LCS: Learning Compressible Subspaces for Efficient, Adaptive, Real-Time Network Compression at Inference Time

Elvis Nunez[*†]
University of California, Los Angeles
elvis.nunez@ucla.edu

Maxwell Horton[*]
Apple
mchorton@apple.com

Anish Prabhu
Apple
anish_prabhu@apple.com

Anurag Ranjan
Apple
anuragr@apple.com

Ali Farhadi
Apple
afarhadi@apple.com

Mohammad Rastegari
Apple
mrastegari@apple.com

Abstract

When deploying deep neural networks (DNNs) 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. In this work, we present a training procedure to produce DNNs that can be compressed in real-time to arbitrary compression levels entirely on-device. This enables the deployment of a single model that can efficiently adapt to its host device's available resources. We formulate this problem as learning an adaptively compressible network subspace, where one end is optimized for accuracy, and the other for efficiency. Our subspace model requires no recalibration nor retraining when changing compression levels. Moreover, our generic training framework is amenable to multiple forms of compression. We present results 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. To our knowledge, our method is the first to enable adaptive on-device network compression with little to no computational overhead.*

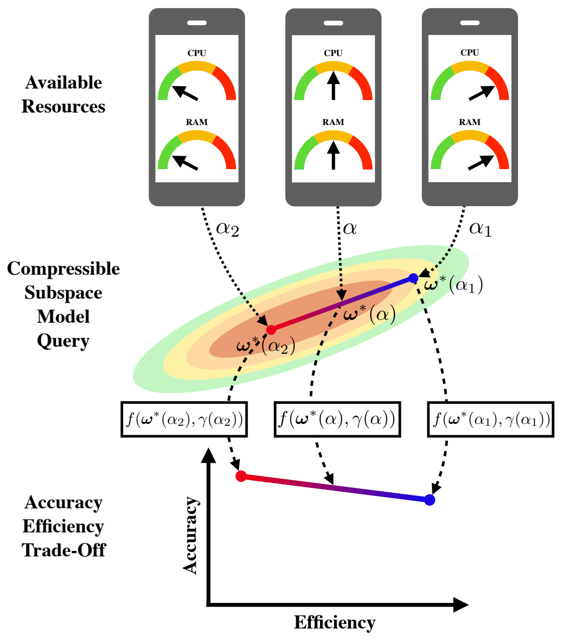
[*]Equal contribution. Correspondence to mchorton@apple.com.
[†]Work done during an internship at Apple.

1. Introduction

Deep neural network models are deployed to a variety of computing platforms, including phones, tablets, and watches [5]. 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 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 [10, 19, 9]. 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 [31] and Universal Slimming [30] 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 Batch-Norm [11] statistics to be recalibrated for every accuracy-efficiency configuration before deployment. This requires users to know every compression level in advance. If a



Available Resources
CPU
RAM
α_2
α
α_1
Compressible Subspace Model Query
$\omega^*(\alpha_1)$
$\omega^*(\alpha)$
$\omega^*(\alpha_2)$
$f(\omega^*(\alpha_2),\gamma(\alpha_2))$
$f(\omega^*(\alpha),\gamma(\alpha))$
$f(\omega^*(\alpha_1),\gamma(\alpha_1))$
Accuracy Efficiency Trade-Off
Accuracy
Efficiency

(a)

Learning Compressible Subspaces

```
Input: Network subspace $\boldsymbol{\omega}^*(\alpha)$, compression level calculator $\gamma(\alpha)$, compression function $f(\boldsymbol{\omega}, \gamma)$, stochastic sampling function $\boldsymbol{\alpha}_n \in [\alpha_1, \alpha_2]^n$, dataset $\mathcal{D}$, loss function $\mathcal{L}$.
Replace BatchNorm layers with GroupNorm.
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 1: (a) Depiction of our method for learning a linear subspace of networks $\boldsymbol{\omega}^*$ parameterized by $\alpha \in [\alpha_1, \alpha_2]$. Networks with $\alpha \approx \alpha_2$ exhibit high accuracy and low efficiency, while networks with $\alpha \approx \alpha_1$ trade off accuracy in favor of high efficiency. By varying $\alpha \in (\alpha_1, \alpha_2)$, we obtain a spectrum of networks which demonstrate an accuracy-efficiency trade-off. (b) Our training algorithm.

large number of compression levels are chosen, the storage burden of BatchNorm statistics is significant (Figure 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 aggressive compression of 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 among models.

We address the problem of training a model that can be compressed in real-time and on-device after deployment. Inspired by Wortsman et al. [27], we formulate this problem as learning an adaptively compressible network subspace (Figure 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. (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://github.com/apple/learning-compressible-subspaces` for research purposes.

2. Related Work

Architectures Demonstrating Accuracy-Efficiency Trade-Offs: Many popular network architectures include a hyperparameter to control the number of filters in each layer

Table 1: Our method with a linear subspace (LCS+L) and a point subspace (LCS+P), LEC [16], NS [31], and US [30]. 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 do not support arbitrary compression levels.

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

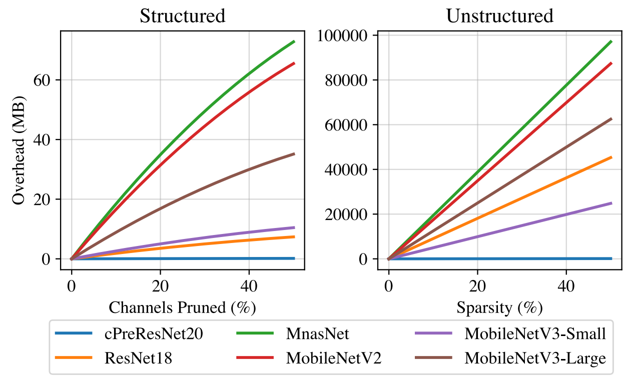


Figure 2: Parameter overhead in Megabytes (MB) for storing an extra set of pre-calibrated BatchNorm statistics for every possible sparsity configuration between 0% sparsity and the given compression level. Our method avoids this overhead by eliminating the need for storing BatchNorm statistics (Section 3.5). See Appendix A.2 for more details.

[10, 19, 9] or the number of blocks in the network [23]. In Once For All [3], 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 do not require deploying a set of weights for every compression level.

A method for post-training quantization to variable bit widths appears in [20], 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) [16], 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 [11] statistics at each sparsity level. These methods include Network Slimming (NS) [31] and Universal Slimming (US) [30]. Similar methods train a single network to perform well at various levels of quantization by storing extra copies of BatchNorm statistics [6, 29], or by recalibrating the BatchNorm statistics [13].

We differ from these methods by avoiding the need to recalibrate or store BatchNorm statistics (Section 3.5) and by allowing for adaptive selection of any compression level at inference time, neither of which have been done before (Table 1). Additionally, we avoid the overhead of storing BatchNorms at every desired compression level. Figure 2 demonstrates the substantial overhead of storing BatchNorm parameters for every possible compression level. Note that [31] avoids this storage overhead by only storing a few sets of BatchNorm statistics. However, this has the drawback of only allowing a few accuracy-efficiency configurations, which is problematic as explained in Section 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 [17], 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 [8], 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 [27] (though another formulation was introduced concurrently in [2]). 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.

3. Compressible Subspaces

3.1. Compressible Lines

Our method involves training a neural network subspace [27] that contains a spectrum of networks that each have a different accuracy-efficiency trade-off. We recast the subspace formulation of [27] to train a linear subspace with high-accuracy solutions at one end and high-efficiency 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 $\alpha \in [\alpha_1, \alpha_2]$, and $0 \leq \alpha_1 < \alpha_2 \leq 1$. 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, tuning $\alpha \in [\alpha_1, \alpha_2]$ allows us to move along our subspace, and we would like different points along our subspace to exhibit different accuracy-efficiency trade-offs. To achieve this, we introduce a function $\gamma(\alpha)$ which determines how much to compress the network at α, and a compression function $f(\boldsymbol{\omega}, \gamma)$ which performs the compression.

To train our subspace, we first sample a position in our subspace by randomly choosing some $\alpha \in [\alpha_1, \alpha_2]$, yielding a network with weights $\boldsymbol{\omega}^*(\alpha)$. We then compute $\gamma(\alpha)$, which determines how much to compress the network. Finally, f compresses the network, 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, backpropagating 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, as depicted in Figure 1(a). 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 A.1).

3.2. Compressible Points

In Section 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 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.

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 [30], 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 [30] in the structured sparsity setting, where we found the accuracy improvements to be more significant.

An overview of our overall algorithm is given in Figure 1. Next, we detail our compression methods f.

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 [32], which prunes a fraction γ of the weights with the smallest absolute value from each layer (we ignore the input and output layers). 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 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 $\gamma(\alpha) : [\alpha_1, \alpha_2] \rightarrow [a, b]$ 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 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 [30].

In quantization experiments, our compression function $f(\boldsymbol{\omega}, \gamma)$ is affine quantization as described in [12]. 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.

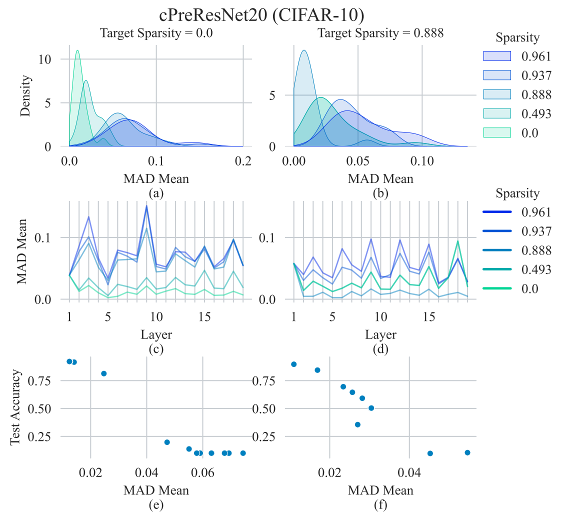


Figure 3: Analysis of observed batch-wise means $\hat{\boldsymbol{\mu}}$ and stored BatchNorm means $\boldsymbol{\mu}$ during testing for models trained with TopK unstructured sparsity. The models are trained with different target sparsities and evaluated with various inference-time sparsities. (a)-(b): 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.

3.5. Circumventing BatchNorm Recalibration

Previous works that train a compressible network require an additional training step to calibrate BatchNorm [11] statistics at each compression level [30, 31]. This precludes both methods from evaluating at arbitrarily fine-grained compression levels after deployment (Table 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 3, we analyze the inaccuracies of BatchNorm statistics for two models trained 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 [7] model on CIFAR-10 [14]. In Figure 3a and Figure 3b, we show the distribution of mean absolute differences (MAD), $|\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}|$, across all layers of the models. Models have lower Batch-

Norm 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 3c and Figure 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 3e and Figure 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 A.3 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 [28]. This computes an alternative normalization over g groups of channels rather than across a batch. It does not 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 [26] (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 A.6.

4. Experiments

We present results in the domains of unstructured sparsity, structured sparsity, and quantization. We train using Pytorch [18] on Nvidia GPUs. On CIFAR-10 [14], we experiment with the pre-activation version of ResNet20 [7] presented in the PyTorch version of the open-source code provided by [16]. We abbreviate it as "cPreResNet20."

We additionally experiment with a variety of architectures on the ImageNet [4] dataset. In particular, we present results using standard convolutional neural network (CNN) architectures: ResNet18 [7], and VGG19 [21]; lightweight CNNs: MnasNet-B1 [22], MobileNetV2 [19], MobileNetV3-Small, and MobileNetV3-Large [9]; and vision transformer models: DeiT-Ti, DeiT-S [24], and CaiT-XXS [25]. All models are trained using an input resolution of 224×224. Our baseline model accuracies are summarized in Appendix A.5.

We train cPreResNet20 for 200 epochs and ImageNet CNNs for 90 epochs. We follow hyperparameter choices in [27] for our methods and baselines (though we do not use the β regularization they describe), with a few architecture-dependent parameters detailed in Appendix A.5. For transformer models, we train for 300 epochs and follow the hyperparameter settings in [24]. Our baselines for each architecture always use the same training hyperparameters as our own methods.

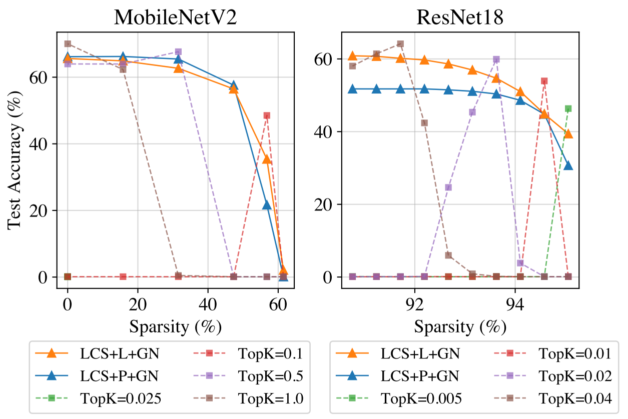


Figure 4: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target. The TopK target refers to the fraction of weights that remain unpruned during training.

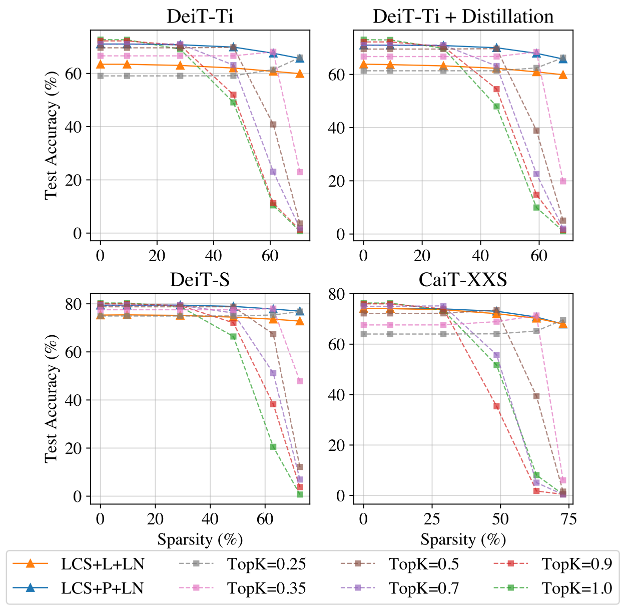


Figure 5: Our method for unstructured sparsity using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular TopK target.

4.1. Unstructured Sparsity

We present results for our method using MobileNetV2 and ResNet18 in Figure 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 accuracy-efficiency trade-off clearer).[1] Additional hyperparameter details are provided in Appendix A.6.

Our method achieves a strong accuracy-efficiency 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 that are trained to perform at a particular TopK sparsity level, and each network is evaluated at a variety of sparsity targets. These methods peak in accuracy near their target sparsity, but decrease sharply at higher sparsities.

We present additional results for our method using transformer architectures in Figure 5. Transformers contain LayerNorm [1] 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 accuracy-efficiency 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 A.9, we provide runtime characteristics of our models. We also present 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.

4.2. Structured Sparsity

We present results for our method using structured sparsity in Figure 6. For all structured sparsity experiments, we use an α range of $[0, 1]$. We use a width factor range of $[0.25, 1]$ for both VGG19 and ResNet18. As discussed in Section 3.5, we use a special case of GroupNorm [28] known as InstanceNorm [26] 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. We provide a table demonstrating the memory, flops, and runtime of structured sparsity models in Table 2. See Appendix A.7 for additional hyperparameter details.

[1] 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 $1 - \alpha$.

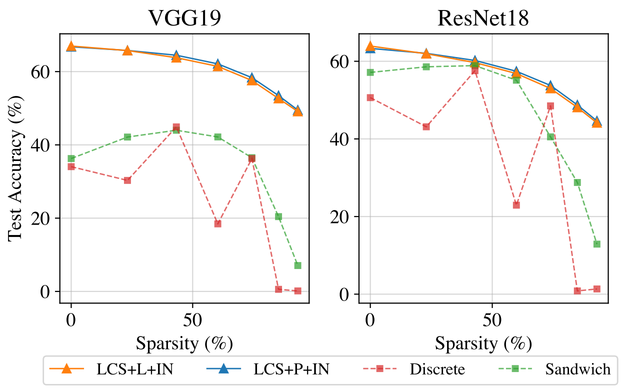


Figure 6: Our method for structured sparsity using a linear subspace (LCS+L+IN) and a point subspace (LCS+P+IN) compared to $Sandwich$ and $Discrete$.

To our knowledge, efficient, adaptive, *real-time* compression has not been explored before for structured sparsity. As such, we compare our methods to two baselines that train a network to operate at different sparsity levels. In the first method, which we denote $Discrete$, we train a network at four discrete width factors of $\{0.25, 0.5, 0.75, 1\}$. In the second method, which we denote $Sandwich$, we train a network using the sandwich rule (Section 3.3). At test time, both methods are evaluated at arbitrary sparsities. We do not perform BatchNorm calibration for either method. Note that our baselines are similar to NS and US, but our baselines operate at arbitrary width factors without BatchNorm calibration.

Models trained with our method demonstrate a strong accuracy-efficiency trade-off. By contrast, the trade-off produced by $Sandwich$ peaks in the middle. We hypothesize that this is due to the 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 $Discrete$ contains peaks and troughs. This method trains only at discrete width factors of $\{0.25, 0.50, 0.75, 1\}$ and produces stronger accuracies at these sparsities than at sparsities that it was not 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

Table 2: Runtime characteristics for structured sparsity. Note that models of a particular architecture and sparsity level all have the same 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 (*Sandwich*)	70.62	83.13	81.11	62.18	40.04	21.81
	Acc (*Discrete*)	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 (*Sandwich*)	58.91	**60.39**	53.76	44.72	22.51	8.34
	Acc (*Discrete*)	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 (*Sandwich*)	36.27	43.99	42.17	36.5	20.42	7.07
	Acc (*Discrete*)	34.05	44.91	18.44	36.26	0.57	0.14

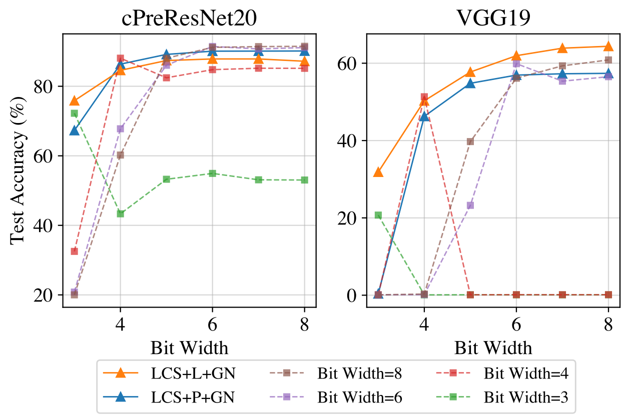


Figure 7: Our method for quantization using a linear subspace (LCS+L+GN) and a point subspace (LCS+P+GN) compared to networks trained for a particular bit width target.

in [15], but we leave more investigation to future work. See also Appendix A.9 for results using MnasNet, MobileNetV2, MobileNetV3-Small, and MobileNetV3-Large, as well as speed and memory usage characteristics.

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. As such, 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 and to verify the versatility of our method.

We present results for our method in Figure 7, comparing to baseline models trained at a fixed bit width and evaluated at a variety of bit widths. See Appendix A.8 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 [28]). See Appendix A.9 for ResNet18 results, and for memory usage characteristics of models.

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.

Our compressible subspaces yield several positive real-world impacts. Devices equipped with our models can dynamically adjust their energy consumption by efficiently compressing models according to the device's available resources. Additionally, our method circumvents the need for training multiple models tailored to multiple devices. Increasingly larger DNNs are inaccessible to older devices and devices with lower compute, and training models specific to each device's hardware constraints would be prohibitively expensive. Using our method, users can train a single model and efficiently compress it to a particular device's hardware constraints prior to deployment.

Acknowledgements

We thank Hanlin Goh and Barry Theobald for their helpful review feedback.

References

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

[2] Gregory Benton, Wesley Maddox, Sanae Lotfi, and Andrew 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Qing Jin, Linjie Yang, and Zhenyu Liao. Adabits: Neural network quantization with adaptive bit-widths. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2146–2156, 2020.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] Haichao Yu, Haoxiang Li, Humphrey Shi, Thomas S Huang, and Gang Hua. Any-precision deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 10763–10771, 2021.

[30] 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.

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

[32] 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.