizing loss functions, most studies of loss landscapes are largely theoretical in nature. A number of authors have studied our ability to minimize neural loss functions. Using random matrix theory and spin glass theory, several authors have shown that local minima are of low objective value [Choromanska et al., 2015, Dauphin et al., 2014]. It can also be shown that local minima are global minima, provided one assumes linear neurons [Hardt and Ma, 2017], very wide layers [Nguyen and Hein, 2017], or full rank weight matrices [Yun et al., 2017]. These assumptions have been relaxed by [Kawaguchi, 2016] and [Lu and Kawaguchi, 2017], although some assumptions (e.g., of the loss functions) are still required. Freeman and Bruna [2017], Soudry and Hoffer [2017], Xie et al. [2017] also analyzed shallow networks with one or two hidden layers under mild conditions.

Another approach is to show that we can expect good minimizers, not simply because of the endogenous properties of neural networks, but because of the optimizers. For restricted network classes such as those with one hidden layer, with some extra assumptions on the sample distribution, globally optimal or near-optimal solutions can be found by common optimization methods [Li and Yuan, 2017, Soltanolkotabi et al., 2017, Tian, 2017]. For networks with specific structures, Safran and Shamir [2016] and Haeffele and Vidal [2017] show there likely exists a monotonically decreasing path from an initialization to a global minimum. Swirszcz et al. [2016] show counterexamples that achieve "bad" local minima for toy problems.

Sharpness/Flatness of local minima Hochreiter and Schmidhuber [1997] defined "flatness" as the size of the connected region around the minimum where the training loss remains low. Keskar et al. [2017] suggested that flatness can be characterized by the eigenvalues of the Hessian, and proposed ϵ-sharpness as an approximation, which looks at the maximum loss in a bounded neighborhood of a minimum. However, Dinh et al. [2017] show that these quantitative measure of sharpness are problematic because they are not invariant to symmetries in the network, and are thus not sufficient to determine the generalization ability. Baldassi et al. [2016], Chaudhari et al. [2017] introduced local entropy as a measure of width of the valley (flatness), which is invariant to the simple transformation.

Theoretical results make some restrictive assumptions such as the independence of the input samples, or restrictions on non-linearities and loss functions. For this reason, visualizations play a key role in verifying the validity of theoretical as-

sumptions, and understanding loss function behavior in real-world systems. In the next section, we briefly review methods that have been used for this purpose.

5.2 The Basics of Loss Function Visualization

Neural nets contain many parameters, and so their loss functions live in a very high-dimensional space. Unfortunately, visualizations are only possible using low-dimensional 1D (line) or 2D (surface) plots. Several methods exist for closing this dimensionality gap.

5.2.1 1-Dimensional Linear Interpolation

One simple and lightweight way to plot loss functions is to choose two sets of parameters θ and θ', and plot the values of the loss function along the line connecting these two points. We can parameterize this line by choosing a scalar parameter α, and defining the weighted average $\theta(\alpha) = (1-\alpha)\theta+\alpha\theta'$. Finally, we plot the function $f(\alpha) = \mathcal{L}(\theta(\alpha))$. This strategy was taken by Goodfellow et al. [2015], who studied the loss surface along the line between a random initial guess, and a nearby minimizer obtained by stochastic gradient descent. This method has been widely used to study the "sharpness" and "flatness" of different minima, and the dependence of sharpness on batch-size [Dinh et al., 2017, Keskar et al., 2017]. Smith and Topin [2017] use the same technique to show different minima and the "peaks" between them, while Im et al. [2016] plot the line between minima obtained via different optimizers. The 1D linear interpolation method suffers from several weaknesses. First,

it is difficult to visualize non-convexities using 1D plots. Indeed, Goodfellow et al. [2015] found that loss functions appear to lack local minima along the minimization trajectory. We will see later, using 2D methods, that some loss functions have extreme non-convexities, and that these non-convexities correlate with the difference in generalization between different network architectures. Second, this method does not consider batch normalization or invariance symmetries in the network. For this reason, the visual sharpness comparisons produced by 1D interpolation plots may be misleading; this issue will be explored in depth in Section 5.4.

5.2.2 2D Contour Plots

To use this approach, one chooses a center point θ^* in the graph, and chooses two direction vectors, δ and η. One then plots a function of the form $f(\alpha) = \mathcal{L}(\theta^* + \alpha\delta)$ in the 1D (line) case, or

$$f(\alpha, \beta) = \mathcal{L}(\theta^* + \alpha\delta + \beta\eta) \tag{5.1}$$

in the 2D (surface) case[1]. This approach was used in Goodfellow et al. [2015] to explore the trajectories of different minimization methods. It was also used in Im et al. [2016] to show that different optimization algorithms find different local minima within the 2D projected space. Because of the computational burden of 2D plotting, these methods generally result in low-resolution plots of small regions that have not captured the complex non-convexity of loss surfaces. Below, we use high-resolution

[1]When making 2D plots, batch normalization parameters are held constant, i.e., random directions are not applied to batch normalization parameters.

visualizations over large slices of weight space to visualize how network design affects non-convex structure.

5.3 Proposed Visualization: Filter-Wise Normalization

This study relies heavily on plots of the form (5.1) produced using random direction vectors, δ and η, each sampled from a random Gaussian distribution with appropriate scaling (described below).

While the "random directions" approach to plotting is simple, it fails to capture the intrinsic geometry of loss surfaces, and cannot be used to compare the geometry of two different minimizers or two different networks. This is because of the *scale invariance* in network weights. When ReLU non-linearities are used, the network remains unchanged if we (for example) multiply the weights in one layer of a network by 10, and divide the next layer by 10. This invariance is even more prominent when batch normalization is used (Figure 5.2). In this case, the size (i.e., norm) of a filter is irrelevant because the output of each layer is re-scaled during batch normalization. For this reason, a network's behavior remains unchanged if we re-scale the weights.

Scale invariance prevents us from making meaningful comparisons between plots, unless special precautions are taken. As shown in Figure 5.3, a neural network with large weights may appear to have a smooth and slowly varying loss function; perturbing the weights by one unit will have very little effect on network performance if the weights live on a scale much larger than one. However, if the

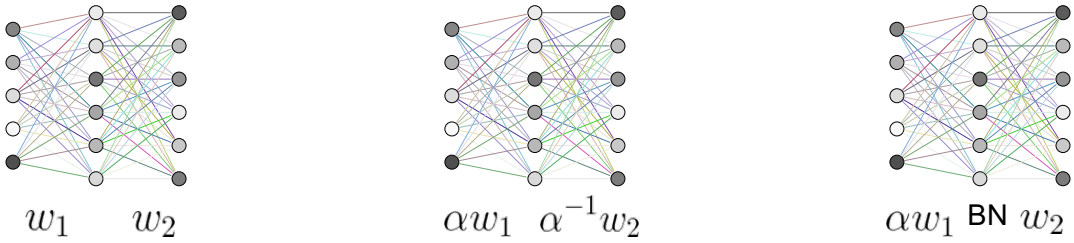


Figure 5.2: The three one-hidden layer rectified feedforward networks (with ReLU as activation function) can be equivalent when the only the scales of weights are different. w_1 and w_2 are the weights for the first and second layer, respectively. The α scale transformation does not affect the generalization as the behavior of the function is identical. When BN layer is added after the first layer, it is equivalent to scale w_1 or w_2 with α^{-1}.

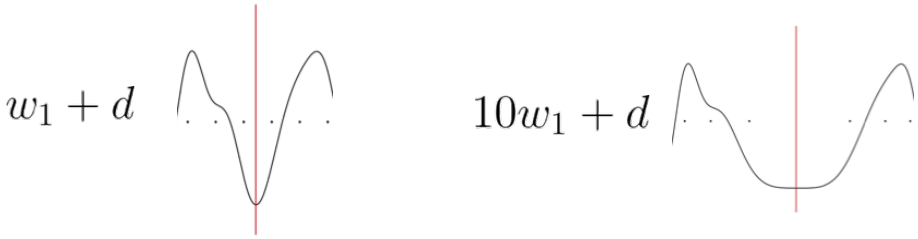


Figure 5.3: A illustration of adding the same perturbation may results in different change of loss landscape depending on the scale of weights. The loss landscape will change quickly in the direction of a small filter, and slowly in the direction of large filter. w_1 are the weights for one filter, d is a random Gaussian vector with the same dimension as w_1. x-axis is along the direction of d and $y - axis$ is the loss value.

weights are much smaller than one, then that same one unit perturbation may have a catastrophic effect, making the loss function appear quite sensitive to weight perturbations. Keep in mind that neural nets are scale invariant; if the small-parameter and large-parameter networks in this example are equivalent (because one is simply a re-scaling of the other), then any apparent differences in the loss function are merely an artifact of scale invariance. This scale invariance was exploited by Dinh et al. [2017] to build pairs of equivalent networks that have different apparent sharpness.

To remove this scaling effect, we plot loss functions using filter-wise normalized

directions. To obtain such directions for a network with parameters θ, we begin by producing a random Gaussian direction vector d with dimensions compatible with θ. Then we normalize each filter in d to have the same norm of the corresponding filter in θ. In other words, we make the replacement

$$d_{i,j} \leftarrow \frac{d_{i,j}}{\|d_{i,j}\|}\|\theta_{i,j}\|, \tag{5.2}$$

where $d_{i,j}$ represents the jth filter (not the jth weight) of the ith layer of d, and $\|\cdot\|$ denotes the Frobenius norm. Note that the filter-wise normalization is different from that of [Im et al., 2016], which normalize the direction without considering the norm of individual filters. Note that filter normalization is not limited to convolutional (Conv) layers but also applies to fully connected (FC) layers. The FC layer is equivalent to a Conv layer with a 1×1 output feature map and the filter corresponds to the weights that generate one neuron.

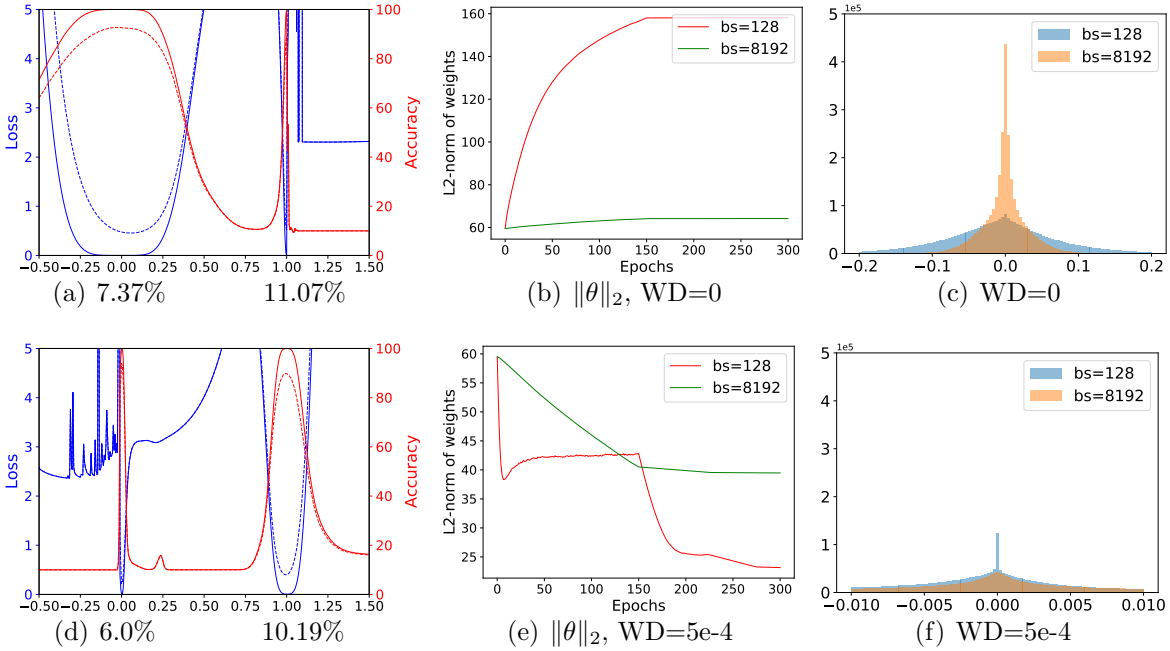
Do filter normalized plots capture the natural distance scale of loss surfaces? We answer this question in the affirmative in Section 5.4 by showing that the sharpness of filter-normalized plots correlates well with generalization error, while plots without filter normalization can be very misleading. In Appendix C, we also compare filter-wise normalization to layer-wise normalization (and no normalization), and show that filter normalization produces superior correlation between sharpness and generalization error.

5.4 The Sharp vs Flat Dilemma

Section 5.3 introduces the concept of filter normalization, and provides an intuitive justification for its use. In this section, we address the issue of whether sharp minimizers generalize better than flat minimizers. In doing so, we will see that the sharpness of minimizers correlates well with generalization error when filter normalization is used. This enables side-by-side comparisons between plots. In contrast, the sharpness of non-normalized plots may appear distorted and unpredictable.

It is widely thought that small-batch SGD produces "flat" minimizers that generalize better, while large batch sizes produce "sharp" minima with poor generalization [Chaudhari et al., 2017, Hochreiter and Schmidhuber, 1997, Keskar et al., 2017]. This claim is disputed though, with Dinh et al. [2017], Kawaguchi et al. [2017] arguing that generalization is not directly related to the curvature of loss surfaces, and some authors proposing specialized training methods that achieve good performance with large batch sizes [De et al., 2017, Goyal et al., 2017, Hoffer et al., 2017]. Here, we explore the difference between sharp and flat minimizers. We begin by discussing difficulties that arise when performing such a visualization, and how proper normalization can prevent such plots from producing distorted results.

We train a CIFAR-10 classifier using a 9-layer VGG network [Simonyan and Zisserman, 2015] with Batch Normalization [Ioffe and Szegedy, 2015] for a fixed number of epochs. We use two batch sizes: a large batch size of 8192 (16.4% of the training data of CIFAR-10), and a small batch size of 128. Let θ^s and θ^l indicate the solutions obtained by running SGD using small and large batch sizes,



(a) 7.37% 11.07% (b) $\|\theta\|_2$, WD=0 (c) WD=0

(d) 6.0% 10.19% (e) $\|\theta\|_2$, WD=5e-4 (f) WD=5e-4

Figure 5.4: (a) and (d) are the 1D linear interpolation of VGG9 solutions obtained by small-batch and large-batch training methods. The blue lines are loss values and the red lines are accuracies. The solid lines are training curves and the dashed lines are for testing. Small batch is at abscissa 0, and large batch is at abscissa 1. The corresponding test errors are shown below. (b) and (e) shows the change of weights norm $\|\theta\|_2$ during training. When weight decay is disabled, the weight norm grows steadily during training without constraints (c) and (f) are the weight histograms, which verify that small-batch methods produce more large weights with zero weight decay and more small weights with non-zero weight decay.

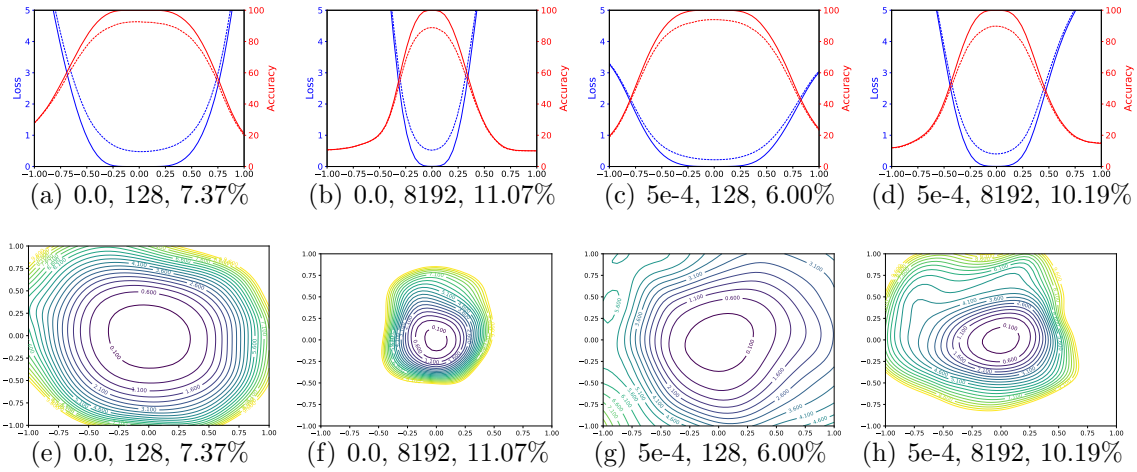
respectively[2]. Using the linear interpolation approach [Goodfellow et al., 2015], we plot the loss values on both training and testing data sets of CIFAR-10, along a direction containing the two solutions, i.e., $f(\alpha) = \mathcal{L}(\theta^s + \alpha(\theta^l - \theta^s))$.

Figure 5.4(a) shows linear interpolation plots with θ^s at x-axis location 0, and θ^l at location 1. The 1D interpolation method for plotting is described in detail in Section 5.2. As observed by Keskar et al. [2017], we can clearly see that the

[2]In this section, we consider the "running mean" and "running variance" as trainable parameters and include them in θ. Note that the original study by [Goodfellow et al., 2015] does not consider batch normalization. These parameters are not included in θ in future sections, as they are only needed when interpolating between two minimizers.

small-batch solution is quite wide, while the large-batch solution is sharp. However, this sharpness balance can be flipped simply by turning on weight decay [Krogh and Hertz, 1992]. Figure 5.4(d) show results of the same experiment, except this time with a non-zero weight decay parameter. This time, the large batch minimizer is considerably flatter than the sharp small batch minimizer. However, we see that small batches generalize better in all experiments; there is no apparent correlation between sharpness and generalization. We will see that these sharpness comparisons are extremely misleading, and fail to capture the endogenous properties of the minima.

The apparent differences in sharpness can be explained by examining the weights of each minimizer. Histograms of the network weights are shown for each experiment in Figure 5.4(c) and (f). We see that, when a large batch is used with zero weight decay, the resulting weights tend to be smaller than in the small batch case. We reverse this effect by adding weight decay; in this case the large batch minimizer has much larger weights than the small batch minimizer. This difference in scale occurs for a simple reason: A smaller batch size results in more weight updates per epoch than a large batch size, and so the shrinking effect of weight decay (which imposes a penalty on the norm of the weights) is more pronounced. The evolution of the weight norms during training is depicted in Figure 5.4(b) and 5.4(e). Figure 5.4 is not visualizing the endogenous sharpness of minimizers, but rather just the (irrelevant) weight scaling. The scaling of weights in these networks is irrelevant because batch normalization re-scales the outputs to have unit variance. However, small weights still appear more sensitive to perturbations, and produce



(a) 0.0, 128, 7.37% (b) 0.0, 8192, 11.07% (c) 5e-4, 128, 6.00% (d) 5e-4, 8192, 10.19%

(e) 0.0, 128, 7.37% (f) 0.0, 8192, 11.07% (g) 5e-4, 128, 6.00% (h) 5e-4, 8192, 10.19%

Figure 5.5: The 1D and 2D visualization of solutions obtained using SGD with different weight decay and batch size. The title of each subfigure contains the weight decay, batch size, and test error.

sharper looking minimizers.

Filter Normalized Plots We repeat the experiment in Figure 5.4, but this time we plot the loss function near each minimizer separately using random filter-normalized directions. This removes the apparent differences in geometry caused by the scaling depicted in Figure 5.4(c) and (f). The results, presented in Figure 5.5, still show differences in sharpness between small batch and large batch minima; however, these differences are much more subtle than it would appear in the un-normalized plots. For comparison, sample un-normalized plots and layer-normalized plots are shown in Section C.2 of the Appendix. We also visualize these results using two random directions and contour plots. The weights obtained with small batch size and non-zero weight decay have wider contours than the sharper large batch minimizers. Results for ResNet-56 appear in Figure C.6 of Appendix C.

Generalization and Flatness Using the filter-normalized plots in Figure 5.5, we can make side-by-side comparisons between minimizers, and we see that now sharpness correlates well with generalization error. Large batches produced visually sharper minima (although not dramatically so) with higher test error. Interestingly, the Adam optimizer attained larger test error than SGD, and, as predicted, the corresponding minima are visually sharper. Results of a similar experiment using ResNet-56 are presented in Appendix C (Figure C.6).

5.5 What Makes Neural Networks Trainable? The (Non)Convexity Structure of Loss Surfaces

Our ability to find global minimizers to neural loss functions is not universal; it seems that some neural architectures are easier to minimize than others. For example, using skip connections, ResNets were able to train extremely deep architectures, while comparable architectures without skip connections are not trainable. Furthermore, our ability to train seems to depend strongly on the initial parameters from which training starts. Using visualization methods, we perform an empirical study of neural architectures to explore why the non-convexity of loss functions seems to be problematic in some situations, but not in others. We aim to provide insight into the following questions: Do loss functions have significant non-convexity at all? If prominent non-convexities exist, why are they not problematic in all situations? Why are some architectures easy to train, and why are results so sensitive to the initialization? We will see that different architectures have extreme differences

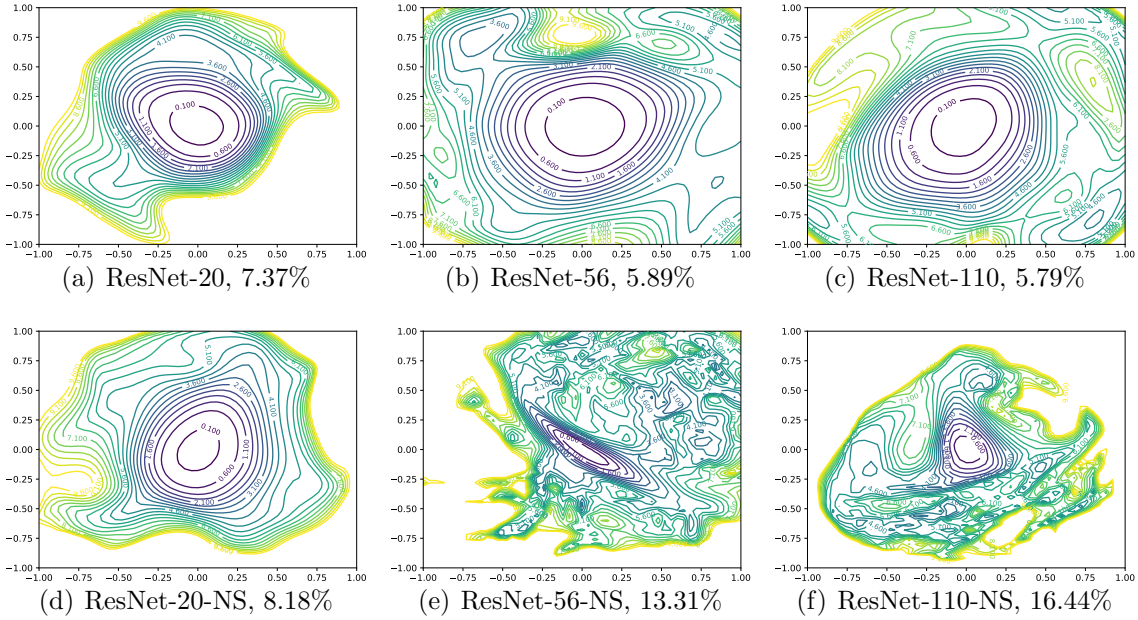
in non-convexity structure that answer these questions, and that these differences correlate with generalization error.

5.5.1 Experimental Setup

To understand the effects of network architecture on non-convexity, we trained a number of networks, and plotted the landscape around the obtained minimizers using the filter-normalized random direction method described in Section 5.3. We consider three classes of neural networks:

- ResNets that are optimized for performance on CIFAR-10. We consider ResNet-20/56/110, where each name is labeled with the number of layers it has.
- "VGG-like" networks that do not contain shortcut/skip connections. We produced these networks simply by removing the skip connections from ResNets. We call these networks ResNet-20/56/110-noshort.
- "Wide" ResNets that have more filters per layer than the CIFAR-10 optimized networks.

All models are trained on the CIFAR-10 dataset using SGD with Nesterov momentum, batch-size 128, and 0.0005 weight decay for 300 epochs. The learning rate was initialized at 0.1, and decreased by a factor of 10 at epochs 150, 225 and 275. Deeper VGG-like networks (e.g., ResNet-110-noshort, as described below) required a smaller initial learning rate of 0.01. High resolution 2D plots of the minimizers for different neural networks are shown in Figure 5.6 and Figure 5.8. Results are shown



(a) ResNet-20, 7.37% (b) ResNet-56, 5.89% (c) ResNet-110, 5.79%

(d) ResNet-20-NS, 8.18% (e) ResNet-56-NS, 13.31% (f) ResNet-110-NS, 16.44%

Figure 5.6: 2D visualization of the loss surface of ResNet and ResNet-noshort with different depth.

as contour plots rather than surface plots because this makes it extremely easy to see non-convex structures and evaluate sharpness. For surface plots of ResNet-56, see Figure 5.1. Note that the center of each plot corresponds to the minimizer, and the two axes parameterize two random directions with filter-wise normalization as in (5.1). We make several observations below about how architecture affects the loss landscape. We also provide loss and error values for these networks in Table C.2, and convergence curves in Figure C.10 of Appendix C.

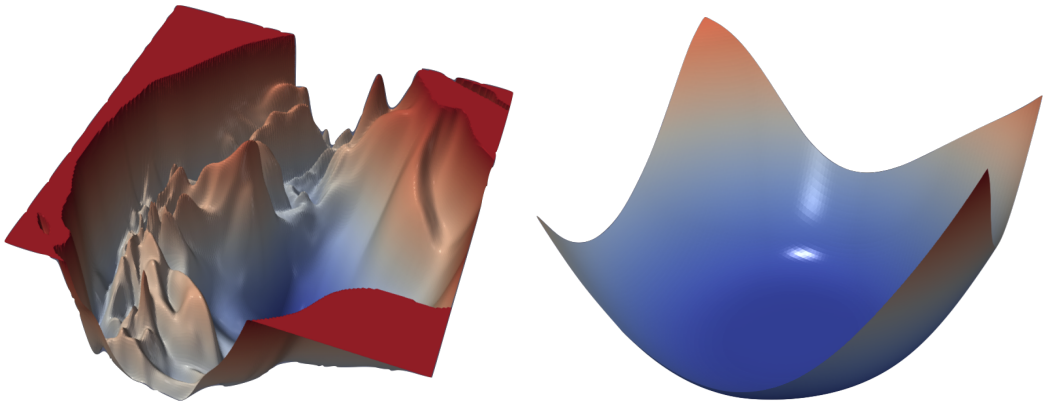
5.5.2 The Effect of Network Depth

From Figure 5.6, we see that network depth has a dramatic effect on the loss surfaces of neural networks when skip connections are not used. The network ResNet-20-noshort has a fairly benign landscape dominated by a region with convex

contours in the center, and no dramatic non-convexity. This isn't too surprising: the original VGG networks for ImageNet had 19 layers and could be trained effectively [Simonyan and Zisserman, 2015]. However, as network depth increases, the loss surface of the VGG-like nets spontaneously transitions from (nearly) convex to chaotic. ResNet-56-noshort has dramatic non-convexities and large regions where the gradient directions (which are normal to the contours depicted in the plots) do not point towards the minimizer at the center. Also, the loss function becomes extremely large as we move in some directions. ResNet-110-noshort displays even more dramatic non-convexities, and becomes extremely steep as we move in all directions shown in the plot. Furthermore, note that the minimizers at the center of the deep VGG-like nets seem to be fairly sharp. In the case of ResNet-56-noshort, the minimizer is also fairly ill-conditioned, as the contours near the minimizer have significant eccentricity.

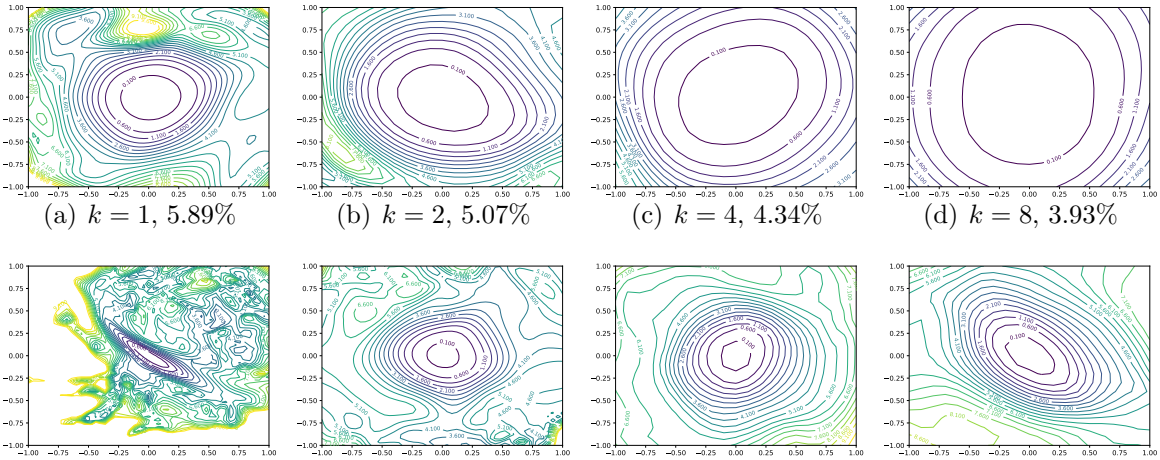
5.5.3 Shortcut Connections to the Rescue

Shortcut connections have a dramatic effect of the geometry of the loss functions. In Figure 5.6, we see that residual connections prevent the transition to chaotic behavior as depth increases. In fact, the width and shape of the 0.1-level contour is almost identical for the 20- and 110-layer networks. Interestingly, the effect of skip connections seems to be most important for deep networks. For the more shallow networks (ResNet-20 and ResNet-20-noshort), the effect of skip connections is fairly unnoticeable. However residual connections prevent the explosion



(a) ResNet-110, no skip connections (b) DenseNet, 121 layers

Figure 5.7: The loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.



(a) $k = 1$, 5.89% (b) $k = 2$, 5.07% (c) $k = 4$, 4.34% (d) $k = 8$, 3.93%

(e) $k = 1$, 13.31% (f) $k = 2$, 10.26% (g) $k = 4$, 9.69% (h) $k = 8$, 8.70%

Figure 5.8: Wide-ResNet-56 on CIFAR-10 both with shortcut connections (top) and without (bottom). The label $k = 2$ means twice as many filters per layer. Test error is reported below each figure.

of non-convexity that occurs when networks get deep. This effect seems to apply to other kinds of skip connections as well; Figure 5.7 show the loss landscape of DenseNet [Huang et al., 2017], which shows no noticeable non-convexity.

5.5.4 Wide Models vs Thin Models

To see the effect of the number of convolutional filters per layer, we compare the narrow CIFAR-optimized ResNets (ResNet-56) with Wide-ResNets [Zagoruyko and Komodakis, 2016], while we multiply the number of filters per layer by $k = 2, 4$, and 8. From Figure 5.8, we see that the wider models have loss landscapes with no noticeable chaotic behavior. Increased network width resulted in flat minima and wide regions of apparent convexity. We see that increased width prevents chaotic behavior, and skip connections dramatically widen minimizers. Finally, note that sharpness correlates extremely well with test error. It helps to explain the observations we made in Chapter 3 about why pruning a wider network with retraining can get better performance than training a thin model from scratch.

5.5.5 Implications for Network Initialization

One interesting observation seen in Figure 5.6 is that loss landscapes for all the networks considered seem to be partitioned into a well-defined region of low loss value and convex contours, surrounded by a well-defined region of high loss value and non-convex contours. This partitioning of chaotic and convex regions may explain the importance of good initialization strategies, and also the easy training behavior of "good" architectures. When using normalized random initialization strategies such as those proposed by Glorot and Bengio [2010], typical neural networks attain an initial loss value less than 2.5. The well behaved loss landscapes in Figure 5.6 (ResNets, and shallow VGG-like nets) are dominated by large, flat, nearly convex

attractors that give rise to a loss value of 4 or greater. For such landscapes, a random initialization will likely lie in the "well- behaved" loss region, and the optimization algorithm might never "see" the pathological non-convexities that occur on the high loss chaotic plateaus.

Chaotic loss landscapes (ResNet-56/110-noshort) have shallower regions of convexity that rise to lower loss values. For sufficiently deep networks with shallow enough attractors, the initial iterate will likely lie in the chaotic region where the gradients are uninformative. In this case, the gradients "shatter" [Balduzzi et al., 2017], and training is impossible. SGD was unable to train a 156 layer network without skip connections (even with very low learning rates), which adds weight to this hypothesis.

5.5.6 Landscape Geometry Affects Generalization

Both Figures 5.6 and 5.8 show that landscape geometry has a dramatic effect on generalization. First, note that visually flatter minimizers consistently correspond to lower test error, which further strengthens our assertion that filter normalization is a natural way to visualize loss function geometry.

Second, we notice that chaotic landscapes (deep networks without skip connections) result in worse training and test error, while more convex landscapes have lower error values. In fact, the most convex landscapes, Wide-ResNets in the top row of Figure 5.8), generalize the best of all, and show no noticeable chaotic behavior.

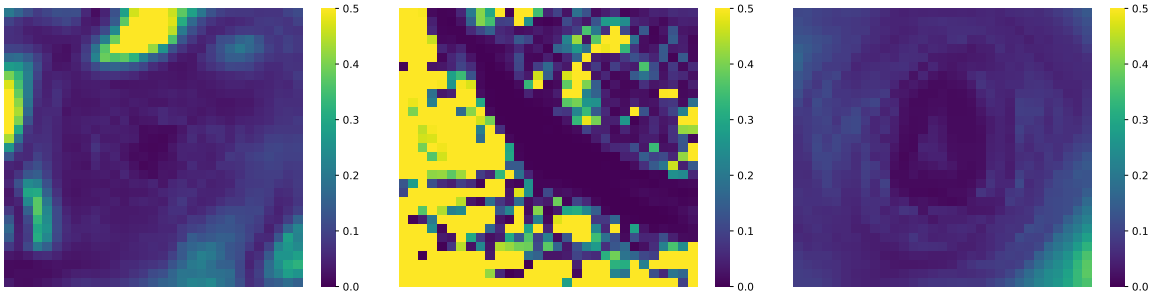
5.5.7 A note of caution: Are we really seeing convexity?

We are viewing the loss surface under a dramatic dimensionality reduction, and we need to be careful how we interpret these plots. One way to measure the level of convexity in a loss function is to compute the *principle curvatures*, which are simply eigenvalues of the Hessian. A truly convex function has non-negative curvatures (a positive semi-definite Hessian), while a non-convex function has negative curvatures. It can be shown that the principle curvatures of a dimensionality reduced plot (with random Gaussian directions) are weighted averages of the principle curvatures of the full-dimensional surface (where the weights are Chi-square random variables).

This has several consequences. First of all, if non-convexity is present in the dimensionality reduced plot, then non-convexity must be present in the full-dimensional surface as well. However, the apparent convexity in the low-dimensional surface does not mean the high-dimensional function is truly convex. Rather it means that the positive curvatures are dominant (more formally, the *mean* curvature, or average eigenvalue, is positive).

While this analysis is reassuring, one may still wonder if there is significant "hidden" non-convexity that these visualizations fail to capture. To answer this question, we calculate the *minimum* and *maximum* eigenvalues of the Hessian, λ_{min} and λ_{max} [3]. Figure 5.9 maps the ratio $|\lambda_{min}/\lambda_{max}|$ across the loss surfaces studied above (using the same minimizer and the same random directions). Blue color

[3] We compute these using an implicitly restarted lanczos method that requires only Hessian-vector products (which are calculated directly using automatic differentiation), and does not require an explicit representation of the Hessian or its factorization.



(a) Resnet-56 (b) Resnet-56-noshort (c) DenseNet-121

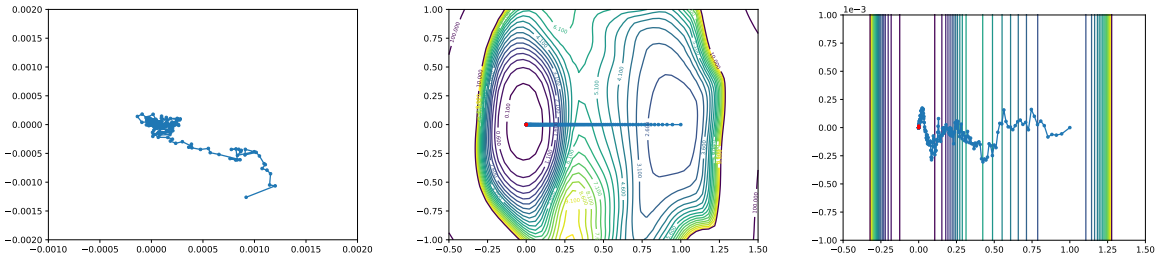
Figure 5.9: For each point in the filter-normalized surface plots, we calculate the minimum and maximum eigenvalue of the Hessian, and map the ratio of these two.

indicates a more convex region (near-zero negative eigenvalues relative to the positive eigenvalues), while yellow indicates significant levels of negative curvature. We see that the convex-looking regions in our surface plots do correspond to regions with insignificant negative eigenvalues (i.e., there are not major non-convex features that the plot missed), while chaotic regions contain large negative curvatures. For convex-looking surfaces like DenseNet, the negative eigenvalues remain extremely small (less than 1% the size of the positive curvatures) over a large region of the plot.

5.6 Visualizing Optimization Paths

We explore methods for visualizing the trajectories of different optimizers. For this application, random directions are ineffective. We will provide a theoretical explanation for why random directions fail, and explore methods for effectively plotting trajectories on top of loss function contours.

Several authors have observed that random directions fail to capture the variation in optimization trajectories, including [Gallagher and Downs, 2003, Liao and Poggio, 2017, Lipton, 2016, Lorch, 2016]. Several failed visualizations are depicted



(a) Two random directions (b) Random direction for y-axis (c) Enlarged version (b)

Figure 5.10: Ineffective visualizations of optimizer trajectories. These visualizations suffer from the orthogonality of random directions in high dimensions.

in Figure 5.10. In Figure 5.10(a), we see the iterates of SGD projected onto the plane defined by two random directions. Almost none of the motion is captured (notice the super-zoomed-in axes and the seemingly random walk). This problem was noticed by [Goodfellow et al., 2015], who then visualized trajectories using one direction that points from initialization to solution, and one random direction. This approach is shown in Figure 5.10(b). As seen in Figure 5.10(c), the random axis captures almost no variation, leading to the (misleading) appearance of a straight line path.

5.6.1 Why Random Directions Fail: Low Dimensional Optimization Trajectories

It is well known that two random vectors in a high dimensional space will be nearly orthogonal with high probability. In fact, the expected cosine similarity between Gaussian random vectors in n dimensions is roughly $\sqrt{2/(\pi n)}$ ([Goldstein and Studer, 2016], Lemma 5).

This is problematic when optimization trajectories lie in extremely low di-

mensional spaces. In this case, a randomly chosen vector will lie orthogonal to the low-rank space containing the optimization path, and a projection onto a random direction will capture almost no variation. Figure 5.10(b) suggests that optimization trajectories are low dimensional because the random direction captures orders of magnitude less variation than the vector that points along the optimization path. Below, we use PCA directions to directly validate this low dimensionality, and also to produce effective visualizations.

5.6.2 Effective Trajectory Plotting using PCA Directions

To capture variation in trajectories, we need to use non-random (and carefully chosen) directions. Here, we suggest an approach based on PCA that allows us to measure how much variation we've captured; we also provide plots of these trajectories along the contours of the loss surface.

Let θ_i denote model parameters at epoch i and the final estimate as θ_n. Given n training epochs, we can apply PCA to the matrix $M = [\theta_0 - \theta_n; \cdots ; \theta_{n-1} - \theta_n]$, and then select the two most explanatory directions. Optimizer trajectories (blue dots) and loss surfaces along PCA directions are shown in Figure 5.11. Epochs where the learning rate was decreased are shown as red dots. On each axis, we measure the amount of variation in the descent path captured by that PCA direction.

We see some interesting behavior in these plots. At early stages of training, the paths tend to move perpendicular to the contours of the loss surface, i.e., along the gradient directions as one would expect from non-stochastic gradient descent.

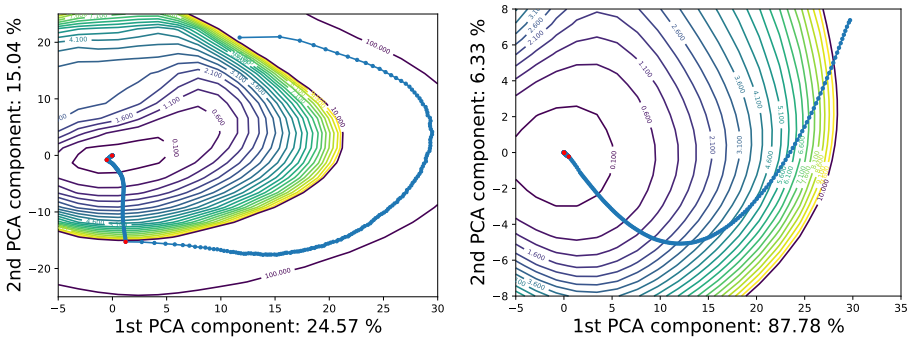
The stochasticity becomes fairly pronounced in several plots during the later stages of training. This is particularly true of the plots that use weight decay and small batches (which leads to more gradient noise, and a more radical departure from deterministic gradient directions). When weight decay and small batches are used, we see the path turn nearly parallel to the contours and "orbit" the solution when the stepsize is large. When the stepsize is dropped (at the red dot), the effective noise in the system decreases, and we see a kink in the path as the trajectory falls into the nearest local minimizer.

Finally, we can directly observe that the descent path is very low dimensional: between 40% and 90% of the variation in the descent paths lies in a space of only 2 dimensions. The optimization trajectories in Figure 5.11 appear to be dominated by movement in the direction of a nearby attractor. This low dimensionality is compatible with the observations in Section 5.5, where we observed that non-chaotic landscapes are dominated by wide, nearly convex minimizers.

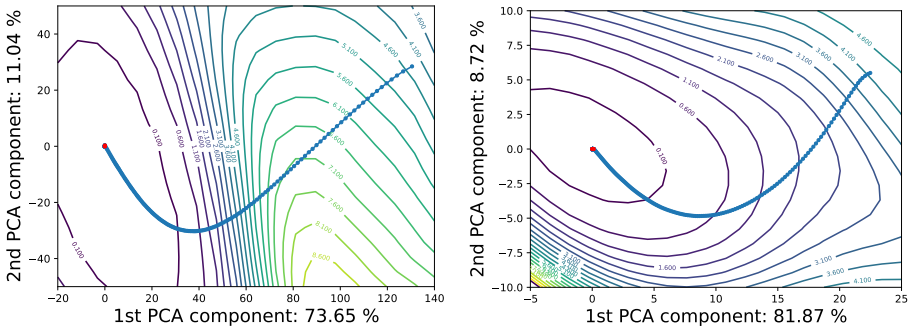
5.7 Summary

In this chapter, we presented a visualization technique that provides insights into the consequences of a variety of choices facing the neural network practitioner, including network architecture, optimizer selection, and batch size. Neural networks have advanced dramatically in recent years, largely on the back of anecdotal knowledge and theoretical results with complex assumptions. For progress to continue to be made, a more general understanding of the structure of neural networks is needed.

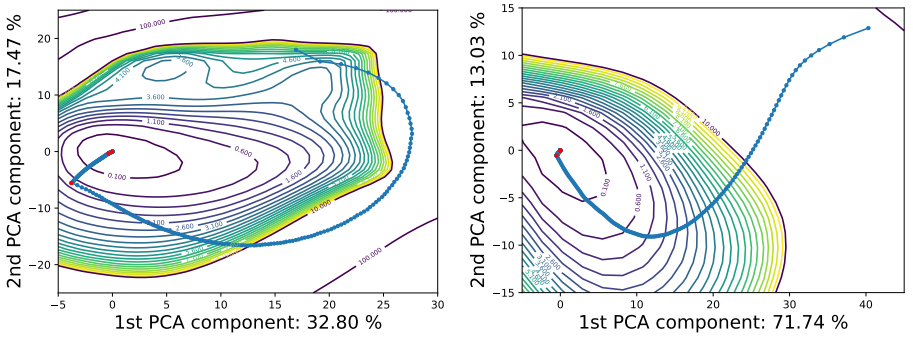
Our hope is that effective visualization, when coupled with continued advances in theory, can result in faster training, simpler models, and better generalization.



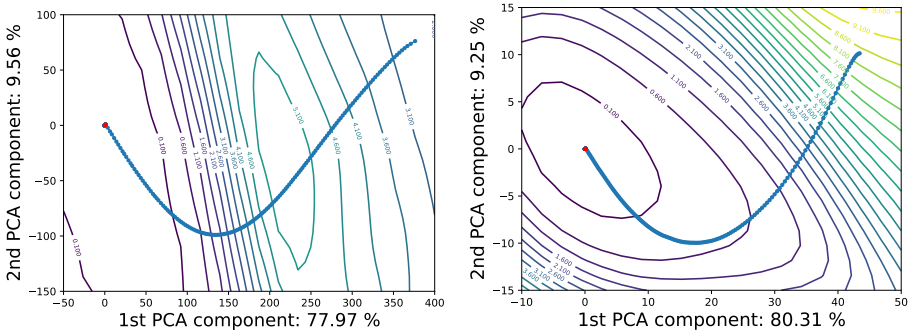
(a) SGD,WD=5e-4



(b) SGD, WD=0



(c) Adam, WD=5e-4



(d) Adam,WD=0

Figure 5.11: Projected learning trajectories use normalized PCA directions for VGG-9. The left plot in each subfigure uses batch size 128, and the right one uses batch size 8192.

Chapter 6: Conclusion

This dissertation focused on efficient inference and training methods along with a better interpretation of generalization for deep neural network. To reduce the computation cost and model size for CNNs, we proposed to prune pre-trained model by removing filters with small L_1-norms from layers that are robust to pruning, which avoids the sparsity patterns introduced by magnitude-based weights pruning. Orthogonal to model pruning, model quantization is another technique for accelerating the inference speed. We explored the behavior of training CNNs with quantized weights without keeping full-precision weights as a reference, which is essential for on device learning on resource-constrained devices. Beyond seeking the efficiency of CNNs for both inference and training, we tried to understand the trainability and generalization ability via loss landscape visualization. We identified issues with previous methods for loss surface visualization, i.e., 1D linear interpolation method is not suitable for comparing the sharpness between two minimizers. We introduced a loss surface visualization method via adding filter-normalized perturbation, which enables accurate and side-by-side loss surfaces comparison across different neural network architectures. Our visualization results provide strong correlation between flatness and generalization errors. We identified interesting connections between the

geometry of the loss surface and its generalization power.

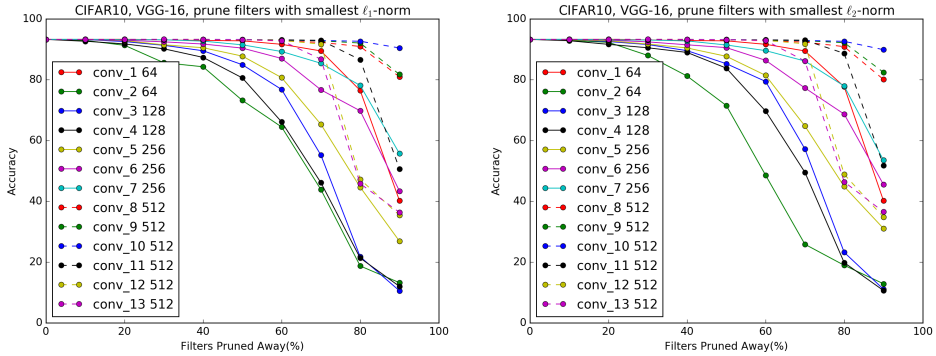
There are still many problems in representation learning that are not clearly understood and worth further exploration, especially about the interpretation of model selection and optimization process. The task of designing efficient neural networks for an unknown dataset and task remains difficult. Automatic architecture search via reinforcement learning are promising but very computationally expensive [Mallat, 2016, Zhang et al., 2017, Zoph et al., 2018]. The structure of automatically designed networks are also limited by the predefined elements and constraints, which is limited for producing novel architectures. We believe a good architecture should be built with a good understanding about the principles rather than stacking more modules. For example, an accurate correlation between model capacity and generalization power is important for performance prediction, which can be useful for guiding architecture search and parameter quantization.

On the optimization side, the relationship between trainability and generalization is still not clear, e.g., if one network architecture is easy to optimize, does it necessarily lead to good generalization? It is also well known that Recursive Neural Networks (RNNs) and Generative Adversarial Networks (GANs) are difficult to train [Goodfellow et al., 2014, Pascanu et al., 2013]. Exploration of the loss surface for these problems through visualization may provide better intuitions and solutions. In the future, we can envision that on-device learning will become pervasive and important as edge devices are getting widely deployed, which bring new opportunities as well as challenges. We can expect more interpretable models and optimization algorithms to be developed with such computation constraints.

Appendix A: Appendix for Pruning Filters for Efficient ConvNets

A.1 ℓ_2-norm based Filter Pruning

We compare the ℓ_1-norm with the ℓ_2-norm for filter pruning. As shown in Figure A.1, ℓ_1-norm works slightly better than ℓ_2-norm for layer `conv_2`. There is no significant difference between the two norms for other layers.



(a) $\|\mathcal{F}_{i,j}\|_1$ (b) $\|\mathcal{F}_{i,j}\|_2$

Figure A.1: Comparison of ℓ_1-norm and ℓ_2-norm based filter pruning for VGG-16 on CIFAR-10.

A.2 FLOP and Wall-Clock Time

FLOP is a commonly used measure to compare the computation complexities of CNNs. It is easy to compute and can be done statically, which is independent of the underlying hardware and software implementations. Since we physically prune

the filters by creating a smaller model and then copy the weights, there are no masks or sparsity introduced to the original dense BLAS operations. Therefore the FLOP and wall-clock time of the pruned model is the same as creating a model with smaller number of filters from scratch.

We report the inference time of the original model and the pruned model on the test set of CIFAR-10 and the validation set of ILSVRC 2012, which contain 10,000 32×32 images and 50,000 224×224 images respectively. The ILSVRC 2012 dataset is used only for ResNet-34. The evaluation is conducted in Torch7 with Titan X (Pascal) GPU and cuDNN v5.1, using a mini-batch size 128. As shown in Table A.1, the saved inference time is close to the FLOP reduction. Note that the FLOP number only considers the operations in the Conv and FC layers, while some calculations such as Batch Normalization and other overheads are not counted.

Table A.1: The reduction of FLOP and wall-clock time for inference.

Model	FLOP	Pruned %	Time (s)	Saved %
VGG-16	3.13×10^8		1.23	
VGG-16-pruned-A	2.06×10^8	34.2%	0.73	40.7%
ResNet-56	1.25×10^8		1.31	
ResNet-56-pruned-B	9.09×10^7	27.6%	0.99	24.4%
ResNet-110	2.53×10^8		2.38	
ResNet-110-pruned-B	1.55×10^8	38.6%	1.86	21.8%
ResNet-34	3.64×10^9		36.02	
ResNet-34-pruned-B	2.76×10^9	24.2%	22.93	28.0%

Appendix B: Appendix for Training Quantized Nets

Here we present proofs of the lemmas and theorems presented in Chapter 4, as well as some additional experimental details and results.

B.1 Proof of Lemma 1

Proof. We want to bound the quantization error r^t. Consider the i-th entry in r^t denoted by r_i^t. Similarly, we define w_i^t and $\nabla \tilde{f}(w^t)_i$. Choose some random number $p \in [0, 1]$. The stochastic rounding operation produces a value of r^t given by

$$
\begin{aligned}
r_i^t &= Q_s(w_i^t - \alpha_t \nabla \tilde{f}(w^t)_i) - w_i^t + \alpha_t \nabla \tilde{f}(w^t)_i \\
&= \Delta \cdot \begin{cases} \dfrac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} + \left\lfloor \dfrac{-\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right\rfloor + 1, & \text{for } p \leq -\dfrac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} - \left\lfloor \dfrac{-\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right\rfloor, \\ \dfrac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} + \left\lfloor \dfrac{-\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right\rfloor, & \text{otherwise,} \end{cases} \\
&= \Delta \cdot \begin{cases} -q + 1, & \text{for } p \leq q, \\ -q, & \text{otherwise,} \end{cases}
\end{aligned}
$$

where we write $q = -\frac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} - \left\lfloor \frac{-\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right\rfloor$ and $q \in [0, 1]$. Now we have

$$\begin{aligned}
\mathbb{E}_p\big[(r_i^t)^2\big] &\leq \Delta^2((-q+1)^2 q + (-q)^2(1-q)) \\
&= \Delta^2 q(1-q) \\
&\leq \Delta^2 \min\{q, 1-q\}.
\end{aligned}$$

Because $\min\{q, 1-q\} \leq \left| \frac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right|$, it follows that $\mathbb{E}_p\big[(r_i^t)^2\big] \leq \Delta^2 \left| \frac{\alpha_t \nabla \tilde{f}(w^t)_i}{\Delta} \right| \leq \Delta \left| \alpha_t \nabla \tilde{f}(w^t)_i \right|$. Summing over the index i yields

$$\begin{aligned}
\mathbb{E}_p \left\| r^t \right\|_2^2 &\leq \Delta \alpha_t \left\| \nabla \tilde{f}(w^t) \right\|_1 \\
&\leq \sqrt{d} \alpha_t \Delta \left\| \nabla \tilde{f}(w^t) \right\|_2. \qquad \text{(B.1)}
\end{aligned}$$

Now, $\left(\mathbb{E}\left\| \nabla \tilde{f}(w^t) \right\|_2\right)^2 \leq \mathbb{E}\left\| \nabla \tilde{f}(w^t) \right\|_2^2 \leq G^2$. Plugging this into (B.1) yields

$$\mathbb{E}\left\| r^t \right\|_2^2 \leq \sqrt{d} \Delta \alpha_t G. \qquad \text{(B.2)}$$

□

B.2 Proof of Theorem 1

Proof. From the update rule (4.4), we get:

$$\begin{aligned} w^{t+1} &= Q\big(w^t - \alpha_t \nabla \tilde{f}(w^t)\big) \\ &= w^t - \alpha_t \nabla \tilde{f}(w^t) + r^t, \end{aligned}$$

where r^t denotes the quantization used on the t-th iteration. Subtracting by the optimal $w^\star$, taking norm, and taking expectation conditioned on w^t, we get:

$$\begin{aligned} \mathbb{E}\|w^{t+1} - w^\star\|^2 &= \|w^t - w^\star\|^2 - 2\mathbb{E}\langle w^t - w^\star, \alpha_t \nabla \tilde{f}(w^t) - r^t\rangle + \mathbb{E}\|\alpha_t \nabla \tilde{f}(w^t) - r^t\|^2 \\ &= \|w^t - w^\star\|^2 - 2\alpha_t \langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \alpha_t^2 \mathbb{E}\|\nabla \tilde{f}(w^t)\|^2 + \mathbb{E}\|r^t\|^2 \\ &\leq \|w^t - w^\star\|^2 - 2\alpha_t \langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \alpha_t^2 G^2 + \sqrt{d}\Delta \alpha_t G, \end{aligned}$$

where we use the bounded variance assumption, $\mathbb{E}[r^t] = 0$, and Lemma 1. Using the assumption that L is μ-strongly convex, we can simplify this to:

$$\mathbb{E}\|w^{t+1} - w^\star\|^2 \leq (1 - \alpha_t \mu)\|w^t - w^\star\|^2 - 2\alpha_t(\mathcal{L}(w^t) - \mathcal{L}(w^\star)) + \alpha_t^2 G^2 + \sqrt{d}\Delta \alpha_t G.$$

Re-arranging the terms, and taking expectation we get:

$$\begin{aligned} &2\alpha_t \mathbb{E}(\mathcal{L}(w^t) - \mathcal{L}(w^\star)) \leq (1 - \alpha_t \mu)\mathbb{E}\|w^t - w^\star\|^2 - \mathbb{E}\|w^{t+1} - w^\star\|^2 + \alpha_t^2 G^2 + \sqrt{d}\Delta \alpha_t G. \\ \Rightarrow \quad &\mathbb{E}(\mathcal{L}(w^t) - \mathcal{L}(w^\star)) \leq \left(\frac{1}{2\alpha_t} - \frac{\mu}{2}\right)\mathbb{E}\|w^t - w^\star\|^2 - \frac{1}{2\alpha_t}\mathbb{E}\|w^{t+1} - w^\star\|^2 + \frac{\alpha_t}{2}G^2 + \frac{\sqrt{d}\Delta G}{2}. \end{aligned}$$

Assume that the stepsize decreases with the rate $\alpha_t = 1/\mu(t+1)$. Then we have:

$$\mathbb{E}(\mathcal{L}(w^t) - \mathcal{L}(w^\star)) \leq \frac{\mu t}{2}\mathbb{E}\|w^t - w^\star\|^2 - \frac{\mu(t+1)}{2}\mathbb{E}\|w^{t+1} - w^\star\|^2 + \frac{1}{2\mu(t+1)}G^2 + \frac{\sqrt{d}\Delta G}{2}.$$

Averaging over $t = 0$ to T, we get a telescoping sum on the right hand side, which yields:

$$\begin{aligned}\frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\mathcal{L}(w^t) - \mathcal{L}(w^\star)) &\leq \frac{G^2}{2\mu T}\sum_{t=0}^{T}\frac{1}{t+1} + \frac{\sqrt{d}\Delta G}{2} - \frac{\mu(T+1)}{2}\mathbb{E}\|w^{T+1} - w^\star\|^2 \\ &\leq \frac{(1+\log(T+1))G^2}{2\mu T} + \frac{\sqrt{d}\Delta G}{2}.\end{aligned}$$

Using Jensen's inequality, we have:

$$\mathbb{E}(\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)) \leq \frac{1}{T}\sum_{t=0}^{T}\mathbb{E}(\mathcal{L}(w^t) - \mathcal{L}(w^\star)),$$

where $\bar{w}^T = \frac{1}{T}\sum_{t=0}^{T} w^t$, the average of the iterates. Thus the final convergence theorem is given by:

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{(1+\log(T+1))G^2}{2\mu T} + \frac{\sqrt{d}\Delta G}{2}.$$

□

B.3 Proof of Theorem 2

Proof. From the update rule (4.4), we have,

$$w^{t+1} = Q\big(w^t - \alpha_t \nabla \tilde{f}(w^t)\big) = w^t - \alpha_t \nabla \tilde{f}(w^t) + r^t,$$

where r^t denotes the quantization error on the t-th iteration. Hence we have

$$\begin{aligned}\|w^{t+1} - w^\star\|^2 &= \|w^t - \alpha_t \nabla \tilde{f}(w^t) + r^t - w^\star\|^2 \\ &= \|w^t - w^\star\|^2 - 2\langle w^t - w^\star, \alpha_t \nabla \tilde{f}(w^t) - r^t\rangle + \|\alpha_t \nabla \tilde{f}(w^t) - r^t\|^2.\end{aligned}$$

Taking expectation, and using $\mathbb{E}[\tilde{f}(w^t)] = \nabla \mathcal{L}(w^t)$ and $\mathbb{E}[r^t] = 0$, we have

$$\begin{aligned}\mathbb{E}\|w^{t+1} - w^\star\|^2 &= \mathbb{E}\|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \mathbb{E}\|\alpha_t \nabla \tilde{f}(w^t) - r^t\|^2 \\ &= \mathbb{E}\|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \alpha_t^2 \mathbb{E}\|\nabla \tilde{f}(w^t)\|^2 + \mathbb{E}\|r^t\|^2.\end{aligned}$$

Using the bounded variance assumption $\mathbb{E}\|\nabla \tilde{f}(w^t)\|^2 \le G^2$ and bounded quantization error in Lemma 1, we have

$$\mathbb{E}\|w^{t+1} - w^\star\|^2 \le \mathbb{E}\|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \alpha_t^2 G^2 + \sqrt{d}\Delta\alpha_t G. \tag{B.3}$$

$\mathcal{L}(x)$ is convex and hence $\langle \nabla \mathcal{L}(x), x_t - x^\star \rangle \geq \mathcal{L}(x_t) - \mathcal{L}(x^*)$, which can be used in (B.3) to get

$$\mathbb{E}\|w^{t+1} - w^\star\|^2 \leq \mathbb{E}\|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}[\mathcal{L}(w^t) - \mathcal{L}(w^\star)] + \alpha_t^2 G^2 + \sqrt{d}\Delta\alpha_t G.$$

Re-arranging the terms, we have,

$$\mathbb{E}[\mathcal{L}(w^t) - \mathcal{L}(w^\star)] \leq \frac{1}{2\alpha_t}\left(\mathbb{E}\|w^t - w^\star\|^2 - \mathbb{E}\|w^{t+1} - w^\star\|^2\right) + \frac{\alpha_t}{2}G^2 + \frac{1}{2}\sqrt{d}\Delta G.$$

Accumulate from $t = 1$ to T to get

$$\begin{aligned}\sum_{t=1}^{T} \mathbb{E}[\mathcal{L}(w^t) - \mathcal{L}(w^\star)] \leq & \frac{1}{2\alpha_1}\mathbb{E}\|w_1 - w^\star\|^2 + \sum_{t=1}^{T}\left(\frac{1}{2\alpha_t} - \frac{1}{2\alpha_{t-1}}\right)\mathbb{E}\|w^t - w^\star\|^2 \\ & + \sum_{t=1}^{T}\frac{\alpha_t}{2}G^2 + \frac{T}{2}\sqrt{d}\Delta G.\end{aligned}$$

Applying $\mathbb{E}\|w^t - w^\star\|^2 \leq D^2$ and $\sum_{t=1}^{T}\alpha_t \leq c\sqrt{T+1}$, we have

$$\sum_{t=1}^{T} \mathbb{E}[\mathcal{L}(w^t) - \mathcal{L}(w^\star)] \leq \frac{\sqrt{T}}{2c}D^2 + \frac{c\sqrt{T+1}}{2}G^2 + \frac{T}{2}\sqrt{d}\Delta G. \tag{B.4}$$

Since $\mathcal{L}(w)$ is convex, we can set $\bar{w}^T = \frac{1}{T}\sum_{t=1}^{T} w^t$, and use Jensen's inequality to arrive at

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\mathcal{L}(w^t) - \mathcal{L}(w^\star)]. \tag{B.5}$$

Combine (B.4) and (B.5) to achieve

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{1}{2c\sqrt{T}} D^2 + \frac{\sqrt{T+1}}{2T} cG^2 + \frac{\sqrt{d}\Delta G}{2}.$$

□

B.4 Proof of Theorem 3

Proof. From the update rule (4.5), we have,

$$w^{t+1} = w^t - \alpha_t \nabla \tilde{f}\big(Q(w^t)\big) = w^t - \alpha_t \nabla \tilde{f}\big(w^t + r^t\big).$$

Taking expectation conditioned on w^t and r^t, we have

$$\begin{aligned}
&\mathbb{E}\|w^{t+1} - w^\star\|^2 \\
&= \mathbb{E}\|w^t - \alpha_t \nabla \tilde{f}\big(w^t + r^t\big) - w^\star\|^2 \\
&= \mathbb{E}\|w^t - \alpha_t \nabla \tilde{f}\big(w^t\big) + \alpha_t \nabla \tilde{f}\big(w^t\big) - \alpha_t \nabla \tilde{f}\big(w^t + r^t\big) - w^\star\|^2 \\
&= \|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \tilde{f}\big(w^t\big)\rangle + 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \tilde{f}\big(w^t\big) - \nabla \tilde{f}\big(w^t + r^t\big)\rangle + \\
&\quad \mathbb{E}\|\alpha_t \nabla \tilde{f}\big(w^t + r^t\big)\|^2 \\
&= \|w^t - w^\star\|^2 - 2\alpha_t \langle w^t - w^\star, \nabla F\big(w^t\big)\rangle + 2\alpha_t \langle w^t - w^\star, \nabla F\big(w^t\big) - \nabla F\big(w^t + r^t\big)\rangle + \\
&\quad \alpha_t^2 \mathbb{E}\|\nabla \tilde{f}\big(w^t + r^t\big)\|^2 \\
&\leq \|w^t - w^\star\|^2 - 2\alpha_t \langle w^t - w^\star, \nabla F\big(w^t\big)\rangle + 2\alpha_t \|w^t - w^\star\| \|\nabla F\big(w^t\big) - \nabla F\big(w^t + r^t\big)\| + \alpha_t^2 G^2 \\
&\leq \|w^t - w^\star\|^2 - 2\alpha_t \langle w^t - w^\star, \nabla F\big(w^t\big)\rangle + 2\alpha_t L \|r^t\| \|w^t - w^\star\| + \alpha_t^2 G^2.
\end{aligned}$$

Using $\|r^t\| \leq \sqrt{d}\Delta$ and the bounded domain assumption, we get

$$\begin{aligned}\mathbb{E}\|w^{t+1} - w^\star\|^2 &\leq \|w^t - w^\star\|^2 - 2\alpha_t\langle w^t - w^\star, \nabla F\big(w^t\big)\rangle + 2\alpha_t L\sqrt{d}\Delta\|w^t - w^\star\| + \alpha_t^2 G^2 \\ &\leq \|w^t - w^\star\|^2 - 2\alpha_t\langle w^t - w^\star, \nabla F\big(w^t\big)\rangle + 2\alpha_t L\sqrt{d}\Delta D + \alpha_t^2 G^2.\end{aligned}$$

Taking expectation, and following the same steps as in Theorem 2, we get the convergence result:

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{1}{2c\sqrt{T}}D^2 + \frac{\sqrt{T+1}}{2T}cG^2 + \sqrt{d}\Delta LD.$$

□

B.5 Proof of Theorem 4

Proof. From the update rule (4.5), we get

$$\begin{aligned}w^{t+1} &= w^t - \alpha_t\nabla\tilde{f}\big(Q(w^t)\big) \\ &= w^t - \alpha_t\nabla\tilde{f}\big(w^t + r^t\big) \\ &= w^t - \alpha_t[\nabla\tilde{f}\big(w^t\big) + \nabla^2\tilde{f}\big(w^t\big)r^t + \hat{r}^t]\end{aligned}$$

where $\|\hat{r}^t\| \leq \frac{L_2}{2}\|r^t\|^2$ from our assumption on the Hessian. Note that in general r^t has mean zero while $\hat{r}^t$ does not. Using the same steps as in the Theorem 1, we get

$$\begin{aligned}\mathbb{E}\|w^{t+1} - w^\star\|^2 &= \|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \tilde{f}(w^t + r^t)\rangle + \alpha_t^2 \mathbb{E}\|\nabla \tilde{f}(w^t + r^t)\|^2. \\ &\leq \|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \mathcal{L}(w^t) + \hat{r}^t\rangle + \alpha_t^2 G^2 \\ &= \|w^t - w^\star\|^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \nabla \mathcal{L}(w^t)\rangle + \alpha_t^2 G^2 - 2\alpha_t \mathbb{E}\langle w^t - w^\star, \hat{r}^t\rangle\end{aligned}$$

Assuming the domain has finite diameter D, and observing that the quantization error for BC-SGD can always be upper-bounded as $\|r^t\| \leq \sqrt{d}\Delta$, we get:

$$-2\alpha_t \mathbb{E}\langle w^t - w^\star, \hat{r}^t\rangle \leq 2\alpha_t D \mathbb{E}\|\hat{r}^t\| \leq 2\alpha_t D \frac{L_2}{2}\|r^t\| \leq \alpha_t D L_2 \sqrt{d}\Delta.$$

Following the same steps as in Theorem 1, we get

$$\mathbb{E}[\mathcal{L}(\bar{w}^T) - \mathcal{L}(w^\star)] \leq \frac{(1 + \log(T+1))G^2}{2\mu T} + \frac{D L_2 \sqrt{d}\Delta}{2}.$$

□

B.6 Proof of Theorem 5

Proof. Let the matrix U_α be a partial transition matrix defined by $U_\alpha(x, x) = 0$, and $U_\alpha(x, y) = T_\alpha(x, y)$ for $x \neq y$. From U_α, we can get back the full transition matrix T_α using the formula

$$T_\alpha = I - \mathrm{diag}(\mathbf{1}^T U_\alpha) + U_\alpha.$$

Note that this formula is essentially "filling in" the diagonal entries of T_α so that every column sums to 1, thus making T_α a valid stochastic matrix.

Let's bound the entries in U_α. Suppose that we begin an iteration of the stochastic rounding algorithm at some point x. Consider an adjacent point y that differs from x at only 1 coordinate, k, with $y_k = x_k + \Delta$. Then we have

$$\begin{aligned} U_\alpha(x,y) &= \frac{1}{\alpha}\int_0^{\Delta} p_{x,k}(x/\alpha)\frac{x}{\Delta}\,dx + \frac{1}{\alpha}\int_{\Delta}^{2\Delta} p_{x,k}(x/\alpha)\frac{2\Delta - x}{\Delta}\,dx \\ &= \frac{1}{\alpha}\int_0^{\Delta/\alpha} p_{x,k}(z)\frac{\alpha z}{\Delta}\alpha\,dz + \frac{1}{\alpha}\int_{\Delta/\alpha}^{2\Delta/\alpha} p_{x,k}(z)\frac{2\Delta - \alpha z}{\Delta}\alpha\,dz \\ &\leq \alpha\int_0^{\Delta/\alpha} p_{x,k}(z)\frac{z}{\Delta}\,dz + \int_{\Delta/\alpha}^{\infty} p_{x,k}(z)\,dz \\ &= \alpha\int_0^{\infty} p_{x,k}(z)\frac{z}{\Delta}\,dz + O(\alpha^2). \end{aligned} \tag{B.6}$$

Note that we have used the decay assumption:

$$\int_\nu^\infty p_{x,k}(z) \leq \frac{C}{\nu^2}.$$

Likewise, if $y_k = x_k - \Delta$, then the transition probability is

$$U_\alpha(x,y) = \alpha\int_{-\infty}^{0} p_{x,k}(z)\frac{z}{\Delta}\,dz + O(\alpha^2), \tag{B.7}$$

and if $y_k = x_k \pm m\Delta$ for an integer $m > 1$,

$$U_\alpha(x,y) = O(\alpha^2). \tag{B.8}$$

We can approximate the behavior of U_α using the matrix

$$\tilde{U}(x,y) = \begin{cases} \int_0^\infty p_{x,k}(z)\frac{z}{\Delta}\,dz, \text{if } x \text{ and } y \text{ differ only at coordinate } k, \text{ and } y_k = x_k + \Delta \\ \int_{-\infty}^0 p_{x,k}(z)\frac{z}{\Delta}\,dz, \text{if } x \text{ and } y \text{ differ only at coordinate } k, \text{ and } y_k = x_k - \Delta \\ 0, \text{otherwise.} \end{cases}$$

Define the associated Markov chain transition matrix

$$\tilde{T}_{\alpha_0} = I - \alpha_0 \cdot \text{diag}(\mathbf{1}^T \tilde{U}) + \alpha_0 \tilde{U}, \tag{B.9}$$

where α_0 is the largest scalar such that the stochastic linear operator $\tilde{T}_{\alpha_0}$ has non-negative entries. For $\alpha < \alpha_0$, $\tilde{T}_\alpha$ has non-negative entries and column sums equal to 1; it thus defines the transition operator of a Markov chain. Let $\tilde{\pi}$ denote the stationary distribution of the Markov chain with transition matrix $\tilde{T}_{\alpha_0}$.

We now claim that $\tilde{\pi}$ is also the stationary distribution of $\tilde{T}_\alpha$ for all $\alpha < \alpha_0$. We verify this by noting that

$$\begin{aligned} \tilde{T}_\alpha &= (I - \alpha \cdot \text{diag}(\mathbf{1}^T \tilde{U})) + \alpha \tilde{U} \\ &= (1 - \frac{\alpha}{\alpha_0})I + \frac{\alpha}{\alpha_0}[I - \alpha_0 \cdot \text{diag}(\mathbf{1}^T \tilde{U}) + \alpha_0 \tilde{U}] \\ &= (1 - \frac{\alpha}{\alpha_0})I + \frac{\alpha}{\alpha_0}\tilde{T}_{\alpha_0}, \end{aligned} \tag{B.10}$$

and so $\tilde{T}_\alpha \tilde{\pi} = (1 - \frac{\alpha}{\alpha_0})\tilde{\pi} + \frac{\alpha}{\alpha_0}\tilde{\pi} = \tilde{\pi}$.

Recall that T_α is the transition matrix for the Markov chain generated by

the stochastic rounding algorithm with learning rate α. We wish to show that this Markov chain is well approximated by $\tilde{T}_\alpha$. Note that

$$T_\alpha(x, y) = \prod_{k, x_k \neq y_k} T_\alpha(x, x + (y_k - x_k)\Delta e_k) \leq O(\alpha^2)$$

when x, y differ at more than 1 coordinate. In other words, transitions between multiple coordinates simultaneously become vanishingly unlikely for small α. When x and y differ by exactly 1 coordinate, we know from (B.6) that

$$T_\alpha(x, y) = \alpha U(x, y) + O(\alpha^2).$$

These observations show that the off-diagonal elements of T_α are well approximated (up to uniform $O(\alpha^2)$ error) by the corresponding elements in αU. Since the columns of T_α sum to one, the diagonal elements are well approximated as well, and we have

$$T_\alpha = (I - \alpha \cdot \mathrm{diag}(\mathbf{1}^T U)) + \alpha U + O(\alpha^2) = \tilde{T}_\alpha + O(\alpha^2).$$

To be precise, the notation above means that

$$|T_\alpha(x, y) - \tilde{T}_\alpha(x, y)| < C\alpha^2, \tag{B.11}$$

for some C that is uniform over (x, y).

We are now ready to show that the stationary distribution of T_α exists and

approaches $\tilde{\pi}$. Re-arranging (B.10) gives us

$$\alpha_0 \tilde{T}_\alpha + (\alpha - \alpha_0) I = \alpha \tilde{T}_{\alpha_0}.$$

Combining this with (B.11), we get

$$\left\| \alpha_0 T_\alpha + (\alpha - \alpha_0) I - \alpha \tilde{T}_{\alpha_0} \right\|_\infty < O(\alpha^2),$$

and so

$$\left\| \frac{\alpha_0}{\alpha} T_\alpha + (1 - \frac{\alpha_0}{\alpha}) I - \tilde{T}_{\alpha_0} \right\|_\infty < O(\alpha). \tag{B.12}$$

From (B.12), we see that the matrix $\frac{\alpha_0}{\alpha} T_\alpha + (1 - \frac{\alpha_0}{\alpha}) I$ approaches $\tilde{T}_{\alpha_0}$. Note that $\tilde{\pi}$ is the Perron-Frobenius eigenvalue of $\tilde{T}_{\alpha_0}$, and thus has multiplicity 1. Multiplicity 1 eigenvalues/vectors of a matrix vary continuously with small perturbations to that matrix (Theorem 8, p130 of Lax [2007]). It follows that, for small α, $\frac{\alpha_0}{\alpha} T_\alpha + (1 - \frac{\alpha_0}{\alpha}) I$ has a stationary distribution, and this distribution approaches $\tilde{\pi}$. The leading eigenvector of $\frac{\alpha_0}{\alpha} T_\alpha + (1 - \frac{\alpha_0}{\alpha}) I$ is the same as the leading eigenvector of T_α, and it follows that T_α has a stationary distribution that approaches $\tilde{\pi}$.

Finally, note that we have assumed $\int_0^{C_2} p_{x,k}(z)\, dz > 0$ and $\int_{-C_2}^0 p_{x,k}(z)\, dz > 0$. Under this assumption, for $\alpha < \frac{1}{C_2}$, $\tilde{T}_{\alpha_0}(x, y) > 0$ whenever x, y are neighbors the differ at a single coordinate. It follows that every state in the Markov chain $\tilde{T}_{\alpha_0}$ is accessible from every other state by traversing a path of non-zero transition probabilities, and so $\tilde{\pi}(x) > 0$ for every state x. □

B.7 Proof of Theorem 6

Proof. Given some distribution π over the states of the Markov chain, and some set A of states, let $[\pi]_A = \sum_{a \in A} \pi(a)$ denote the measure of A with respect to π.

Suppose for contradiction that the mixing time of the chain remains bounded as α vanishes. Then we can find an integer M_ϵ that upper bounds the ϵ-mixing time for all α. By the assumption of the theorem, we can select some set of states A with $[\tilde{\pi}]_A > \epsilon$, and some starting state $y \notin A$. Let e be a distribution (a vector in the finite-state case) with $e_y = 1$, $e_k = 0$ for $k \neq y$. Note that $[e]_A = 0$ because $y \notin A$. Then

$$\left|[e]_A - [\tilde{\pi}]_A\right| > \epsilon.$$

Note that, as $\alpha \to 0$, we have $\|T_\alpha - \tilde{T}_\alpha\| \to 0$ and thus $\|T_\alpha^{M_\epsilon} - \tilde{T}_\alpha^{M_\epsilon}\| \to 0$. We also see from the definition of $\tilde{T}_\alpha$ in (B.9), $\lim_{\alpha \to 0} \tilde{T}_\alpha = I$. It follows that

$$\lim_{\alpha \to 0} \left|[T_\alpha^{M_\epsilon} e]_A - [\tilde{\pi}]_A\right| = \left|[e]_A - [\tilde{\pi}]_A\right| > \epsilon,$$

and so for some α the inequality (4.9) is violated. This is a contradiction because it was assumed that M_ϵ is an upper bound on the mixing time.

□

Table B.1: VGG-9 on CIFAR-10.

layer type	kernel size	input size	output size
Conv_1	3×3	$3 \times 32 \times 32$	$64 \times 32 \times 32$
Conv_2	3×3	$64 \times 32 \times 32$	$64 \times 32 \times 32$
Max Pooling	2×2	$64 \times 32 \times 32$	$64 \times 16 \times 16$
Conv_3	3×3	$64 \times 16 \times 16$	$128 \times 16 \times 16$
Conv_4	3×3	$128 \times 16 \times 16$	$128 \times 16 \times 16$
Max Pooling	2×2	$128 \times 16 \times 16$	$128 \times 8 \times 8$
Conv_5	3×3	$128 \times 8 \times 8$	$256 \times 8 \times 8$
Conv_6	3×3	$256 \times 8 \times 8$	$256 \times 8 \times 8$
Conv_7	3×3	$256 \times 8 \times 8$	$256 \times 8 \times 8$
Max Pooling	2×2	$256 \times 8 \times 8$	$256 \times 4 \times 4$
Linear	1×1	1×4096	1×256
Linear	1×1	1×256	1×10

Table B.2: VGG-BC for CIFAR-10.

layer type	kernel size	input size	output size
Conv_1	3×3	$3 \times 32 \times 32$	$128 \times 32 \times 32$
Conv_2	3×3	$128 \times 32 \times 32$	$128 \times 32 \times 32$
Max Pooling	2×2	$128 \times 32 \times 32$	$128 \times 16 \times 16$
Conv_3	3×3	$128 \times 16 \times 16$	$256 \times 16 \times 16$
Conv_4	3×3	$256 \times 16 \times 16$	$256 \times 16 \times 16$
Max Pooling	2×2	$256 \times 16 \times 16$	$256 \times 8 \times 8$
Conv_5	3×3	$256 \times 8 \times 8$	$512 \times 8 \times 8$
Conv_6	3×3	$512 \times 8 \times 8$	$512 \times 8 \times 8$
Max Pooling	2×2	$512 \times 8 \times 8$	$512 \times 4 \times 4$
Linear	1×1	1×8192	1×1024
Linear	1×1	1×1024	1×1024
Linear	1×1	1×1024	1×10

B.8 Neural Net Architecture & Training Details

The default minibatch size is 128. However, the big-batch SR-ADAM method adopts a large minibatch size (512 for WRN-56-2 and ResNet-18 and 1024 for other models). Following [Courbariaux et al., 2015], we do not use weight decay during

training. We implement all models in Torch7 [Collobert et al., 2011] and train the quantized models with NVIDIA GPUs.

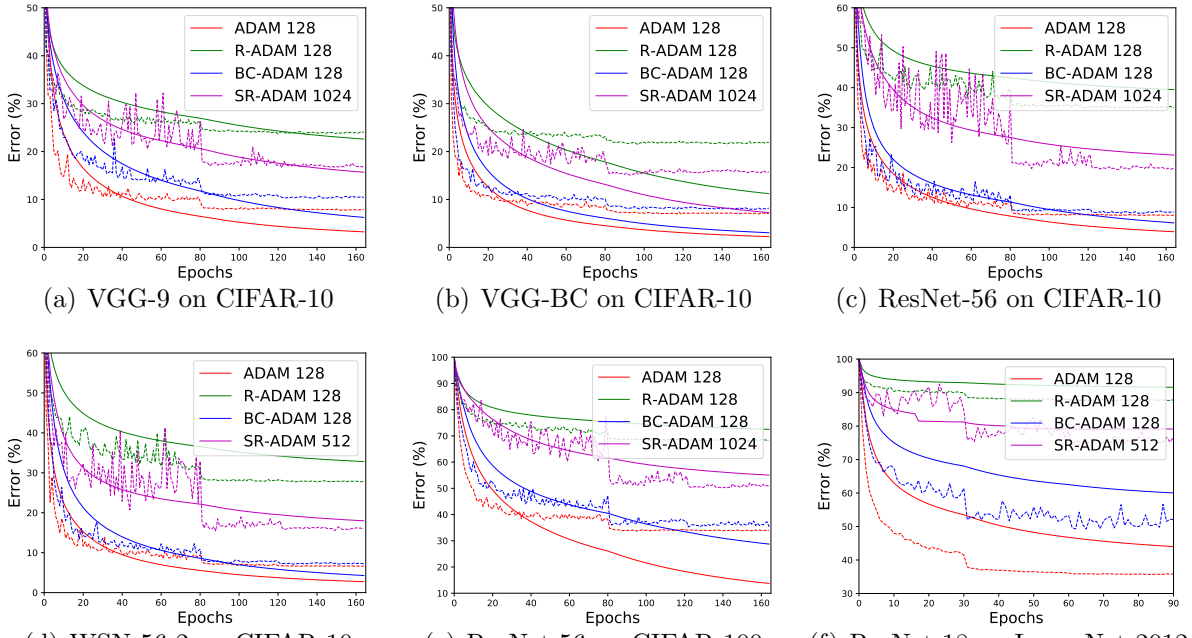
Binarizing linear layers causes some performance drop without much computational speedup. This is because fully connected layers have a very small computation overhead compared to Conv layers. Also, for state-of-the-art CNNs, the number of FC parameters is quite small. The number of params of Conv/FC layers for CNNs in Table B.1 are (in millions): VGG-9: 1.7/1.1, VGG-BC: 4.6/9.4, ResNet-56: 0.84/0.0006, WRN-56-2: 3.4/0.001, ResNet-18: 11.2/0.5. While the VGG-like nets have many FC parameters, the more efficient and higher performing ResNets are almost entirely convolutional.

B.9 Convergence Curves

The convergence curves for training and testing errors reported in Table 4.1 are shown in Figure B.1.

B.10 Weight Initialization and Learning Rate

For experiments on SR-Adam and R-Adam, the weights of convolutional layers are intitialized with random Rademacher (± 1) variables. The authors of BC [Courbariaux et al., 2015] adopt a small initial learning rate (0.003) and it takes 500 epochs to converge. It is observed that large binary weights ($\Delta = 1$) will generate small gradients when batch normalization is used [Ioffe and Szegedy, 2015], hence a large learning rate is necessary for faster convergence. We experiment with a



(a) VGG-9 on CIFAR-10 (b) VGG-BC on CIFAR-10 (c) ResNet-56 on CIFAR-10

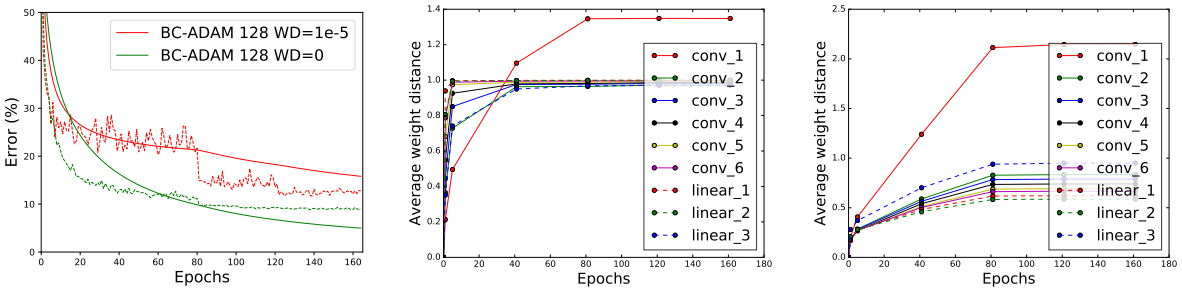
(d) WSN-56-2 on CIFAR-10 (e) ResNet-56 on CIFAR-100 (f) ResNet-18 on ImageNet 2012

Figure B.1: Training and testing errors of different training methods for VGG-9, VGG-BC, ResNet-56, Wide-ResNet-56-2 and ResNet-18. The solid line is the training error and the dashed line is the testing error.

larger learning rate (0.01) and find it converges to the same performance within 160 epochs, comparing with 500 epochs in the original paper [Courbariaux et al., 2015].

B.11 Weight Decay

Figure B.2 shows the effect of applying weight decay to BC-ADAM. As shown in Figure B.2(a), BC-ADAM with 1e-5 weight decay yields worse performance compared to zero weight decay. Applying weight decay in BC-ADAM will shrink w_r to 0, as well as increase the distance between w_b and w_r. Figure B.2(b) and B.2(c) shows the distance between w_b and w_r during training. With 1e-5 weight decay, the average weight difference between w_b and w_r approaches 1, which indicates w_r is close to zero. Weight decay cannot "decay" the weight of SR as $\|w_b\|_2$ is the same



(a) WD=1e-5 vs WD=0 (b) $|w_b^t - w_r^t|$, WD=1e-5 (c) $|w_b^t - w_r^t|$, WD=0

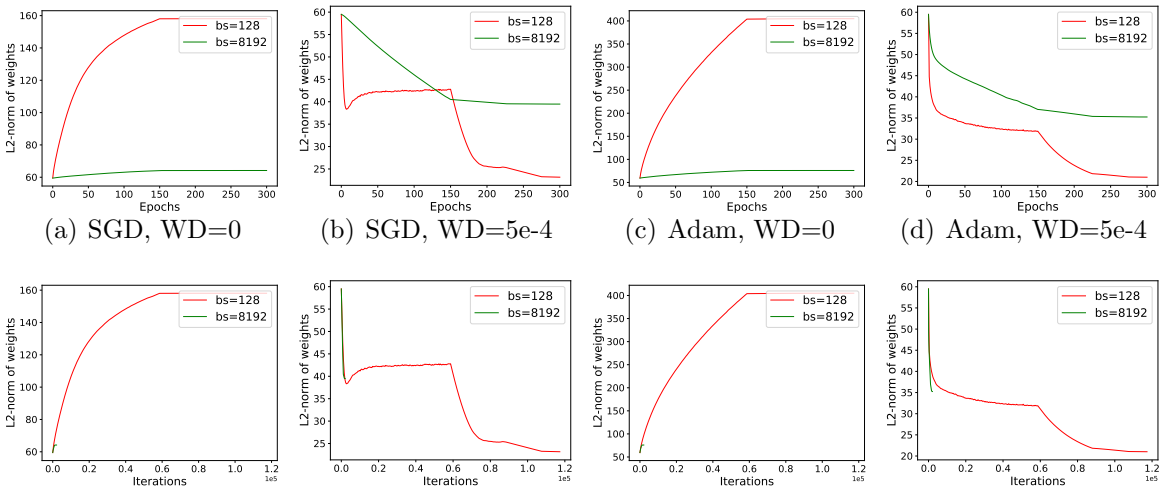
Figure B.2: The effect of weight decay (WD) on BC-ADAM for training VGG-BC. The y-axis of (b) and (c) is the averaged weight difference between the binary weights w_b and the real-valued weights w_r, i.e., $\frac{1}{d}\|w_b^t - w_r^t\|_1$. where d is the number of weights in w_b.

for all binarized networks.

Appendix C: Appendix for Loss Surface Visualization

C.1 The Change of Weights Norm during Training

Figure C.1 shows the change of weights norm during training in terms of epochs and iterations.



(a) SGD, WD=0 (b) SGD, WD=5e-4 (c) Adam, WD=0 (d) Adam, WD=5e-4

(e) SGD, WD=0 (f) SGD, WD=5e-4 (g) Adam, WD=0 (h) Adam, WD=5e-4

Figure C.1: The change of weights norm during training for VGG-9. When weight decay is disabled, the weight norm grows steadily during training without constraints. While when nonzero weight decay is adopted, the weight norm decreases rapidly at the beginning and becomes stable until the learning rate is decayed. Since we use a fixed number of epochs for different batch sizes, the difference in weight norm change between large-batch and small-batch training is mainly caused by the larger number of updates when a small batch is used. As shown in the second row, the changes of weight norm are at the same pace for both small and large batch training in terms of iterations.

C.2 Comparison of Normalization Methods

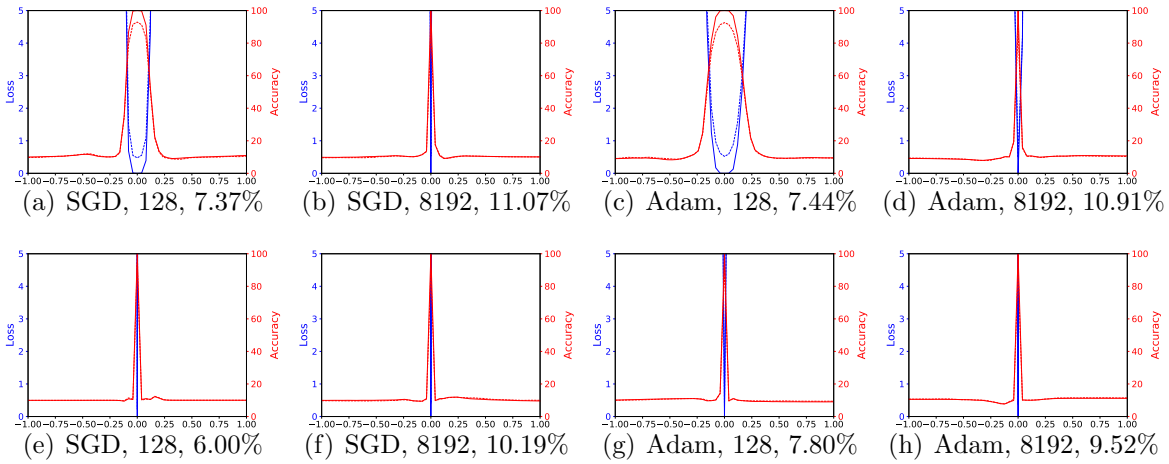
Here we compare several normalization methods for a given random normal direction d. Let d_i denote the weights of layer i and $d_{i,j}$ represent the j-th filter in the i-th layer.

- **No Normalization** In this case, the direction d is added to the weights directly without processing.
- **Layer Normalization** The direction d is normalized in the layer level so that the direction for each layer has the same norm as the corresponding layer of θ, $d_i \leftarrow \frac{d_i}{\|d_i\|}\|\theta_i\|$.

Figure C.2 shows the 1D randomalized plots without normalization. One issue with the non-normalized plots is that the x-axis range must be chosen carefully. Figure C.3 shows enlarged plots with $[-0.2, 0.2]$ as the range for x-axis. Without normalization, the plots fail to show consistency between flatness and generalization error. Here we compare filter normalization with layer normalization. We find filter normalization is more accurate than layer normalization. One failing case for layer normalization is shown in Figure C.4, where Figure C.4(g) is flatter than Figure C.4(c), but with worse generalization error.

C.3 Small-Batch vs Large-Batch for ResNet-56

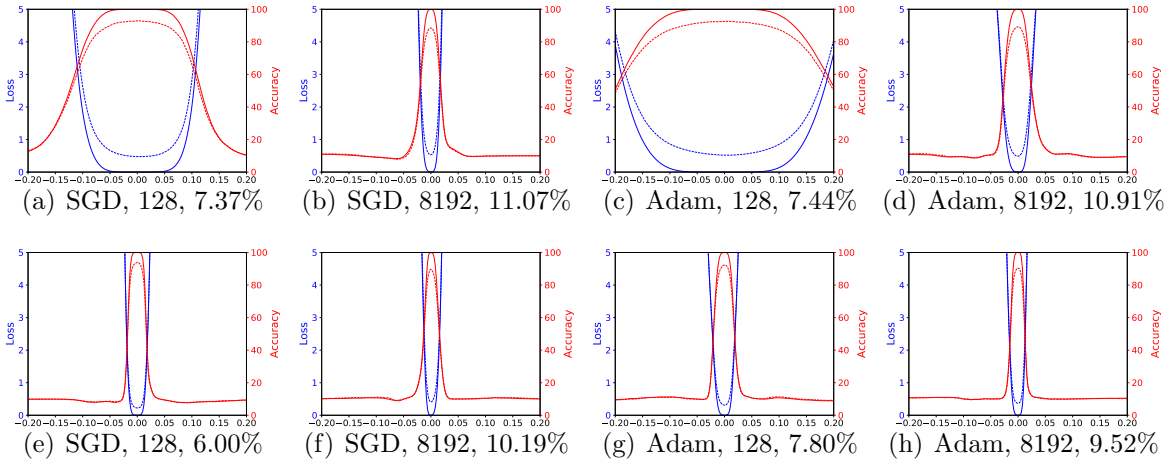
Similar to the observations made in Section 5.4, the "sharp vs flat dilemma" also applies to ResNet-56 as shown in Figure C.5. The generalization error for each



(a) SGD, 128, 7.37% (b) SGD, 8192, 11.07% (c) Adam, 128, 7.44% (d) Adam, 8192, 10.91%

(e) SGD, 128, 6.00% (f) SGD, 8192, 10.19% (g) Adam, 128, 7.80% (h) Adam, 8192, 9.52%

Figure C.2: 1D loss surface for VGG-9 without normalization. The first row has no weight decay and the second row uses weight decay 0.0005.

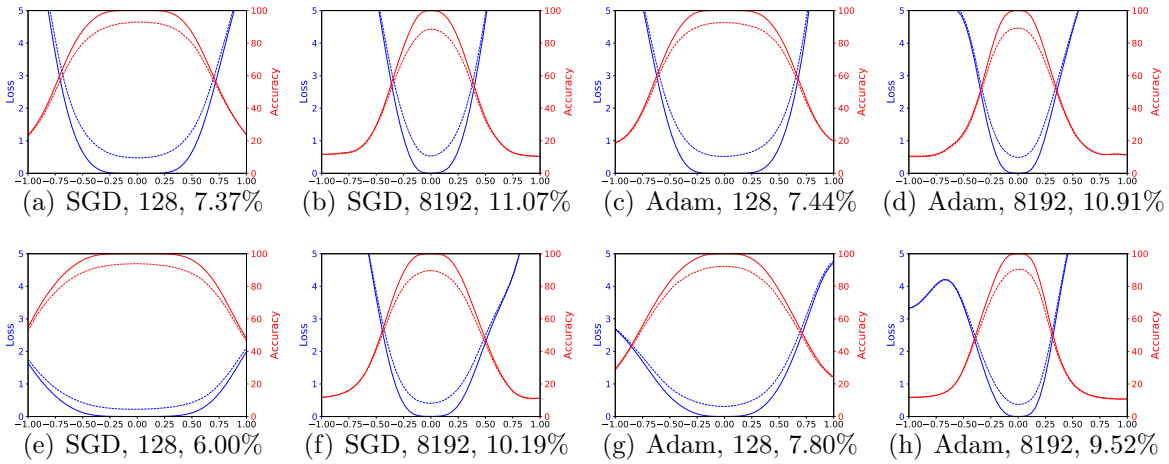


(a) SGD, 128, 7.37% (b) SGD, 8192, 11.07% (c) Adam, 128, 7.44% (d) Adam, 8192, 10.91%

(e) SGD, 128, 6.00% (f) SGD, 8192, 10.19% (g) Adam, 128, 7.80% (h) Adam, 8192, 9.52%

Figure C.3: Enlarged Figure C.2. The range of x-axis is [-0.2, 0.2] instead of [-1.0, 1.0]. The first row has no weight decay and the second row uses weight decay 0.0005. The pairs of (a, e) and (c, g) show that sharepness of minima does not correlate well with test error.

solution is shown in Table C.1. The 1D and 2D visualization with filter normalized directions are shown in Figure C.6.



(a) SGD, 128, 7.37% (b) SGD, 8192, 11.07% (c) Adam, 128, 7.44% (d) Adam, 8192, 10.91%

(e) SGD, 128, 6.00% (f) SGD, 8192, 10.19% (g) Adam, 128, 7.80% (h) Adam, 8192, 9.52%

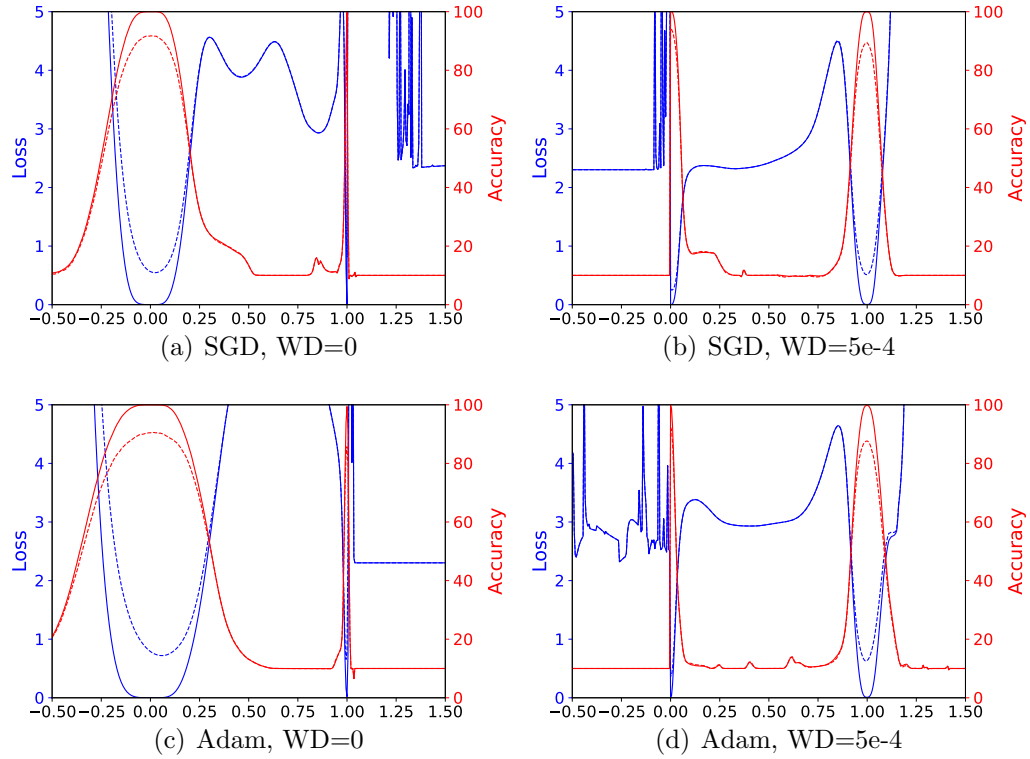
Figure C.4: 1D loss surface for VGG-9 with layer normalization. The first row has no weight decay and the second row uses weight decay 5e-4.

Table C.1: Test errors for ResNet-56 with different optimizater, batch-size and weight-decay.

	SGD		Adam	
	bs=128	bs=4096	bs=128	bs=4096
WD = 0	8.26	13.93	9.55	14.30
WD = 5e-4	**5.89**	**10.59**	**7.67**	**12.36**

C.4 Repeatability of the Loss Surface Visualization

Do different directions produce dramatically different surfaces? We plot the 1D loss surface of VGG-9 with 10 random filter-normalized directions. As shown in Figure C.7, the shapes of the different plots are very close, which indicates good generalization ability of the minima. We also repeat the 2D loss surface plots multiple times for ResNet-56-noshort, which has worse generalization error. As shown in Figure C.8, there are apparent changes in the loss surface for different plots; however, the choatic behaviour is quite consistent across plots.



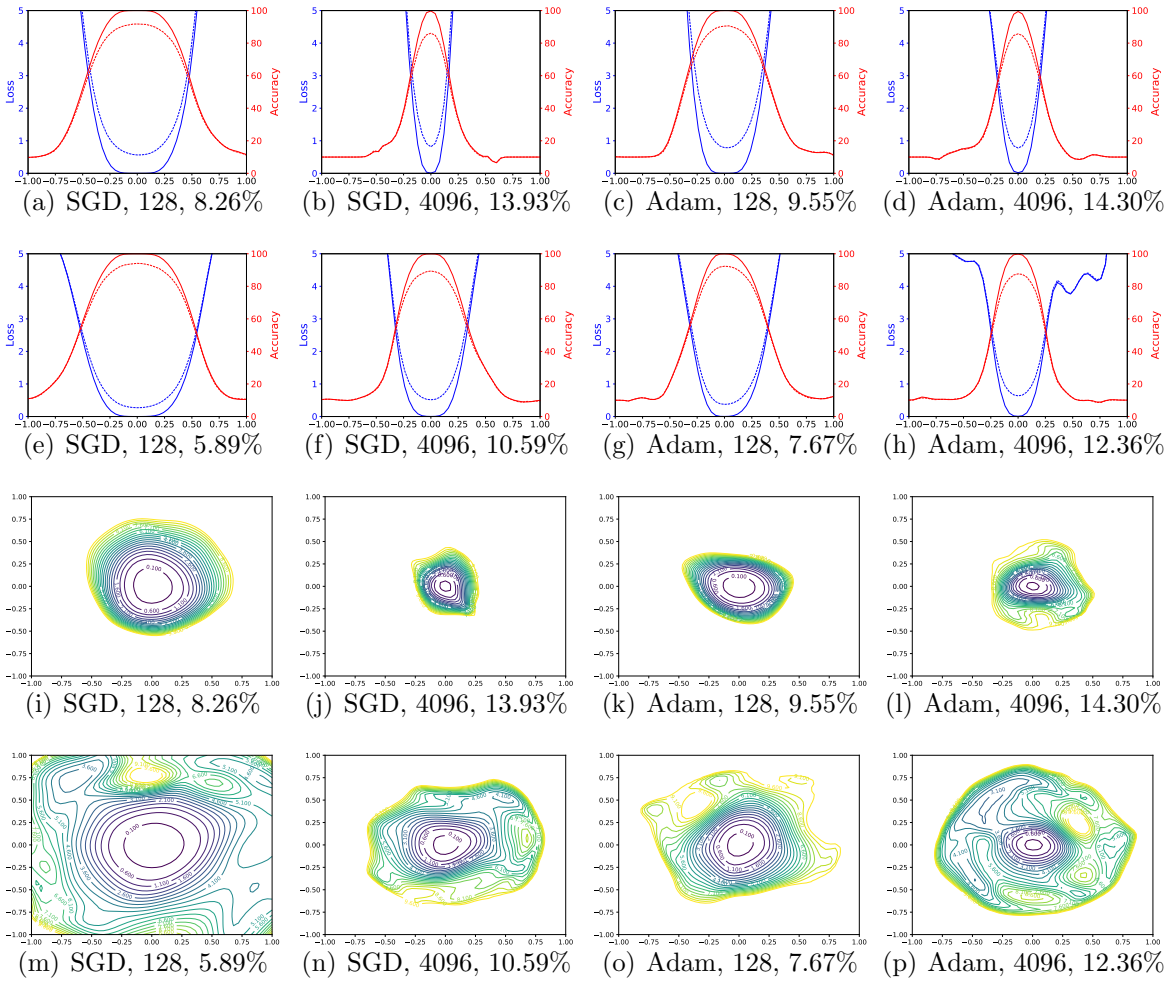
(a) SGD, WD=0 (b) SGD, WD=5e-4

(c) Adam, WD=0 (d) Adam, WD=5e-4

Figure C.5: 1D linear interpolation of solutions obtained by small-batch and large-batch methods for ResNet-56. The blue lines are loss values and the red lines are error.

C.5 Implementation Details

Computing resources for generating the figures Our PyTorch code can be executed in a multiple GPU workstation as well as a HPC with hundreds of GPUs using `mpi4py`. The computation time depends on the model's inference speed on the training set, the resolution of the plots, and the number of GPUs. The resolution for the 1D plots in Figure 5.5 is 401. The default resolutions used for the 2D contours in Figure 5.5 and Figure 5.6 is 51×51. We use higher resolutions (251×251) for the ResNet-56-noshort used in Figure 5.1 to show more details. For example, a 2D contour of ResNet-56 model with a (relatively low) resolution of 51×51 will take



(a) SGD, 128, 8.26% (b) SGD, 4096, 13.93% (c) Adam, 128, 9.55% (d) Adam, 4096, 14.30%

(e) SGD, 128, 5.89% (f) SGD, 4096, 10.59% (g) Adam, 128, 7.67% (h) Adam, 4096, 12.36%

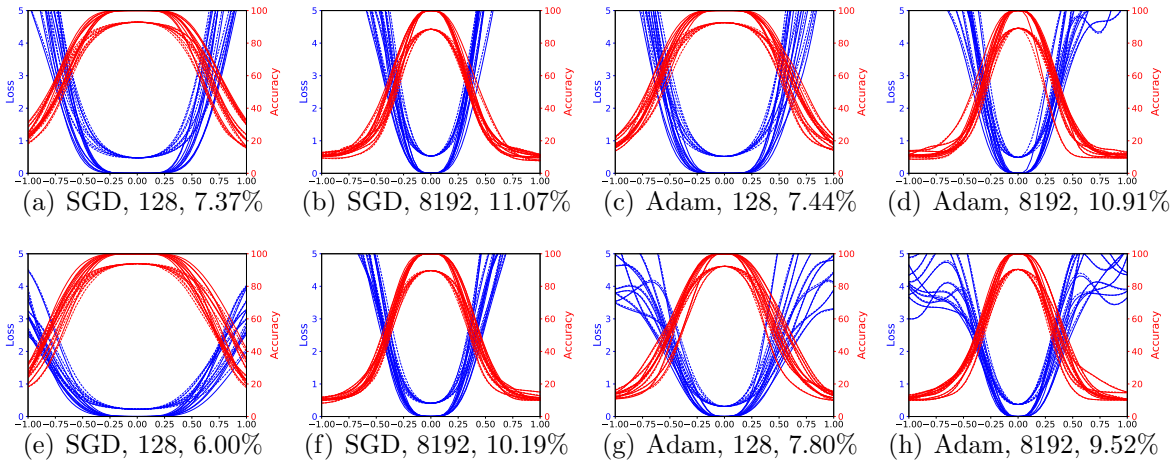
(i) SGD, 128, 8.26% (j) SGD, 4096, 13.93% (k) Adam, 128, 9.55% (l) Adam, 4096, 14.30%

(m) SGD, 128, 5.89% (n) SGD, 4096, 10.59% (o) Adam, 128, 7.67% (p) Adam, 4096, 12.36%

Figure C.6: 1D and 2D visualization of ResNet-56 trained with different optimizer, batch size and weight decay. The first and third row uses zero weight decay and the second and fourth row uses 5e-4 weight decay.

about 1 hour on a workstation with 4 GPUs (Titan X Pascal or 1080 Ti).

Batch Normalization parameters In the 1D linear interpolation methods, the Batch Normalization (BN) parameters including the "running mean" and "running variance" need to be considered as part of θ. If these parameters are not considered, then it is not possible to reproduce the exact loss values for both minimizers. In the filter-normalized visualization, the random direction applies to all weights but not



(a) SGD, 128, 7.37% (b) SGD, 8192, 11.07% (c) Adam, 128, 7.44% (d) Adam, 8192, 10.91%

(e) SGD, 128, 6.00% (f) SGD, 8192, 10.19% (g) Adam, 128, 7.80% (h) Adam, 8192, 9.52%

Figure C.7: Repeatability of the surface plots for VGG-9 with filter normalization. The shape of minima obtained using 10 different random filter-normalized directions.

the weights in BN. Note that the filter normalization process removes the effect of weight scaling, and so the batch normalization can be ignored.

The VGG-9 architecture and parameters for Adam VGG-9 is a cropped version of VGG-16, which keeps the first 7 Conv layers in VGG-16 with 2 FC layers. A BN layer is added after each conv layer and the first FC layer. We find VGG-9 is an efficient network with better performance comparing to VGG-16 on CIFAR-10. We use the default values for β_1, β_2 and ϵ in Adam with the same learning rate schedule as used in SGD.

C.6 Training Curves for VGG-9 and ResNets

The loss curves for training VGG-9 used in Section 5.4 are shown in Figure C.9. Figure C.10 shows the loss curves and error curves of architectures used in Section 5.5 and Table C.2 shows the final error and loss values. The default setting for training

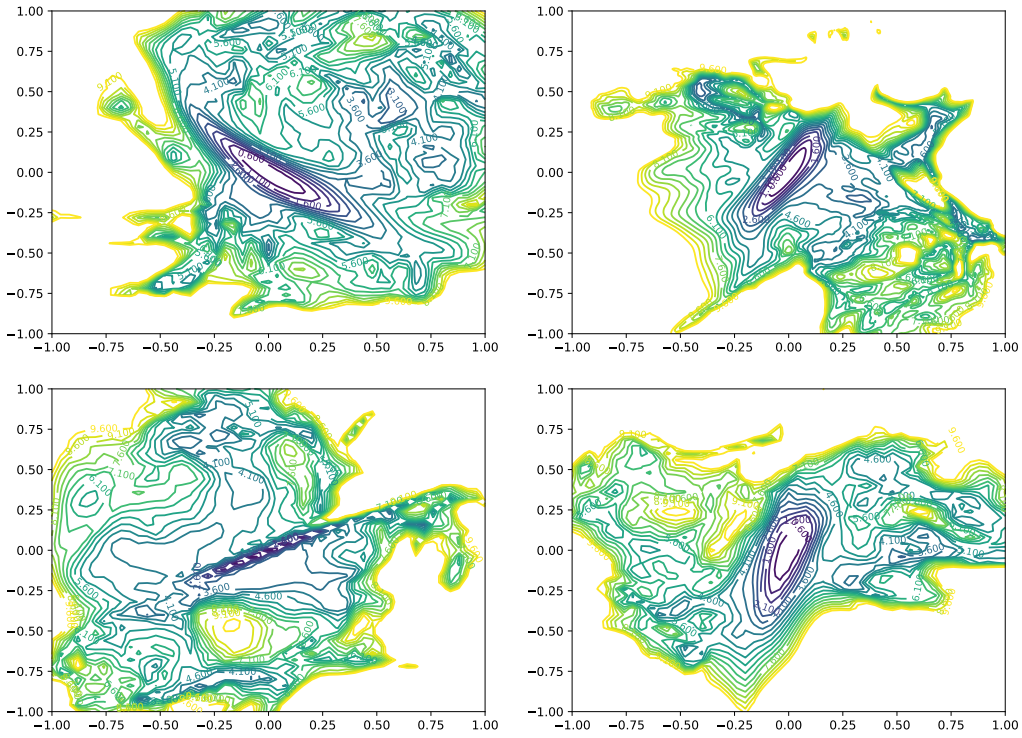
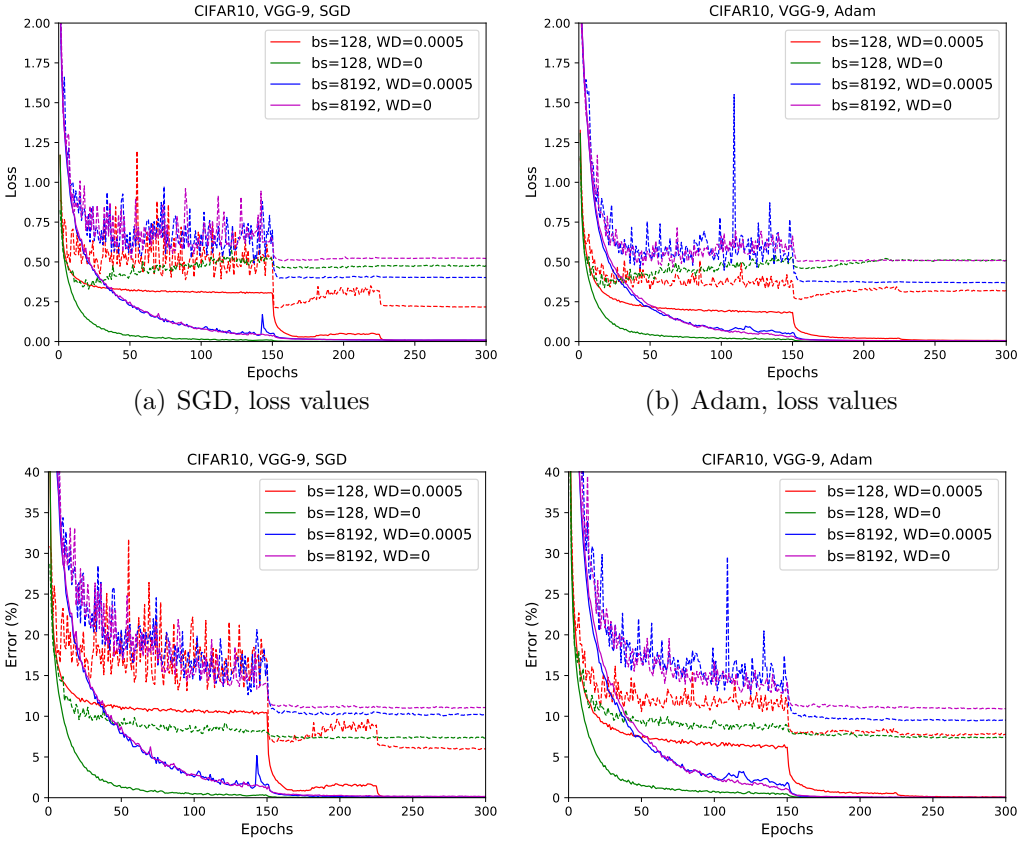


Figure C.8: Repeatability of the 2D surface plots for ResNet-56-noshort. The model is trained with batch size 128, initial learning rate 0.1 and weight decay 5e-4. The final training loss is 0.192, the training error is 6.49 and the test error is 13.31.

is using SGD with Nesterov momentum, batch-size 128, and 0.0005 weight decay for 300 epochs. The default learning rate was initialized at 0.1, and decreased by a factor of 10 at epochs 150, 225 and 275.

Table C.2: Loss values and errors for different architectures trained on CIFAR-10.

	init LR	Training Loss	Training Error	Test Error
ResNet-20	0.1	0.017	0.286	7.37
ResNet-20-noshort	0.1	0.025	0.560	8.18
ResNet-56	0.1	0.004	0.052	5.89
ResNet-56-noshort	0.1	0.192	6.494	13.31
ResNet-56-noshort	0.01	0.024	0.704	10.83
ResNet-110	0.1	0.002	0.042	5.79
ResNet-110-noshort	0.01	0.258	8.732	16.44



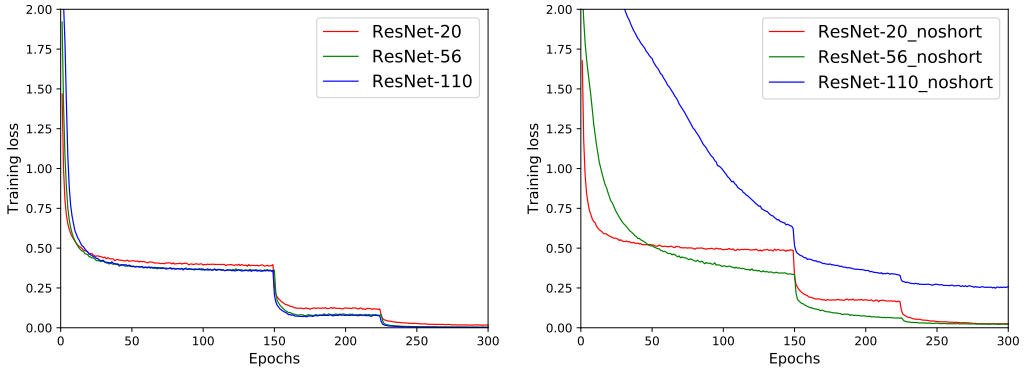
(a) SGD, loss values

(b) Adam, loss values

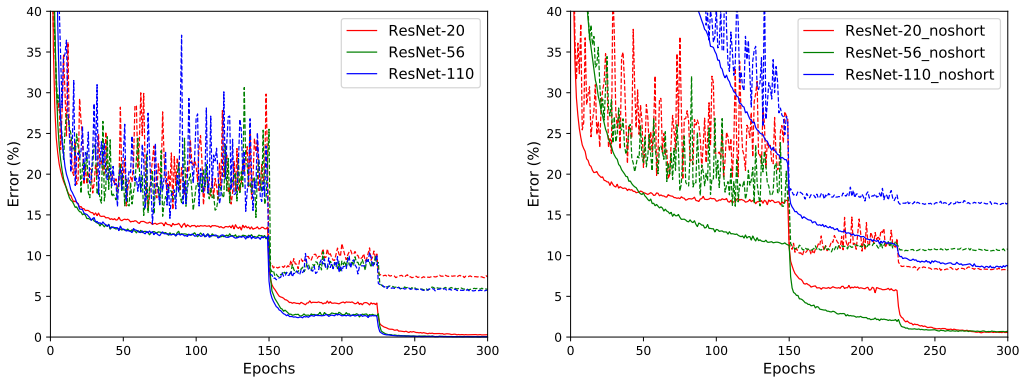
(c) SGD, errors

(d) Adam, errors

Figure C.9: Training loss/error curves for VGG-9 with different optimization methods. The first row shows loss curves and the second is about the error curves. Dashed lines are for testing, solid for training.



(a) ResNet-CIFAR (b) ResNet-CIFAR-noshort



(c) ResNet-CIFAR (d) ResNet-CIFAR-noshort

Figure C.10: Convergence curves for different architectures. The first row is for training loss and the second row shows error curves.

Bibliography

Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns: Application to face recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):2037–2041, 2006.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *ICASSP*, 2015a.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured Pruning of Deep Convolutional Neural Networks. *arXiv preprint arXiv:1512.08571*, 2015b.

Aharon Azulay and Yair Weiss. Why do deep convolutional networks generalize so poorly to small image transformations? *arXiv preprint arXiv:1805.12177*, 2018.

Carlo Baldassi, Federica Gerace, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Learning may need only a few bits of synaptic precision. *Physical Review E*, 93(5):052313, 2016.

David Balduzzi, Marcus Frean, Lennox Leary, JP Lewis, Kurt Wan-Duo Ma, and Brian McWilliams. The shattered gradients problem: If resnets are the answer, then what is the question? In *ICML*, 2017.

Ronen Basri and David Jacobs. Efficient Representation of Low-Dimensional Manifolds using Deep Networks. *arXiv preprint arXiv:1602.04723*, 2016.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013a.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013b.

Avrim Blum and Ronald L Rivest. Training a 3-node neural network is np-complete. In *NIPS*, 1989.

Pratik Chaudhari, Anna Choromanska, Stefano Soatto, and Yann LeCun. Entropy-SGD: Biasing Gradient Descent into Wide Valleys. In *ICLR*, 2017.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

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

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.

Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005.

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, 2014.

Soham De, Abhay Yadav, David Jacobs, and Tom Goldstein. Automated inference with adaptive batches. In *AISTATS*, 2017.

Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *NIPS*, 2013.

Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. 2017.

Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for Energy-Efficient Neuromorphic Computing. In *NIPS*, 2015.

Li Fei-Fei and Pietro Perona. A bayesian hierarchical model for learning natural scene categories. In *CVPR*, 2005.

C Daniel Freeman and Joan Bruna. Topology and geometry of half-rectified network optimization. In *ICLR*, 2017.

Marcus Gallagher and Tom Downs. Visualization of learning in multilayer perceptron networks using principal component analysis. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 33(1):28–34, 2003.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Tom Goldstein and Christoph Studer. Phasemax: Convex phase retrieval via basis pursuit. *arXiv preprint arXiv:1610.07531*, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014.

Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. In *ICLR*, 2015.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, 2015.

Benjamin D Haeffele and René Vidal. Global optimality in neural network training. In *CVPR*, 2017.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both Weights and Connections for Efficient Neural Network. In *NIPS*, 2015.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016a.

Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2016b.

Moritz Hardt and Tengyu Ma. Identity matters in deep learning. In *ICLR*, 2017.

Babak Hassibi and David G Stork. Second Order Derivatives for Network Pruning: Optimal Brain Surgeon. In *NIPS*, 1993.

Kaiming He and Jian Sun. Convolutional Neural Networks at Constrained Time Cost. In *CVPR*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1): 1–42, 1997.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *NIPS*, 2017.

Markus Höhfeld and Scott E Fahlman. Probabilistic rounding in neural network learning with limited precision. *Neurocomputing*, 4(6):291–299, 1992.

Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *CVPR*, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.

Forrest Iandola, Matthew Moskewicz, Khalidand Ashraf, Song Han, William Dally, and Keutzer Kurt. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡ 1MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Daniel Jiwoong Im, Michael Tao, and Kristin Branson. An empirical analysis of deep network loss surfaces. *arXiv preprint arXiv:1612.04010*, 2016.

Yani Ioannou, Duncan Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training CNNs with Low-Rank Filters for Efficient Image Classification. In *ICLR*, 2016.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 2015.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *BMVC*, 2014.

Kenji Kawaguchi. Deep learning without poor local minima. In *NIPS*, 2016.

Kenji Kawaguchi, Leslie Pack Kaelbling, and Yoshua Bengio. Generalization in deep learning. *arXiv preprint arXiv:1710.05468*, 2017.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

Minje Kim and Paris Smaragdis. Bitwise neural networks. In *ICML Workshop on Resource-Efficient Machine Learning*, 2015.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *NIPS*, 1992.

Guanghui Lan, Arkadi Nemirovski, and Alexander Shapiro. Validation analysis of mirror descent stochastic approximation method. *Mathematical programming*, 134 (2):425–458, 2012.

Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *CVPR*, 2016.

P.D. Lax. *Linear Algebra and Its Applications*. Number v. 10 in Linear algebra and its applications. Wiley, 2007. ISBN 9780471751564. URL `https://books.google.com/books?id=e7FJM6aqZD8C`.

Quoc V Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress, 2011.

Yann Le Cun, John S Denker, and Sara A Solla. Optimal Brain Damage. In *NIPS*, 1989.

Vadim Lebedev and Victor Lempitsky. Fast Convnets Using Group-wise Brain Damage. In *CVPR*, 2016.

Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *NIPS*, 1990.

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.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436, 2015.

David Asher Levin, Yuval Peres, and Elizabeth Lee Wilmer. *Markov chains and mixing times*. American Mathematical Soc., 2009.

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

Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. In *NIPS*, 2017a.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient Convnets. In *ICLR*, 2017b.

Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913*, 2017c.

Yuanzhi Li and Yang Yuan. Convergence analysis of two-layer neural networks with relu activation. *arXiv preprint arXiv:1705.09886*, 2017.

Qianli Liao and Tomaso Poggio. Theory of deep learning ii: Landscape of the empirical risk in deep learning. *arXiv preprint arXiv:1703.09833*, 2017.

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *ICML*, 2016a.

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

Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. In *ICLR*, 2016b.

Zachary C Lipton. Stuck in a what? adventures in weight space. In *ICLR Workshop*, 2016.

Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse Convolutional Neural Networks. In *CVPR*, 2015.

Eliana Lorch. Visualizing deep network training trajectories with pca. *ICML Workshop on Visualization for Deep Learning*, 2016.

David G Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

Haihao Lu and Kenji Kawaguchi. Depth creates no bad local minima. *arXiv preprint arXiv:1702.08580*, 2017.

S Mallat. Understanding deep convolutional networks. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 374(2065), 2016.

Michele Marchesi, Gianni Orlandi, Francesco Piazza, and Aurelio Uncini. Fast neural networks without multipliers. *IEEE Transactions on Neural Networks*, 4 (1):53–62, 1993.

Zelda Mariet and Suvrit Sra. Diversity Networks. In *ICLR*, 2016.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *ICML*, 2015.

Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast Training of Convolutional Networks through FFTs. *arXiv preprint arXiv:1312.5851*, 2013.

Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

Quynh Nguyen and Matthias Hein. The loss surface of deep and wide neural networks. In *ICML*, 2017.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML*, 2013.

Adam Polyak and Lior Wolf. Channel-Level Acceleration of Deep Face Representations. *IEEE Access*, 2015.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

Itay Safran and Ohad Shamir. On the quality of the initial basin in overspecified neural networks. In *ICML*, 2016.

Shreyas Saxena and Jakob Verbeek. Convolutional Neural Fabrics. In *NIPS*, 2016.

Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

Leslie N Smith and Nicholay Topin. Exploring loss function topology with cyclical learning rates. *arXiv preprint arXiv:1702.04283*, 2017.

Mahdi Soltanolkotabi, Adel Javanmard, and Jason D Lee. Theoretical insights into the optimization landscape of over-parameterized shallow neural networks. *arXiv preprint arXiv:1707.04926*, 2017.

Daniel Soudry and Elad Hoffer. Exponentially vanishing sub-optimal local minima in multilayer neural networks. *arXiv preprint arXiv:1702.05777*, 2017.

Suraj Srinivas and R Venkatesh Babu. Data-free Parameter Pruning for Deep Neural Networks. In *BMVC*, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *JMLR*, 2014.

Grzegorz Swirszcz, Wojciech Marian Czarnecki, and Razvan Pascanu. Local minima in training of deep networks. *arXiv preprint arXiv:1611.06310*, 2016.

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*, 2015a.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. *arXiv preprint arXiv:1512.00567*, 2015b.

Cheng Tai, Tong Xiao, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. In *ICLR*, 2016.

Yuandong Tian. An analytical formula of population gradient for two-layered relu network and its applications in convergence and critical point analysis. In *ICML*, 2017.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning Structured Sparsity in Deep Learning. In *NIPS*, 2016.

Bo Xie, Yingyu Liang, and Le Song. Diverse neural network learns true target functions. In *AISTATS*, 2017.

Chulhee Yun, Suvrit Sra, and Ali Jadbabaie. Global optimality conditions for deep neural networks. *arXiv preprint arXiv:1707.02444*, 2017.

Sergey Zagoruyko. 92.45% on CIFAR-10 in Torch. `http://torch.ch/blog/2015/07/30/cifar.html`, 2015.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. In *ECCV*, 2014.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *ICLR*, 2017.

Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *CVPR*, 2015.

Hao Zhou, Jose Alvarez, and Fatih Porikli. Less Is More: Towards Compact CNNs. In *ECCV*, 2016a.

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*, 2016b.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.