Data-Constrained Model Compression

Maxwell Horton

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Reading Committee:
Ali Farhadi, Chair
Mohammad Rastegari
Hannaneh Hajishirzi

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Data-Constrained Model Compression

Maxwell Horton

Chair of the Supervisory Committee:
Professor Ali Farhadi
Computer Science and Engineering

In recent years, strong progress has been made in compressing compute-heavy machine learning models to enable them to execute in real-time on edge devices. Typically, model compression techniques require retraining a model on the original dataset of interest. This is problematic if the original dataset is unavailable due to privacy or legal concerns, or if the model to be compressed was obtained from a third party. We explore the challenges associated with compressing a model in three different data-constrained scenarios. In the first scenario, labels are unavailable. We approach this problem through knowledge distillation, training a smaller model using predictions made from a larger model on unlabeled data. In the second scenario, both data and labels are unavailable. We approach this problem by separately compressing every layer of a pretrained model to obtain a compressed approximation of the original model. Our method is computationally efficient, achieving strong compression rates while maintaining accuracy. In the third scenario, we explore the problem of dynamic, real-time compression after model deployment. We demonstrate a training technique in which we condition a model to achieve high accuracy across a variety of compression levels, allowing for efficient, real-time model selection along the efficiency-accuracy trade-off curve after model deployment. We present these works to elucidate the challenges associated with data-constrained model compression, and to provide solutions for compressing models in these challenging scenarios.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

CONVOLUTIONAL NEURAL NETWORK (CNN): a variety of neural network containing convolutional operations, typically used in computer vision.

NETWORK COMPRESSION: any method used to reduce the number of parameters of a network, or to increase its execution speed. These methods include, but are not limited to, efficient network design, quantization, and pruning.

NEURAL NETWORK: a function containing learned parameters. The parameters are typically learned through an iterative training procedure.

NEURAL NETWORK SUBSPACE: a collection of trained neural networks with the property that their weights can be interpolated to produce another accurate network.

PRUNING: a method of neural network compression which removes unnecessary network weights.

QUANTIZATION: a method of neural network compression which quantizes floating-point weights to integral representations for reduced size and increased speed. For example, compressing 32-bit floating point weights to 8-bit integers.

SPARSITY: see pruning.

TRANSFORMER: a variety of neural network originally developed for natural language processing and recently adapted for computer vision.

ACKNOWLEDGMENTS

First and foremost, I'd like to thank my academic advisors. Ali Farhadi has provided me with encouragement, support, motivation, and vision. Mohammad Rastegari taught me to dream, to pursue outlandish and bold ideas. I feel lucky to have had such thoughtful mentorship from both of them.

I would like to thank colleagues, mentors, and advisors who convinced me to pursue computer science research. My undergraduate advisor, Maria Spiropulu, persuaded me to pursue a PhD. Without her mentorship, I would have missed out on the joy of life as a graduate student. I'd also like to thank Brooklyn Schlamp, Connor DeFanti, and Robert Karl for cultivating my interest in computer science as an undergraduate. I never would have figured out how to close vim without you.

I'd like to thank my collaborators: Hessam Bagherinezhad, Yanzi Jin, Mitchell Wortsman, Elvis Nunez, Anish Prabhu, and Anurag Ranjan. Your inspiration and thoughtful criticism helped shape our work and my thinking. I am indebted to Hessam for sharing ideas and helping me identify my interests in my early years of graduate school.

Raj Katti encouraged me to pursue graduate school. My brothers Nick Horton and Alex Horton provided inspiration and guidance. My parents Patty Horton and Jeff Horton provided lots of encouragement. A long list of friends and coworkers at University of Washington, Xnor.Ai, and Apple Inc. inspired me to pursue interesting challenges and to view problems with a new perspective.

DEDICATION

A small contribution towards understanding the behavior of machine learning models in data-constrained scenarios.

Chapter 1

INTRODUCTION

In recent years, research focused on machine learning model efficiency has brought models from cloud computing centers to edge devices [54, 110, 107, 39, 72, 61]. Enabling on-device execution of machine learning models permits a wide range of applications not possible through cloud computing alone. For example, consider a smart-home security system capable of identifying people in or around a home. Analyzing the video feed on-device avoids the security risk of sending a live video stream of one's house over the internet [1]. It also saves a large amount of bandwidth. Furthermore, applications requiring real-time responsiveness may not be suitable for sending inference samples to the cloud. Virtual reality, augmented reality, and self-driving vehicles require a more rapid response time than offline tasks like photo library analysis. Such applications may benefit from avoiding the latency of sending information to the cloud. Finally, lack of connectivity can prohibit cloud-based inference. As machine learning capabilities broaden, the importance of algorithms supporting on-device machine learning increases.

When studying methods for compressing machine learning models, two limiting assumptions are made. First, researchers usually assume that the dataset of interest is available. In many circumstances, this is not the case. For instance, privacy or legal concerns may limit the ability to distribute datasets. Additionally, it is not uncommon for datasets to be retracted after being published [2]. In these cases, a pretrained model may already be available, but if it requires too much compute, a compressed version may be desired.

The second limiting assumption is that a device has a fixed amount of available resources. Usually networks with increased accuracy require increased compute. When deploying a model to a device, a natural question arises: how much compute is the user willing to spend

on network inference? This question rarely has a straightforward answer. The available resources on a device are always changing. A user may wish to devote less compute to a neural network if his or her device is low on battery. The resulting sacrifice in accuracy may be acceptable it if it avoids draining too much battery. Many network architectures contain a "width factor" parameter that controls the trade-off between efficiency and accuracy [54, 130, 115]. Deploying a suite of such models to enable an efficiency-accuracy trade-off will incur a relatively large parameter overhead, which can be problematic for edge devices. Moreover, switching between models may be relatively time-consuming if the extra model weights cannot be held in memory. Finally, this method requires separate training runs for every efficiency-accuracy configuration, as well as knowledge of all run-time configurations before deployment. This inflexibility prohibits models from adapting with maximum efficiency. Consider the case of a large number of models working together to support an intelligent application. If only a few efficiency-accuracy configurations are deployed for each model, it's not possible to slightly reduce the compute of all models simultaneously. Instead, a few models will be greatly compressed, which could destabilize the intelligent application.

We develop methods for compressing models to enable real-world applications in data-constrained scenarios. We present three works in detail to achieve this. In the first, *Label Refinery* (Chapter 2), we examine model compression when data is present, but labels are missing. In the second, *Layer-Wise Data-Free CNN Compression* (Chapter 3), we examine model compression when data and labels are both missing. Finally, in *Learning Compressible Subspaces* (Chapter 4), we examine model compression when we need to efficiently compress to arbitrary compression levels after model deployment.

In *Label Refinery*, we explore the scenario in which data is present, but labels are missing. This situation arises when the original dataset is unavailable due to legal or privacy issues, but unlabeled replacement data is available. This scenario assumes that labels are difficult or expensive to obtain (e.g. if samples require expert annotation, or detailed manual labor). In this scenario, obtaining a compressed model requires using a pretrained model capable of labeling existing data-points. This setup is typically referred to as Knowledge Distilla-

tion [50], in which a "teacher" model distills learned information into a "student" model. Typically, the teacher is a larger network and the student is the same network or a smaller network. Training the student to match the teacher's outputs gives superior accuracy to training the student on the original labels. Common intuition is that the teacher can help the student model label noise more effectively, resulting in a more accurate student model. We show that this pattern extends beyond the typical case of a single round of supervision. We train several student models in succession, using each student as the teacher for the next student model. Interestingly, this repetition provides increased model accuracy. We analyze the generalization behavior of such sequentially-trained models. We find that repeating this process results in accuracy saturation after roughly 5 rounds of training. We also demonstrate that knowledge distillation can provide significant accuracy increases for compressed models on ImageNet, compared to the same model architecture trained on labeled data. For example, we improve the accuracy of MobileNet0.25 from 50.65% to 55.59% by training with our distillation method.

In *Layer-Wise Data-Free CNN Compression*, we examine model compression when both data and labels are missing. This scenario arises if legal or privacy issues prevent the original dataset from being distributed, or if a third party trains a model that needs to be compressed. One common approach to this problem is to generate data using a Generative Adversarial Network (GAN) [34, 152, 18], but this approach has the drawback that GANs are generally difficult to train in a stable manner. In a truly data-free scenario, no validation set is available, which makes it difficult to know whether the GAN has diverged. Furthermore, parameter settings for a GAN may not generalize to different datasets or models.

We take a simpler approach that involves compressing each layer of the network separately. We model the inputs and outputs to the network using randomly-generated Gaussian data that matches the network's BatchNorm [58] statistics. By minimizing the per-layer error during compression, we obtain a compressed network that preserves accuracy. Furthermore, our method is orders of magnitude more efficient than GAN-based methods. When pruning, we outperform baselines of a similar compute envelope, achieving 1.5 times the sparsity rate

at the same accuracy. We also show how to combine our efficient method with high-compute generative methods to improve upon their results.

Finally, in *Learning Compressible Subspaces*, we explore methods for post-training compression after model deployment. In this scenario, our goal is to compress a model to arbitrary compression levels after deployment. As discussed previously, deploying multiple models to allow for an efficiency-accuracy trade-off is costly in terms of device storage. Providing only a few efficiency-accuracy configurations has the additional drawback of preventing a large number of models from reducing their accuracy slightly in order to save compute. Instead, a few models must incur a larger loss of accuracy.

To address this, we demonstrate a method for model compression to arbitrary compression levels. Our method conditions a set of weights to achieve high accuracy when uncompressed, and maintain accuracy when varying levels of compression are applied. Our method is efficient (in that it requires no extra parameters), adaptive (in that the compression level can be chosen after deployment), and real-time (in that the compression level can be changed in less than the cost of a forward pass). We allow for operation at any compression level in a wide range (e.g. 0% to 90% for structured sparsity with ResNet18 [45] on ImageNet [22]). We achieve accuracies comparable with trained models trained for the given compression level.

In the chapters that follow, we discuss our methods and results in detail. Additional results are provided in the Appendix. We hope this work will encourage further research and exploration into the area of model compression in data-constrained environments.

Chapter 2

LABEL REFINERY[1]

Among the three main components (data, labels, and models) of any supervised learning system, data and models have been the main subjects of active research. However, studying labels and their properties has received very little attention. Current principles and paradigms of labeling impose several challenges to machine learning algorithms. Labels are often incomplete, ambiguous, and redundant. In this paper we study the effects of various properties of labels and introduce the *Label Refinery*: an iterative procedure that updates the ground truth labels after examining the entire dataset. We show significant gain using refined labels across a wide range of models. Using a Label Refinery improves the state-of-the-art top-1 accuracy of (1) AlexNet from 59.3 to 67.2, (2) MobileNet[1] from 70.6 to 73.39, (3) MobileNet$^{0.25}$ from 50.6 to 55.59, (4) VGG19 from 72.7 to 75.46, and (5) Darknet19 from 72.9 to 74.47.

2.1 Introduction

There are three main components in the pipeline of a typical supervised learning system: the *data*, the *model*, and the *labels*. Sources of data have expanded drastically in past several years. We have observed the impact of large-scale datasets for several visual tasks. A variety of data augmentation methods [136, 138, 150, 120] have effectively expanded these datasets and improved the performance of learning systems. Models have also been extensively studied in the literature. Recognition systems have shown improvements by increasing the depth of the architectures [46, 121], introducing new activation and normalization layers [58, 69], and developing optimization techniques and loss functions [64, 148]. In contrast to the improvements in data and models, little effort has focused on improving labels.

[1]WORK COMPLETED AT XNOR.AI.

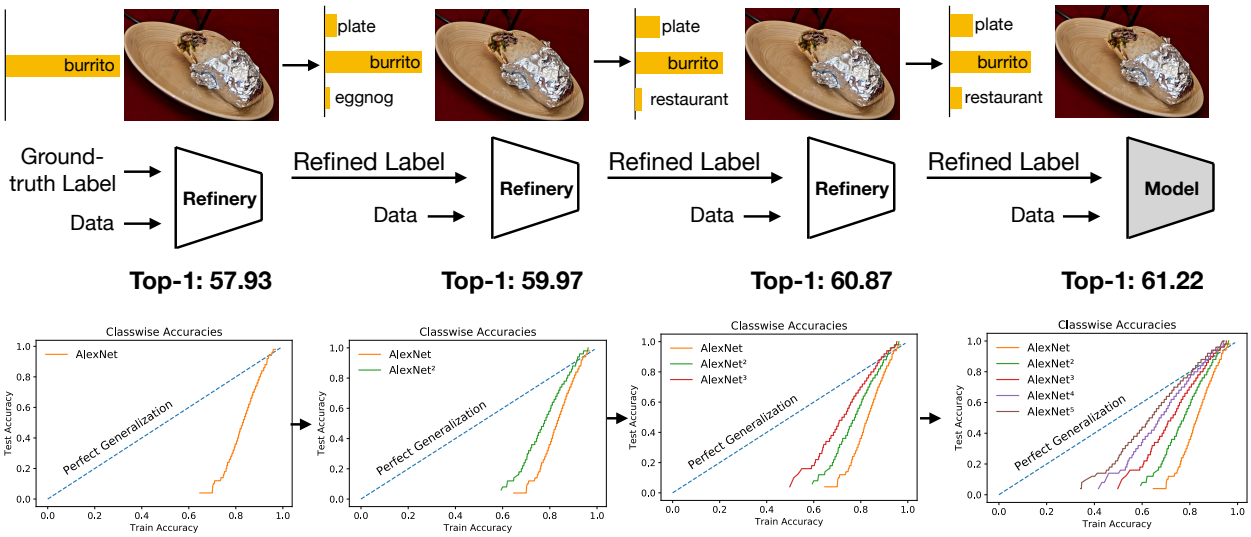


Figure 2.1: Current labeling principles impose challenges for machine learning models. We introduce the *Label Refinery*, an iterative procedure to update ground truth labels using a visual model trained on the entire dataset. The Label Refinery produces soft, multi-category, dynamically-generated labels consistent with the visual signal. The training image shown is labelled with the single category "burrito". After a few iterations of label refining, the labels from which the final model is trained are informative, unambiguous, and smooth. This results in major improvements in the model accuracy during successive stages of refinement as well as improved model generalization. These plots show that as models proceed through successive stages of refinement, the gaps between train and test results and approach ideal generalization.

Current labeling practices impose specific challenges on our learning algorithms. 1) **Incompleteness:** A natural image of a particular category will contain other object categories as well. For example, Figure 2.2(a) shows an image from ImageNet that is labeled "cat" but the image contains a "ball" as well. This problem is rooted in the nature of how researchers define and collect labels, and is not unique to a specific dataset. 2) **Taxonomy Dependency:** Categories that are far from each other in the taxonomy structure can be visually similar (Figure 2.3). 3) **Inconsistency:** To prevent overfitting, various loss functions and regularization techniques have been introduced into the training process. Data augmentation [136] is one of the most effective methods employed to prevent neural networks from memorizing the training data. Most modern state-of-the-art architectures for image classifi-

(a)

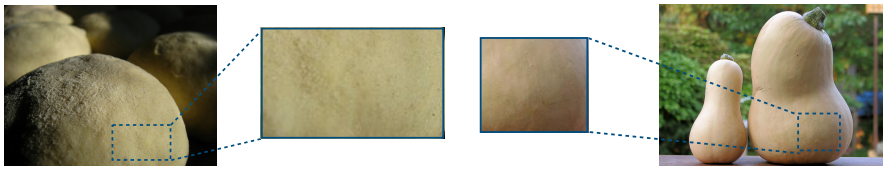
(b)

Figure 2.2: Figure 2.2(a) shows a sample image from the "persian cat" category of ImageNet's training set. The standard technique to train modern state-of-the-art architectures is to crop patches as small as 8% area of the original image, and label them with the original image's label. This will often result in inaccurate labels for the augmented data. Figure 2.2(b) shows a sample crop of the original image where the "persian cat" is no longer in the crop. A trained ResNet-50 labels Figure 2.2(a) by "persian cat", and labels Figure 2.2(b) by "golf ball". We claim that using a model to generate labels for the patches results in more accurate labels and therefore more accurate models.

cation are trained with crop-level data augmentation, in which crops of the image used for training can be as small as 8% of the area of the original image [46]. For many categories, such small crops will frequently result in patches in which the object of interest is no longer visible (Figure 2.2), resulting in an inconsistency with the original label.

To address the aforementioned shortcomings, we argue that several characteristics should apply to ideal labels. Labels should be *soft* to provide more coverage for co-occurring and visually-related objects. Traditional one-hot vector labels introduce challenges in the modeling stage. Labels should be *informative* of the specific image, meaning that they should not be identical for all the images in a given class. For example, an image of a "dog" that has similar appearance to a "cat" should have a different label than an image of a "dog" that has similar appearance to a "fox". This also suggest that labels should be defined at the instance-level rather than the category-level. Determining the best label for each instance may require observing the entire dataset to establish intra- and inter-category relations, suggesting that labels should be *collective* across the whole dataset. Labels should also be

consistent with the image content when crops are taken. Therefore, labels should be *dynamic* in the sense that the label for a crop should depend on the content of the crop.



(a) (b)

Figure 2.3: Sample training examples from "dough" and "butternut squash" categories of ImageNet. While the two sample images are visually distinctive, their random crops are quiet similar. A trained ResNet-50 labels both cropped patches softly over categories of "dough", "butternut squash", "burrito", "french loaf", and "spaghetti squash". We claim that labelling the crops softly by a trained model makes the training of the same model more stable, and therefore results in more accurate models.

In this paper we introduce *Label Refinery*, a solution that uses a neural network model and the data to modify crop labels during training. Refining the labels while training enables us to generate soft, informative, collective, and dynamic labels. Figure 2.1 depicts an example of a label refinery. As models progress through the stages of the refinery, labels are updated using the previous models. This results in major improvements in model accuracy and generalization. The output of the label refinery is a set of labels from which one can learn a model. The model trained from the produced labels are much more accurate and more robust to over-fitting.

Our experiments show that Label Refining consistently improves the accuracy of image classification networks by a large margin across a variety of popular network architectures. Our improvements in Top-1 accuracy on the ImageNet validation set include: AlexNet from 59.3% to 67.2%, VGG19 from 72.7% to 75.46%, ResNet18 from 69.57% to 72.52%, ResNet50 from 75.7% to 76.5%, DarkNet19 from 72.9% to 74.47%, $\text{MobileNet}_{0.25}$ from 50.65% to 55.59%, and MobileNet_{1} from 70.6% to 73.39%. Collective and dynamic labels enable stan-

dard models to generalize better, resulting in significant improvements in image classification. Figure 2.1 Plots the train versus test accuracies as models go through the label refinery procedure. The gap between train and test accuracies is getting smaller and closer to an ideal generalization.

We further demonstrate that a trained model can serve as a Label Refinery for another model of the same architecture. For example, we iterate through several successions of training a new AlexNet model by using the previously trained AlexNet model as a Label Refinery. Our results show major improvements (from 59.3% to 61.2%) on using AlexNet to refine labels for another AlexNet. Note that the final AlexNet has not seen the actual ground-truth labels in the past few stages. The final AlexNet models demonstrate greatly reduced overfitting compared to the original models (Figure 2.4(a) and Figure 2.5). We also experiment with using a model of one architecture as a Label Refinery for a model of another architecture. Further, we have also shown that adversarially modifying image examples improves the accuracy when using label refinery.

Our contributions include: (1) introducing the Label Refinery for crop-level label augmentation, (2) improving state-of-the-art accuracy on ImageNet for a variety of existing architectures, (3) demonstrating the ability of a network to improve accuracy by training from labels generated by another network of the same architecture, and (4) generating adversarial examples to improve the performance of the Label Refinery method.

2.2 Related Work

Label Smoothing and Regularization: Softening labels has been used to improve generalization. [128] uniformly redistributes 10% of the weight from the ground-truth label to other classes to help regularize during training. DisturbLabel [148] replaces some of the labels in a training batch with random labels. This helps regularize training by preventing overfitting to ground-truth labels. [112] augments noisy labels using other models to improve label consistency. [93] introduces a notion of local distributional smoothness in model outputs based on the smoothness of the model's outputs when inputs are perturbed. The

smoothness criterion is enforced with the purpose of regularizing models. The work of [106] explores penalizing networks by regularizing the entropy of the model outputs. Unlike our method, these approaches can not address the inconsistency of the labels.

Incorporating Taxonomy: Several methods have explored using taxonomy to improve label and model quality. [77] uses cross-category relationships from knowledge graphs to mitigate the issues caused by noisy labels. [144] designs a hierarchical loss to reduce the penalty for predictions that are close in taxonomy to the ground-truth. [143] investigates learning multi-label classification with missing labels. They incorporate instance-level information as well as semantic hierarchies in their solution. Incorporating taxonomic information directly into the model's architecture is explored in [15]. [87] uses the output of existing binary classifiers to address the problem of training models on single-label examples that contain multiple training categories. These methods fail to address the incompleteness of the labels. Instead of directly using taxonomy, our model collectively infer the visual relations between categories to impose these knowledge into the training while capturing a complete description of the image.

Data Augmentation: To preserve generalization, several data augmentations such as cropping, rotating, and flipping input images have been applied in training models [69, 121, 46, 126]. [138] proposes data warping and synthetic over-sampling to generate additional training data. [136] and [120] explore using GANs to generate training examples. Most of such augmentation techniques further confuse the model with inconsistent labels. For example, a random crop of an image might not contain the main object the that image We propose augmenting the labels alongside with the data by refining them during training when augmenting the data.

Teacher-Student Training: Using another network or an ensemble of multiple networks as a teacher model to train a student model has been explored in [13, 5, 75, 118, 49, 114]. [5] explores training a shallow student network from a deeper teacher network. A teacher model is used in [114, 118] to train a compressed student network. Most similar to our work is [49], where they introduce distillation loss for training a model from an ensemble of its

own. We show that Label Refinery must be done at the crop level, it benefits from being performed iteratively, and models benefit by learning off of the labels generated by the exact same model.

Model	**Top-1**	**Top-5**
AlexNet	57.93	79.41
AlexNet^2	59.97	81.44
AlexNet^3	60.87	82.13
AlexNet^4	61.22	**82.56**
AlexNet^5	**61.37**	**82.56**

Model	**Top-1**	**Top-5**
ResNet50	75.7	92.81
ResNet50^2	**76.5**	**93.12**

Model	**Top-1**	**Top-5**
MobileNet	68.51	88.13
MobileNet^2	**69.52**	**88.7**

Model	**Top-1**	**Top-5**
VGG16	70.1	88.54
VGG16^2	71.85	90.07
VGG16^3	**72.49**	**90.76**

Model	**Top-1**	**Top-5**
VGG19	71.39	89.44
VGG19^2	72.66	90.75
VGG19^3	**73.32**	**91.30**

Model	**Top-1**	**Top-5**
Darknet19	70.6	89.13
Darknet19^2	72.74	90.73
Darknet19^3	**73.01**	**90.92**

Table 2.1: Self-Refining results on the ImageNet 2012 validation set. Each model is trained using labels refined by the model right above it. That is, AlexNet^3 is trained by the labels refined by AlexNet^2, and AlexNet^2 is trained by the labels refined by AlexNet. The first row models are trained using the image level ground-truth labels.

2.3 Label Refinery

Previous works have shown that data augmentation using cropping significantly improves the performance of classification models [69, 127]. Given a dataset $\mathcal{D} = \{(X_i, Y_i)\}$, we

can formalize data augmentation by defining a new dataset $\tilde{\mathcal{D}} = \{(f(X_i), Y_i)\}$, where f is a stochastic function that generates crops on-the-fly for the image X_i. The image labels assigned to the augmented crops are often not accurate (Figure 2.2 and Figure 2.3). We address this problem by passing the dataset through multiple Label Refiners. The first Label Refinery network C_{θ_1} is trained over the dataset $\tilde{\mathcal{D}}$ with the inaccurate crop labels. The second Label Refinery network C_{θ_2} is trained over the same set of images, but uses labels generated by C_{θ_1}. More formally, we can view this procedure as training C_{θ_2} on a new augmented dataset $\tilde{\mathcal{D}}_1 = \{(f(X_i), C_{\theta_1}(f(X_i)))\}$. Once C_{θ_2} is trained, we can similarly use it to train a subsequent network C_{θ_3}.

We train the first Label Refinery network C_{θ_1} using the cross-entropy loss against the image-level ground-truth labels. We train all subsequent Label Refinery models C_{θ_t} for $t > 1$ by minimizing the KL-divergence between its output and the soft label generated by the previous Label Refinery $C_{\theta_{t-1}}$. Letting $p_c^t(z) \triangleq C_{\theta_t}(z)[c]$ be the probability assigned to class c in the output of model C_{θ_t} on some crop z, our loss function for training model C_{θ_t} is:

$$\begin{aligned} L_t(f(X_i)) &= -\sum_c p_c^{t-1}(f(X_i)) \log \left(\frac{p_c^t(f(X_i))}{p_c^{t-1}(f(X_i))} \right) \\ &= -\sum_c p_c^{t-1}(f(X_i)) \log p_c^t(f(X_i)) \\ &\quad + \sum_c p_c^{t-1}(f(X_i)) \log p_c^{t-1}(f(X_i)) \end{aligned} \tag{2.1}$$

The second term is the entropy of the soft labels, and is constant with respect to C_{θ_t}. We can remove it and instead minimize the cross entropy loss:

$$\tilde{L}_t(f(X_i)) = -\sum_c p_c^{t-1}(f(X_i)) \log p_c^t(f(X_i)) \tag{2.2}$$

Note that training C_{θ_1} using cross entropy loss can be viewed as a special case of our sequential training method using KL-divergence in which C_{θ_1} is trained from the original image-level labels. It's worth emphasizing that the subsequent models do not see the original ground

truth labels Y_i. The information in the original labels is propagated by the sequence of Label Refinery networks.

If any of the Label Refinery networks have Batch Normalization [58], we put them in training mode even at the label generation step. That is, their effective mean and standard deviation to be computed from the current training batch as opposed to the saved running mean and running variance. We have observed that this results in more accurate labels and, therefore, more accurate models. We believe that this is due to the fact that the Label Refinery has been trained with the Batch Normalization layers in the training mode. Hence it produces more accurate labels *for the training set* if it's in the same mode.

It is possible to use the same network architecture for some (or all) of the Label Refinery networks in the sequence. We have empirically observed that the dataset labels improve iteratively even when the same network architecture is used multiple times (Section 2.4). This is because the same Label Refinery network trained on the new refined dataset becomes more accurate that its previous versions over each pass. Thus, subsequent networks are trained with more accurate labels.

The accuracy of a trained model heavily depends on the consistency of the labels provided to it during training. Unfortunately, assessing the quality of crop labels quantitatively is not possible because there crop level labels are not provided. Asking human annotators to evaluate individual crops is infeasible both due to the number of possible crops and due to the difficulty of evaluating soft labels to a large number of categories for a crop in which there may not be a single main object. We can use a network's validation set accuracy as a measure of its ability to produce correct labels for crops. Intuitively, this measurement serves as an indication of the quality of a Label Refinery network. However, we observe that models with higher validation accuracy do not always produce better crop labels if the model with higher validation accuracy is severely overfit to the training set. Intuitively, this is because the model will reproduce the ground-truth image labels for training set images. We explore this more in Section 2.4.1.

One popular way to augment ImageNet data is to crop patches as small as 8% of the area

Model	Paper Number		Our Impl.		Label Refinery	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
AlexNet [69]	59.3	81.8	57.93	79.41	**66.28**†	**86.13**†
MobileNet [56]	70.6	N/A	68.53	88.14	**73.39**	**91.07**
MobileNet0.75 [56]	68.4	N/A	65.93	86.28	**70.92**	**89.68**
MobileNet0.5 [56]	63.7	N/A	63.03	84.55	**66.66**†	**87.07**†
MobileNet0.25 [56]	50.6	N/A	50.65	74.42	**54.62**†	**77.92**†
ResNet-50 [46]	N/A	N/A	75.7	92.81	**76.5**	**93.12**
ResNet-34 [46]	N/A	N/A	73.39	91.32	**75.06**	**92.35**
ResNet-18 [46]	N/A	N/A	69.7	89.26	**72.52**	**90.73**
ResNetXnor-50 [108]	N/A	N/A	63.1	83.61	**70.34**	**89.18**
VGG16 [121]	73	91.2	70.1	88.54	**75**	**92.22**
VGG19 [121]	72.7	91	71.39	89.44	**75.46**	**92.52**
Darknet19 [111]	72.9	91.2	70.6	89.13	**74.47**	**91.94**

Table 2.2: Using refined labels improves the accuracy of a variety of network architectures to new state-of-the-art accuracies. The Label Refinery used in these experiments is a ResNet-50 model trained with weight decay.

† These models can be further improved by training with adversarial inputs (Table 2.3).

of the image [127]. In the presence of such aggressive data augmentation, the original image label is often very inaccurate for the given crop. Whereas traditional methods only augment the image input data through cropping, we additionally augment the labels using Label Refinery networks to produce labels for the crops. Smaller networks such as MobileNet [56] usually aren't trained with such small crops. Yet, we observe that such networks can benefit from small crops if a Label Refinery is used. This demonstrates that a primary cause in accuracy degradation of such networks is inaccurate labels on small crops.

2.3.1 Adversarial Jittering

Using a Label Refinery network allows us to generate labels for any set of images. Our training dataset $\tilde{\mathcal{D}}_t = \{(f(X_i), C_{\theta_t}(f(X_i)))\}$ depends only on the input images X_i, and labels are generated on-the-fly by the Refinery network C_{θ_t}. This means that we are no longer limited to using images in the training set $\mathcal{D}$. We could use another unlabeled image dataset as a source of X_i. We could even use synthetic images. We experiment with using the Label Refinery in conjunction with the network being trained in order to generate adversarial examples on which the two networks disagree.

Let $C_{\theta_{t-1}}$ and C_{θ_t} be two of the networks in a sequence of Label Refinery networks. Given a crop $f(X_i)$, we define $\alpha_t(f(X_i))$ to be a modification of $f(X_i)$ for which $C_{\theta_{t-1}}$ and C_{θ_t} output different probability distributions. Following the practice of [103] for generating adversarial examples, we define α_t as

$$\alpha_t(X) = X + \eta \frac{\partial L_t}{\partial X} \tag{2.3}$$

, where L_t is the KL-divergence loss defined in Equation 2.1. This update performs one step of gradient ascent in the direction of increasing the KL-divergance loss. In other words, the input is modified to exacerbate the discrepancy between the output probability distributions. In order to prevent the model being trained from becoming confused by the unnatural inputs $\alpha_t(f(X_i))$, we batch the adversarial examples with their corresponding natural crops $f(X_i)$.

Model	**GT Labels**		**Label Refinery**		**Adversarial**	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
AlexNet	57.93	79.41	66.28	86.13	**67.2**	**86.92**
MobileNet0.5	63.03	84.55	66.66	87.07	**67.33**	**87.4**
MobileNet0.25	50.65	74.42	54.62	77.92	**55.59**	**78.58**

Table 2.3: Smaller models are further improved by training over adversarial inputs. The Adversarial Label Refinery is ResNet-50.

2.4 Experiments

We evaluate the effect of label refining for a variety of network architectures on the standard ImageNet, ILSRVC2012 [23] classification challenge. We first explore the effect of label refining when the Label Refinery network architecture is identical to the architecture of the network being trained. We then evaluate the effect of label refining when the Label Refinery uses a more accurate network architecture. Finally, we present some ablation studies and analysis to investigate the source of the improvements. Note that all experiments are done with a single model over a single validation crop.

Implementation Details: All models are trained using PyTorch [104] on 4 GPUs for 200 epochs to ensure convergence. The learning rate is constant for the first 140 epochs. It is divided by 10 after epoch 140 and again divided by 10 after epoch 170. We use an initial learning rate of 0.01 to train AlexNet and an initial learning rate of 0.1 for all other networks. We use image cropping and horizontal flipping to augment the training set. When cropping, we follow the data augmentation practice of [127] in which the crop areas are chosen uniformly from 8% to 100% of the area of the image. We use a batch size of 256 for all models except the MobileNet variations, for which we use batch size of 512. Except for adversarial inputs experiments, we train models from refined labels starting from a random initialization. Our source code is available at `http://github.com/hessamb/label-refinery`.

Self-Refinement: We first explore using a Label Refinery to train another network with the same architecture. Table 2.1 shows the results for self-refinement on various architectures. Each row represents a randomly-initialized instance of the network architecture trained with labels refined by the model directly one row above it in the table. All six network architectures improve their accuracy through self-refinement. For AlexNet the self-refining process must be repeated 4 times before convergence, whereas MobileNet and ResNet-50 converge much faster. We argue that this is because AlexNet is more overfit to the training set. Therefore, it takes more training iterations to forget the information that it has memorized from training examples. One might argue that the accuracy improvements are due to

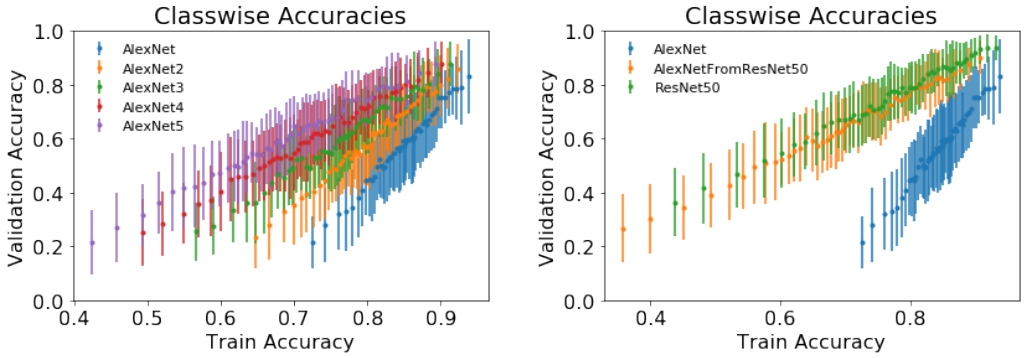
Model	Top-1	Top-5
AlexNet – no refinery	57.93	79.41
AlexNet – soft static refinery	63.55	84.16
AlexNet – hard dynamic refinery	64.41	84.53
AlexNet – soft dynamic refinery	66.28	86.13

Table 2.4: AlexNet benefits from both soft labeling and dynamic labeling. When combined the improvement is increased over both, suggesting that they capture different aspects of label errors. Label Refinery is ResNet-50.

the extended training time of models. However, we experimented with training models for an equal number of total epochs and the model accuracies did not improve further. This is discussed further in Section 2.4.1.

Cross-Architecture Refinement: The architecture of a Label Refinery network can be different from that of the trained network. A high-quality Label Refinery should not overfit on training data even if its validation accuracy is high. In other words, under the same validation accuracy, a network with lower training accuracy is a better Label Refinery. Intuitively, this property allows the refinery to generate high-quality crop labels that are reflective of the true content of the crops. This property prevents the refinery from simply predicting the training labels. We observe that a ResNet-50 model trained to 75.7% top-1 validation accuracy on ImageNet can serve as a high-quality refinery. Table 2.2 shows that a variety of network architectures benefit significantly from training with refined labels. All network architectures that we tried using Label Refineries gained significant accuracy improvement over their previous state-of-the-art. AlexNet and ResNetXnor-50[2] achieve more than a 7 point improvement in top-1 accuracy. Efficient and compact models such as MobileNet benefit significantly from cross-architecture refinement. VGG networks have a very high capacity and they overfit to the training set more than the other networks. Providing more

[2]ResNetXnor-50 is the XNOR-net [108] version of ResNet-50 in which layers are binary.



(a) (b)

Figure 2.4: Per category train and test accuracy. For each model, labels were sorted according to training set accuracies and divided into bins. each point in the plot shows the average validation set accuracy and the associated standard deviation for each bin. These figures show that training with a refinery results in models with less overfitting.

accurate training set labels helps them to fit to more accurate signals and perform better at validation time. Darknet19, the backbone architecture of YOLOv2 [111], improves almost 4 points when trained with refined labels.

Adversarial Inputs: As discussed in Section 2.3.1 we can adversarially augment our training set with patches on which the refinery network and the trained model disagree. We used a gradient step of $\eta = 1$, as defined in Equation 2.3 to augment the dataset. We batch each adversarially modified crop with the original crop during training. This helps to ensure the trained model does not drift too far from natural images. We observe in Table 2.3 that smaller models further improve beyond the improvements from using a Label Refinery alone.

2.4.1 Analysis

We explore the characteristics of models trained using a Label Refinery. We first explore how much of the improvement comes from the dynamic labeling of the image crops and how much of it comes from softening the target labels. We then explore the overfitting characteristics

Model	Top-1	Top-5
AlexNet – vanilla	57.93	79.41
AlexNet – taxonomy refined categories	56.73	77.69
AlexNet – visually refined categories	58.54	80.77
AlexNet – visually refined images	62.69	83.46

Table 2.5: Comparing refining labels at category level vs. image level. Note that "AlexNet – visually refined images" is trained over image level refined labels as opposed to crop level. For fairness, we fixed the batch normalization layers of label refinery (which harms the quality of label refinery) in all visually refined labels experiments. Label Refinery is ResNet-50.

of models trained with a Label Refinery. Finally, we explore using various loss functions to train models against the refined labels. Most of the analyses are performed on AlexNet architecture because it trains relatively fast ($\sim$ 1 day) on the ImageNet dataset.

Dynamic Labels vs. Soft Labels: The benefits of using a label refinery are twofold: (1) Each crop is dynamically re-labeled with more accurate labels for the crop (Figure 2.2), and (2) images are softly labeled according to the distribution of visually similar objects in the crop (Figure 2.3). We find that both aspects of the refinement process improve performance. To assess the improvement from dynamic labeling alone, we perform label refinement with hard dynamic labels. Specifically, we assign a one-hot label to each crop by passing the crop to the Label Refinery and choosing the most-likely category from the output. To observe the improvement from soft labeling alone, we perform label refinement with soft static labels. To compute these labels for a given crop, we pass a center crop of the original image to the refiner rather than using the training crop. We compare the results for soft static labels and hard dynamic labels in Table 2.4. Both dynamic labeling and soft labeling significantly improve the accuracy of AlexNet. When they are combined we observe an additional improvement, suggesting that they address different issues with labels in the

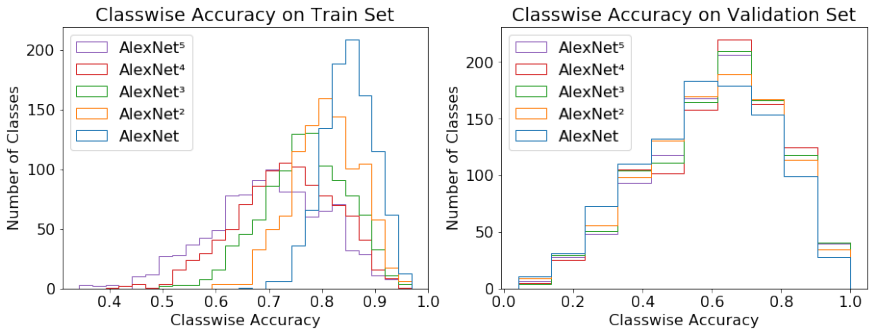


Figure 2.5: The train and validation accuracy distribution of AlexNet models trained sequentially. AlexNet is trained off of the ground-truth labels, and the successive models AlexNet^{i+1} are trained off of the labels generated by AlexNet^{i}.

Model	**Refinery**		**AlexNet**	
	Top-1	Top-5	Top-1	Top-5
AlexNet – vanilla	N/A	N/A	57.93	79.41
AlexNet – VGG16 refinery	70.1	88.54	60.78	81.80
AlexNet – MobileNet refinery	68.53	88.14	65.22	85.69
AlexNet – ResNet-50 refinery	75.7	92.81	66.28	86.13

Table 2.6: Different architecture choices for the refinery network.

dataset.

Category Level Refining vs. Image Level Refining: Labels can be refined at the category level. That is, all images in a class can be assigned a unique soft label that models intra-category similarities. At the category level, labels can be refined either by visual cues (based on the visual similarity between the categories) or by semantic relations (based on the taxonomic relationship between the categories). Since ImageNet categories are drawn from WordNet, we can use taxonomy-based distances to refine the labels. We experiment with using the Wu-Palmer similarity [147] of the WordNet [91] categories to refine the category

Model	Top-1	Top-5
AlexNet – vanilla	57.93	79.41
AlexNet – l_2 loss	63.16	85.56
AlexNet – KL from output to label	65.36	85.41
AlexNet – KL from label to output	66.28	86.13

Table 2.7: Different loss function choices. Label Refinery is ResNet-50.

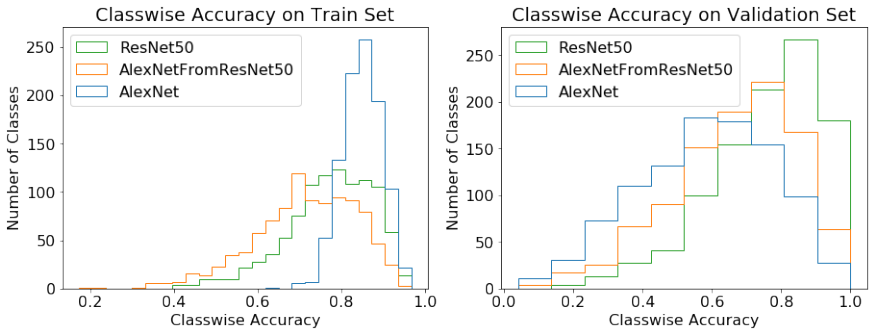


Figure 2.6: The train and validation accuracies for AlexNet, ResNet, and AlexNet trained off of labels generate by ResNet50. `AlexNetFromResNet50` has a train accuracy profile that more closely resembles ResNet50 than AlexNet.

labels. Table 2.5 compares refining labels at the category level with refining at the image level. We observe larger improvements when the labels are refined at the image level. Our experiment shows that taxonomy-based refinement does not improve training. We believe this is because WordNet similarities do not correlate well with visual similarities in the image space. Refining category labels based off of their WordNet distance can confuse the target model.

Model Generalization: Figure 2.4(a) and 2.4(b) show the per-category train and validation accuracies of ImageNet categories for models trained with a Label Refinery. Each point in the plot shows the average and standard deviation of the accuracies for a set of categories. Figure 2.4(a) shows the accuracies of a sequence of AlexNet models (in different

colors). AlexNet trained using the ground-truth labels has much higher train accuracy. Successive models demonstrate less overfitting as shown by the decrease in the ratio between train accuracy and validation accuracy. Figure 2.4(b) shows the per-category accuracies of AlexNet and ResNet-50, as well as an AlexNet model trained with a ResNet50 Label Refinery. ResNet-50 trained with weight decay generalizes better compared to AlexNet, which has two fully connected layers. Intuitively, the generalization of ResNet-50 enables it to generate accurate per-crop labels for the training set. Thus, training AlexNet with a ResNet-50 Label Refinery allows AlexNet to perform well on the test set without overfiting to the original ground-truth labels.

Figure 2.5 shows the training set and validation set accuracies of a sequence of AlexNet models trained with a Label Refinery. The AlexNet trained with ground-truth labels achieves $\sim 86\%$ training accuracy for the majority of classes, but achieves much lower validation set accuracies. By contrast, AlexNet^5 has a training accuracy profile more closely resembling its validation accuracy profile. Figure 2.6 shows a similar phenomena training AlexNet with a ResNet-50 refinery. It's interesting to note that the training and validation profiles of AlexNet trained with a ResNet50 Label Refinery more closely resemble the refinery than the original AlexNet.

Choice of Label Refinery Network: A good Label Refinery network should generate accurate labels for the *training set* crops. A Label Refinery's validation accuracy is an informative signal of its quality. However, if the Label Refinery network is heavily overfitted on the training set, it will not be helpful during training because it will produce the same ground-truth label for all image crops. Table 2.6 compares different architecture choices for refinery network. VGG16 is a worse choice of Label Refinery than MobileNet, even though VGG16 is more accurate. This is because VGG16 severely overfits to the training set and therefore produces labels too similar to the ground-truth.

Choice of Loss Function: We can use a variety of loss functions to train our target networks to match the soft labels. The KL-divergence loss function that we use is a generalization of the standard cross-entropy classification loss. Note that KL-divergence is not a

symmetric function (i.e. $D_{KL}(P||Q) \neq D_{KL}(Q||P)$). Table 2.7 shows the model accuracy if other standard loss functions are used.

2.5 Conclusion

In this paper we address shortcomings commonly found in the labels of supervised learning pipelines. We introduce a solution to refine the labels during training in order to improve the generalization and the accuracy of learning models. The proposed Label Refinery allows us to dynamically label augmented training crops with soft targets. Using a Label Refinery, we achieve a significant gain in the classification accuracy across a wide range of network architectures. Our experimental evaluation shows improvement in the state-of-the-art accuracy for popular architectures including AlexNet, VGG, ResNet, MobileNet, and XNOR-Net.

Chapter 3

LAYER-WISE DATA-FREE CNN COMPRESSION[1]

We present a computationally efficient method for compressing a trained neural network without using real data. We break the problem of data-free network compression into independent layer-wise compressions. We show how to efficiently generate layer-wise training data using only a pretrained network. We use this data to perform independent layer-wise compressions on the pretrained network. We also show how to precondition the network to improve the accuracy of our layer-wise compression method. We present results for layer-wise compression using quantization and pruning. When quantizing, we compress with higher accuracy than related works while using orders of magnitude less compute. When compressing MobileNetV2 and evaluating on ImageNet, our method outperforms existing methods for quantization at all bit-widths, achieving a +0.34% improvement in 8-bit quantization, and a stronger improvement at lower bit-widths (up to a +28.50% improvement at 5 bits). When pruning, we outperform baselines of a similar compute envelope, achieving 1.5 times the sparsity rate at the same accuracy. We also show how to combine our efficient method with high-compute generative methods to improve upon their results.

3.1 Introduction

The increasing popularity of Convolutional Neural Networks (CNNs) for visual recognition has driven the development of efficient networks capable of running on low-compute devices [54, 110, 107]. Domains such as smart-home security, factory automation, and mobile applications often require efficient networks capable of running on edge devices rather than running on expensive, high-latency cloud infrastructure. Quantization [61, 107] and pruning

[1]WORK COMPLETED AT APPLE INC.

[72, 139, 39, 30] are two of the main areas of active research focused on compressing CNNs for improved execution time and reduced memory usage.

Most methods for CNN compression require retraining on the original training set. Applying post-training quantization usually results in poor network accuracy [61] unless special care is taken in adjusting network weights [98]. Low-bit quantization provides additional challenges when data is not available. Popular methods for compressing through sparsity [72, 139, 39, 30] also require training on the original data. However, many real-world scenarios require compression of an existing model but prohibit access to the original dataset. For example, data may be legally sensitive or may have privacy restrictions [92]. Additionally, data may not be available if a model which has already been deployed needs to be compressed.

As CNNs move from cloud computing centers to edge devices, the need for efficient algorithms for compression has also arisen. The increasing popularity of federated learning [88] has emphasized the importance of efficient on-device machine learning algorithms. Additionally, models deployed to edge devices may need to be compressed on-the-fly to support certain use cases. For example, models running on edge devices in areas of low connectivity may need to compress themselves when battery power is low. Such edge devices usually do not have enough storage to hold a variety of models with different performance profiles [17]. To enable these use cases requires efficient data-free network compression.

We propose a simple and efficient method for data-free network compression. Our method is a layer-wise optimization based on the teacher-student paradigm [50]. A pretrained model is used as a "teacher" that will help train a compressed "student" model. During our layer-wise optimization, we generate data to approximate the input to a layer of the teacher network. We use this data to optimize the corresponding layer in the compressed student network. Figure 3.1 illustrates an overview of our method.

Our contributions are as follows. (1) We demonstrate that network compression can be broken into a series of layer-wise compression problems. (2) We develop a generic, computationally efficient algorithm for compressing a network using generated data. (3) We

demonstrate the efficacy of our algorithm when compressing with quantization and pruning. (4) We achieve higher accuracy in our compressed models than related works.

3.2 Related Work

Pruning and Quantization:

Quantization involves reducing the numerical precision of weights and/or activations from 32-bit floating point to a lower-bit integral representation [67, 38, 80, 32, 76, 145, 11, 61, 67, 151, 107, 19, 27, 86] to reduce size and improve execution time. The most commonly used quantization scheme is affine quantization, described in [61, 67]. Pruning involves the deletion of weights from a neural network to reduce size, and to improve execution time if specialized hardware is available. Many formulations exist [30, 74, 43, 25, 21, 78]. See [72] for a comprehensive overview.

Data-Light and Data-Free Compression: A few recent works have explored methods for quantizing a model using little data ("data-light" methods) or no data ("data-free" methods). In the data-light method AdaRound [96], the authors devise a method for optimizing the rounding choices made when quantizing a weight matrix. In the data-free method Data-Free Quantization [98], the authors manipulate network weights and biases to reduce the post-training quantization error. Other equalization formulations have been explored [89].

A few works have explored data-light and data-free pruning. In the data-light method PFA [124], activation correlations are used to prune filters. An iterative data-free pruning method is presented in [123], though they only prune fully-connected layers. Overparameterized networks are explored in [21] and [43]. Layer-wise approaches are explored in [28, 35].

Generative Methods for Data-Free Compression: Given a trained model, it's possible to create synthetic images that match characteristics of the training set's statistics. These images can be used to retrain a more efficient model. These methods are usually computationally expensive.

In the data-free method Deep Inversion (DI) [152], the authors use a pretrained model

Input: Pretrained network Θ to compress.

1: Copy Θ to obtain student Θ_s and teacher Θ_t.

2: Store BatchNorm statistics $\mu_{\mathcal{B}_i}$, $\sigma_{\mathcal{B}_i}$ for each BatchNorm layer $\mathcal{B}_i$ of Θ_s. Retain them for future use in the data generation function $\mathcal{G}_{\mathcal{C}_i}$ (Section 3.3.2).

3: Perform BatchNorm fusion (Section 3.3.1) on Θ_s and Θ_t.

4: Perform Assumption-Free Cross-Layer Equalization (Section 3.3.1) on Θ_s and Θ_t.

5: Perform layer-wise compression on Θ_s using the data generated by $\mathcal{G}_{\mathcal{C}_i}$ (see Section 3.3.3 for quantization, Section 3.3.4 for pruning).

6: return the compressed model Θ_s

Figure 3.1: An overview of our method. We use a layer-wise training scheme to individually compress each layer of a pretrained network Θ to produce a compressed network.

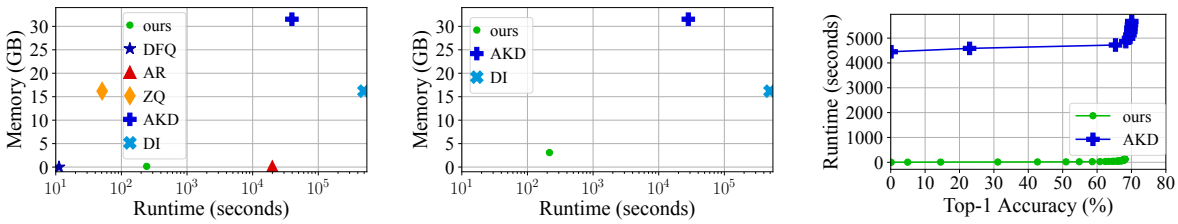
to generate a dataset which is then used to train a sparse model. A set of noise images are trained to match the network's BatchNorm [59] statistics, and to look realistic. The authors use these images to train a sparse model using Knowledge Distillation [50]. A similar data-light method appears in The Knowledge Within [41].

In Adversarial Knowledge Distillation [18], the authors generate synthetic images for model training using a GAN [34]. The images are used in conjunction with Knowledge Distillation [50] to train a network. Similar GAN-based formulations have been developed [149, 132, 47, 155, 90].

In ZeroQ [16], the authors tune noise images to match BatchNorm statistics of a network. Those images are then used to set quantizers. This method runs quickly, but requires substantial memory for backpropagation (Figure 3.2).

3.3 Layer-Wise Data-Free Compression

Our method for data-free network compression begins with a fully trained network and creates a compressed network of the same architecture. This is conceptually similar to



(a) (b) (c)

Figure 3.2: Runtime and memory overhead for data-free quantization and pruning of MobileNetV1 on an NVidia Tesla V100. Note DFQ [98], AR [96], and ZQ [16] are quantization-only methods. (a) Quantization to 8 bits. (b) Pruning. (c) ImageNet validation accuracy as a function of training time when pruning. DI [152] requires orders of magnitude more computation than AKD [18] (data generation alone takes 492,000 seconds) so we omit it for clarity.

Knowledge Distillation [50] in that a pretrained "teacher" network is used to train a "student" network. However, Knowledge Distillation requires training data. Previous approaches have addressed this through generating data [18, 152], but these methods are computationally expensive (Figure 3.2).

We take a simpler approach illustrated in Figure 3.1. We view each layer of the student as a compressed approximation of the corresponding layer in the teacher. As long as the approximation in each layer is accurate, the overall student network will produce the same outputs as the teacher. Because our data generation doesn't require generating realistic images (unlike AKD [18] and DI [152]), our method takes less time and memory than these approaches (Figure 3.2). Because our method trains layers separately, it doesn't require large gradient buffers needed for backpropagating through the whole neural network, making it more memory efficient than ZeroQ [16], AKD, and DI. This allows our method to run efficiently on edge devices with limited memory, supporting use cases for on-the-fly compression without deploying extra model weights. Our method also achieves higher accuracy than other low-memory methods, such as AR [96] and DFQ [98] (Section 3.4).

Our method is described in detail in the following subsections. We begin by performing fusion and Assumption-Free Cross-Layer Equalization (Section 3.3.1). Then, we perform layer-wise compression using generated data. We discuss data generation in Section 3.3.2. Our layer-wise compression method for quantization is discussed in Section 3.3.3, and our method for pruning is discussed in Section 3.3.4.

3.3.1 Fusion and Equalization

Two issues complicate the matter of assembling a compressed network from compressed individual layers. For simplicity, we describe these issues in the case of a fully connected layer, but the extension to convolutions is straightforward.

The first issue relates to the BatchNorm [59] layers commonly used in CNNs. A BatchNorm layer consists of the parameters μ, σ, γ, and β, which correspond to the mean of its inputs, the standard deviation of its inputs, the weight of its affine transformation, and the bias of its affine transformation. Consider a linear layer W with bias b, which is followed by a BatchNorm layer. The output of the linear layer, followed by the BatchNorm layer, is

$$f(x) = \frac{Wx + b - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta, \tag{3.1}$$

where ϵ is a small number used to avoid division by 0, and $\odot$ denotes elementwise multiplication [59, 104].

Note that the function $f(x)$ is overparameterized. There are multiple different sets of parameters that yield identical functions $f(x)$. For example, consider the effect of multiplying the c^{th} row of W, b, and μ by some nonzero scalar a, and multiplying the c^{th} row of γ by $1/a$. The adjustments to W, b, μ, and γ will cancel out, and $f(x)$ will remain unchanged.

Thus, the magnitude of rows of W can be rescaled if the corresponding BatchNorm elements are rescaled. This is problematic when pruning or quantizing W because we expect the weight values' magnitudes to reflect their importance [139, 72]. This coupling of the values of W with BatchNorm parameters prevents this. To address this issue, we fuse the BatchNorm parameters μ, σ, γ, and β into the preceding convolution [85], so that the

BatchNorm parameters' effective influence on weight magnitudes is accounted for.

The second complication with breaking data-free network compression into layer-wise compression subproblems is that the relative magnitude of weights are not calibrated within a layer. Consider the output of a pair of layers with weights W_1 and W_2, and biases b_1 and b_2. Suppose the network uses ReLU activations [4], such that the output of the pair of layers is

$$f(x) = \text{ReLU}(W_1(\text{ReLU}(W_2x + b_2)) + b_1). \tag{3.2}$$

This function $f(x)$ is overparameterized. If the c^{th} row of W_2 is multiplied by a scale factor a, and if the c^{th} element of b_2 is multiplied by a, and if the c^{th} column of W_1 is multiplied by $1/a$, then the function $f(x)$ remains unchanged. In this rescaled network, the c^{th} row of W_2 has changed, but no other rows in W_2 have changed. Thus, the weights' magnitudes do not necessarily reflect their importance to the network, which is problematic for pruning and quantization [139, 72].

To address this scaling inconsistency we employ a method we call Assumption-Free Cross-Layer Equalization (AFCLE). Our method is inspired by Cross-Layer Equalization [98]. With each layer $\mathcal{L}_j$ with weight $W_j \in \mathbb{R}^{c_o \times c_i}$ and bias $b_j \in \mathbb{R}^{c_o}$, we associate a pair of vectors, $v_j^i \in \mathbb{R}^{c_i}$ and $v_j^o \in \mathbb{R}^{c_o}$. We now compute our layer's output as

$$\mathcal{L}_j(x) = (W_j(x \odot v_j^i) + b_j) \odot v_j^o. \tag{3.3}$$

Each element of the vectors v_j^i and v_j^o is initialized to 1.

AFCLE progresses iteratively across the network. In each iteration, we consider a channel in a pair of adjacent network layers. Consider the c^{th} row of W_2 with weights W_2^c, and the corresponding c^{th} column of W_1 with weights W_1^c. Let $[\cdot]_c$ denote the c^{th} element of a vector.

We rescale the network as

$$s_c = \frac{\sqrt{\max(|W_1^c|)\max(|W_2^c|)}}{\max(|W_2^c|)} \tag{3.4}$$

$$W_1^c \leftarrow W_1^c / s_c \tag{3.5}$$

$$[v_1^o]_c \leftarrow [v_1^o]_c * s_c \tag{3.6}$$

$$[b_1]_c \leftarrow [b_1]_c / s_c \tag{3.7}$$

$$W_2^c \leftarrow W_2^c * s_c \tag{3.8}$$

$$[v_2^i]_c \leftarrow [v_2^i]_c / s_c. \tag{3.9}$$

Essentially, each v_j^i and v_j^o act as a buffer that reverses the changes to W_j and b_j, so that we can equalize W_j across layers without changing the network's outputs at any layer. We iterate over all channels and over all pairs of adjacent layers in the network. We continue until the mean of all the scale parameters s_c for one round of equalization deviates from 1 by less than 10^{-3}, since weight updates are negligibly small at this point. Note that AFCLE results in no real increase in parameter count, since the vectors v_j^i and v_j^o can be folded into W_j and b_j after data-free compression is completed.

Our method is inspired by Cross-Layer Equalization (CLE) [98]. The main difference between AFCLE and CLE is that AFCLE method uses v_j^i and v_j^o to record weight updates, whereas CLE assumes that the updates to W_1^c and W_2^c do not alter the network. CLE's assumption holds if the network's activation functions are piecewise linear (e.g. ReLU). When experimenting with networks with only ReLU activations, we simply drop the v_j^i and v_j^o buffers, since they are not needed.

3.3.2 Layer-Wise Data Generation

We now describe our method for generating data used by our layer-wise optimization algorithms. The details of how this generated data is used for quantization and pruning are described in Sections 3.3.3 and 3.3.4.

Consider a network composed of blocks containing a convolution, a BatchNorm [59], and

an activation. Let $\mathcal{B}_i$ denote the BatchNorm layer associated with a block of index i. It has stored parameters $\mu_{\mathcal{B}_i}$, $\sigma_{\mathcal{B}_i}$, $\gamma_{\mathcal{B}_i}$, and $\beta_{\mathcal{B}_i}$ corresponding to the input mean, input standard deviation, scale factor, and bias. Because the BatchNorm layer normalizes its inputs before applying an affine transformation, the mean of its outputs is $\beta_{\mathcal{B}_i}$ and the standard deviation of its outputs is $\gamma_{\mathcal{B}_i}$.

We exploit this information to generate layer-wise inputs. Let $\mathcal{C}_i$ denote the convolutional layer in block i of the network, and f_i denote the activation function.

Consider the case in which block i accepts multiple input tensors from blocks indexed by elements $j \in \mathcal{K}$. Let $x_{\mathcal{B}_j}$ denote the input into the BatchNorm $\mathcal{B}_j$ from a training batch (when training with real data). Assuming the inputs to block i are combined by an addition function, the input $x_{\mathcal{C}_i}$ to convolution $\mathcal{C}_i$ is

$$x_{\mathcal{C}_i} = \sum_{j \in \mathcal{K}} f_j(\mathcal{B}_j(x_{\mathcal{B}_j})). \tag{3.10}$$

During data-free compression, we do not have access to $x_{\mathcal{B}_j}$, so we estimate it. Let $\mathcal{G}_{\mathcal{C}_i}$ be a function that generates an input used to train layer $\mathcal{C}_i$. Using our observation above regarding the output statistics of BatchNorms, we estimate

$$x_{\mathcal{B}_j} \sim \mathcal{N}(\beta_{\mathcal{B}_j}, \gamma_{\mathcal{B}_j}) \tag{3.11}$$

$$\mathcal{G}_{\mathcal{C}_i} = \sum_{j \in \mathcal{K}} f_j(x_{\mathcal{B}_j}), \tag{3.12}$$

where $\mathcal{N}(a, b)$ denotes a Gaussian function with mean a and standard deviation b. If a layer is not preceded by a BatchNorm, we generate data from $\mathcal{N}(0, 1)$ (as in the case of the network's first layer, or if BatchNorms aren't present before the convolution). We ignore the effect of other layers (such as Average Pooling). Note that BatchNorm statistics need to be gathered before BatchNorm fusion (Section 3.3.1) because the parameters are modified during fusion.

Input: BatchNorm parameters $\beta_{\mathcal{B}_j}, \gamma_{\mathcal{B}_j}$, input activation functions f_j, number of steps N, quantization bits b.

$\mathcal{G}_{\mathcal{C}_i} = \sum_{j \in \mathcal{K}} f_j(\mathcal{N}(\beta_{\mathcal{B}_j}, \gamma_{\mathcal{B}_j}))$ (Equation 3.12)
$X \sim \mathcal{G}_{\mathcal{C}_i}$
$x_{\max} \leftarrow \max(X)$, $x_{\min} \leftarrow \min(X)$
$h \leftarrow -\infty$, $l \leftarrow \infty$, $L \leftarrow \infty$
for $h_i \in [1, 2, ..., N]$ **do**
  for $l_i \in [1, 2, ..., N]$ **do**
    $\tilde{h} = (h_i/N)(\max(x_{\max}, 0))$
    $\tilde{l} = (l_i/N)(\min(x_{\min}, 0))$
    if $||X - Q(X, \tilde{l}, \tilde{h}, b)||_2 < L$ **then**
      $l \leftarrow \tilde{l}$, $h \leftarrow \tilde{h}$
      $L \leftarrow ||X - Q(X, \tilde{l}, \tilde{h}, b)||_2$
    end if
  end for
end for
return l, h

Figure 3.3: Computing the high end h and the low end l of the activation range for our activation quantizers.

3.3.3 Data-Free Quantization

We now describe how to use our generated data to quantize a neural network. Our method quantizes both weights and activation ranges. As is standard, we set our weight quantizers to the minimum and maximum of the weight tensors [61].

We set activation ranges by performing a simple optimization (Figure 3.3). Recall that the quantized version of a floating-point activation tensor X can be expressed as $Q(X, l, h, b)$:

$$I(X, l, h, b) = \left\lfloor \frac{\min(\max(X, l), h) - l}{(h - l)/(2^b - 1)} \right\rceil \tag{3.13}$$

$$Q(X, l, h, b) = \frac{h - l}{2^b - 1} I(X, l, h, b) + l, \tag{3.14}$$

where l is the minimum of the quantization range, h is the maximum, b is the number of bits in the quantization scheme, and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. We perform a grid search jointly over $h \in [0, \max(X_{\max}, 0]]$ and $l \in [\min(0, X_{\min}), 0]$ to minimize $|X - Q(X, l, h, b)|_2$.

Our overall method for data-free quantization follows Figure 3.1. First, we perform BatchNorm fusion and AFCLE on the student network (Section 3.3.1). Then, we perform bias absorption, as in [98]. Next, we set activation quantizers (Figure 3.3). Then, we perform bias correction, as in [98]. We then set activation quantizers again (Figure 3.3), because the bias correction step adjusted weights and output statistics slightly.

3.3.4 Data-Free Pruning

We now describe how to use our generated data (Section 3.3.2) to prune neural network weights. Our overall method follows Figure 3.1. We begin by duplicating our pretrained network Θ to obtain a teacher Θ_t and a student Θ_s. We perform fusion and AFCLE as a preprocessing step (Section 3.3.1). For the remainder of our method, the teacher will remain unchanged, and the student will be pruned.

We prune using Soft Threshold Reparameterization (STR) [72] with gradient descent [63]. In convolutional and fully-connected layers, an intermediate weight W_s is calculated as

$$W_s = \text{sign}(W) \cdot \text{ReLU}(|W| - \text{sigmoid}(s)), \tag{3.15}$$

where W is the weight tensor, and s is a model parameter used to control sparsity. This intermediate tensor W_s is used in place of W in the forward pass. In the backward pass, the gradients are propagated to the original weight matrix W. A weight decay parameter λ is

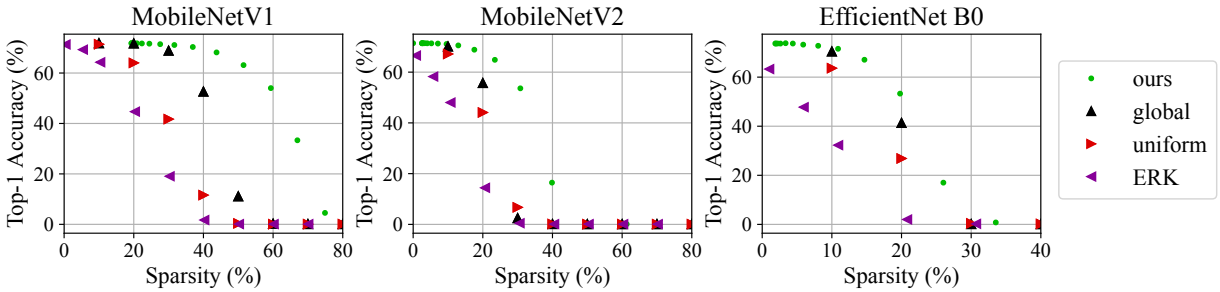


Figure 3.4: ImageNet results comparing efficient pruning methods. Our method produces the strongest sparsity/accuracy tradeoff.

used to drive s upwards from an initial value $s_0 < 0$, which increases sparsity. See [72] for further details.

We prune each layer of Θ_s separately. Let i be an index assigned to the convolutional or fully-connected layer $\mathcal{C}_i^s$ in the student, and let the corresponding layer in the teacher be $\mathcal{C}_i^t$. Let $\mathcal{C}_i^s(\cdot)$ denote the application of a layer to an input tensor. Given a loss function L_i associated with layer i, we compute the loss for layer i as

$$L = L_i(\mathcal{C}_i^s(x), \mathcal{C}_i^t(x)), \tag{3.16}$$

where $x \sim \mathcal{G}_{\mathcal{C}_i^t}$ is generated from Equation 3.12. In our experiments, we choose L_i to be the mean square error loss for each i. We also freeze the student's bias during training, since our goal is to induce sparsity only in the weights. We empirically found that including the activations f_j in Equation 3.12 is not important during pruning, so we omit them for simplicity.

3.4 Experiments

We provide results for data-free quantization and pruning. We separately discuss efficient methods and expensive methods to clarify which results are obtainable with on-device compression in the low-compute scenario. We evaluate on ImageNet [22]. We train in PyTorch

Table 3.1: ImageNet results comparing quantization methods. (†) denotes relatively memory-intensive methods (see Figure 3.2).

Bits	ours	DFQ	AR	ZQ†	AKD†	DI†	ours	DFQ	AR	ZQ†	AKD†	DI†
	MNV1 1.0						MNV2					
8	71.13	71.18	**71.35**	64.36	70.13	24.53	**70.50**	70.16	68.61	62.03	19.33	7.21
7	**70.26**	69.76	70.22	54.31	67.77	21.35	**69.67**	68.89	66.65	45.42	4.87	5.43
6	**66.95**	61.81	63.68	28.23	7.11	19.17	**66.97**	63.06	51.00	5.16	9.70	3.54
5	**55.37**	25.63	29.17	0.51	0.10	9.06	**56.33**	23.41	27.83	1.98	0.86	2.62
4	**5.62**	0.23	0.28	0.09	0.10	0.10	**8.23**	0.49	0.48	0.07	0.10	0.09
	MNV1 0.5						RN18					
8	**63.77**	63.65	56.39	46.74	62.83	25.75	**68.67**	67.72	56.24	43.69	62.13	19.69
7	**61.64**	60.47	51.01	39.26	57.20	19.05	**67.92**	66.40	54.45	39.65	61.38	19.05
6	**55.51**	44.67	34.44	7.31	48.00	22.17	**63.72**	62.86	50.57	41.12	54.96	16.98
5	**23.79**	7.29	5.48	0.23	16.86	6.70	35.47	34.76	**43.64**	12.74	41.41	12.43
4	**1.71**	0.24	0.28	0.09	0.10	0.10	0.88	1.06	**10.64**	0.14	0.10	6.33

[104] using NVIDIA Tesla V100 GPUs. When training with backpropagation, we use Adam [65] with a cosine learning rate decaying from 0.001 to 0 over 10^5 iterations (though in practice, our method converges in only a few hundred iterations). We use batch size 128 for all methods, reducing it to fit on a single GPU as needed.

In all MobileNetV2 [115] experiments, we replace ReLU6 with ReLU [4] as in DFQ [98]. The accuracy of this modified network is unchanged. For all experiments, we place activation quantizers after skip connections, and we quantize per-tensor (not per-channel) as in Equation 3.14. This differs from previous works [16], so our quantization baselines differ slightly from results reported in original works.

3.4.1 Data-Free Quantization

Efficient Quantization: In Table 3.1, we compare our results with methods with a similar compute envelope. When generating data for quantizing activations (Figure 3.3), we use a batch size of 2000, and optimize for $N = 100$ steps. We evaluate MobileNetV1 [54], MobileNetV2 [115], and ResNet18 [45].

We compare to DFQ [98]. DFQ performs preprocessing similar to our method, but simply sets activation quantizers to be 6 standard deviations from the mean (using BatchNorm statistics). We also compare to AdaRound (AR) [96], which optimizes over rounding decisions. AR requires training data, but we use our generated data for a fair comparison.

Note that, since our method and DFQ don't require backpropagation, we simply report the accuracy. For AR, which requires backpropagation, we report the final-epoch accuracy. We do this since a truly data-free scenario would not allow for a validation set on which to perform early stopping. Our method outperforms DFQ and AR in nearly all cases.

Expensive Quantization: We evaluate more memory-intensive methods ZeroQ (ZQ) [16], Adversarial Knowledge Distillation (AKD) [18], and Deep Inversion (DI) [152] in Table 3.1. As before, we report final-epoch accuracies. Our method outperforms these generative methods using orders of magnitude less memory usage (Figure 3.2) in almost all cases. DI had difficulty providing an effective training signal to the quantized student, and failed to produce accuracies above 25%, even at 8 bits. AKD failed to produce good results on MobileNets at 6 bits and below, whereas our method produced strong results across all networks. Note that AKD requires training a GAN, which may require parameter tuning for different network architectures or datasets. Such tuning is not possible in a data-free setup, so AKD is unlikely to produce results that transfer well across architectures and datasets.

3.4.2 Data-Free Pruning

Efficient Pruning: We present results for our pruning method in Figure 3.4. We fix the sparsity-inducing weight decay parameter to $\lambda = 1.55 \times 10^{-5}$, as in [72]. We investigate

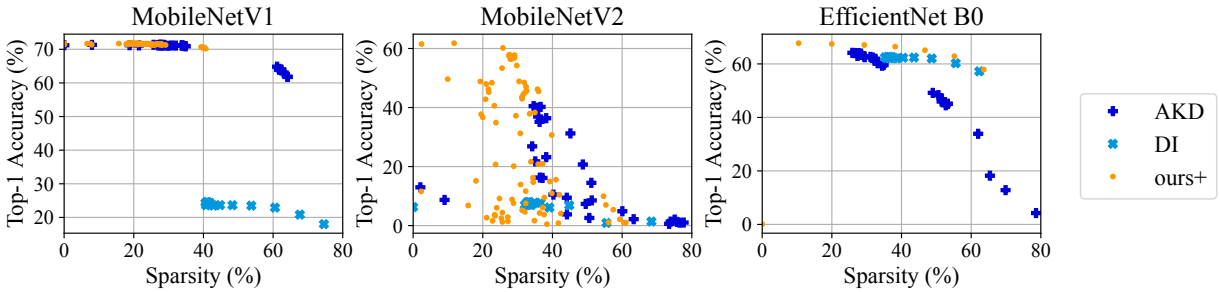


Figure 3.5: Comparison of computationally expensive pruning methods on ImageNet.

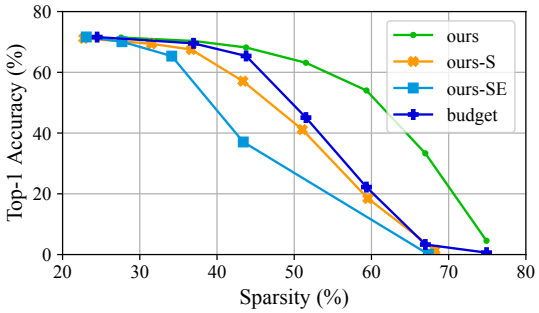


Figure 3.6: Ablation study of pruning MobileNetV1 on ImageNet.

MobileNets as before. We also investigate EfficientNet [131] to include results for networks with non-ReLU activations. (We did not investigate EfficientNet for quantization because several baseline methods required only ReLU activations.)

Because our efficient quantization baselines DFQ [98] and AR [96] do not support pruning, we compare to a different set of efficient baselines. The "global" baseline corresponds to pruning every weight whose magnitude is smaller than a given threshold [12]. The "uniform" baseline corresponds to applying a uniform sparsity level to each layer by pruning the weights with smallest magnitude [12]. The Erdosh-Renyi Kernel (ERK) baseline corresponds to a budgeted layer-wise pruning [30]. Each layer's fraction of pruned weights is

$$p = \frac{c_o + c_i + k_h + k_w}{c_o c_i k_h k_w}, \tag{3.17}$$

where c_o is the number of output dimensions, c_i is the number of input dimensions, and k_h, k_w are the kernel height and width. For these baselines, we perform fusion [85] because it is a standard technique, but we do not perform AFCLE.

Our method outperforms these baselines for efficient pruning, producing a better sparsity/accuracy tradeoff. In Figure 3.6, we present an ablation. "Ours-S" represents generating data from $\mathcal{N}(0, 1)$, with no other changes to our method. "Ours-SE" represents generating data from $\mathcal{N}(0, 1)$ and skipping AFCLE. "Budget" represents a model that uses the layer-wise pruning budget learned by our method (without retraining weights). We find that generating data from $\mathcal{N}(0, 1)$ and skipping AFCLE both reduce performance substantially.

Expensive Pruning: We compare our method to more computationally expensive methods in Figure 3.5. We compare to Adversarial Knowledge Distillation (AKD) [18] and Deep Inversion (DI) [152] (note that ZeroQ [16] does not support pruning). These methods involve generating end-to-end training data. We found we needed to sweep across more values of the sparsity-controlling weight decay parameter λ to encourage varying levels of sparsity in AKD. In addition to $\lambda = 1.55 * 10^{-5}$, we used $\{2\lambda, 10\lambda, 100\lambda\}$. We report final-epoch accuracies.

Note that, because STR controls the sparsity level implicitly (not explicitly), it does not give direct control over the final sparsity levels. This is why some methods never converge to lower-sparsity, higher-accuracy solutions (for example, DI for MobileNetV1). For MobileNets, DI failed to produce solutions with more than 30% accuracy, which has limited practical value. For AKD, performance was stronger in general, although solutions for MobileNetV2 did not exceed 40% accuracy. Some parameter settings for MobileNetV2 resulted in very poor performance.

For each network, we combined our method with the top-performing baseline method for that network ("ours+", Figure 3.5). To do this, we add our layer-wise loss (Equation 3.16) to the baseline's loss function and backpropagate as usual (we omit fusion and AFCLE for simplicity in this case). For MobileNetV1, we combine our method with AKD, producing a slight increase in accuracy for moderately sparse models. For MobileNetV2, we combine

our method with AKD to produce high-accuracy solutions that cannot be obtained by AKD or DI alone. For EfficientNet B0, we combine our method with DI and improve the overall sparsity/accuracy tradeoff.

Although these methods achieve a stronger sparsity/accuracy tradeoff than efficient pruning methods, getting these methods to work may be difficult in practice without data. AKD and DI use many more parameters than the efficient methods. These parameters may need more tuning to work on data from another domain or for other architectures. Moreover, many trials of AKD and DI converged to low accuracy, which isn't detectable without data.

3.5 Conclusion

We present an efficient, effective method for data-free compression. We break this problem into the subproblem of compressing individual layers without data. We precondition networks to better maintain accuracy during compression. Then, we compress individual layers using data generated from BatchNorm [59] statistics from previous layers. Our method outperforms baselines on quantization. Our method outperforms other computationally efficient methods for pruning, and can be combined with computationally expensive methods to improve their results.

Chapter 4

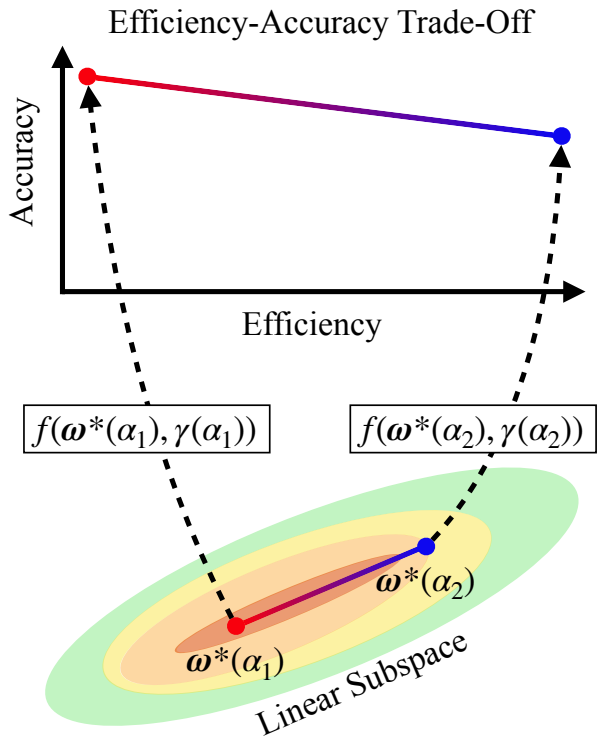
LEARNING COMPRESSIBLE SUBSPACES[1]

When deploying deep learning models to a device, it is traditionally assumed that available computational resources (compute, memory, and power) remain static. However, real-world computing systems do not always provide stable resource guarantees. Computational resources need to be conserved when load from other processes is high, or available memory is low. Inspired by recent works on neural network subspaces, we propose a method for training a *compressible subspace* of neural networks that contains a fine-grained spectrum of models that range from highly efficient to highly accurate. Our subspace model requires no retraining when changing compression levels. Thus, our subspace model can be deployed on-device to allow for adaptive network compression at inference time. We present results for our method for unstructured sparsity, structured sparsity, and quantization on a variety of architectures. We present models that require a single extra copy of network parameters, as well as models that require no extra parameters. Both models allow for operation at any compression level within a wide range (for example, 0% to 90% for structured sparsity with ResNet18 on ImageNet). At each compression level, our models achieve an accuracy comparable to a baseline model optimized for that particular compression level.

4.1 Introduction

Deep neural network models are deployed to a variety of computing platforms, including phones, tablets, and watches [26]. These models are generally designed to consume a fixed budget of resources, but the compute resources available on a device can vary over time. Computational burden from other processes, as well as battery life, may influence the

[1]WORK COMPLETED AT APPLE INC.



(a)

Learning Compressible Subspaces

Input: Network subspace $\boldsymbol{\omega}^*(\alpha)$, compression level $\gamma(\alpha)$, compression function $f(\boldsymbol{\omega}, \gamma)$, stochastic sampler $\boldsymbol{\alpha}_n \in [\alpha_1, \alpha_2]^n$, dataset $\mathcal{D}$, loss $\mathcal{L}$.

Replace BN layers with GN.

while $\boldsymbol{\omega}^*$ has not converged **do**

  for batch $\mathcal{B}$ in $\mathcal{D}$ **do**

    Sample a vector $\boldsymbol{\alpha} \sim \boldsymbol{\alpha}_n$

    $l \leftarrow 0$

    for $\alpha \in \boldsymbol{\alpha}$ **do**

      # compute loss on batch

      $l \mathrel{+}= \mathcal{L}(f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha)), \mathcal{B})$

    end for

    backpropagate loss l

    apply gradient update to $\boldsymbol{\omega}^*$

  end for

end while

return $\boldsymbol{\omega}^*$

(b)

Figure 4.1: (a) Depiction of our method for learning a linear subspace of networks $\boldsymbol{\omega}^*$ parameterized by $\alpha \in [\alpha_1, \alpha_2]$. When compressing with compression function f and compression level γ, we obtain a spectrum of networks which demonstrate an efficiency-accuracy trade-off. (b) Our algorithm

availability of resources to a model. Adaptively adjusting inference-time load is beyond the capabilities of traditional neural networks, which are designed with a fixed architecture and a fixed resource usage.

A simple approach to the problem of providing an accuracy-efficiency trade-off is to train multiple neural networks of different sizes. Multiple networks are stored on the device and loaded into memory when needed. There is a breadth of research in the design of efficient architectures that can be trained with different capacities, then deployed on a device [56, 116, 55]. However, there are a few drawbacks to using multiple networks to provide an accuracy-efficiency trade-off: (1) it requires training and deploying multiple networks (which induces training-time computational burden and on-device storage burden), (2) it requires all compression levels to be specified before deployment, and (3) it requires new networks to be loaded into memory when changing the compression level, which prohibits real-time model switching on memory-constrained edge devices.

Previous methods such as Network Slimming [154] and Universal Slimming [153] address the first issue in the setting of structured sparsity by training a single network conditioned to perform well when varying the number of channels pruned. However, these methods require BatchNorm [60] statistics to be recalibrated for every accuracy-efficiency configuration before deployment. This requires users to know every compression level in advance. If a large number of compression levels are chosen, the storage burden of BatchNorm statistics is significant (Figure 4.2), especially for low-compute devices. If only a few compression levels are chosen, a user will have to sacrifice accuracy and underutilize available resources by choosing a smaller model if the desired compression level is not available. This situation is exacerbated when multiple models are running on-device. Consider an intelligent system dependent on the output of a large number of separate models. Reducing resource usage when few compression levels are available will require strongly compressing a few models, which may strongly degrade overall performance. If a large number of compression levels are available, the user can instead slightly reduce the size of each model. In other words, providing more compression levels allows the user to finely control the resource distribution

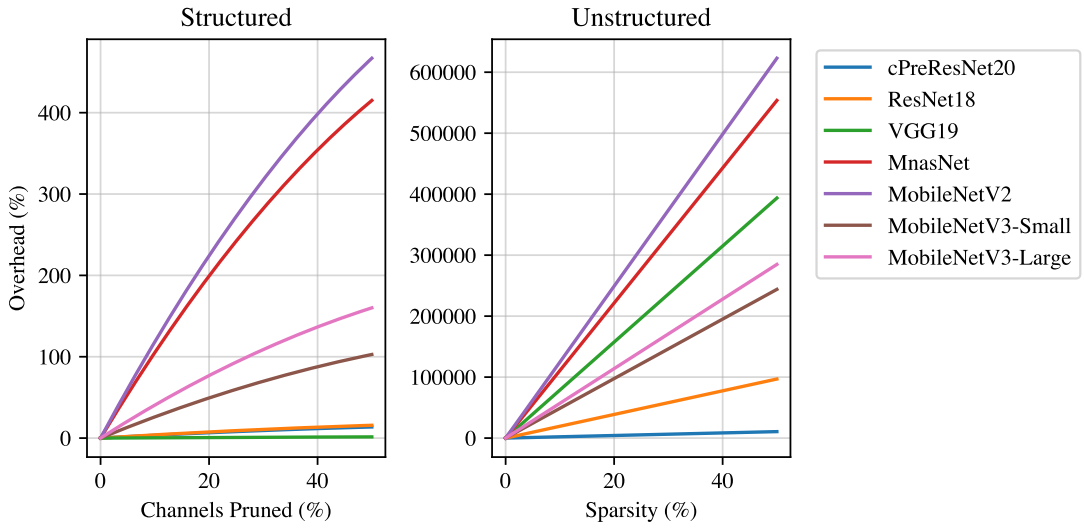


Figure 4.2: Parameter overhead for storing an extra set of pre-calibrated BatchNorm statistics for every possible sparsity configuration between 0% sparsity and the given compression level. Our method avoids this overhead by eliminating the need for storing BatchNorm statistics (Section 4.3.6). See Appendix B.1 for more details

among models.

We address the problem of deploying a model that can be compressed after deployment. Inspired by Wortsman et al. [141], we formulate this problem as computing an adaptively compressible network subspace (Figure 4.1(a)). Our solution is **efficient**, meaning we provide models that incur no parameter overhead on-device compared to a single model of the same architecture (though we also provide results for models that do incur parameter overhead). Our solution is **adaptive**, meaning our model can run at any compression level after deployment (rather than a predefined set of compression levels). Our solution is **real-time**, meaning we can adjust compression levels at inference time at negligible computational cost.

Contributions: Our contributions are as follows. (1) We introduce our method, Learning Compressible Subspaces (LCS), for training models capable of efficient, adaptive, real-time compression after deployment. To our knowledge this has not been done previously.

Table 4.1: Our method with a linear subspace (LCS+L) and a point subspace (LCS+P), LEC [82], NS [154], and US [153]. Note that "Adaptive" refers to post-deployment compression at any compression level. $|\boldsymbol{\omega}|$ denotes the number of network parameters, $|b|$ denotes the number of BatchNorm parameters, and n denotes the number of compression levels for models that don't support arbitrary compression levels

	LCS+P	LCS+L	LEC	NS	US
No Retraining	✔	✔	✘	✔	✔
No Norm Recalibration	✔	✔	✘	✘	✘
Adaptive	✔	✔	✘	✘	✘
Stored Parameters	$\|\boldsymbol{\omega}\|$	$2\|\boldsymbol{\omega}\|$	$n\|\boldsymbol{\omega}\|$	$\|\boldsymbol{\omega}\| + n\|b\|$	$\|\boldsymbol{\omega}\| + n\|b\|$

(2) We demonstrate that neural network subspaces can be used to encode models that specialize at one end for high-accuracy and at the other end for high-efficiency. (3) We provide an empirical evaluation of our method using unstructured sparsity, structured sparsity, and quantization. (4) We open source our code at `https://withheld.for.review` for research purposes.

4.2 Related Work

Architectures Demonstrating Efficiency-Accuracy Trade-Offs: Many popular network architectures include a hyperparameter to control the number of filters in each layer [56, 116, 55] or the number of blocks in the network [130]. In Once For All [14], the need for individually training separate networks is circumvented. Instead, a single large network is trained, then queried for subnetworks.

We differ from these methods in two ways. First, we compress on-device adaptively (without specifying the compression levels before deployment). Previous works require training

separate networks (or in the case of Once For All, querying a larger model for a compressed network) before deployment. Second, we don't require deploying a set of weights for every compression level.

A method for post-training quantization to variable bit widths appears in [119], but it relies on calibration data and cannot be run on low-compute edge devices. Thus, compression levels must be specified before deployment. They also present a method for quantization-aware training to variable bit widths, but most of their results keep activation bit widths fixed, whereas we vary it.

Training-Aware Compression: Recent works train a single neural network which can be configured at inference time to execute at different compression levels. These methods are the closest to our work. In Learning Efficient Convolutions (LEC) [82], the authors train a single network, then fine-tune it at different structured sparsity rates. Other methods train a single set of weights conditioned to perform well when channels are pruned, but require recalibration (or preemptive storage) of BatchNorm [60] statistics at each sparsity level. These methods include Network Slimming (NS) [154] and Universal Slimming (US) [153]. Similar methods train a single network to perform well at various levels of quantization by storing extra copies of BatchNorm statistics [36].

We differ from these methods by avoiding the need to recalibrate or store BatchNorm statistics (Section 4.3.6) and by allowing for adaptive selection of any compression level at inference time, neither of which have been done before (Table 4.1). Additionally, we avoid the overhead of storing BatchNorms at every desired compression level. Figure 4.2 demonstrates the substantial overhead of storing BatchNorm parameters for every possible compression level. Note that [154] avoids this storage overhead by only storing a few sets of BatchNorm statistics. However, this has the drawback of only allowing a few efficiency-accuracy configurations, which is problematic as explained in Section 4.1.

Note that in the case of quantization, it's feasible to store BatchNorm statistics for every compression level, since there are a small, discrete number of compression levels to choose from (e.g. different bit widths). We include a few preliminary quantization experiments to

show that our general approach applies to this compression method, but we point out that avoiding BatchNorm storage costs is not essential in this case. Note that our method is broadly applicable to a variety of compression methods, whereas previous works all focus on a single compression method.

Other Post-Training Compression Methods: Other works have investigated post-training compression. In [99], a method is presented for compressing 32 bit models to 8 bits, though it has not been evaluated in the low-bit regime. It involves running an equalization step and assuming a Conv-BatchNorm-ReLU network structure. A related post-training compression method is shown in [53], which shows results for both quantization and sparsity. However, their sparsity method requires a lightweight training phase. We differ from these methods in providing real-time post-deployment compression for both sparsity and quantization without making assumptions about the network structure.

Neural Network Subspaces: The idea of learning a neural network subspace is introduced in [141] (though another formulation was introduced concurrently in [10]). Multiple sets of network weights are treated as the corners of a simplex, and an optimization procedure updates these corners to find a region in weight space in which points inside the simplex correspond to accurate networks. This approach is shown to produce models with improved accuracy and calibration.

4.3 Compressible Subspaces

4.3.1 Compressible Lines

Our method involves training a neural network subspace [141] that contains a spectrum of networks that each have a different efficiency-accuracy trade-off. We recast the subspace formulation of [141] to train a linear subspace with high-efficiency solutions at one end and high-accuracy solutions at the other end.

To learn a compressible subspace, we choose a model architecture and denote its collection of weights by $\boldsymbol{\omega}$. We randomly initialize two sets of network weights, $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$, to define

the endpoints of our subspace. Our network subspace spans the line between $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$ and is defined by $\boldsymbol{\omega}^*(\alpha) = \alpha\boldsymbol{\omega}_1 + (1-\alpha)\boldsymbol{\omega}_2$, where $\alpha \in [\alpha_1, \alpha_2]$, $\alpha_1, \alpha_2 \in [0, 1]$, $\alpha_1 < \alpha_2$. In other words, by varying our subspace parameter α, we can obtain a set of weights $\boldsymbol{\omega}^*(\alpha)$ through interpolation.

We now adjust our subspace so that one end (e.g. $\alpha \approx \alpha_1$) yields highly compressed networks, but the other end (e.g. $\alpha \approx \alpha_2$) yields highly accurate networks. Intermediate values (e.g. $\alpha \approx (\alpha_1 + \alpha_2)/2$) should exhibit moderate compression. In other words, we want to be able to tune $\alpha \in [\alpha_1, \alpha_2]$ to choose our desired compression level. To achieve this, we introduce a function $\gamma(\alpha)$ used to control the compression level and a compression function $f(\boldsymbol{\omega}, \gamma)$.

To train our subspace, we first sample a position in our subspace by randomly choosing some $\alpha \in [\alpha_1, \alpha_2]$. We then compress $\boldsymbol{\omega}^*(\alpha)$, obtaining a network with weights $f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha))$. We then perform a standard forward and backward pass of gradient descent with it, back-propagating gradients to $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$. We continue training in this manner until convergence.

Once our model is trained, a user can deploy $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$ on-device to allow efficient, adaptive, real-time compression. To change compression levels in real-time, the user first determines how many resources are available on the device. This step is application-dependent, and may involve looking at the amount of currently available memory or the current CPU load. The user chooses the compression level γ_0 based on currently available resources. From this quantity, the user calculates the appropriate $\alpha_0 = \gamma^{-1}(\gamma_0)$ value corresponding to the desired compression level. Next, the user computes the compressed network, $f(\boldsymbol{\omega}^*(\alpha_0), \gamma_0)$. This network is used until a new compression level is desired by the user. Note that computing f is negligible compared to the cost of a network forward pass in all our experiments (see Section 4.3.5).

4.3.2 Compressible Points

In Section 4.3.1, we discussed formulating our subspace as a line connected by two endpoints in weight space. This formulation requires additional storage resources to deploy

the subspace (Table 4.1), since an extra copy of network weights is stored. For many cost-efficient computing devices, this overhead may be significant. To eliminate this need, we propose training a degenerate subspace with a single point in weight-space (rather than two endpoints). We still use $\alpha \in [\alpha_1, \alpha_2]$ to control our compression ratio, but our subspace is parameterized by a single set of weights, $\boldsymbol{\omega}^*(\alpha) = \boldsymbol{\omega}$. The compressed weights are now expressed as $f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha)) \equiv f(\boldsymbol{\omega}, \gamma(\alpha))$. This corresponds to applying varying levels of compression during each forward pass.

This method still produces a subspace of models in the sense that, for each value of α, we obtain a different compressed network $f(\boldsymbol{\omega}, \gamma(\alpha))$. However, we no longer use different endpoints of a linear subspace to specialize one end of the subspace for accuracy and the other for efficiency. Instead, we condition one set of network weights to tolerate varying levels of compression.

4.3.3 Sampling the Subspace Parameter

When training our compressible subspaces, we need to sample our subspace parameter α at each iteration of training. In [153], a "sandwich method" for training with varying levels of structured sparsity is proposed. This method involves performing n rounds of forward and backward passes in each iteration of training. One round uses the maximum sparsity level, another round uses the minimum sparsity level, and the remaining $n - 2$ rounds use randomly chosen sparsity levels. After all n rounds of forward and backward passes, the gradient update is applied.

To incorporate this method into our algorithm, we introduce a stochastic function $\boldsymbol{\alpha}_n : \boldsymbol{\Omega} \rightarrow [\alpha_1, \alpha_2]^n$, where $\boldsymbol{\Omega}$ represents the state of the stochastic function (e.g. the internal state of a random number generator). For each training batch, we sample $\boldsymbol{\alpha} \in [0, 1]^n$ from $\boldsymbol{\alpha}_n$. We perform n forward and backward passes using compressed networks $f(\boldsymbol{\omega}^*(\alpha_i), \gamma(\alpha_i))$ for $i \in \{1, ..., n\}$, where α_i is the i^{th} element of $\boldsymbol{\alpha}$. Then, the gradient update is applied.

In most experiments, we omit the sandwich rule (by setting $n = 1$) because we found the benefit to be marginal compared to the increased training cost. However, we use the

sandwich rule with $n = 4$ when comparing to [153] in the structured sparsity setting, where we found the accuracy improvements more significant.

An overview of our overall algorithm is given in Figure 4.1. Next, we detail our compression methods f.

4.3.4 Compression Methods

We experiment with three different formulations for our compression function $f(\boldsymbol{\omega}, \gamma)$. These correspond to unstructured sparsity, structured sparsity, and quantization.

In our unstructured sparsity experiments, our compression function $f(\boldsymbol{\omega}, \gamma)$ is TopK sparsity [156], which prunes a fraction γ of the weights with the smallest absolute value from each layer (we ignore the input and output layer). Our compression level calculator is $\gamma(\alpha) = 1 - \alpha$. Our stochastic sampling function $\boldsymbol{\alpha}_n$ samples a single α value uniformly along an interval $[\alpha_1, \alpha_2]$. We experiment with different settings of $[\alpha_1, \alpha_2]$ corresponding to the wide-sparsity regime and high-sparsity regime. See Section 4.4 for experimental details.

In our structured sparsity experiments, our compression function $f(\boldsymbol{\omega}, \gamma)$ retains a fraction γ of the input and output channels in each layer and prunes away the rest (we ignore the input channels in the first layer, and the output channels in the last layer). Our compression level calculator $\gamma(\alpha) : [\alpha_1, \alpha_2] \to [a, b]$ is the unique affine transformation over its domain and range. Here, $[a, b]$ is the width factor range where a and b are the minimum and maximum fraction of channels retained (see Section 4.4 for model-specific parameter settings). Our stochastic sampling function $\boldsymbol{\alpha}_n$ samples $n = 4$ values as $[a, b, U(a, b), U(a, b)]$, where $U(a, b)$ samples uniformly in the range $[a, b]$. This choice mirrors the "sandwich rule" used in [153].

In quantization experiments, our compression function $f(\boldsymbol{\omega}, \gamma)$ is affine quantization as described in [62]. Our compression level calculator is $\gamma(\alpha) = 2 + 6\alpha$. Our stochastic sampling function $\boldsymbol{\alpha}_n$ samples a single α value uniformly over the set $\{1/6, 2/6, ..., 6/6\}$. This corresponds to training with bit widths 3 through 8. We avoid lower bit widths to circumvent training instabilities we encountered in baselines.

4.3.5 Computational Cost of Compression

We justify our claim that our network compression functions $f(\omega, \gamma)$ can be computed in real-time. Consider a network layer with weights $\boldsymbol{\omega}_l$. The computational cost of a forward pass is $\mathcal{O}(|\boldsymbol{\omega}_l|L)$, where L is the spatial dimension (in the case of a 2D convolution, $L = HW$, where H, W are the height and width of the output buffer). In the LCS+L algorithm, the cost of computing $\boldsymbol{\omega}^*$ is $\mathcal{O}(|\boldsymbol{\omega}_l|)$, which is far less than the cost of a forward pass. In the case of LCS+P, no computation is needed. Similarly, the cost of computing $\gamma(\alpha)$ is $O(1)$.

Now, consider the cost of computing $f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha))$. Our unstructured sparsity method takes $\mathcal{O}(|\boldsymbol{\omega}_l|)$ for each layer (to calculate the threshold, then discard parameters below the threshold). Our structured sparsity method takes $\mathcal{O}(1)$ for each layer, since we only need to mark each layer with the number of filters to ignore. Since most inference libraries support tensor slicing operations, we can simply pass a subset of the filters to the underlying matrix multiplication or convolution. Our quantization method [62] takes $\mathcal{O}(|\boldsymbol{\omega}_l|)$ for each layer (to calculate the affine transform parameters and apply them). In all cases, the complexity of computing $f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha))$ is at least L times lower than the cost of the convolutional forward passes, meaning our compression methods can be considered real-time.

4.3.6 Circumventing BatchNorm Recalibration

Previous works that train a compressible network require an additional training step to calibrate BatchNorm [60] statistics at each compression level [153, 154]. This precludes both methods from evaluating at arbitrarily fine-grained compression levels after deployment (Table 4.1). We seek to eliminate the need for recalibration or storage of statistics.

To understand the need for recalibration in previous works, recall that BatchNorm layers store the per-channel mean of the inputs $\boldsymbol{\mu}$ and the per-channel variance of the inputs $\boldsymbol{\sigma}^2$. The recalibration step is needed to correct $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$, which are corrupted when a network is adjusted. Adjustments that corrupt statistics include applying sparsity and quantization.

In Figure 4.3, we analyze the inaccuracies of BatchNorm statistics for two models trained

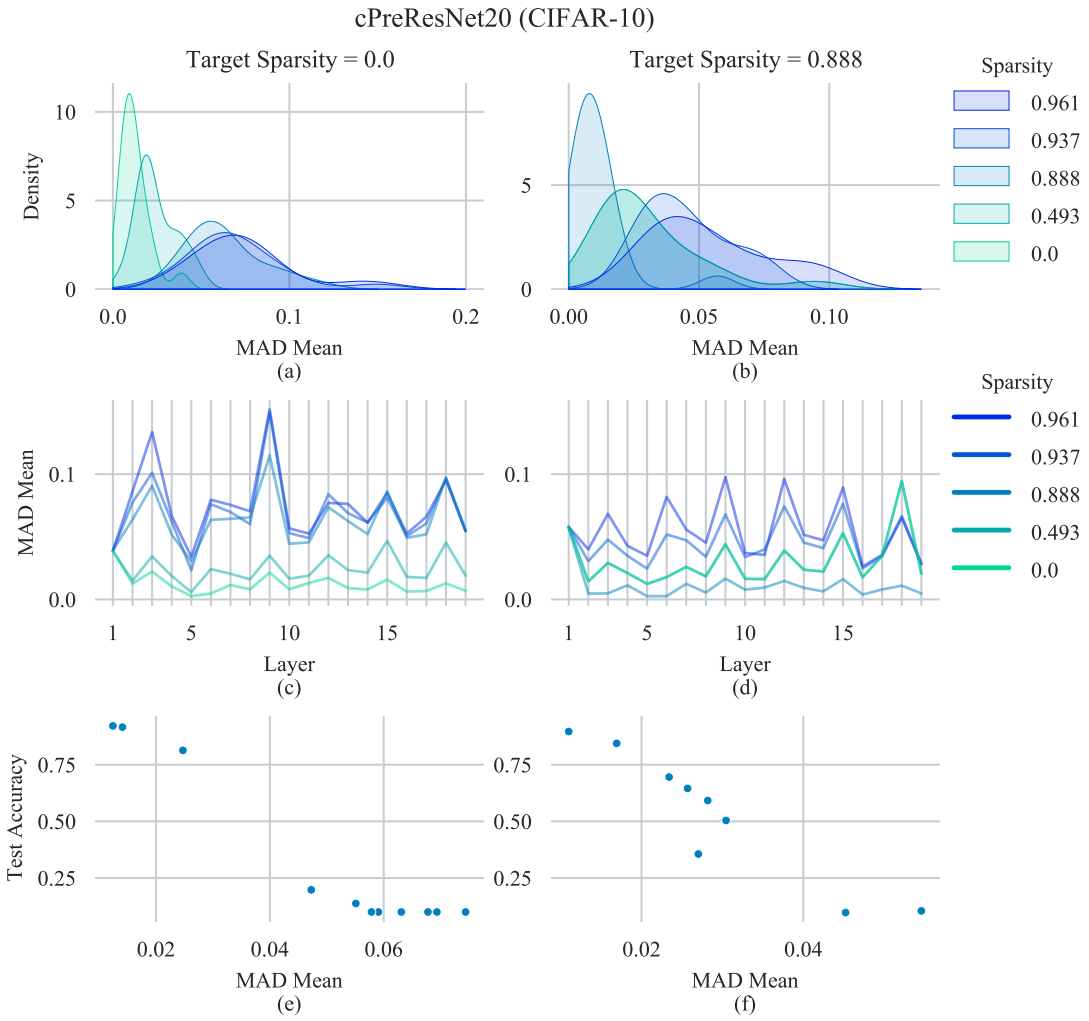


Figure 4.3: Analysis of observed batch-wise means $\hat{\boldsymbol{\mu}}$ and stored BatchNorm means $\boldsymbol{\mu}$ during testing for models trained with TopK unstructured sparsity. The models are trained with different target sparsities and evaluated with various inference-time sparsities. (a)-(b): The distribution of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ across all layers. (c)-(d): The average value of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ for individual layers. (e)-(f): The correlation between the average of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ and test set error. Note that in (b) and (d), sparsities of 0 and 0.493 produce near-identical results, thus those curves are overlapping

with specific unstructured sparsity levels and tested with a variety of inference-time unstructured sparsity levels. We calculate the differences between stored BatchNorm means $\boldsymbol{\mu}$ and

the true mean of a batch $\hat{\boldsymbol{\mu}}$ during the test epoch of a cPreResNet20 [44] model on CIFAR-10 [68]. In Figure 4.3a and Figure 4.3b, we show the distribution of mean absolute differences $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ across all layers of the models. Models have lower BatchNorm errors when evaluated near sparsity levels that match their training-time target sparsity. Applying mismatched levels of sparsity shifts the distribution of these errors away from 0. In Figure 4.3c and Figure 4.3d, we show the average of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ across the test set for each of the BatchNorm layers. Across layers, the lowest error is achieved when the level of sparsity matches training. In Figure 4.3e and Figure 4.3f, we show the average of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ and the corresponding test set accuracy for various levels of inference-time sparsity. We find that the increased error in BatchNorm is correlated with decreased accuracy. See Appendix B.2 for similar analyses with structured sparsity and quantization.

Thus, BatchNorm layers' stored statistics can become inaccurate during inference-time compression, which can lead to accuracy degradation. To circumvent the need for BatchNorm, we adjust our networks to use GroupNorm [146]. This computes an alternative normalization over g groups of channels rather than across a batch. It doesn't require maintaining a running average of the mean and variance across batches of input, so there are no stored statistics that can be corrupted if the network changes.

GroupNorm typically uses $g = 32$ groups, but it also includes InstanceNorm [135] (in which $g = c$, where c is the number of channels) as a special case. We use $g = c$ in structured sparsity experiments, since the number of channels is determined dynamically and is not always divisible by 32. For all other experiments, we use $g \in \{1, 8, 32\}$ depending on the architecture as discussed in Appendix B.5.

4.4 Experiments

We present results in the domains of unstructured sparsity, structured sparsity, and quantization. We train using Pytorch [105] on Nvidia GPUs. On CIFAR-10 [68], we experiment with the pre-activation version of ResNet20 [44] presented in the PyTorch version of the open-source code provided by [82]. We abbreviate it as "cPreResNet20."

We additionally experiment with a variety of architectures on the ImageNet [24] dataset. In particular, we present results using standard convolutional neural network (CNN) architectures: ResNet18 [44], and VGG19 [122]; lightweight CNNs: MnasNet-B1 [129], MobileNetV2 [116], MobileNetV3-Small, and MobileNetV3-Large [55]; and vision transformer models: DeiT-Ti, DeiT-S [133], and CaiT-XXS [134]. All models are trained using an input resolution of 224×224. Our baseline model accuracies are summarized in Appendix B.4.

We train cPreResNet20 for 200 epochs and ImageNet CNNs for 90 epochs. We follow hyperparameter choices in [141] for our methods and baselines (though we don't use the β regularization they describe), with a few architecture-dependent parameters detailed in Appendix B.4. For transformer models, we train for 300 epochs and follow the hyperparameter settings in [133]. Our baselines for each architecture always use the same training hyperparameters as our own methods.

4.4.1 Unstructured Sparsity

We present results for our method using MobileNetV2 and ResNet18 in Figure 4.4. For MobileNetV2, we use an α range of $[0.025, 1]$, corresponding to a wide sparsity training regime, while for ResNet18 we use a range of $[0.005, 0.05]$, corresponding to a high sparsity training regime (because ResNet18 is overparameterized, we operate over a high sparsity range to make the efficiency-accuracy trade-off clearer).[2] Additional hyperparameter details are provided in Appendix B.5.

Our method achieves a strong efficiency-accuracy trade-off in both cases. Our line subspace (LCS+L+GN) achieves a higher accuracy at high sparsities, at the expense of a lower accuracy at low sparsities. To our knowledge, efficient, adaptive, real-time compression has not been previously explored for unstructured sparsity. Thus, our baselines are networks trained for a particular TopK sparsity target, and evaluated at a variety of targets. These methods peak in accuracy near their target sparsity, but decrease sharply at higher sparsities.

[2] In the unstructured setting, we do not compress the first and last layers of our models. Hence a compressed model's sparsity rate may not be exactly $1 - \alpha$.

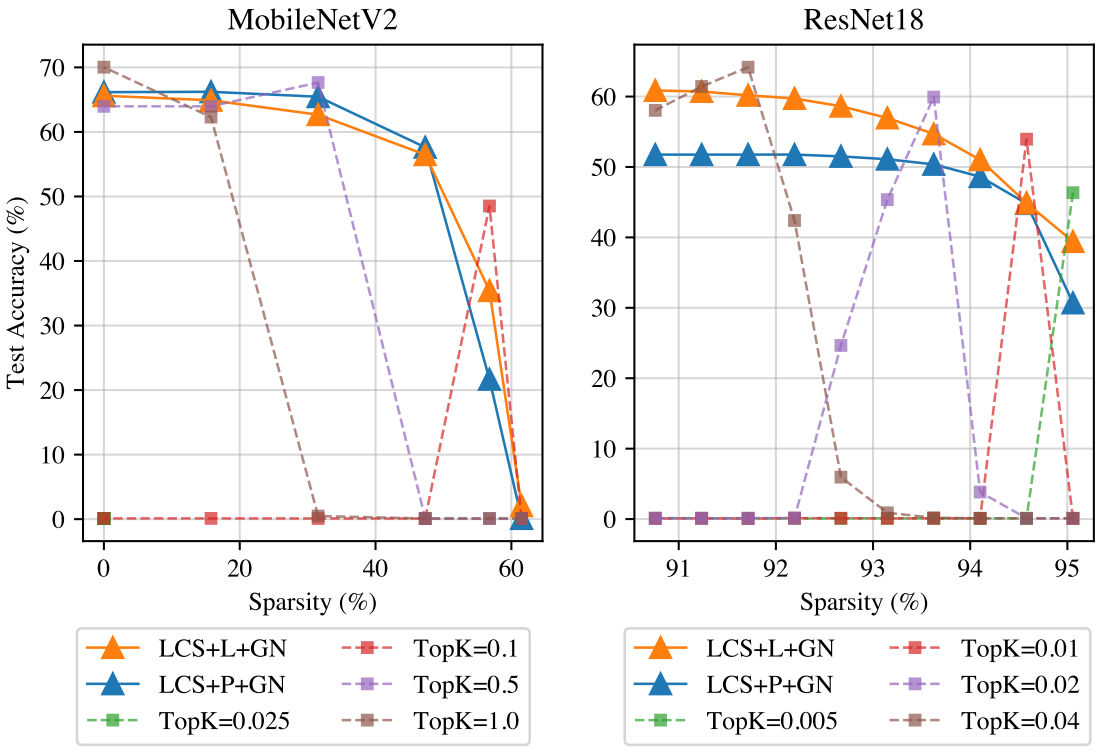


Figure 4.4: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target. The TopK target refers to the fraction of weights that remain unpruned during training

We present additional results for our method using transformer architectures in Figure 4.5. Transformers contain LayerNorm [6] rather than BatchNorm, which does not require recalibration. Hence, we do not need to modify normalization layers in this case. As before, our method produces a strong efficiency-accuracy trade-off across a variety of sparsity levels. Our LCS+L+LN method underperforms on DeiT models relative to LCS+P+LN, but still achieves stronger results than baselines at high sparsities. We hypothesize that the benefits of learning fewer parameters outweighs the benefits of increased capacity in this case, but we leave more investigation to future work.

In Appendix B.8, we provide runtime characteristics of our models. We also present

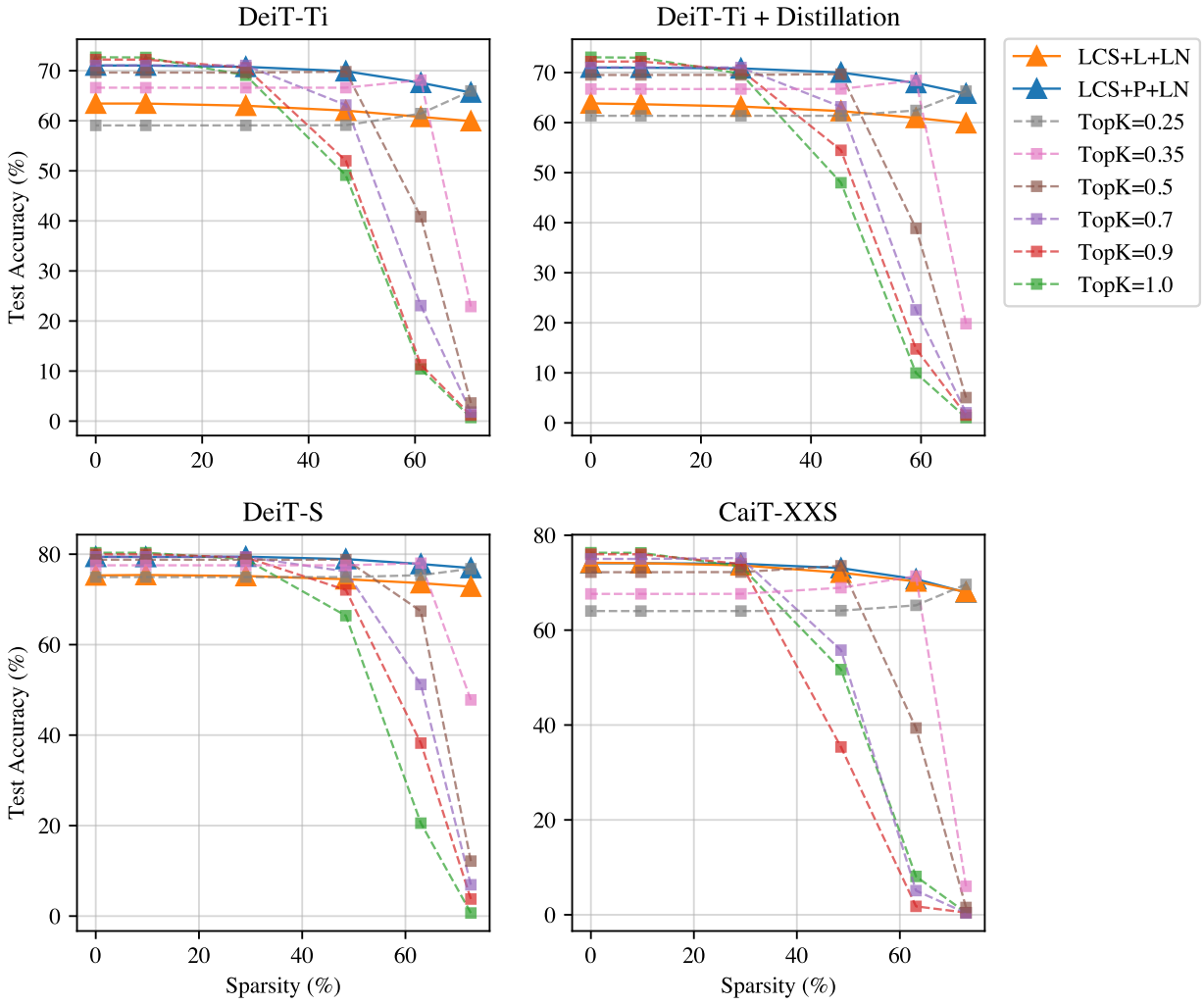


Figure 4.5: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target

results for the wide sparsity regime using cPreResNet20, ResNet18, VGG19, MnasNet, MobileNetV3-Small, and MobileNetV3-Large; additionally, we show results for the high sparsity regime using DeiT-Ti and DeiT-S.

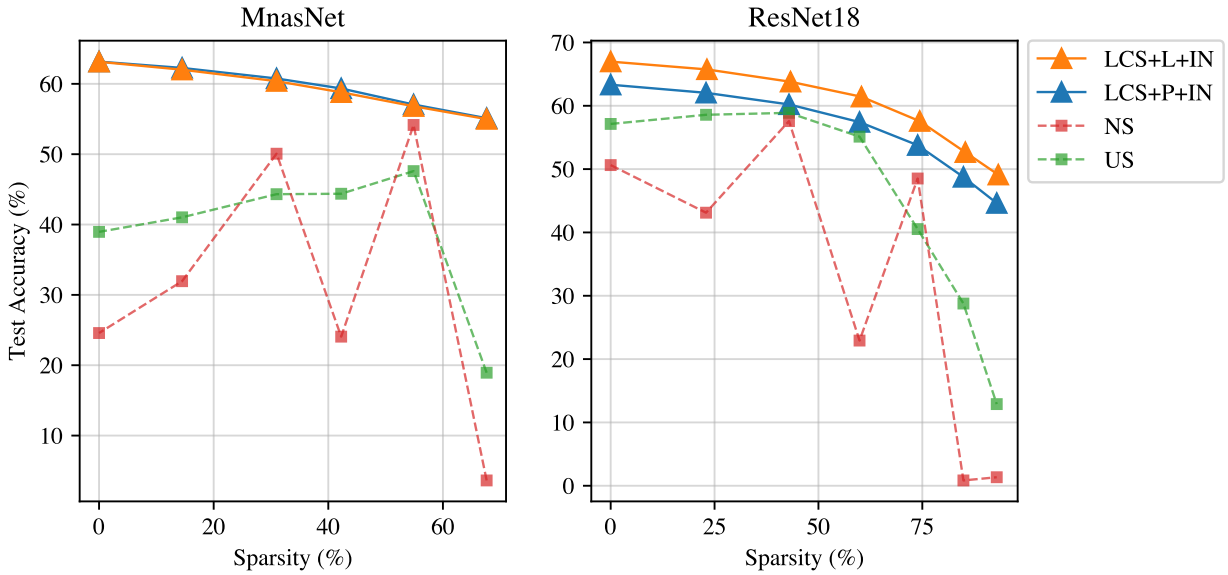


Figure 4.6: Our method for structured sparsity using a linear subspace (LCS+L+IN) and a point subspace (LCS+P+IN) compared to Universal Slimming (US) [153] and Network Slimming (NS) [154]. We do not allow recalibration

4.4.2 Structured Sparsity

We present results for our method using structured sparsity in Figure 4.6. For all structured sparsity experiments, we use an α range of $[0, 1]$. For MnasNet, we use a width factor range of $[0.5, 1]$. For ResNet18, our width factor range is $[0.25, 1]$. As discussed in Section 4.3.6, we use a special case of GroupNorm [146] known as InstanceNorm [135] since the number of channels in the network varies. We performed preliminary experiments with LayerNorm, but InstanceNorm achieved stronger results in our case. In the case of structured sparsity, filters are able to specialize without the need for an extra copy of network weights, since some filters are only used when the model is lightly pruned. Therefore, we only create extra copies of our InstanceNorm parameters when using LCS+L+IN, as an extra copy of network weights was unnecessary. See Appendix B.6 for additional hyperparameter details.

To our knowledge, efficient, adaptive real-time compression has not been explored before for structured sparsity. Thus, we compare to modified versions of Network Slimming [154] and Universal Slimming [153] in which BatchNorm recalibration is not performed. Additionally, NS does not allow for evaluation at arbitrary sparsities, so we alter it slightly to use a single BatchNorm layer (choosing subsets of its channels to achieve adaptive compression). After these changes, the main difference between our US and NS baselines is, NS uses network width factors of $[0.25, 0.5, 0.75, 1]$ at training and inference, whereas US uses the sandwich rule (Section 4.3.3).

Our method demonstrates a strong efficiency-accuracy trade-off. The trade-off produced by US peaks in the middle. We hypothesize that this is due to their sandwich rule training formulation in which sparsity levels are randomly sampled. This could cause the BatchNorm statistics to be more accurate (on average) near the middle of the sparsity range. The trade-off produced by NS contains peaks and troughs. This method trains only at discrete width factors of $[0.25, 0.50, 0.75, 1]$. This method produces stronger accuracies at these sparsities than at sparsities that it wasn't explicitly trained for.

In preliminary experiments with transformers, we found that adaptive compression for structured sparsity did not converge to high accuracies. We hypothesize this may be due to inter-channel variation of transformers described in [79], but we leave more investigation to future work. See also Appendix B.8 for results using VGG19, MobileNetV2, MobileNetV3-Small, and MobileNetV3-Large, as well as speed and memory usage characteristics.

4.4.3 Quantization

We also provide preliminary experiments for quantization. Note that in the quantization setting, there are a small number of discrete compression levels. In this case, it is usually feasible to simply store extra BatchNorm parameters for all desired parameter settings before model deployment. Thus, our main purpose for experimenting in this setting is to characterize the behavior of our method under another compression technique besides pruning, to verify the generality of our method.

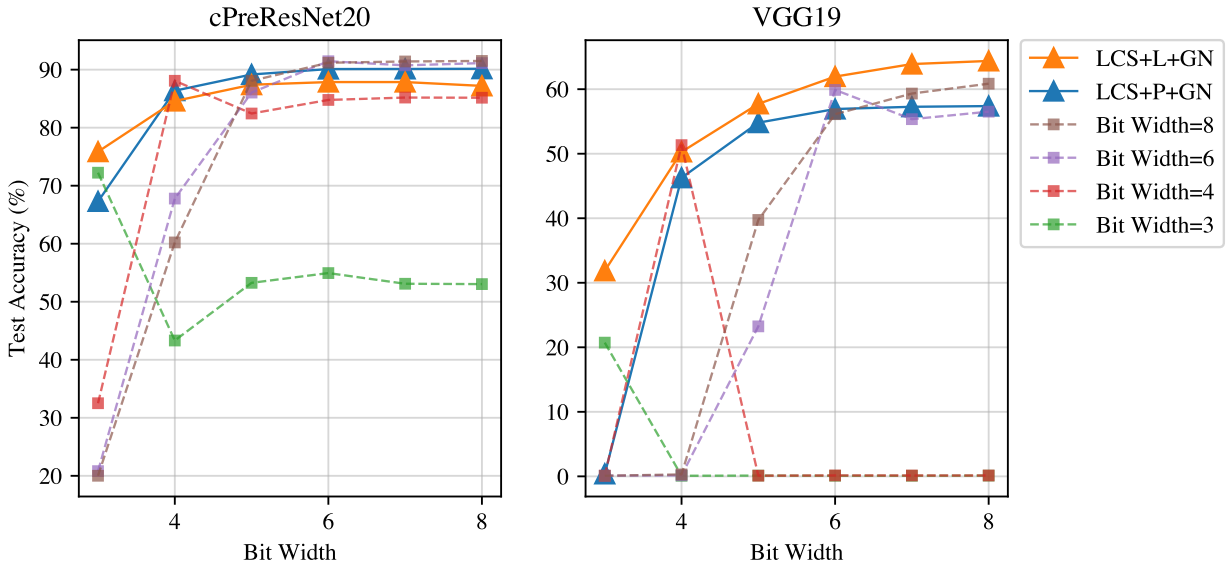


Figure 4.7: Our method for quantization using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular bit width target

We present results for our method in Figure 4.7, comparing to models trained at a fixed bit width and evaluated at a variety of bit widths. See Appendix B.7 for training details. Generally, baselines achieve high accuracy at the bit width at which they were trained, and reduced accuracy at other bit widths. By contrast, our method using a linear subspace (LCS+L+GN) achieves high accuracy at all bit widths, matching or exceeding accuracies of individual networks trained for target bit widths. In the case of VGG19, we found that our accuracy even exceeded the baselines. We believe part of the increase is due to GroupNorm demonstrating improved results on this network compared to BatchNorm (which does not happen with ResNets, as reported in [146]). See Appendix B.8 for ResNet18 results, and for memory usage characteristics of models.

4.5 Conclusion

We present a method for learning a compressible subspace of neural networks. Our method produces a model that can be deployed on-device and used for efficient, adaptive, real-time model compression. Our model can be compressed after deployment in real-time, to any compression level, without retraining, and without specifying the compression levels before deployment. Additionally, our LCS+P method incurs no parameter overhead. We show that our generic algorithm outperforms baselines in the domains of unstructured sparsity and structured sparsity. We demonstrate that it is flexible enough to apply to quantization.

Chapter 5
CONCLUSION

We investigate the challenges of compressing models in data-constrained scenarios. We have explored three scenarios. In the first, labels are unavailable. This scenarios arises if the original dataset is unavailable, but unlabeled data can be easily obtained. In the second, data and labels are unavailable. This scenario arises if the original dataset is unavailable, and data is difficult to obtain. In the third, compression occurs after model deployment. This requires compressing without data, and with constrained compute. This scenario arises when we want to compress to varying compression levels after deployment to allow for dynamic resource consumption. As on-device machine learning continues to grow, we hope that this exploration and similar efforts will enable previously unseen functionalities for challenging scenarios in which traditional machine learning methods cannot be used.

We believe there is opportunity for strong future contributions in data-constrained model compression. We encourage researchers to create a competition-style evaluation on a held-out test set. In this competition, contestants would be given a model to compress without being told what dataset it operates on, or what domain its inputs and outputs come from. This would more realistically model the scenario in which no dataset is available, because researchers would not have a validation set on which to tune their model. Another rich area for exploration is in pushing the accuracy of data-constrained models to match the performance of models trained on the original data. Because accuracy of data-free model retraining is lower than the results on the original dataset, there should be opportunities to improve upon the results demonstrated in existing works.

BIBLIOGRAPHY

[1] Are your home security cameras vulnerable to hacking? `https://www.cnet.com/home/security/stop-home-security-camera-hacking/`. Accessed: 2022-03-26.

[2] Megaface. `https://exposing.ai/megaface/`. Accessed: 2022-03-26.

[3] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13:1 – 18, 2017.

[4] Raman Arora, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee. Understanding deep neural networks with rectified linear units. In *International Conference on Learning Representations*, 2018.

[5] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[6] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

[7] Hessam Bagherinezhad, Maxwell Horton, Mohammad Rastegari, and Ali Farhadi. Label refinery: Improving imagenet classification through label progression. *CoRR*, abs/1805.02641, 2018.

[8] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*, 2018.

[9] Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. MIT Press, 2007.

[10] Gregory Benton, Wesley Maddox, Sanae Lotfi, and Andrew Gordon Gordon Wilson. Loss surface simplexes for mode connecting volumes and fast ensembling. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 769–779. PMLR, 18–24 Jul 2021.

[11] Aishwarya Bhandare, V. Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and V. Saletore. Efficient 8-bit quantization of transformer neural machine language translation model. *ArXiv*, abs/1906.00532, 2019.

[12] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[13] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 535–541, New York, NY, USA, 2006. ACM.

[14] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020.

[15] Lijuan Cai and Thomas Hofmann. Exploiting known taxonomies in learning overlapping concepts. In *IJCAI*, volume 7, pages 708–713, 2007.

[16] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13166–13175, 2020.

[17] Yao Chen, Cole Hawkins, Kaiqi Zhang, Zheng Zhang, and Cong Hao. 3u-edgeai: Ultra-low memory training, ultra-low bitwidthquantization, and ultra-low latency acceleration. *CoRR*, abs/2105.06250, 2021.

[18] Yoojin Choi, Jihwan Choi, Mostafa El-Khamy, and Jungwon Lee. Data-free network quantization with adversarial knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[19] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[20] 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.

[21] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems*, pages 598–605. Morgan Kaufmann, 1990.

[22] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[25] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance, 2019.

[26] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. On-device machine learning: An algorithms and learning theory perspective. *arXiv preprint arXiv:1911.00623*, 2019.

[27] Giuseppe Di Guglielmo, Javier Mauricio Duarte, Philip Harris, Duc Hoang, Sergo Jindariani, Edward Kreinar, Mia Liu, Vladimir Loncar, Jennifer Ngadiuba, Kevin Pedro, and et al. Compressing deep neural networks on fpgas to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology*, Jun 2020.

[28] Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 4860–4874, Red Hook, NY, USA, 2017. Curran Associates Inc.

[29] Zhen Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 293–302, 2019.

[30] Erich Elsen, Pablo Samuel Castro Rivadeneira, Trevor Gale, and Utku Evci. Rigging the lottery: Making all tickets winners. In *International Conference of Machine Learning*, 2020.

[31] Tommaso Furlanello, Zachary C Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. Born again neural networks. *arXiv preprint arXiv:1805.04770*, 2018.

[32] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[33] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

[34] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[35] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. Wootz: A compiler-based framework for fast cnn pruning via composability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 717–730, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Luis Guerra, Bohan Zhuang, Ian Reid, and Tom Drummond. Switchable precision neural networks. *arXiv preprint arXiv:2002.02815*, 2020.

[37] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 1387–1395, Red Hook, NY, USA, 2016. Curran Associates Inc.

[38] Hai Victor Habi, Roy H. Jennings, and Arnon Netzer. Hmq: Hardware friendly mixed precision quantization block for cnns, 2020.

[39] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural network. *ArXiv*, abs/1506.02626, 2015.

[40] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. *arXiv preprint arXiv:1506.02626*, 2015.

[41] Matan Haroush, Itay Hubara, Elad Hoffer, and Daniel Soudry. The knowledge within: Methods for data-free model compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[42] B. Hassibi and D. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1992.

[43] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1992.

[44] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[47] Xiangyu He, Qinghao Hu, Peisong Wang, and Jian Cheng. Generative zero-shot network quantization. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2994–3005, 2021.

[48] Yihui He, X. Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, 2017.

[49] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[50] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[51] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

[52] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *CoRR*, abs/2102.00554, 2021.

[53] Maxwell Horton, Yanzi Jin, Ali Farhadi, and Mohammad Rastegari. Layer-wise data-free CNN compression. *CoRR*, abs/2011.09058, 2020.

[54] A. Howard, Menglong Zhu, Bo Chen, D. Kalenichenko, W. Wang, Tobias Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[55] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

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

[57] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, 2017.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456, 2015.

[59] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

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

[61] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[62] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[63] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[64] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[65] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[66] Animesh Koratana, Daniel Kang, Peter Bailis, and Matei Zaharia. LIT: block-wise intermediate representation training for model compression. *CoRR*, abs/1810.01937, 2018.

[67] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018.

[68] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

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

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

[71] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, S. Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. *ArXiv*, abs/2002.03231, 2020.

[72] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. In *Proceedings of the International Conference on Machine Learning*, July 2020.

[73] Y. LeCun, J. Denker, and S. Solla. Optimal brain damage. In *NIPS*, 1989.

[74] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

[75] Jinyu Li, Rui Zhao, Jui-Ting Huang, and Yifan Gong. Learning small-size dnn with output-distribution-based criteria. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[76] R. Li, Y. Wang, F. Liang, H. Qin, J. Yan, and R. Fan. Fully quantized network for object detection. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2805–2814, 2019.

[77] Yuncheng Li, Jianchao Yang, Yale Song, Liangliang Cao, Jiebo Luo, and Jia Li. Learning from noisy labels with distillation. *arXiv preprint arXiv:1703.02391*, 2017.

[78] Mingbao Lin, Rongrong Ji, Shaojie Li, Qixiang Ye, Yonghong Tian, J. Liu, and Q. Tian. Filter sketch for network pruning. *ArXiv*, abs/2001.08514, 2020.

[79] Yang Lin, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. Fq-vit: Fully quantized vision transformer without retraining. *ArXiv*, abs/2111.13824, 2021.

[80] Bin Liu, Yue Cao, Mingsheng Long, Jianmin Wang, and Jingdong Wang. Deep triplet quantization. In *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, page 755–763, New York, NY, USA, 2018. Association for Computing Machinery.

[81] Xuanqing Liu, Minhao Cheng, Huan Zhang, and Cho-Jui Hsieh. Towards robust neural networks via random self-ensemble. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 369–385, 2018.

[82] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2755–2763, 2017.

[83] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[84] Christos Louizos, Max Welling, and Diederik P. Kingma. Learning sparse neural networks through l_0 regularization. In *International Conference on Learning Representations*, 2018.

[85] Nenad Markus. Fusing batchnorm with convolution in runtime. https://nenadmarkus.com/p/fusing-batchnorm-and-conv/. Accessed: 2021-12-22.

[86] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. Training binary neural networks with real-to-binary convolutions. In *International Conference on Learning Representations*, 2020.

[87] Julian J McAuley, Arnau Ramisa, and Tibério S Caetano. Optimization of robust loss functions for weakly-labeled image taxonomies. *International journal of computer vision*, 104(3):343–361, 2013.

[88] H. Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[89] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different: Recovering neural network quantization error through weight factorization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4486–4495. PMLR, 09–15 Jun 2019.

[90] Paul Micaelli and Amos J. Storkey. Zero-shot knowledge transfer via adversarial belief matching. In *NeurIPS*, 2019.

[91] George A Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J Miller. Introduction to wordnet: An on-line lexical database. *International journal of lexicography*, 3(4):235–244, 1990.

[92] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Praneeth Vepakomma, Abhishek Singh, Ramesh Raskar, and Hadi Esmaeilzadeh. Privacy in deep learning: A survey. *CoRR*, abs/2004.12254, 2020.

[93] Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, and Shin Ishii. Distributional smoothing by virtual adversarial examples. *stat*, 1050:2, 2015.

[94] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv: Learning*, 2017.

[95] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks, 2015.

[96] M. Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. *ArXiv*, abs/2004.10568, 2020.

[97] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. *ArXiv*, abs/2004.10568, 2020.

[98] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[99] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1325–1334, 2019.

[100] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alexander M. Bronstein, and Avi Mendelson. Loss aware post-training quantization. *ArXiv*, abs/1911.07190, 2021.

[101] Sharan Narang, G. Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *ArXiv*, abs/1704.05119, 2017.

[102] Sharan Narang, Greg Diamos, S. Sengupta, and E. Elsen. Exploring sparsity in recurrent neural networks. *ArXiv*, abs/1704.05119, 2017.

[103] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[104] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[105] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[106] Gabriel Pereyra, George Tucker, Jan Chorowski, Łukasz Kaiser, and Geoffrey Hinton. Regularizing neural networks by penalizing confident output distributions. *arXiv preprint arXiv:1701.06548*, 2017.

[107] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[108] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

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

[110] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 6517–6525. IEEE, 2017.

[111] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 6517–6525. IEEE, 2017.

[112] Scott Reed, Honglak Lee, Dragomir Anguelov, Christian Szegedy, Dumitru Erhan, and Andrew Rabinovich. Training deep neural networks on noisy labels with bootstrapping. *arXiv preprint arXiv:1412.6596*, 2014.

[113] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[114] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[115] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[116] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[117] Abigail See, Minh-Thang Luong, and Christopher D. Manning. Compression of neural machine translation models via pruning. In *CoNLL*, 2016.

[118] Jonathan Shen, Noranart Vesdapunt, Vishnu N Boddeti, and Kris M Kitani. In teacher we trust: Deep network compression for pedestrian detection.

[119] Moran Shkolnik, Brian Chmiel, Ron Banner, Gil Shomron, Yury Nahshan, Alex Bronstein, and Uri Weiser. Robust quantization: One model to rule them all. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 5308–5317. Curran Associates, Inc., 2020.

[120] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[121] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[122] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[123] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *BMVC*, 2015.

[124] X. Suau, u. Zappella, and N. Apostoloff. Filter distillation for network compression. In *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 3129–3138, 2020.

[125] Farhana Sultana, Abu Sufian, and Paramartha Dutta. Evolution of image segmentation using deep convolutional neural network: A survey. *Knowledge-Based Systems*, 201-202:106062, Aug 2020.

[126] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[127] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. CVPR, 2015.

[128] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[129] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[130] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[131] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[132] Jialiang Tang, Mingjin Liu, Ning Jiang, Huan Cai, Wenxin Yu, and Jinjia Zhou. Data-free network pruning for model compression. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2021.

[133] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers & distillation through attention. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 10347–10357. PMLR, 18–24 Jul 2021.

[134] Hugo Touvron, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. Going deeper with image transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 32–42, 2021.

[135] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[136] Jason Wang and Luis Perez. The effectiveness of data augmentation in image classification using deep learning. Technical report, Technical report, 2017.

[137] Jeremy HM Wong and Mark John Gales. Sequence student-teacher training of deep neural networks. 2016.

[138] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. Understanding data augmentation for classification: when to warp? In *Digital Image Computing: Techniques and Applications (DICTA), 2016 International Conference on*, pages 1–6. IEEE, 2016.

[139] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. In *NeurIPS*, 2019.

[140] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. In *NeurIPS*, 2019.

[141] Mitchell Wortsman, Maxwell C Horton, Carlos Guestrin, Ali Farhadi, and Mohammad Rastegari. Learning neural network subspaces. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11217–11227. PMLR, 18–24 Jul 2021.

[142] Baoyuan Wu, Fan Jia, Wei Liu, Bernard Ghanem, and Siwei Lyu. Multi-label learning with missing labels using mixed dependency graphs. *International Journal of Computer Vision*, pages 1–22, 2018.

[143] Baoyuan Wu, Siwei Lyu, and Bernard Ghanem. Ml-mg: Multi-label learning with missing labels using a mixed graph. In *Proceedings of the IEEE international conference on computer vision*, pages 4157–4165, 2015.

[144] Cinna Wu, Mark Tygert, and Yann LeCun. Hierarchical loss for classification. *arXiv preprint arXiv:1709.01062*, 2017.

[145] Hao Wu, P. Judd, Xiaojie Zhang, M. Isaev, and P. Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. *ArXiv*, abs/2004.09602, 2020.

[146] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[147] Zhibiao Wu and Martha Palmer. Verbs semantics and lexical selection. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 133–138. Association for Computational Linguistics, 1994.

[148] Lingxi Xie, Jingdong Wang, Zhen Wei, Meng Wang, and Qi Tian. Disturblabel: Regularizing cnn on the loss layer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4753–4762, 2016.

[149] Shoukai Xu, Haokun Li, Bohan Zhuang, Jing Liu, Jiezhang Cao, Chuangrun Liang, and Mingkui Tan. Generative low-bitwidth data free quantization. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, Lecture Notes in Computer Science, pages 1–17. Springer, 2020. European

Conference on Computer Vision 2020, ECCV 2020 ; Conference date: 23-08-2020 Through 28-08-2020.

[150] Yan Xu, Ran Jia, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. Improved relation classification by deep recurrent neural networks with data augmentation. *arXiv preprint arXiv:1601.03651*, 2016.

[151] Y. Yang, Shuang Wu, Lei Deng, Tianyi Yan, Yuan Xie, and Guoqi Li. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural networks : the official journal of the International Neural Network Society*, 125:70–82, 2020.

[152] Hongxu Yin, Pavlo Molchanov, Jose M. Alvarez, Zhizhong Li, Arun Mallya, Derek Hoiem, Niraj K Jha, and Jan Kautz. Dreaming to distill: Data-free knowledge transfer via deepinversion. In *The IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[153] Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1803–1811, 2019.

[154] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *International Conference on Learning Representations*, 2018.

[155] Xiangguo Zhang, Haotong Qin, Yifu Ding, Ruihao Gong, Qing Yan, Renshuai Tao, Yuhang Li, Fengwei Yu, and Xianglong Liu. Diversifying sample generation for accurate data-free quantization. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15653–15662, 2021.

[156] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

Appendix A

LAYER-WISE DATA-FREE COMPRESSION

A.1 Note on Quantization With Residuals

When quantizing a network with residual connections, one must take care to ensure that the activation tensor created by adding the output of quantized convolutions is quantized *after* the addition from the residual is applied. Otherwise, the input to the following layer will not be truly quantized.

To ensure this, we place our activation quantization modules after the skip connections in the case of residual networks. In our DFQ [98] implementation, we need batch norm input statistics for the layer following a skip connection. It is not clear from the discussion in [98] where their activation modules are, or how this detail is handled. When estimating statistics needed for the output of the skip connections (which is needed for DFQ), we estimate the per-channel output statistics from a skip connection as $\mu = \mu_1 + \mu_2$, $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$, where μ_i, σ_i are the mean and standard deviation of the batch norm statistics from the inputs to the residual addition.

A.2 Notes on Experimental Setup

We note here a few details of our experimental setup for Adversarial Knowledge Distillation (AKD) [18] and Deep Inversion (DI) [152].

When using DI to generate data for MobileNet, MobileNetV2, ResNet18, and EfficientNet B0, we use the parameters for MobileNetV2 [115] from the public GitHub repository (settings for these other networks were not in the repository). We validated that these settings generated images that could be recognized by an independent CNN as belonging to the given class. As in [152], we generate $165,000$ images and use them to retrain a student

network from a teacher network. We initialize both the student and the teacher to the same pretrained network. We then train the student with STR [72] to match the teacher using the Label Refinery [7] formulation for Knowledge Distillation [50]. We train for 10 epochs over this training set, using Adam [65] with a cosine learning rate decaying from 0.001 to 0.

When training with AKD [18], we use the parameter settings described in [18]. Specifically, we trained the discriminator with Nesterov gradient descent with momentum 0.9 and initial learning rate 0.1. We used Adam [65] to optimize the generator, with an initial learning rate 0.001. We decay both learning rates to 0 over the course of training, using a cosine learning rate schedule. We use a loss-scaling alpha of $\alpha = 0.1$ (see [18] for details). We adjusted batch sizes to fit in a single NVIDIA Tesla V100 for all experiments. We initialized the student and teacher networks to the same pretrained model, then trained to produce a quantized or sparse model. Since we reloaded pretrained students and teachers, we found we could shorten the training regime and still reach convergence. Optimizing the training regime is important for fair runtime comparison in Figure 3.2. Our epochs in this case were shortened to 12800 samples, and we perform warmup for 50 epochs (enough to plateau the generator loss) and regular training for 250 epochs (enough to reach convergence in all cases).

When training to induce sparsity with AKD, we found it important to adjust the sparsity-inducing weight decay λ (see STR [72] for details). In addition to setting $\lambda = 0.00001551757813$ (as for our other experiments), we used values of 2λ, 10λ, and 100λ to achieve a richer variety of points on the sparsity/accuracy trade-off curve. We found this important in practice. However, in the data-free scenario, we would not in general have access to a validation set, so we show all final-epoch results in Figure 3.1. We also found it useful to adjust our sweep over s_0 values to values that induce an initial sparsity of $[0\%, 10\%, ..., 70\%]$.

When combining our method with DeepInversion (for EfficientNet B0) in Figure 3.5, we simply add our loss to the DeepInversion loss. When combining our method with AKD (for MobileNetV1, MobileNetV2, and ResNet18) in Figure 3.5, we scale our loss by factors of $[0.1, 1, 10, 100]$, add it to the discriminator loss, and plot all final-epoch results.

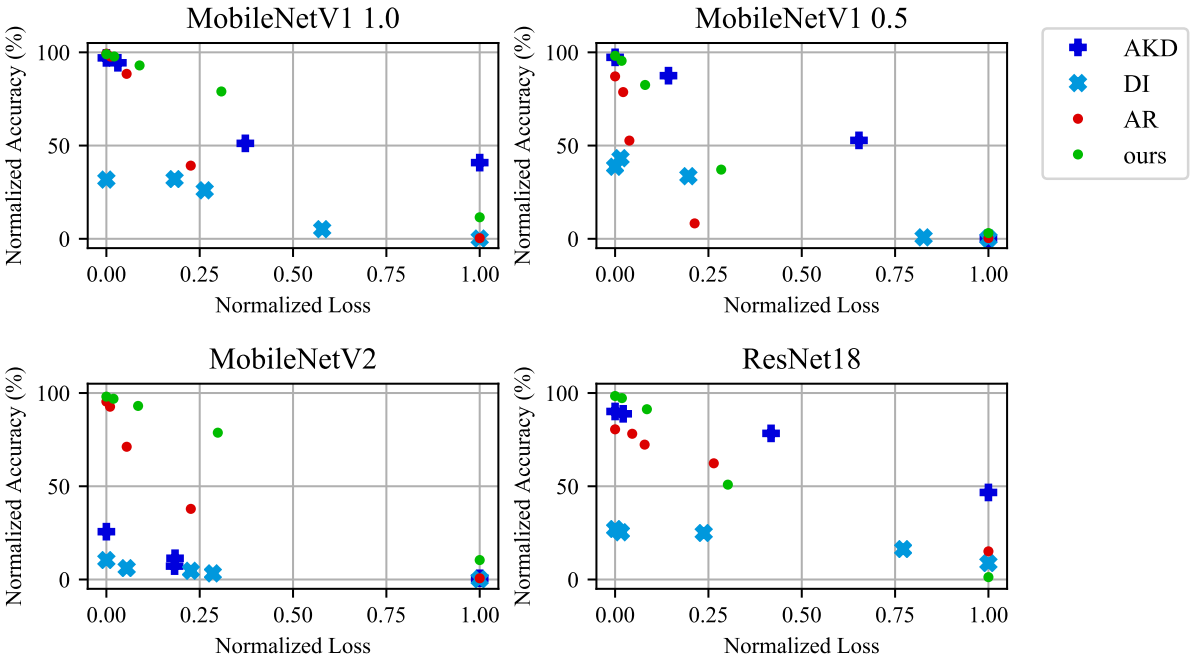


Figure A.1: Correlation between normalized training loss and normalized validation accuracy on ImageNet for quantization methods.

A.3 Analysis of Stability in Quantized Methods

We present the results for best-epoch performance (rather than last-epoch performance) of efficient quantization methods in Table A.1. Since only AdaRound [96] requires backpropagation, only the AdaRound entries are different than the last-epoch results in Table 3.1. The results for DFQ [98] and our method are unchanged. Results for best-epoch performance of expensive quantization methods appear in Table A.2. Note that these best-epoch results give an unrealistic advantage to AdaRound [96], Adversarial Knowledge Distillation [18], and Deep Inversion [152], because a data-free scenario would not allow for a validation set in practice.

Note that gains are modest in some cases, but very large in other cases. We noticed

Bits	ours	DFQ	AR	ours	DFQ	AR
	MNV1 1.0			MNV2		
8	71.13	71.18	**71.36** (+0.01)	**70.50**	70.16	69.73 (+1.12)
7	70.26	69.76	**70.27** (+0.05)	**69.67**	68.89	68.65 (+2.00)
6	**66.95**	61.81	64.28 (+0.60)	**66.97**	63.06	51.93 (+0.93)
5	**55.37**	25.63	29.17 (+0.00).	**56.33**	23.41	27.83 (+0.00)
4	**5.62**	0.23	0.30 (+0.02)	**8.23**	0.49	0.51 (+0.03)
	MNV1 0.5			RN18		
8	**63.77**	63.65	62.18 (+5.79)	**68.67**	67.72	67.64 (+11.40)
7	**61.64**	60.47	53.71 (+2.70)	**67.92**	66.40	66.78 (+12.33)
6	**55.51**	44.67	35.42 (+0.98)	**63.72**	62.86	62.49 (+11.92)
5	**23.79**	7.29	6.02 (+0.54)	35.47	34.76	**51.53** (+7.89)
4	**1.71**	0.24	0.31 (+0.03)	0.88	1.06	**11.15** (+0.51)

Table A.1: Results for computationally-efficient quantization methods on MobileNetV1, MobileNetV2, and ResNet18 on ImageNet. This presents the results in Table 3.1, but showing the best epoch (rather than last epoch) for AR. The difference between the best epoch and the last epoch is reported in parenthesis. As our method and DFQ do not require backpropagation, the results are the same as in Table 3.1. (DFQ): Data-Free Quantization [98]. (AR): AdaRound [96].

Bits	ours	AKD	DI	ours	AKD	DI
	MNV1 1.0			MNV2		
8	**71.13**	70.31 (+0.18)	24.77 (+0.24)	**70.50**	19.57 (+0.24)	7.21 (+0.00)
7	**70.26**	68.16 (+0.39)	25.00 (+3.65)	**69.67**	5.15 (+0.28)	5.60 (+0.17)
6	**66.95**	61.38 (+54.27)	19.24 (+0.07)	**66.97**	10.05 (+0.35)	3.65 (+0.11)
5	**55.37**	31.59 (+31.49)	9.06 (+0.00)	**56.33**	2.08 (+1.22)	2.62 (+0.00)
4	**5.62**	0.10 (+0.00)	0.10 (+0.00)	**8.23**	0.10 (+0.00)	0.16 (+0.07)
	MNV1 0.5			RN18		
8	**63.77**	63.28 (+0.45)	27.98 (+2.23)	**68.67**	62.27 (+0.14)	20.16 (+0.47)
7	**61.64**	60.11 (+2.91)	27.10 (+8.05)	**67.92**	61.39 (+0.01)	19.14 (+0.09)
6	**55.51**	52.16 (+4.16)	22.58 (+0.41)	**63.72**	55.22 (+0.26)	17.50 (+0.52)
5	**23.79**	21.18 (+4.32)	6.70 (+0.00)	35.47	**41.41** (+0.00)	12.43 (+0.00)
4	**1.71**	0.10 (+0.00)	0.10 (+0.00)	0.88	0.10 (+0.00)	**6.62** (+0.29)

Table A.2: Results for our efficient method compared to computationally-expensive quantization methods on MobileNetV1, MobileNetV2, and ResNet18 on ImageNet. This presents the results in Table 3.1, but showing the best epoch (rather than last epoch) for AKD and DI. The difference between the best epoch and the last epoch is reported in parenthesis. As our method does not require backpropagation, the results are the same as in Table 3.1. (AKD): Adversarial Knowledge Distillation [18]. (DI): DeepInversion [18].

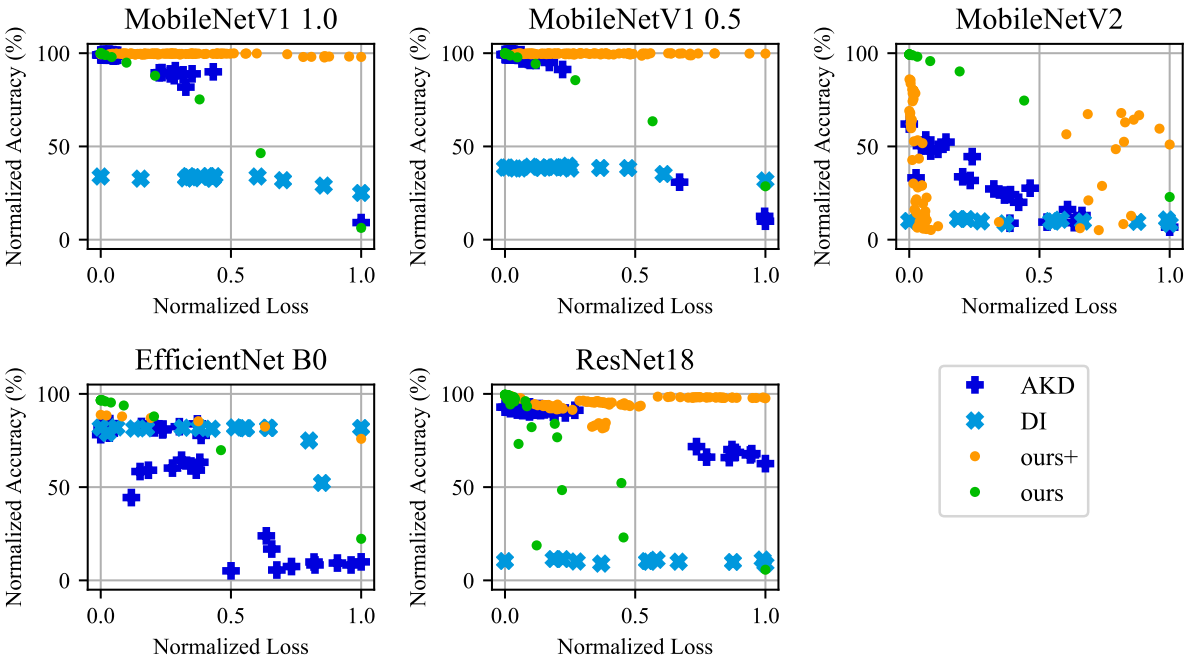


Figure A.2: Correlation between normalized training loss and normalized validation accuracy on ImageNet for pruning methods.

that in some cases (e.g. ResNet18), AdaRound accuracy actually decreases with training, indicating that the preprocessing step provided reasonably accurate quantization, but the training step introduced instability. Additionally, 6-bit MobileNetV1 1.0 achieved a lower final-epoch accuracy than 6-bit MobileNetV1 0.5 (Table 3.1) when using Adversarial Knowledge Distillation (AKD). In Table A.2, we see that the best-epoch accuracy of MobileNetV1 1.0 was much higher than MobileNetV1 0.5 when using AKD. Instability in the optimization lead to the poor final-epoch results in Table 3.1 for 6-bit MobileNetV1 1.0 with AKD.

A.4 Loss and Accuracy Correlation

We also investigate the correlation between the loss function and the accuracy of methods. This correlation is of particular interest in data-free compression, because if no data is available, it's likely that no validation set is available. In this case, we may need to judge the relative quality of a variety of models by examining their loss values.

We show the correlation between training loss and validation accuracy for quantization methods in Figure A.1. Since DFQ does not specify a method for determining the loss, we omit it. For our method, the loss is represented by the sum of squared differences between our generated activations and the quantized activations during our first round of activation quantization (see Section 3.4.1). For all other methods, we report training loss. When reporting the loss for AKD, we ignore the generator loss, since the generator loss is not directly used to train the compressed student. For ease of visualization, we normalize the losses to fall in $[0, 1]$ for the given training jobs. We normalize model accuracies by the accuracy of the original floating-point model from which we reloaded our weights.

We find that loss and accuracy are correlated. When using our method or AdaRound, a lower loss always corresponds to a higher accuracy. For AKD and DI, the trend generally holds, but with at least one exception.

In Figure A.2, we compare the loss and accuracies of our pruning method compared to DI [152], AKD [18], and "ours+". Since the "global," "uniform,", and "ERK" methods do not specify a method for calculating the loss, we omit them. Recall that "ours+" represents

combining our method with DI for EfficientNet B0, and represents combining our method with AKD for all other networks. We show final-epoch accuracies and losses for jobs that have a final-epoch accuracy of at least 5%. We ignore jobs with lower accuracy for better visualization, because some loss functions are unbounded for models that achieve very poor accuracy. Note that, when reporting the loss for AKD, we again ignore the generator loss. As before, we normalize losses to fall in $[0, 1]$, and we normalize model accuracies by the accuracy of the original floating-point model from which we reload.

We find our method typically exhibits a strong loss/accuracy correlation. The noisiest example is ResNet18, in which some models achieve poor accuracy at relatively low loss. With our method, we are also always able to reproduce near-original model accuracy at a low loss. The AKD, DI, and "ours+" methods generally show a trend of higher accuracy when loss is lower, though some exceptions break the pattern. For MobileNetV1 1.0 and MobileNetV1 0.5, the "ours+" method never produces low-accuracy solutions, so increased loss does not show a decrease in accuracy, simply because low-accuracy solutions were not found.

Appendix B

LEARNING COMPRESSIBLE SUBSPACES

B.1 Overhead Calculation

Unstructured Sparsity: In this setting, we prune individual network weights. As such, the maximum number of compressed network configurations is determined by the layer with the largest number of parameters. Let L denote the number of parameters in a given model's largest layer and $[0, s]$ a compression sparsity range. Then the maximum number of configurations is given by $\lfloor Ls \rfloor$. In this setting, the number of pre-calibrated BatchNorm parameters that we need to store do not vary for each configuration. Consequently, the total number of additional parameters that must be stored is $B(L-1)$ where B is the total number of BatchNorm parameters in the uncompressed model (note that we subtract 1 from L since storing an uncompressed model requires storing one set of BatchNorm parameters anyway). Hence, enabling compression at every possible configuration would incur a total overhead of

$$\left(\frac{100B(L-1)}{T}\right)\%, \tag{B.1}$$

where T is the total number of model parameters.

Structured Sparsity: Let M denote the width of the widest layer of a given model, $[w, 1]$ a compression width factor range, and B the total number of BatchNorm parameters. Then $L = \lfloor M(1-w) \rfloor$ gives the maximum number of compressed network configurations obtainable after channel pruning. For each $\ell \in \{1, \ldots, L\}$, pruning ℓ channels from the widest layer corresponds to compressing the network with a width factor of $1 - \ell/M$. Hence, for each compressed network configuration, storing pre-calibrated BatchNorm statistics would require storing an additional $B(1 - \ell/M)$ parameters. To enable compression at every pos-

sible configuration would therefore require storing a total of $\sum_{\ell=1}^{L} B(1-\ell/M)$ additional parameters, incurring a total overhead of

$$\left(\frac{100B}{T}\sum_{\ell=1}^{L}(1-\ell/M)\right)\%, \tag{B.2}$$

where T is the total number of model parameters.

B.2 Further BatchNorm Analysis

In Figure 4.3, we analyzed the inaccuracies of BatchNorm statistics for models trained with unstructured sparsity. We show a similar analysis for the case of Universal Slimming (US) [153] and Network Slimming (NS) [154] in Figure B.1. We show the case of quantization in Figure B.2.

B.3 Linear Subspace Analysis

In Figure B.3, we provide additional experimental evidence that our linear subspace method (LCS+L) trains a subspace specialized for high-accuracy at one end and high-efficiency at the other end. We plot the validation accuracy along our subspace, as well as the validation accuracy along our subspace when compressing with $\tilde{f}(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha)) \equiv f(\boldsymbol{\omega}^*(\alpha), \gamma(1-\alpha))$. In other words, the weights that were trained for low compression levels are evaluated with high compression levels, and vice versa. We see that this leads to a large drop in accuracy, confirming that our method has conditioned one side of the line to achieve high accuracy at high sparsities, and the other side of the line to achieve high accuracy at low sparsities.

B.4 Global Model Details

Our CNNs warm up to an initial learning rate of 0.1 (0.045 for MobileNetV2) over 5 epochs, which then decays to 0 over 85 epochs (or 195 for cPreResNet20) using a cosine schedule. We use a batch size of 128 on a single GPU for cPreResNet20 and ResNet18. We use the version of VGG19 provided by [82]. This implementation modifies VGG19 slightly by adding BatchNorm layers and removing the last two fully connected layers. For VGG19, we use a

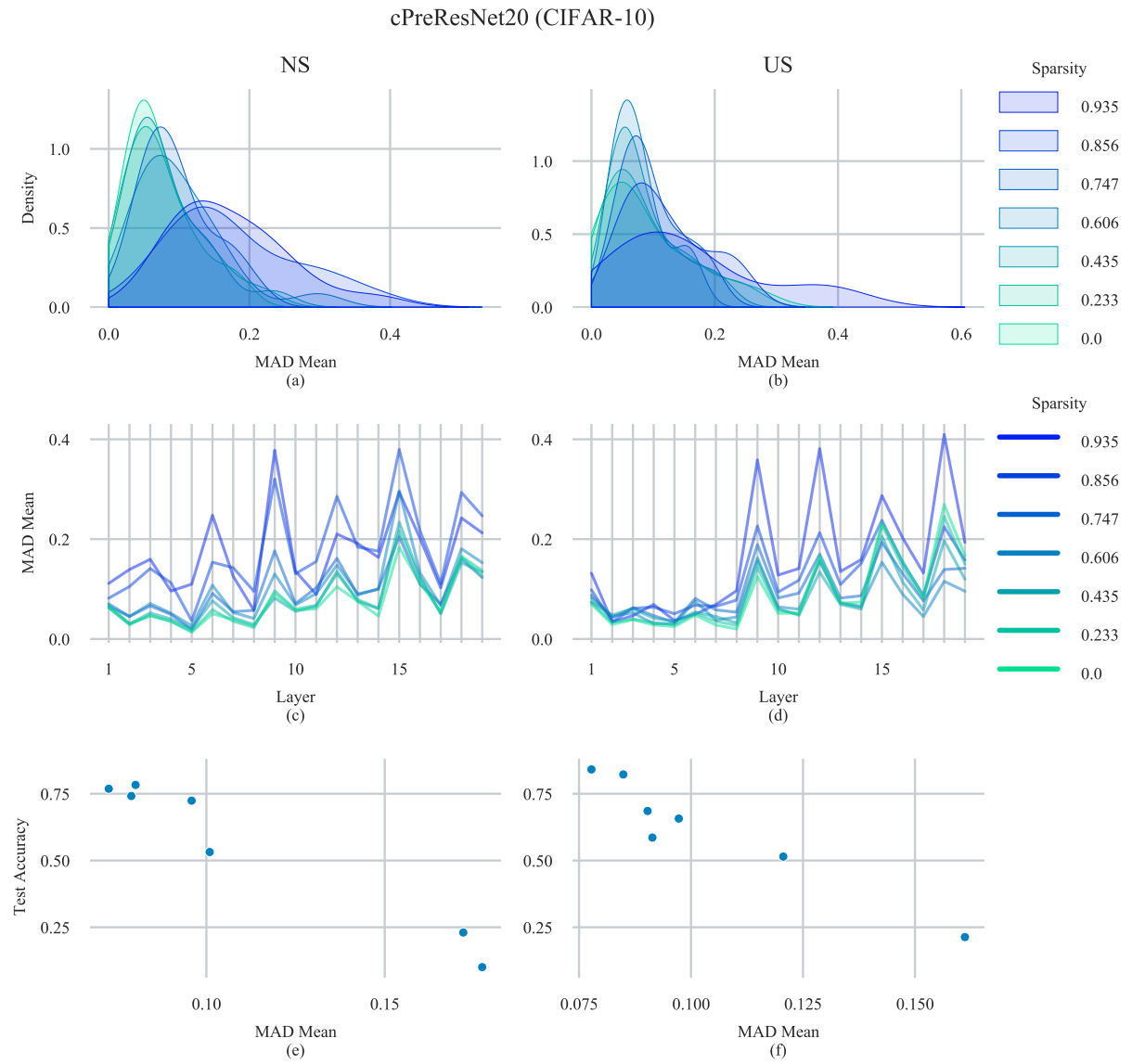


Figure B.1: Analysis of the mean absolute difference between observed batch-wise means $\hat{\boldsymbol{\mu}}$ and stored BatchNorm means $\boldsymbol{\mu}$ during testing for cPreResNet models trained with NS [154] or US [153]. (a)-(b): The distribution of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ across all layers. (c)-(d): The average value of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ for each individual BatchNorm layer. (e)-(f): The correlation between the average of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ and test set error

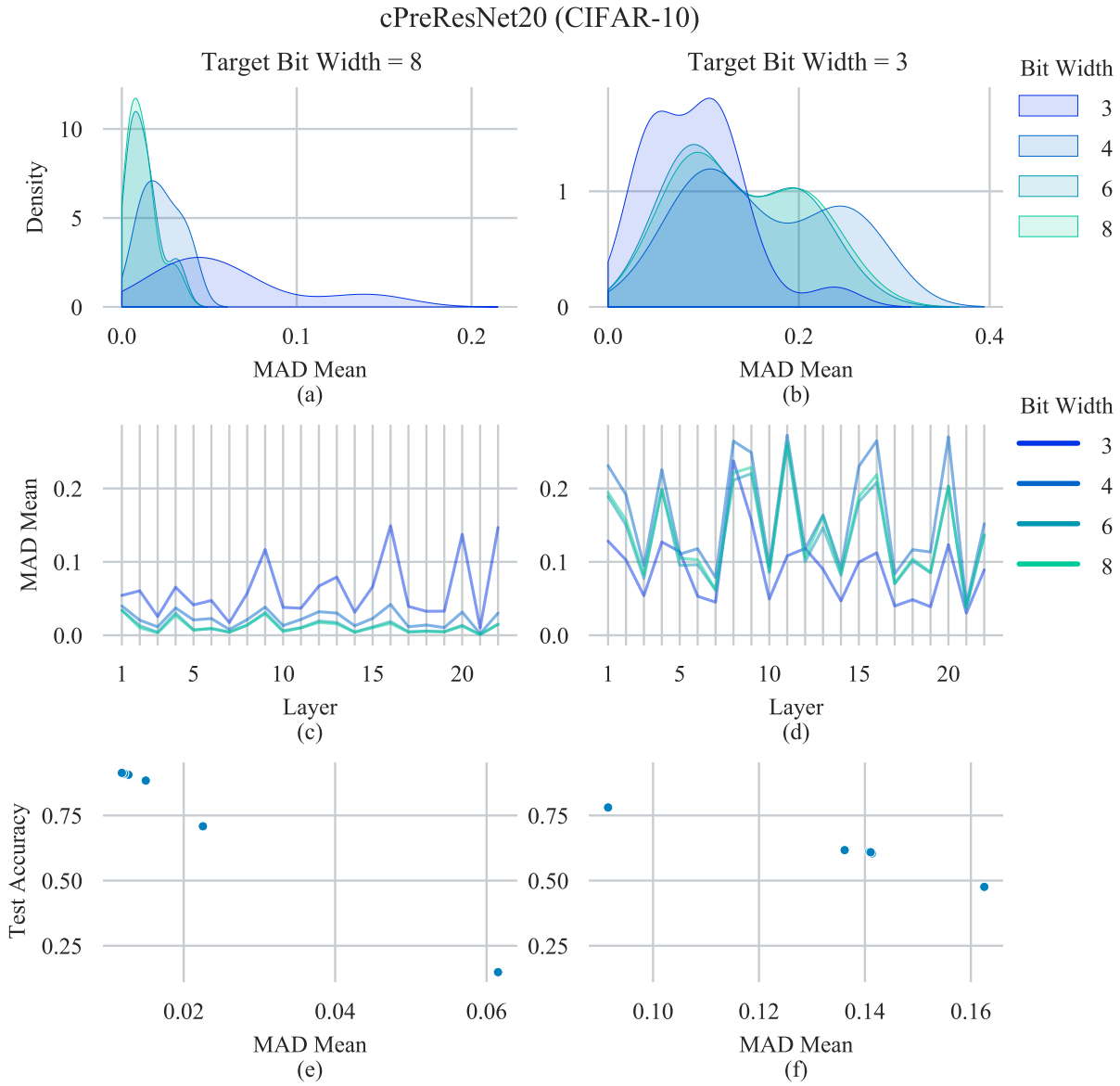


Figure B.2: Analysis of the mean absolute difference between observed batch-wise means $\hat{\boldsymbol{\mu}}$ and stored BatchNorm means $\boldsymbol{\mu}$ during testing for cPreResNet models trained with different quantization bit widths. (a)-(b): The distribution of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ across all layers. (c)-(d): The average value of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ for each individual BatchNorm layer. (e)-(f): The correlation between the average of $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$ and test set error

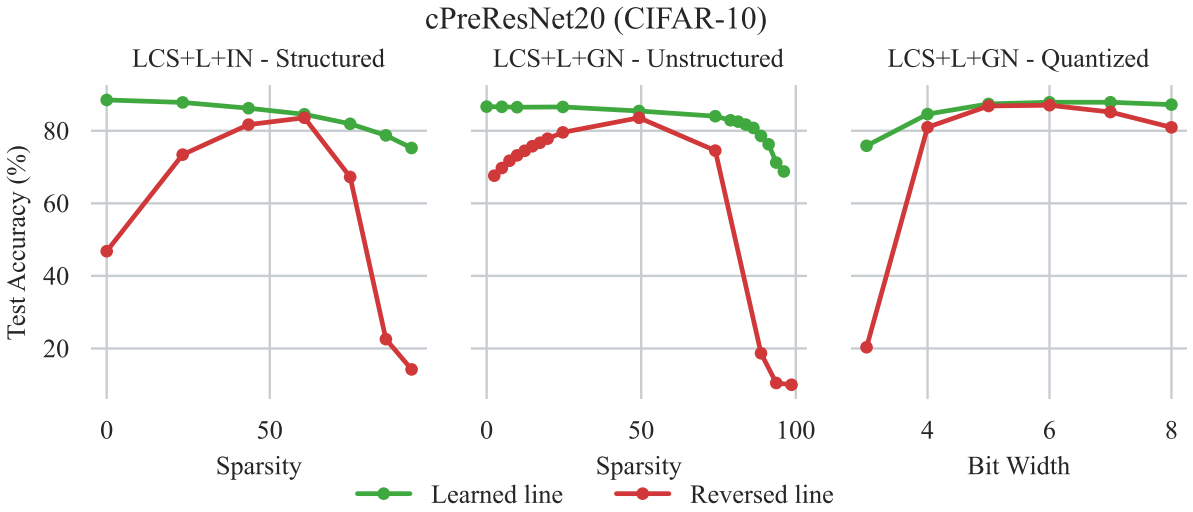


Figure B.3: Standard evaluation of a linear subspace with network $f(\boldsymbol{\omega}^*(\alpha), \gamma(\alpha))$ (Learned line), and evaluation when evaluating with reversed compression levels, $f(\boldsymbol{\omega}^*(\alpha), \gamma(1-\alpha))$ (Reversed line)

batch size of 256 with 4 GPUs. We train MnasNet, MobileNetV2, MobileNetV3-Small, and MobileNetV3-Large with a batch size of 128 using two GPUs. We train transformer models using a batch size of 1024 with 8 GPUs. For cPreResNet20, VGG19, and ResNet18, we use a weight decay of 5×10^{-4}. For MobileNetV2, we use a weight decay of 4×10^{-5}, and 10^{-5} for MnasNet, MobileNetV3-Small, and MobileNetV3-Large.

B.5 Unstructured Sparsity Details

It is typical to include a warmup phase when training models with TopK sparsity [156]. In our baselines in Section 4.4.1, we increase the sparsity level from 0% to its final value over the first 80% of training epochs. For our method, sparsity values fall within a range, so there is no single target sparsity value to warm up to.

For our point method (LCS+P), we simply train for the first 80% of training with the lowest sparsity value in our sparsity range. We finish training by sampling uniformly between the lowest and highest sparsity levels.

Table B.1: Our baseline models' (with BatchNorm) accuracies. cPreResNet20 is trained on CIFAR-10 and all other models on ImageNet.

Model	BatchNorm Baseline Accuracy (%)
cPreReseNet20	91.69
ResNet18	70.72
VGG19	62.21
MnasNet-B1	72.58
MobileNetV2	70.03
MobileNetV3-Small	66.5
MobileNetV3-Large	73.09
DeiT-Ti	72.7
DeiT-Ti + Distillation	73.1
DeiT-S	80.3
CaiT-XXS	76.02

For our line method (LCS+L), our choice of sparsity level is tied to our choice of weight-space parameters through α. We implement our warmup by simply adjusting $\gamma(\alpha)$ to apply less sparsity early in training, warming up to our final sparsity rates over the first 80% of training.

For transformer models in particular, our LCS+P training resembles our LCS+L method whereby $\gamma(\alpha)$ applies less sparsity early in training and gradually warms up to our final sparsity rates over a fraction of the training epochs. Moreover, we do not apply sparsity to the patch embedding layer.

In detail, let $\alpha_{\min}$ and $\alpha_{\max}$ correspond to our minimum and maximum alpha values (for example, for ResNet18 in Section 4.4.1, $\alpha_{\min} = 0.005$, and $\alpha_{\max} = 0.05$). As motivated in Section 4.3.3, we bias sampling of $\boldsymbol{\alpha}$ towards the endpoints of our line. We set $\boldsymbol{\alpha} = [\alpha_{\min}]$

with 25% probability, $\boldsymbol{\alpha} = [\alpha_{\max}]$ with 25% probability, and $\boldsymbol{\alpha} = [U(\alpha_{\min}, \alpha_{\max})]$ with 50% probability, where $U(a, b)$ samples uniformly in the range $[a, b]$. To warm up our sparsity rates, we choose

$$d = \max(1 - c/t, 0) \tag{B.3}$$

$$\gamma(\alpha) = (1 - \alpha)(1 - d), \tag{B.4}$$

where c is the current iteration number, and t is the total number of iterations in the first 80% of training. At the beginning of training, $d = 1$, and $\gamma(\alpha) = 0$, corresponding to a sparsity level of 0 for all values of α. Once 80% of training is finished, $d = 0$, and $\gamma(\alpha) = 1 - \alpha$ for the remainder of training. This corresponds to our final sparsity range.

Our methods use GroupNorm in the unstructured setting. For cPreResNet20, ResNet18, and VGG19, we set the number of groups to $g = 32$ (we set $g = c$ for the first few layers of cPreResNet20, because it has fewer than 32 channels). For MnasNet, MobileNetV2, MobileNetV3-Small, and MobileNetV3-Large, we use $g = 8$. Transformer models use LayerNorm (equivalent to $g = 1$).

B.6 Structured Sparsity Details

When training our method with a line in the structured sparsity setting, we do not use two sets of weights (e.g. $\boldsymbol{\omega}_1$ and $\boldsymbol{\omega}_2$, Section 4.3.1) for convolutional filters. Instead, we only use two sets of weights for affine transforms in GroupNorm [146] layers. For the convolutional filters, we instead use a single set of weights, similar to our point formulation (and similar to US [153] and NS [154]). By contrast, we use two sets of weights for convolutional filters as well as for affine transforms when experimenting with unstructured sparsity (Section 4.4.1) and quantization (Section 4.4.3).

The reason for only using one set of convolutional filters in the structured sparsity setting is that the filters themselves are able to specialize, even without an extra copy of network weights. Some filters are only used in larger networks, so they can learn to identify different signals than the filters used in all subnetworks. Note that this filter specialization argument

does not apply to our unstructured or quantized settings.

In preliminary experiments, we found that using a single set of weights for convolutions in our structured sparsity experiments gave a slight improvement over using two sets of weights (roughly 2% for cPreResNet20 [44] on CIFAR-10 [68]). We hypothesize that this slight difference may be attributed to the ease of learning fewer network parameters.

B.7 Quantization Details

Our method applied to quantization trains without quantizing the activations for the first 80% of training, and then adds activation quantization for the remainder of training. Weights are quantized throughout training.

The number of groups in our GroupNorm layers is the same as described in Appendix B.5.

B.8 Additional Results

Unstructured Sparsity: We present additional results in the unstructured wide sparsity regime in Figure B.4 and Figure B.5. For MnasNet, our α range is $[0.0325, 1]$. For the remaining models, our α range is $[0.025, 1]$. All other training details are unchanged. We find that our method is able to produce a higher accuracy over a wider range of sparsities than our baselines.

We also present results for vision transformer models in the high sparsity regime in Figure B.6. For these models, our α range is $[0.05, 0.2]$. Additionally, for these models we use a 65% warm-up phase instead of 80% as was discussed in Appendix B.5.

We also provide a table of unstructured sparsity results in the high sparsity regime (Table B.2) and the wider sparsity regime (Table B.3), showing memory usage and FLOPS.

Structured Sparsity: We present results for VGG19 on ImageNet in the structured sparsity setting in Figure B.7. We use a width factor range of $[0.25, 1]$. Our method produces a better efficiency-accuracy trade-off than baselines. Note that VGG19 with GroupNorm [146] had a higher baseline accuracy than VGG19 with BatchNorm [60] (as noted in Section 4.4.3).

Additional results on lightweight networks in the structured sparsity setting are presented in Figure B.8. Our width factor ranges for MobileNetV2, MobileNetV3-Small, and MobileNetV3-Large are $[0.55, 1]$, $[0.57, 1]$, and $[0.4427, 1]$, respectively. Our method yields a better accuracy-efficiency trade-off for a wider range of sparsity levels. We also provide a table demonstrating the memory, flops, and runtime of structured sparsity models in Table B.4.

Quantization: We present quantization results for ResNet18 in Figure B.9. We find that our models approach the accuracy of models trained at a single bit width, and our models generalize better to other bit widths.

We also provide a table of quantization results in Table B.5, showing memory usage of the models.

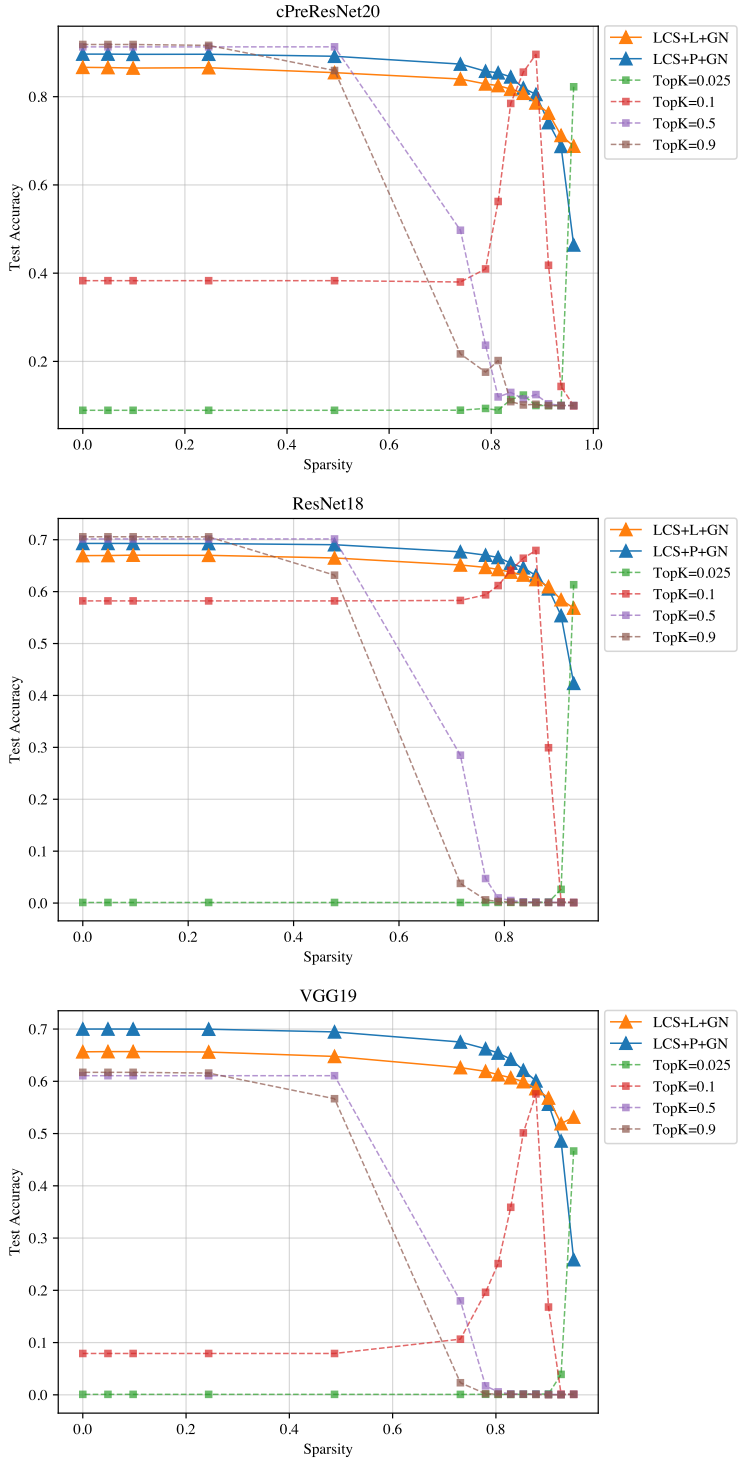


Figure B.4: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target

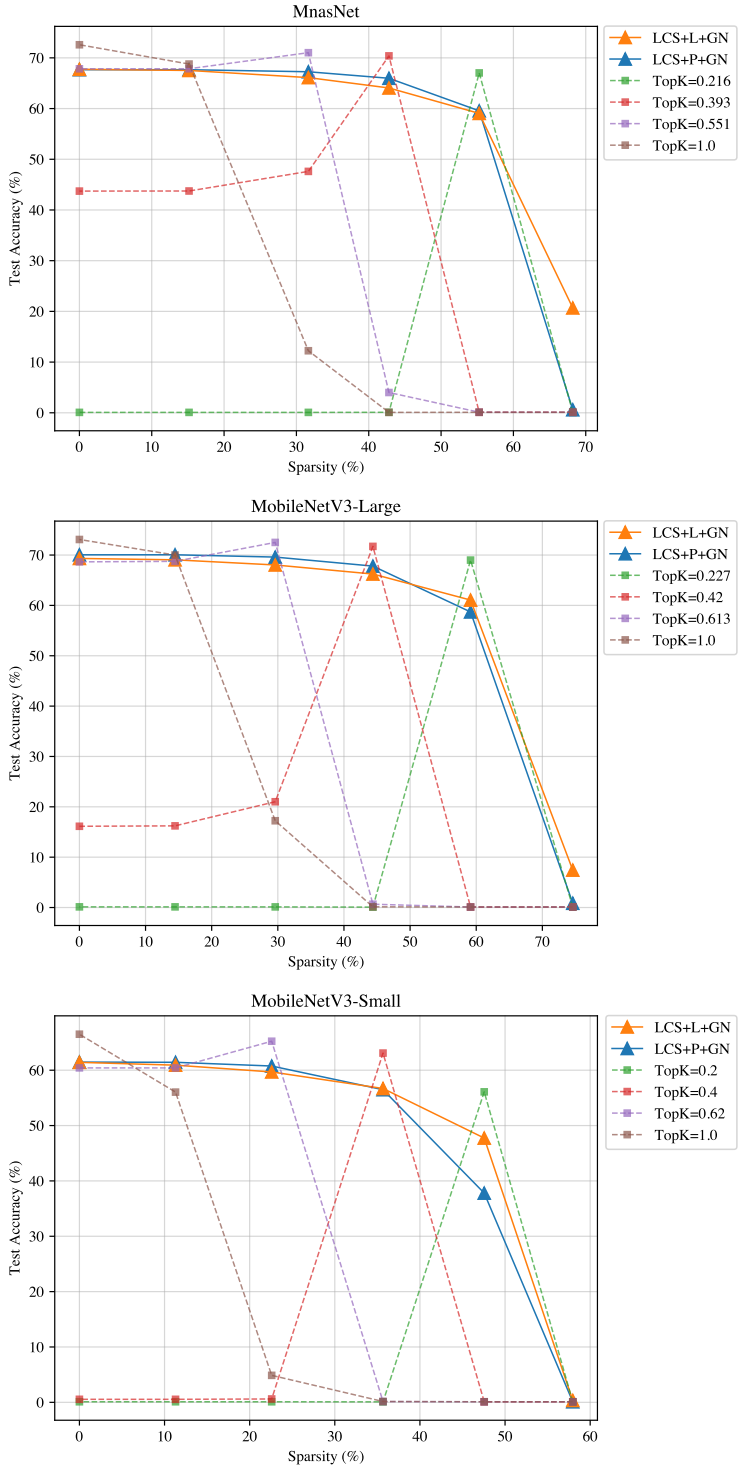


Figure B.5: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target

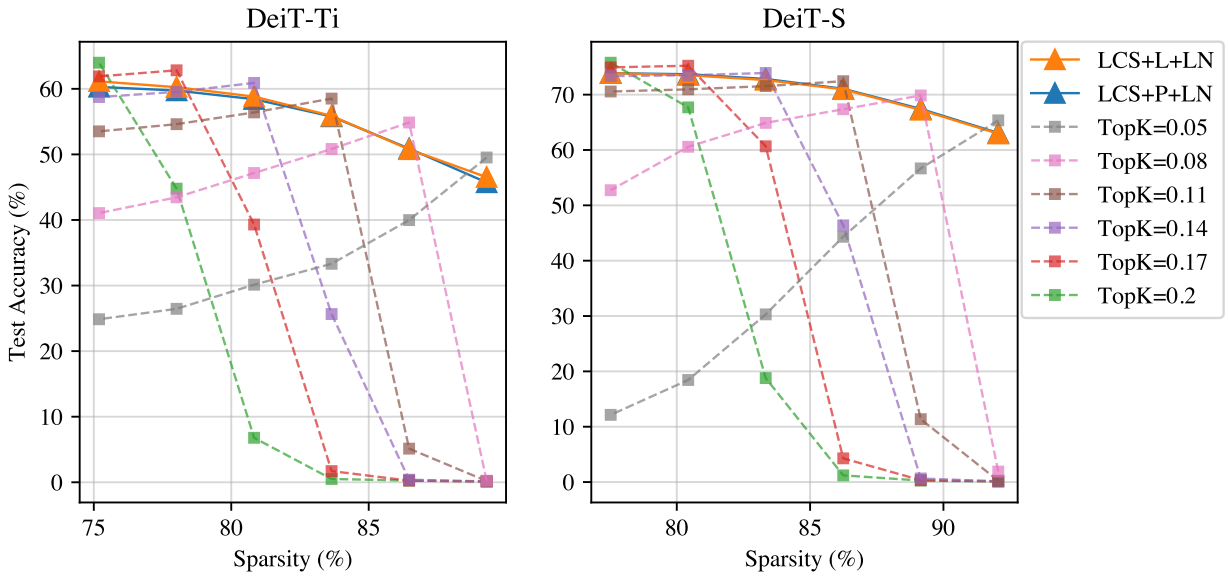


Figure B.6: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target

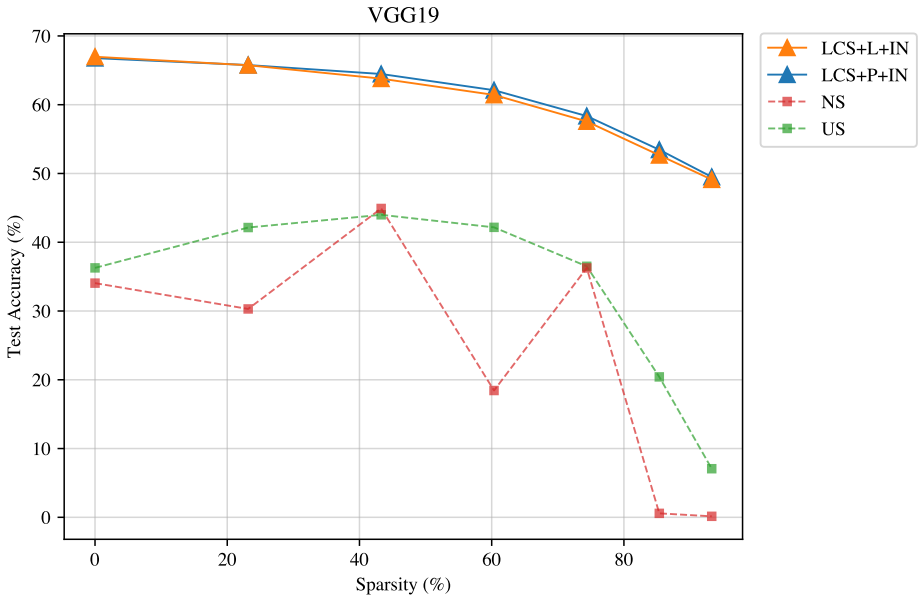


Figure B.7: Our method for structured sparsity using a linear subspace (LCS+L+IN) and a point subspace (LCS+P+IN), compared to Universal Slimming (US) [153] and Network Slimming (NS) [154]

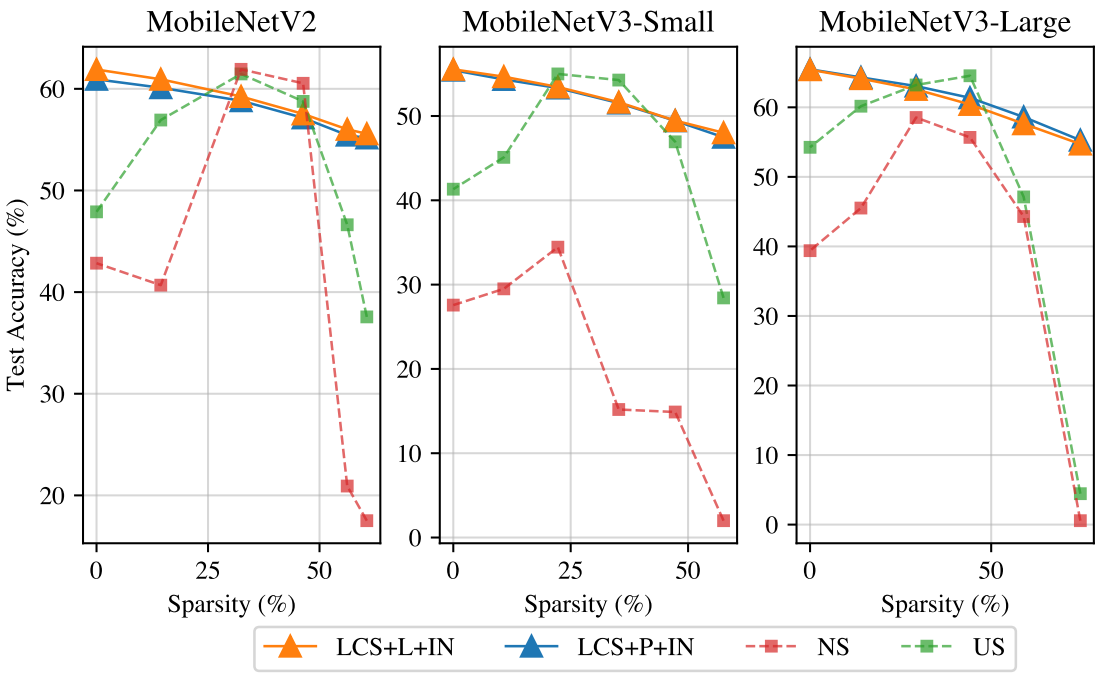


Figure B.8: Our method for structured sparsity using a linear subspace (LCS+L+IN) and a point subspace (LCS+P+IN), compared to Universal Slimming (US) [153] and Network Slimming (NS) [154]

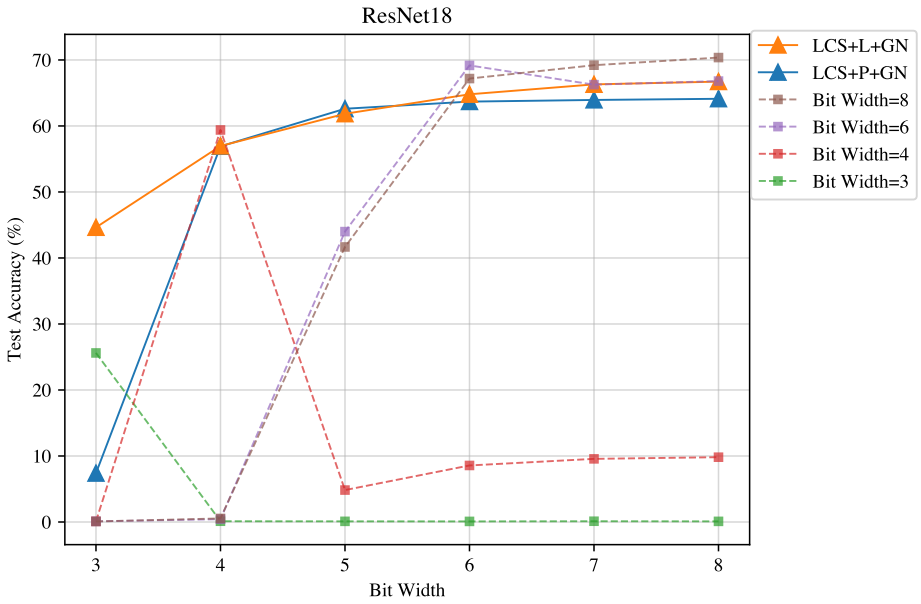


Figure B.9: Our method for quantization using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular bit width target

Table B.2: Results for unstructured sparsity in the high sparsity regime. Note that models of a particular architecture and sparsity level all have the same runtime characteristics (memory and FLOPS), so we only report one value. Runtime was not measured because it requires specialized hardware. So, we follow the standard practice of only reporting memory and flops. Memory consumption refers to the size of *nonzero* model weights in the currently executing model

cPreResNet20 (CIFAR-10)	Sparsity (%)	95.66	96.15	96.64	97.14	97.63	98.12
	FLOPS ($\times 10^6$)	1.46	1.29	1.13	0.96	0.79	0.63
	Memory (MB)	0.04	0.03	0.03	0.02	0.02	0.02
	Acc (LCS+P+GN)	70.18	69.22	67.74	**63.78**	54.91	24.92
	Acc (LCS+L+GN)	**75.53**	**72.30**	67.02	52.86	39.63	34.12
	Acc (TopK=0.04)	14.83	12.83	10.8	9.66	10.01	9.79
	Acc (TopK=0.02)	10.25	10.53	**78.7**	10.56	10.05	10.09
	Acc (TopK=0.01)	10.02	9.97	9.96	10.64	**59.2**	10.79
	Acc (TopK=0.005)	10.0	10.08	9.81	10.03	10.0	**41.44**
ResNet18 (ImageNet)	Sparsity (%)	92.67	93.15	93.62	94.1	94.58	95.06
	FLOPS ($\times 10^6$)	169.42	160.94	152.46	143.98	135.51	127.03
	Memory (MB)	3.42	3.2	2.98	2.76	2.53	2.31
	Acc (LCS+P+GN)	51.5	51.1	50.37	48.62	44.8	30.69
	Acc (LCS+L+GN)	**58.63**	**56.96**	54.70	**51.02**	44.87	39.41
	Acc (TopK=0.04)	5.96	0.9	0.18	0.11	0.1	0.1
	Acc (TopK=0.02)	24.66	45.37	**59.92**	3.84	0.1	0.11
	Acc (TopK=0.01)	0.12	0.1	0.1	0.11	**53.95**	0.11
	Acc (TopK=0.005)	0.1	0.12	0.11	0.1	0.11	**46.35**

Table B.3: Results for unstructured sparsity in the wide sparsity regime. Note that models of a particular architecture and sparsity level all have the same runtime characteristics (memory and FLOPS), so we only report one value. Runtime was not measured, because it requires specialized hardware (so most unstructured pruning works report memory and flops). Memory consumption refers to the size of *nonzero* model weights in the currently executing model

cPreResNet20 (CIFAR-10)	Sparsity (%)	0.0	49.31	86.29	91.22	93.68	96.15
	FLOPS ($\times 10^6$)	33.75	17.11	4.62	2.96	2.13	1.29
	Memory (MB)	0.87	0.44	0.12	0.08	0.05	0.03
	Acc (LCS+P+GN)	89.64	89.34	**82.34**	75.24	67.55	47.1
	Acc (LCS+L+GN)	86.65	85.45	80.78	**76.27**	**71.22**	68.78
	Acc (TopK=0.9)	**91.66**	83.17	10.54	10.0	9.76	10.0
	Acc (TopK=0.5)	91.16	**91.17**	10.64	10.14	10.0	10.0
	Acc (TopK=0.1)	40.74	40.74	78.56	39.54	11.67	10.26
	Acc (TopK=0.025)	9.64	9.64	10.65	10.0	9.98	**82.15**
ResNet18 (ImageNet)	Sparsity (%)	0.0	47.77	83.59	88.37	90.76	93.15
	FLOPS ($\times 10^6$)	1814.1	966.32	330.49	245.72	203.33	160.94
	Memory (MB)	46.72	24.4	7.66	5.43	4.32	3.2
	Acc (LCS+P+GN)	69.25	69.12	64.53	60.36	55.58	41.48
	Acc (LCS+L+GN)	66.94	66.49	63.20	**60.96**	**58.44**	56.83
	Acc (TopK=0.9)	**70.57**	63.17	0.1	0.1	0.1	0.1
	Acc (TopK=0.5)	70.15	**70.15**	0.24	0.17	0.1	0.12
	Acc (TopK=0.1)	58.21	58.21	**66.44**	29.93	0.24	0.1
	Acc (TopK=0.025)	0.12	0.12	0.13	0.17	2.64	**61.33**
VGG19 (ImageNet)	Sparsity (%)	0.0	48.75	85.31	90.19	92.62	95.06
	FLOPS ($\times 10^6$)	19533.52	9822.65	2539.51	1568.42	1082.88	597.34
	Memory (MB)	82.12	42.09	12.06	8.06	6.06	4.06
	Acc (LCS+P+GN)	**70.0**	**69.45**	**62.11**	55.62	48.58	25.87
	Acc (LCS+L+GN)	65.64	64.76	59.93	**56.80**	**51.89**	**53.15**
	Acc (TopK=0.9)	61.72	56.67	0.1	0.0	0.0	0.1
	Acc (TopK=0.5)	61.07	61.07	0.15	0.09	0.06	0.1
	Acc (TopK=0.1)	7.91	7.91	50.13	16.8	0.13	0.09
	Acc (TopK=0.025)	0.09	0.09	0.1	0.16	3.91	46.66

Table B.4: Results for structured sparsity. Note that models of a particular architecture and sparsity level all have the same runtime characteristics (memory, FLOPS, and runtime), so we only report one value. Runtime was measured on a MacBook Pro (16-inch, 2019) with a 2.6 GHz 6-Core Intel Core i7 processor and 16GB 2667 MHz DDR4 RAM. Memory consumption refers to the size of model weights in the currently executing model

cPreResNet20 (CIFAR-10)	Sparsity (%)	0	43.491	60.614	74.655	85.614	93.491
	FLOPS ($\times 10^6$)	33.75	19.07	13.29	8.55	4.85	2.2
	Memory (MB)	0.87	0.49	0.34	0.22	0.12	0.06
	Runtime (ms)	3.13	2.64	2.09	1.83	1.64	1.28
	Acc (LCS+P+IN)	87.51	86.07	84.46	**82.02**	78.39	**75.96**
	Acc (LCS+L+IN)	**88.49**	**86.22**	**84.54**	81.92	**78.73**	75.25
	Acc (US)	70.62	83.13	81.11	62.18	40.04	21.81
	Acc (NS)	72.87	75.46	57.09	70.07	16.86	19.76
ResNet18 (ImageNet)	Sparsity (%)	0.0	42.91	59.89	73.88	84.89	92.91
	FLOPS ($\times 10^6$)	1814.1	1042.66	736.42	483.16	282.89	135.61
	Memory (MB)	46.72	26.67	18.74	12.2	7.06	3.31
	Runtime (ms)	45.85	30.34	22.51	14.31	9.84	6.02
	Acc (LCS+P+IN)	63.32	60.21	**57.42**	**53.77**	**48.75**	**44.62**
	Acc (LCS+L+IN)	**63.93**	59.66	56.84	53.00	48.11	44.14
	Acc (US)	58.91	**60.39**	53.76	44.72	22.51	8.34
	Acc (NS)	50.63	57.58	22.93	48.52	0.84	1.34
VGG19 (ImageNet)	Sparsity (%)	0.0	43.28	60.35	74.37	85.35	93.28
	FLOPS ($\times 10^6$)	19533.52	11008.56	7656.48	4911.33	2773.1	1241.81
	Memory (MB)	82.12	46.58	32.56	21.04	12.03	5.52
	Runtime (ms)	388.49	246.81	172.64	105.77	60.0	29.55
	Acc (LCS+P+IN)	66.77	**64.47**	**62.11**	**58.35**	**53.45**	**49.5**
	Acc (LCS+L+IN)	**66.97**	63.79	61.42	57.57	52.66	49.11
	Acc (US)	36.27	43.99	42.17	36.5	20.42	7.07
	Acc (NS)	34.05	44.91	18.44	36.26	0.57	0.14

Table B.5: Results for quantization. Note that models of a particular architecture and quantization bit width all use the same memory, so we only report one value. Runtime was not measured, because it requires specialized hardware. Memory consumption refers to the size of model weights in the currently executing model

	Bit Widths	8	7	6	5	4	3
	Memory (MB)	0.22	0.19	0.17	0.14	0.11	0.08
	Acc (LCS+P+GN)	89.97	90.0	89.88	**89.26**	86.25	65.26
cPreResNet20	Acc (LCS+L+GN)	87.20	87.86	87.86	87.40	84.59	**75.86**
(CIFAR-10)	Acc (Bit Width=8)	**91.36**	**91.02**	90.47	87.98	65.91	16.65
	Acc (Bit Width=6)	91.07	90.89	**91.26**	87.12	63.1	18.78
	Acc (Bit Width=4)	84.93	84.65	84.77	82.37	**88.22**	25.19
	Acc (Bit Width=3)	55.71	55.56	57.01	55.66	44.83	73.89
	Memory (MB)	11.69	10.23	8.77	7.31	5.84	4.38
ResNet18	Acc (LCS+P+GN)	63.59	63.51	63.15	**61.82**	55.89	5.48
(ImageNet)	Acc (LCS+L+GN)	66.72	66.30	64.80	61.84	56.96	**44.63**
	Acc (Bit Width=8)	**70.36**	**69.2**	67.18	41.67	0.54	0.08
	Acc (Bit Width=6)	66.8	66.26	**69.17**	44.0	0.43	0.1
	Acc (Bit Width=4)	9.82	9.57	8.57	4.84	**59.39**	0.1
	Acc (Bit Width=3)	0.1	0.12	0.09	0.1	0.12	25.61
	Memory (MB)	20.542	17.974	15.406	12.839	10.271	7.703
VGG19	Acc (LCS+P+GN)	57.83	57.74	57.24	55.64	47.11	0.25
(ImageNet)	Acc (LCS+L+GN)	**64.38**	**63.89**	**61.95**	**57.72**	50.28	**31.85**
	Acc (Bit Width=8)	60.85	59.32	56.13	39.73	0.3	0.09
	Acc (Bit Width=6)	56.52	55.35	59.89	23.24	0.16	0.1
	Acc (Bit Width=4)	0.13	0.13	0.14	0.11	**51.33**	0.1
	Acc (Bit Width=3)	0.12	0.1	0.1	0.12	0.09	20.72