Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization

Hesham Mostafa [1] **Xin Wang** [1 2]

Abstract

Modern deep neural networks are typically highly overparameterized. Pruning techniques are able to remove a significant fraction of network parameters with little loss in accuracy. Recently, techniques based on dynamic reallocation of non-zero parameters have emerged, allowing direct training of sparse networks without having to pre-train a large dense model. Here we present a novel dynamic sparse reparameterization method that addresses the limitations of previous techniques such as high computational cost and the need for manual configuration of the number of free parameters allocated to each layer. We evaluate the performance of dynamic reallocation methods in training deep convolutional networks and show that our method outperforms previous static and dynamic reparameterization methods, yielding the best accuracy for a fixed parameter budget, on par with accuracies obtained by iteratively pruning a pre-trained dense model. We further investigated the mechanisms underlying the superior generalization performance of the resultant sparse networks. We found that neither the structure, nor the initialization of the non-zero parameters were sufficient to explain the superior performance. Rather, effective learning crucially depended on the continuous exploration of the sparse network structure space during training. Our work suggests that exploring structural degrees of freedom during training is more effective than adding extra parameters to the network.

[1]Artificial Intelligence Products Group, Intel Corporation, San Diego, CA, USA. [2]Currently with Cerebras Systems, Los Altos, CA, USA, work done while at Intel Corporation. Correspondence to: Xin Wang <xin@cerebras.net>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

1. Introduction

Deep neural networks' success in a wide range of application domains, ranging from computer vision to machine translation to automatic speech recognition, stems from their ability to learn complex transformations by data examples while achieving superior generalization performance. Though they generalize well, deep networks learn more effectively when they are highly overparameterized (Brutzkus et al., 2017; Zhang et al., 2016). Emerging evidence has attributed this need for overparameterization to the geometry of the high-dimensional loss landscapes (Dauphin et al., 2014; Choromanska et al., 2014; Goodfellow et al., 2014; Im et al., 2016; Wu et al., 2017; Liao & Poggio, 2017; Cooper, 2018; Novak et al., 2018), and to the implicit regularization properties of stochastic gradient descent (SGD) (Brutzkus et al., 2017; Zhang et al., 2018a; Poggio et al., 2017), though a thorough theoretical understanding is not yet complete.

In practice, multiple techniques are able to compress large trained models, including distillation (Bucilua et al., 2006; Hinton et al., 2015), weight precision reduction (Hubara et al., 2016; McDonnell, 2018), low-rank decomposition (Jaderberg et al., 2014; Denil et al., 2013), and pruning (Han et al., 2015a; Zhang et al., 2018b). While these methods are highly effective in reducing the size of network parameters with little degradation in accuracy, they either operate on a pre-trained model or require the full overparameterized model to be stored and updated during, or at least at a certain stage of, training. Thus, training remains memory-inefficient despite the compact size of the resultant network produced by compression. The effectiveness of these compression methods, however, indicates the existence of compact network parameter configurations that are able to generalize on par with large networks. This raises a tantalizing hypothesis that overparameterization during training might not be a strict necessity and alternative training or reparameterization methods might exist to discover and train compact networks directly.

The problem of achieving training-time parameter efficiency is being approached in a number of ways. Most straightforward is to search for more parameter efficient network architectures. Innovations in this direction for deep convolutional neural networks (CNNs) include adoption of skip connections (He et al., 2015), replacement of fully-connected layers with global average pooling layers followed directly by the classifier layer (Lin et al., 2013), and depthwise separable convolutions (Sifre & Mallat, 2014; Howard et al., 2017). These modern CNN architectures drastically improved the accuracies achievable at a given parameter budget.

Instead of inventing new network architectures, an alternative approach is to reparameterize an existing model architecture, which is the approach we take in this work. In general, any *differentiable reparameterization* can be used to augment training of a given model. Let an original network (or a layer therein) be denoted by $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta} \in \boldsymbol{\Theta}$. Reparameterize it by $\boldsymbol{\phi} \in \boldsymbol{\Phi}$ and $\boldsymbol{\psi} \in \boldsymbol{\Psi}$ through $\boldsymbol{\theta} = g(\boldsymbol{\phi}; \boldsymbol{\psi})$, where g is differentiable w.r.t. $\boldsymbol{\phi}$ but not necessarily w.r.t. $\boldsymbol{\psi}$. Denote the reparameterized network by $f_{\boldsymbol{\psi}}$, considering $\boldsymbol{\psi}$ as *metaparameters* [*]:

对θ可微 求梯度!

$$\boldsymbol{y} = f\left(\boldsymbol{x}; g(\boldsymbol{\phi}; \boldsymbol{\psi})\right) \triangleq f_{\boldsymbol{\psi}}\left(\boldsymbol{x}; \boldsymbol{\phi}\right). \tag{1}$$

$f_{\boldsymbol{\psi}}$ is trained by backpropagating errors through g, as $\frac{\partial}{\partial \boldsymbol{\phi}} = \frac{\partial g}{\partial \boldsymbol{\phi}} \frac{\partial}{\partial g}$. If it is so chosen that $\dim(\boldsymbol{\Phi}) + \dim(\boldsymbol{\Psi}) < \dim(\boldsymbol{\Theta})$ and $f_{\boldsymbol{\psi}} \approx f$ in terms of generalization performance, then $f_{\boldsymbol{\psi}}$ is a more parameter efficient function approximator than f.

?

Sparse reparameterization is a special case where g is a linear projection; $\boldsymbol{\phi}$ is the non-zero entries (i.e. "weights") and $\boldsymbol{\psi}$ their indices (i.e. "connectivity") in the original parameter tensor $\boldsymbol{\theta}$. Likewise, *parameter sharing* is a similar special case of linear reparameterization where $\boldsymbol{\phi}$ is the tied parameters and $\boldsymbol{\psi}$ the indices at which each parameter is placed (with repetition) in the original parameter tensor $\boldsymbol{\theta}$. If metaparameters $\boldsymbol{\psi}$ are fixed during the course of training, the reparameterization is *static*, whereas if $\boldsymbol{\psi}$ is adjusted adaptively during training, we call it *dynamic* reparameterization.

In this paper, we investigate multiple static and dynamic reparameterizations of deep residual CNNs for efficient training. Inspired by previous techniques, we developed a novel dynamic reparameterization method that yielded the highest parameter efficiency in training sparse deep residual networks, outperforming existing static and dynamic reparameterization methods.

Our method dynamically changes the sparse structure of the network during training. Its superior performance suggests that, given a limited storage and computational budget for training a CNN, it is better to allocate part of the resources to describing and evolving the structure of the network, than to spend it entirely on the parameters of a dense network.

Furthermore, we show that the success of dynamic sparse reparameterization is not solely due to the final sparse structure of the resultant networks, nor to a combination of final structure and initial weight values. Rather, training-time structural exploration is necessary for best generalization, even if a high-performance structure and its initial values are known *a priori*. Thus, simultaneous exploration of network structure and parameter optimization through gradient descent are synergistic. Structural exploration improves the trainability of sparse deep CNNs.

2. Related work

Training of differentiably reparameterized networks has been proposed in numerous studies before.

Dense reparameterization Several dense reparameterization techniques sought to reduce the size of fully connected layers. These include low-rank decomposition (Denil et al., 2013), fastfood transform (Yang et al., 2014), ACDC transform (Moczulski et al., 2015), HashedNet (Chen et al., 2015), low displacement rank (Sindhwani et al., 2015) and block-circulant matrix parameterization (Treister et al., 2018).

Note that similar reparameterizations were also used to introduce certain algebraic properties to the parameters for purposes other than reducing model sizes, e.g. to make training more stable as in unitary evolution RNNs (Arjovsky et al., 2015) and in weight normalization (Salimans & Kingma, 2016), to inject inductive biases (Thomas et al., 2018), and to alter (Dinh et al., 2017) or to measure (Li et al., 2018) properties of the loss landscape. These dense reparameterization methods are static.

Sparse reparameterization Successful training of sparse reparameterized networks usually employs iterative pruning and retraining, e.g. (Han et al., 2015b; Narang et al., 2017; Zhu & Gupta, 2017) [†]. Training typically starts with a large pre-trained model and sparsity is gradually increased by pruning and fine-tuning. Training a small, static, and sparse model *de novo* fares worse than compressing a large dense model to the same sparsity (Zhu & Gupta, 2017).

fine-tuning

[*]We use the term *metaparameter* to refer to the parameters $\boldsymbol{\psi}$ of the reparameterization function g. They differ from parameters $\boldsymbol{\phi}$ in that they are not optimized through gradient descent, and they differ from hyperparameters in that they define meaningful features of the model which are required for inference.

[†]Note that these, as well as all other sparse techniques we benchmark against in this paper, impose *non-structured* sparsification on parameter tensors, yielding *sparse* models. There also exist a class of *structured* pruning methods that "sparsify" at channel or layer granularity, e.g. (Luo et al., 2017) and (Huang & Wang, 2017), generating essentially small *dense* models. We describe a full landscape of existing methods in Appendix D.

(Frankle & Carbin, 2018) identified small and sparse subnetworks post-training which, when trained in isolation, reached a similar accuracy as the enclosing big network. They further showed that these subnetworks were sensitive to initialization, and hypothesized that the role of overparameterization is to provide a large number of candidate subnetworks, thereby increasing the likelihood that one of these subnetworks will have the necessary structure and initialization needed for effective learning.

Most closely related to our work are dynamic sparse reparameterization techniques that emerged only recently. Like ours, these methods adaptively alter, by certain heuristic rules, the location of non-zero parameters during training. Sparse evolutionary training (SET) (Mocanu et al., 2018) used magnitude-based pruning and random growth at the end of each training epoch. NeST (Dai et al., 2017; 2018) iteratively grew and pruned parameters and neurons during training; parameter growth was guided by parameter gradient and pruning by parameter magnitude. Deep Rewiring (DeepR) (Bellec et al., 2017) combined dynamic sparse parameterization with stochastic parameter updates for training. These methods were primarily demonstrated with sparsification of fully-connected layers and applied to relatively small and shallow networks. They also required manual configuration of sparsity levels for different layers of the model. The method we propose in this paper is more scalable and computationally efficient than these previous approaches, while achieving better performance on deep CNNs.

3. Methods

We sparsely reparameterize the majority of layers in deep CNNs. All sparse parameter tensors are randomly initialized at the same sparsity (i.e. fraction of zeros). During training, free parameters are moved within and across weight tensors every few hundred training iterations, following a two-phase procedure (Algorithm 1): magnitude-based pruning followed by random growth. Throughout training, we always maintain the same total number of non-zero parameters in the network.

Our magnitude-based pruning is based on an adaptive global threshold H where all sparse weights with absolute values smaller than H are pruned. H is adjusted via a setpoint negative feedback loop to maintain approximately (with tolerance δ) a fixed number of pruned/grown parameters N_p during each reallocation step.

Immediately after removing K parameters during the pruning phase, K zero-initialized parameters are redistributed among the sparse parameter tensors, following a heuristic rule: layers having larger fractions of non-zero weights receive proportionally more free parameters (see Algorithm 1). Intuitively, one should allocate more parameters to layers such that training loss is more quickly reduced. This means, to the first order, free parameters should be redistributed to layers whose parameters receive larger loss gradients. If a layer has been heavily pruned, this indicates that, for a large portion of its parameters, the training loss gradients are not large or consistent enough to counteract the pull towards zero exerted by weight-decay regularization. This layer is therefore to receive a smaller share of new free parameters during the growth phase. The reallocated parameters are randomly placed at zero positions in the target weight tensors. To ensure the numbers of pruned and grown free parameters are exactly the same, we impose extra guards against rounding errors, as well as against special cases where more free parameters are allocated to a tensor than there are zero positions. For simplicity of exposition, we omit these minor details in Algorithm 1. See Appendix A for a more detailed description of the algorithm.

grow back rule

Our algorithm differs from *SET* (Mocanu et al., 2018) in two important aspects. First, instead of pruning a fixed fraction of weights at each reallocation step, 1 we use an adaptive threshold for pruning. Second, 2 we automatically reallocate parameters across layers during training and do not impose a fixed, manually configured, sparsity level on each layer. The first difference leads to reduced computational overhead as it obviates the need for sorting operations, and the second to better performing networks (see Section 4) and the ability to train extremely sparse networks as shown in Appendix F.

We evaluated our method on the deep residual CNNs listed in Table 1, and compared its performance against existing static and dynamic reparameterization methods[‡]. We did not experiment with AlexNet (Krizhevsky et al., 2012) and VGG-style networks (Simonyan & Zisserman, 2014) as their parameter efficiency is inferior to modern residual networks. Such a choice makes the improvement in parameter efficiency achieved by our dynamic sparse training method more practically relevant. Dynamic sparse reparameterization was applied to all weight tensors of convolutional layers (with the exception of downsampling convolutions and the first convolutional layer acting on the input image), while all biases and parameters of normalization layers were kept dense.

At a specific global sparsity[§] s, we compared our method (*dynamic sparse*) against six baselines:

1. *Full dense*: original large and dense model, with N free parameters;

[‡]Code is available at `https://github.com/IntelAI/dynamic-reparameterization`.

[§]Global sparsity s is defined as the overall sparsity of all sparse parameter tensors, not the entire model, which has a small fraction of dense parameters.

Algorithm 1: Reallocation of non-zero parameters within and across parameter tensors

```
1: for each sparse parameter tensor $\mathbf{W}_i$ do
2:     $(\mathbf{W}_i, k_i) \leftarrow$ prune_by_threshold($\mathbf{W}_i, H$)        ▷ $k_i$ is the number of pruned weights
3:     $l_i \leftarrow$ number_of_nonzero_entries($\mathbf{W}_i$)        ▷ Number of surviving weights after pruning
4: $(K, L) \leftarrow (\sum_i k_i, \sum_i l_i)$        ▷ Total number of pruned and surviving weights
5: $H \leftarrow$ adjust_pruning_threshold($H, K, \delta$)        ▷ Adjust pruning threshold
6: for each sparse parameter tensor $\mathbf{W}_i$ do
7:     $\mathbf{W}_i \leftarrow$ grow_back($\mathbf{W}_i, \frac{l_i}{L}K$)        ▷ Grow $\frac{l_i}{L}K$ zero-initialized weights at random in $\mathbf{W}_i$
```

2. *Thin dense*: original model with less wide layers, such that it had the same size as *dynamic sparse*;
3. *Static sparse*: original model initialized at sparsity level s with random sparseness pattern, then trained with connectivity (sparseness pattern) fixed;
4. *Compressed sparse*: sparse model obtained by iteratively pruning and re-retraining a large and dense pre-trained model to target sparsity s (Zhu & Gupta, 2017);
5. *DeepR*: sparse model trained by using Deep Rewiring (Bellec et al., 2017);
6. *SET*: sparse model trained by using Sparse Evolutionary Training (SET) (Mocanu et al., 2018).

Appendix C compares our method against an additional static (dense) parameterization method based on weight sharing: *HashedNet* (Chen et al., 2015).

Because sparse tensors require storage of both the free parameter values and their locations, we compare models that have the same size in descriptive length, instead of the same number of weights. While the number of bits needed to specify the connectivity is implementation dependent, we assume one bit is used for each position in the weight tensors indicating whether this position is zero or not. A sparse tensor is fully defined by this bit-mask, together with the non-zero entries. This scheme was previously used in CNN accelerators that natively operate on sparse structures (Aimar et al., 2018). For a network with N 32-bit weights in its dense form, a sparse version at sparsity s has a descriptive length of $(32s + 1)N$ bits, and is thus equivalent in size to a thinner dense network with $(s + \frac{1}{32})N$ weights. We use this formula to determine the parameter counts of the *thin dense* baseline, which has $\frac{N}{32}$ more weights than comparable sparse models.

A recent study (Liu et al., 2018) showed that training small networks *de novo* can almost always match the generalization performance obtained by post-training pruning of larger networks, so long as the small networks were trained for long enough. To address concerns that the superior performance of *dynamic sparse* might be matched by training *thin dense* or *static sparse* networks for more epochs, we always train *thin dense* and *static sparse* baselines for double the number of epochs used to train *dynamic sparse* models.

Note that *compressed sparse* is a compression method that first trains a large dense model and then iteratively prunes it down, whereas *dynamic sparse* and all other baselines maintain a constant small model size throughout training. For *compressed sparse*, we train the large dense model for the same number of epochs used for our *dynamic sparse*, and then iteratively prune and fine-tune it across additional training epochs. *Compressed sparse* thus trains for more epochs than *dynamic sparse*. See Appendix B for hyperparameters used for all experiments.

4. Experimental results

WRN-28-2 on CIFAR10 We ran experiments on a Wide Resnet model WRN-28-2 (Zagoruyko & Komodakis, 2016) trained to classify CIFAR10 images (see Appendix B for details of implementation). We varied global sparsity levels and evaluated test accuracy of different training methods based on dynamic and static reparameterization. As shown in Figure 1a, *static sparse* and *thin dense* significantly underperformed *compressed sparse* model as expected, whereas our *dynamic sparse* performed on par or slightly better than *compressed sparse* on average. *DeepR* significantly underperformed all other method. Though *SET* was generally on par with *compressed sparse* and *dynamic sparse* at low sparsity levels, it underperformed *dynamic sparse* at high sparsity levels. Even though the statically reparameterized models *static sparse* and *thin dense* were trained for twice the number of epochs, they still failed to reach the accuracy of *dynamic sparse* or *SET*. Note that *thin dense* had even more trainable free parameters than all sparse models (see Section 3).

Further, we inspected the layer-wise sparsity patterns that emerged from automatic parameter reallocation across layers (Algorithm 1) during *dynamic sparse* training. We observed consistent patterns at different sparsity levels: (a) larger parameter tensors tended to be sparser than smaller ones, and (b) deeper layers tended to be sparser than shallower ones. Figure 1b shows a breakdown of the final sparsity levels of different residual blocks at different sparsity levels.

Table 1: Datasets and models used in experiments

Dataset	CIFAR10	Imagenet
Model	WRN-28-2 (Zagoruyko & Komodakis, 2016)	Resnet-50 (He et al., 2015)
Architecture	C16/3×3 [C16/3×3,C16/3×3]×4 [C64/3×3,C64/3×3]×4 [C128/3×3,C128/3×3]×4 GlobalAvgPool, F10	C64/7×7-2, MaxPool/3×3-2 [C64/1×1, C64/3×3, C256/1×1]×3 [C128/1×1, C128/3×3, C512/1×1]×4 [C256/1×1, C256/3×3, C1024/1×1]×6 [C512/1×1, C512/3×3, C2048/1×1]×3 GlobalAvgPool, F1000
# Parameters	1.5M	25.6M

For brevity architecture specifications omit batch normalization and activations. Pre-activation batch normalization was used in all cases. Convolutional (C) layers are specified with output size and kernel size and Max pooling (MaxPool) layers with kernel size. Brackets enclose residual blocks postfixed with repetition numbers; the downsampling convolution in the first block of a scale group is implied.

Resnet-50 on Imagenet Next, we trained the Resnet-50 bottleneck architecture (He et al., 2015) on Imagenet (see Appendix B for details of implementation). We ran experiments at two global sparsity levels, 0.8 and 0.9 (Table 2). Models obtained by our (*dynamic sparse*) method outperformed all dynamic and static reparameterization baseline methods, slightly outperforming *compressed sparse* models obtained through post-training compression of a large dense model. In Table 2, we also list two additional representative methods of *structured* pruning (see Appendix D), *ThiNet* (Luo et al., 2017) and *Sparse Structure Selection* (Huang & Wang, 2017), which, consistent with recent criticisms (Liu et al., 2018), underperformed static dense baselines. Similar to *dynamic sparse* WRN-28-2, reliable sparsity patterns across parameter tensors in different layers emerged from dynamic parameter reallocation during training, displaying the same empirical trends described above (Figure 2).

Computational overhead of dynamic parameter reallocation We assessed the additional computational cost incurred by dynamic parameter reallocation steps (Algorithm 1) during training, and compared ours with existing dynamic sparse reparameterization techniques, *DeepR* and *SET* (Table 3). Because both *SET* and *dynamic sparse* reallocate parameters only intermittently (every few hundred training iterations), the computational overhead was negligible for the experiments presented here[¶]. *DeepR*, however, requires adding noise to gradient updates as well as reallocating parameters every training iteration, leading to a significantly larger overhead.

[¶]Because of the rather negligible overhead, the reduced operation count thanks to the elimination of sorting operations did not amount to a substantial improvement over SET on GPUs. Our method's another advantage over SET lies in its ability to produce better sparse models and to reallocate free parameters automatically (see Appendix F).

Understanding the effects of dynamic parameter reallocation Why did dynamic parameter reallocation yield sparse models that generalize better than static models trained *de novo*? To address this question, we investigated whether our method discovered more trainable sparse network structures, following the reasoning of the recently proposed "lottery ticket" hypothesis (Frankle & Carbin, 2018).

First, we did the following experiment with WRN-28-2 trained on CIFAR10: after training with *dynamic sparse* method, we retained the final network sparseness pattern (i.e. positions of non-zero entries in all sparse parameter tensors), and then randomly re-initialized this network and re-trained with its structure fixed (Figure 3a). It failed to reach comparable accuracy, suggesting that the sparse connectivity discovered by our method is not sufficient to explain the high generalization performance.

Next, we asked whether the particular weight initialization of the sparse network in addition to its sparseness structure led to high accuracies (Frankle & Carbin, 2018). We used the final network structure as described above, and re-initialized it with the exact same initial values used in the original training. As shown in Figure 3a, the combination of final structure and original initialization still fell significantly short of the level of accuracy achieved by training with dynamic parameter reallocation, not significantly different from training the same network with random re-initialization.

For control, we also show the performance of *static sparse* models where the sparse network structure and its initialization were both random (Figure 3a), which, not surprisingly, performed the worst. Similar trends were observed for Resnet-50 trained on Imagenet (Figure 3b). All static networks, in all sparseness pattern and re-initialization configurations, were trained for double the number of epochs used for dynamic training.

Table 2: Test accuracy% (top-1, top-5) of Resnet-50 trained on Imagenet

		Final overall sparsity (# Parameters)	0.8 (7.3M)		0.9 (5.1M)		0.0 (25.6M)	
Reparameterization	Static	*Thin dense*	72.4 [-2.5]	90.9 [-1.5]	70.7 [-4.2]	89.9 [-2.5]		
		Static sparse	71.6 [-3.3]	90.4 [-2.0]	67.8 [-7.1]	88.4 [-4.0]		
	Dynamic	*DeepR* (Bellec et al., 2017)	71.7 [-3.2]	90.6 [-1.8]	70.2 [-4.7]	90.0 [-2.4]	74.9 [0.0]	92.4 [0.0]
		SET (Mocanu et al., 2018)	72.6 [-2.3]	91.2 [-1.2]	70.4 [-4.5]	90.1 [-2.3]		
		Dynamic sparse (Ours)	**73.3 [-1.6]**	**92.4 [0.0]**	**71.6 [-3.3]**	**90.5 [-1.9]**		
Compression		*Compressed sparse* (Zhu & Gupta, 2017)	73.2 [-1.7]	91.5 [-0.9]	70.3 [-4.6]	90.0 [-2.4]		
		ThiNet (Luo et al., 2017)	*68.4 [-4.5]*	*88.3 [-2.8]*	(at 8.7M parameter count)			
		SSS (Huang & Wang, 2017)	*71.8 [-4.3]*	*90.8 [-2.1]*	(at 15.6M parameter count)			

Numbers in square brackets are differences from the *full dense* baseline. Romanized numbers are results of our experiments, and italicized ones taken directly from the original paper. Performance of two structured pruning methods, *ThiNet* and *Sparse Structure Selection* (*SSS*), are also listed for comparison (below the double line, see Appendix D for a discussion of their relevance); note the difference in parameter counts.

These results suggest that the dynamic evolution of the network through parameter reallocation is crucial to effective learning, because the superior generalization performance cannot be solely attributed to the network's structure, nor to its initialization, nor to a combination of the two.

Finally, to investigate whether the convergence of sparse network structures and that of parameter values had similar time courses, we did extra experiments with WRN-28-2, where at various stages during training, we stopped dynamic parameter reallocation, fixing the network structure while continuing the optimization of parameter values. As shown in Figure 4, dynamic parameter reallocation does not need to be active for the entire course of training, but only for some initial epochs. This suggests the network structure converges faster than the network parameters, which might be exploited in practice to further reduce computational cost.

Table 3: Wall-clock training time comparison

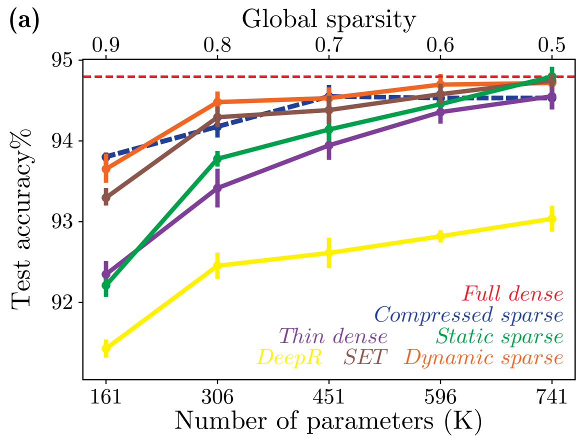
	WRN-28-2 on CIFAR10	Resnet-50 on Imagenet
DeepR	4.466 ± 0.358	5.636 ± 0.218
SET	1.087 ± 0.049	1.009 ± 0.002
Dynamic sparse (ours)	$\mathbf{1.083 \pm 0.051}$	$\mathbf{1.005 \pm 0.004}$

Median $\pm$ standard deviation of wall-clock training epoch times (from 25 epochs) for WRN-28-2 and Resnet-50 for different dynamic reparameterization methods. Results are relative ratios to the epoch time of a sparse network trained without dynamic parameter reallocation. WRN-28-2 is trained on a single, while Resnet-50 on four, Nvidia TITAN Xp GPU(s).

5. Discussion

In this work, we addressed the following problem: given a small, fixed budget of parameters for a deep residual CNN throughout training time, how to train it to yield the best generalization performance. We showed that training with dynamic parameter reallocation can achieve significantly better accuracies than static reparameterization at the same model size. Dynamic sparse reparameterization techniques have received relatively little attention to date, two existing methods (*SET* and *DeepR*) being applied only to relatively small and shallow networks. We proposed a dynamic parameterization method that adaptively reallocates free parameters across the network based on a simple heuristic during training. Our method yielded sparse models that generalize better than those produced by previous dynamic parameterization methods and outperformed all the static reparameterization methods we benchmarked against[‖].

[‖]Additionally, we show that our method outperformed a static dense reparameterization method *HashedNet* (Chen et al., 2015) (Appendix C), and that it is also able to train networks at extreme



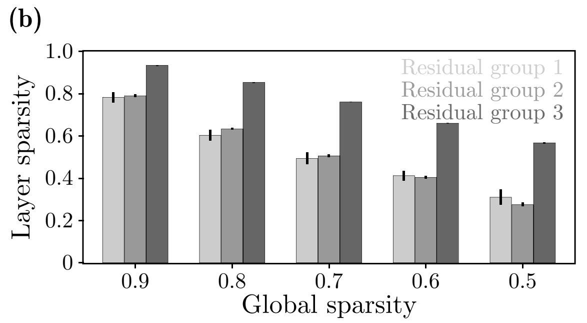
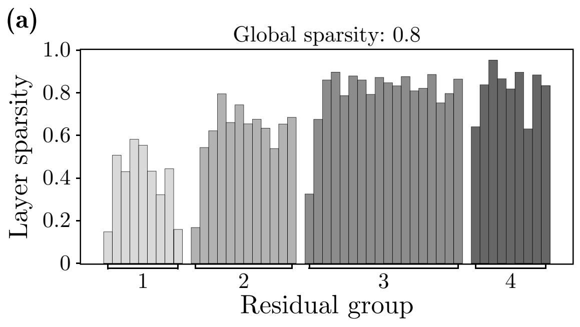


Figure 1: WRN-28-2 on CIFAR10. (a) Test accuracy plotted against number of trainable parameters in the sparse models for different methods. Dashed lines are used for the full dense model and for models obtained through compression, whereas all methods that maintain a constant parameter count throughout training and inference are represented by solid lines. Circular symbols mark the median of 5 runs, and error bars are the standard deviation. Parameter counts include all trainable parameters, i.e, parameters in sparse tensors plus all other dense tensors, such as those of batch normalization layers. (b) Breakdown of the final sparsities of the parameter tensors in the three residual blocks that emerged from our dynamic sparse parameterization algorithm (Algorithm 1) at different levels of global sparsity.



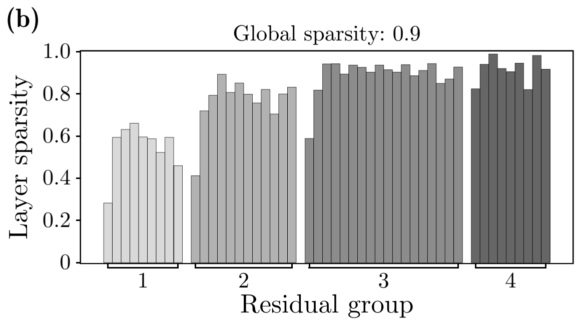
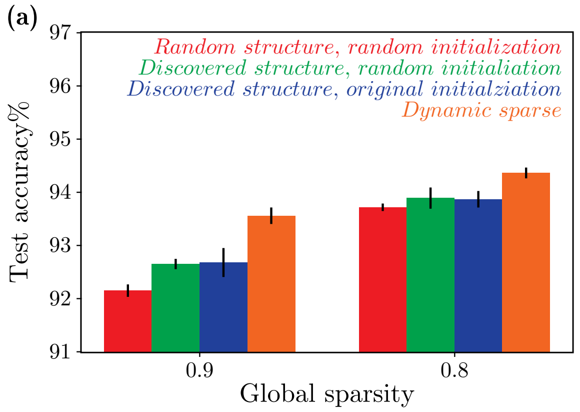


Figure 2: layer-wise breakdown of the final parameter tensor sparsities of Resnet-50 trained on Imagenet. (a) At overall sparsity 0.8. (b) At overall sparsity of 0.9.

High-performance sparse networks are often obtained by post-training pruning of dense networks. A number of recent studies have attempted direct training of sparse networks using *post hoc* information obtained from a pruned model. (Liu et al., 2018) argued that the sparse structure of the pruned model alone suffices to yield high accuracy, i.e. training a model of the same structure, starting with random weights, almost always reaches comparable levels of accuracy as the pruned model. In contrast, (Frankle & Carbin, 2018) argued that a post-compression sparse network structure alone is not sufficient, but necessary, to attain high accuracy, which, as the authors argue, requires both the pruned network connectivity *and* its initial weights when it was trained as part of the dense model pre-compression. In our experiments, we found that neither the *post hoc* sparseness pattern, nor the combination of connectivity and initialization, managed to explain the high performance of sparse networks produced by our *dynamic sparse* training method. Thus, the value of dynamic parameter reallocation goes beyond discovering trainable sparse network structure: the evolutionary process of structural exploration itself seems helpful for SGD to converge to better weights. Extra work is needed to explain the mechanisms underlying this phenomenon. One hypothesis is that the discontinuous jumps in parameter space when parameters are reallocated across layers helped training escape sharp minima that generalize badly (Keskar et al., 2016).

Structural degrees of freedom are qualitatively different from the degrees of freedom introduced by overparameterization. The latter directly expands the dimensionality of the parameter space in which SGD directly optimizes, whereas structural degrees of freedom are realized and explored using non-differentiable heuristics that only interact indirectly with the dynamics of gradient-based optimization, e.g. regularization pulling weights towards zero causing connections to be pruned. Our results suggest that, for residual CNNs under a given descriptive complexity (i.e. memory storage) constraint, it is better (in the sense of producing models that generalize better) to allocate some memory to describe and explore structural degrees of freedom, than to allocate all memory to conventional weights. This makes a potentially compelling case for hardware ac-

sparsity levels where previous static and dynamic parameterization methods often fail catastrophically (Appendix F).

(a)


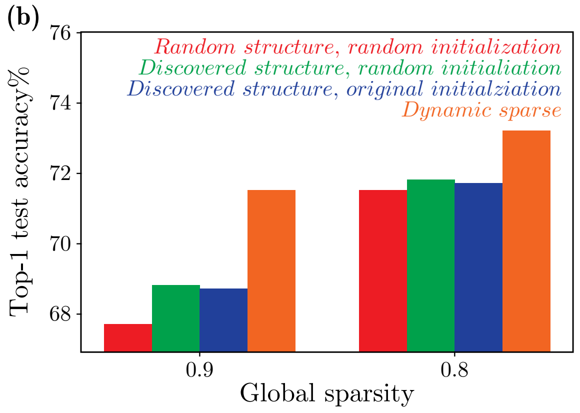
(b)


Figure 3: Comparison of training using our dynamic reparameterization method against training a number of related statically parameterized networks. All statically parameterized networks were trained for double the number of epochs used by our method. (a) WRN-28-2 on CIFAR10. Mean and standard deviation from 5 runs. (b) Resnet-50 on Imagenet. Single run.

celeration of sparse computations for more parameter efficient training.

Beside storage (spatial complexity), computational efficiency (temporal complexity) is also of primary concern. Current mainstream computing hardware architectures such as CPUs and GPUs cannot efficiently handle unstructured sparsity patterns. To maintain structured network configurations, various pruning techniques prune a trained model at the level of entire feature maps or layers. Emerging evidence suggests that the resulting pruned networks perform no better than directly-trained thin networks (Liu et al., 2018), calling into question the value of such coarse-grained pruning. We show in Appendix E additional results applying *dynamic sparse* training at an intermediate level of structured sparseness, i.e. pruning 3×3 kernel slices. Imposing this sparseness structure led to significantly worse generalization, producing sparse networks performing on par with statically parameterized *thin dense* networks trained for double the number of epochs.

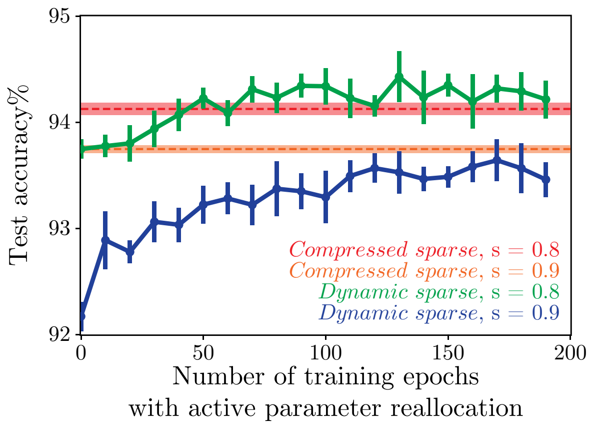


Figure 4: Test accuracies of sparse WRN-28-2 trained on CIFAR10 when dynamic parameter reallocation was switched off at certain epochs. Results are shown for two global sparsity levels: 0.8 and 0.9. Horizontal bands indicate the accuracy of the *compressed sparse* baselines where the band widths represent the standard deviation. For all data points, we ran training for 200 epochs, regardless of when dynamic parameter reallocation was stopped. Mean and standard deviation from 5 independent runs.

In summary, we show in this paper that it is possible to train deep sparse CNNs directly to reach generalization performances comparable to those achieved by iterative pruning and fine-tuning of pre-trained large dense models. The performance level achieved by our proposed method is significantly higher than that achieved by training dense models of the same size. Our method is the first to reallocate free parameters effectively and automatically within and across layers. Furthermore, we show that dynamic exploration of structural degrees of freedom during training is crucial to effective learning. Our work does not contradict the common wisdom that extra degrees of freedom are helpful for training deep networks to achieve better generalization, but it suggests that adding and dynamically exploring structural degrees of freedom is often a more effective and efficient alternative than simply increasing the parameter counts of the model.

References

Aimar, A., Mostafa, H., Calabrese, E., Rios-Navarro, A., Tapiador-Morales, R., Lungu, I.-A., Milde, M. B., Corradi, F., Linares-Barranco, A., Liu, S.-C., et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE transactions on neural networks and learning systems*, (99):1–13, 2018.

Arjovsky, M., Shah, A., and Bengio, Y. Unitary Evolution Recurrent Neural Networks. nov 2015. URL `http://arxiv.org/abs/1511.06464`.

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. Deep

Rewiring: Training very sparse deep networks. nov 2017. URL http://arxiv.org/abs/1711.05136.

Brutzkus, A., Globerson, A., Malach, E., and Shalev-Shwartz, S. SGD Learns Over-parameterized Networks that Provably Generalize on Linearly Separable Data. oct 2017. URL http://arxiv.org/abs/1710.10174.

Bucilua, C., Caruana, R., and Niculescu-Mizil, A. Model Compression. Technical report, 2006. URL https://www.cs.cornell.edu/{~}caruana/compression.kdd06.pdf.

Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. Compressing Neural Networks with the Hashing Trick. apr 2015. URL http://arxiv.org/abs/1504.04788.

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., and LeCun, Y. The Loss Surfaces of Multilayer Networks. nov 2014. URL http://arxiv.org/abs/1412.0233.

Cooper, Y. The loss landscape of overparameterized neural networks. apr 2018. URL http://arxiv.org/abs/1804.10200.

Dai, X., Yin, H., and Jha, N. K. NeST: A Neural Network Synthesis Tool Based on a Grow-and-Prune Paradigm. pp. 1–15, 2017. URL http://arxiv.org/abs/1711.02017.

Dai, X., Yin, H., and Jha, N. K. Grow and Prune Compact, Fast, and Accurate LSTMs. may 2018. URL http://arxiv.org/abs/1805.11797.

Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv*, pp. 1–14, 2014. URL http://arxiv.org/abs/1406.2572.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. Predicting Parameters in Deep Learning. jun 2013. URL http://arxiv.org/abs/1306.0543.

Dinh, L., Pascanu, R., Bengio, S., and Bengio, Y. Sharp Minima Can Generalize For Deep Nets. 2017. ISSN 1938-7228. URL http://arxiv.org/abs/1703.04933.

Frankle, J. and Carbin, M. The Lottery Ticket Hypothesis: Finding Small, Trainable Neural Networks. mar 2018. URL http://arxiv.org/abs/1803.03635.

Goodfellow, I. J., Vinyals, O., and Saxe, A. M. Qualitatively characterizing neural network optimization problems. dec 2014. URL http://arxiv.org/abs/1412.6544.

Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. pp. 1–14, 2015a. doi: abs/1510.00149/1510.00149. URL http://arxiv.org/abs/1510.00149.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both Weights and Connections for Efficient Neural Networks. jun 2015b. URL http://arxiv.org/abs/1506.02626.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. *Arxiv.Org*, 7(3): 171–180, 2015. ISSN 1664-1078. doi: 10.3389/fpsyg.2013.00124. URL http://arxiv.org/pdf/1512.03385v1.pdf.

He, Y., Zhang, X., and Sun, J. Channel Pruning for Accelerating Very Deep Neural Networks. jul 2017. URL http://arxiv.org/abs/1707.06168.

Hinton, G., Vinyals, O., and Dean, J. Distilling the Knowledge in a Neural Network. pp. 1–9, 2015. ISSN 0022-2488. doi: 10.1063/1.4931082. URL http://arxiv.org/abs/1503.02531.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. apr 2017. URL http://arxiv.org/abs/1704.04861.

Huang, Z. and Wang, N. Data-Driven Sparse Structure Selection for Deep Neural Networks. jul 2017. URL https://arxiv.org/abs/1707.01213.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. sep 2016. URL http://arxiv.org/abs/1609.07061.

Im, D. J., Tao, M., and Branson, K. An empirical analysis of the optimization of deep network loss surfaces. dec 2016. URL http://arxiv.org/abs/1612.04010.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up Convolutional Neural Networks with Low Rank Expansions. may 2014. URL http://arxiv.org/abs/1405.3866.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. sep 2016. URL `http://arxiv.org/abs/1609.04836`.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. Technical report, 2012.

Lebedev, V. and Lempitsky, V. Fast ConvNets Using Group-wise Brain Damage. jun 2015. URL `https://arxiv.org/abs/1506.02515`.

Li, C., Farkhoor, H., Liu, R., and Yosinski, J. Measuring the Intrinsic Dimension of Objective Landscapes. apr 2018. URL `http://arxiv.org/abs/1804.08838`.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning Filters for Efficient ConvNets. aug 2016. URL `http://arxiv.org/abs/1608.08710`.

Liao, Q. and Poggio, T. Theory of Deep Learning II: Landscape of the Empirical Risk in Deep Learning. *arXiv*, mar 2017. URL `http://arxiv.org/abs/1703.09833`.

Lin, M., Chen, Q., and Yan, S. Network In Network. *arXiv preprint*, pp. 10, 2013. ISSN 03029743. doi: 10.1109/ASRU.2015.7404828. URL `https://arxiv.org/pdf/1312.4400.pdfhttp://arxiv.org/abs/1312.4400`.

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., and Zhang, C. Learning Efficient Convolutional Networks through Network Slimming. aug 2017. URL `https://arxiv.org/abs/1708.06519`.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the Value of Network Pruning. oct 2018. URL `http://arxiv.org/abs/1810.05270`.

Luo, J.-H., Wu, J., and Lin, W. ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. jul 2017. URL `http://arxiv.org/abs/1707.06342`.

McDonnell, M. D. Training wide residual networks for deployment using a single bit for each weight. feb 2018. URL `http://arxiv.org/abs/1802.08530`.

Mittal, D., Bhardwaj, S., Khapra, M. M., and Ravindran, B. Recovering from Random Pruning: On the Plasticity of Deep Convolutional Neural Networks. jan 2018. URL `http://arxiv.org/abs/1801.10447`.

Mocanu, D. C., Mocanu, E., Stone, P., Nguyen, P. H., Gibescu, M., and Liotta, A. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1):2383, dec 2018. ISSN 2041-1723. doi: 10.1038/s41467-018-04316-3. URL `http://www.nature.com/articles/s41467-018-04316-3`.

Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. ACDC: A Structured Efficient Linear Layer. nov 2015. URL `http://arxiv.org/abs/1511.05946`.

Narang, S., Elsen, E., Diamos, G., and Sengupta, S. Exploring Sparsity in Recurrent Neural Networks. apr 2017. URL `http://arxiv.org/abs/1704.05119`.

Novak, R., Bahri, Y., Abolafia, D. A., Pennington, J., and Sohl-Dickstein, J. Sensitivity and Generalization in Neural Networks: an Empirical Study. feb 2018. URL `http://arxiv.org/abs/1802.08760`.

Poggio, T., Kawaguchi, K., Liao, Q., Miranda, B., Rosasco, L., Boix, X., Hidary, J., and Mhaskar, H. Theory of Deep Learning III: explaining the non-overfitting puzzle. 2017. URL `http://arxiv.org/abs/1801.00173`.

Salimans, T. and Kingma, D. P. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. feb 2016. URL `http://arxiv.org/abs/1602.07868`.

Sifre, L. and Mallat, S. *Rigid-motion scattering for image classification*. PhD thesis, Citeseer, 2014.

Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. sep 2014. URL `http://arxiv.org/abs/1409.1556`.

Sindhwani, V., Sainath, T. N., and Kumar, S. Structured Transforms for Small-Footprint Deep Learning. oct 2015. URL `http://arxiv.org/abs/1510.01722`.

Suau, X., Zappella, L., and Apostoloff, N. Network Compression using Correlation Analysis of Layer Responses. jul 2018. URL `http://arxiv.org/abs/1807.10585`.

Thomas, A. T., Gu, A., Dao, T., Rudra, A., and Christopher, R. Learning invariance with compact transforms. pp. 1–7, 2018.

Treister, E., Ruthotto, L., Sharoni, M., Zafrani, S., and Haber, E. Low-Cost Parameterizations of Deep Convolution Neural Networks. may 2018. URL `http://arxiv.org/abs/1805.07821`.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.

Wu, L., Zhu, Z., and E, W. Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes. jun 2017. URL `http://arxiv.org/abs/1706.10239`.

Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., and Wang, Z. Deep Fried Convnets. dec 2014. URL `http://arxiv.org/abs/1412.7149`.

Zagoruyko, S. and Komodakis, N. Wide Residual Networks. may 2016. URL `http://arxiv.org/abs/1605.07146`.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. Understanding deep learning requires rethinking generalization. nov 2016. URL `http://arxiv.org/abs/1611.03530`.

Zhang, C., Liao, Q., Rakhlin, A., Miranda, B., Golowich, N., and Poggio, T. Theory of Deep Learning IIb: Optimization Properties of SGD. jan 2018a. URL `http://arxiv.org/abs/1801.02254`.

Zhang, T., Ye, S., Zhang, K., Tang, J., Wen, W., Fardad, M., and Wang, Y. A Systematic DNN Weight Pruning Framework using Alternating Direction Method of Multipliers. apr 2018b. URL `http://arxiv.org/abs/1804.03294`.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. 2017. URL `http://arxiv.org/abs/1710.01878`.

Appendices

A. A full description of the dynamic parameter reallocation algorithm

Algorithm 1 in the main text informally describes our parameter reallocation scheme. In this appendix, we present a more rigorous description of the algorithm.

Let all reparameterized weight tensors in the original network be denoted by $\{\mathbf{W}_l\}$, where $l = 1, \cdots, L$ indexes layers. Let N_l be the number of parameters in $\mathbf{W}_l$, and $N = \sum_l N_l$ the total parameter count.

Sparse reparameterize $\mathbf{W}_l = g(\boldsymbol{\phi}_l; \boldsymbol{\psi}_l)$, where function g places components of parameter $\boldsymbol{\phi}_l$ into positions in $\mathbf{W}_l$ indexed by $\boldsymbol{\psi}_l \in \boldsymbol{\Psi}_{M_l}(\{1, \cdots, N_l\})$ [**], s.t. $W_{l,\psi_{l,i}} = \phi_{l,i}, \forall i$ indexing components. Let $M_l < N_l$ be the dimensionality of $\boldsymbol{\phi}_l$ and $\boldsymbol{\psi}_l$, i.e. the number of non-zero weights in $\mathbf{W}_l$. Define $s_l = 1 - \frac{M_l}{N_l}$ as the *sparsity* of $\mathbf{W}_l$. Global sparsity is then defined as $s = 1 - \frac{M}{N}$ where $M = \sum_l M_l$.

fraction of zero para

During the whole course of training, we kept global sparsity constant, specified by hyperparameter $s \in (0, 1)$. Reparameterization was initialized by uniformly sampling positions in each weight tensor at the global sparsity s, i.e. $\boldsymbol{\psi}_l^{(0)} \sim \mathcal{U}\left[\boldsymbol{\Psi}_{M_l^{(0)}}(\{1, \cdots, N_l\})\right], \forall l$, where $M_l^{(0)} = \lfloor (1-s) N_l \rceil$. Associated parameters $\boldsymbol{\phi}_l^{(0)}$ were randomly initialized.

Dynamic reparameterization was done periodically by repeating the following steps during training:

1. Train the model (currently reparameterized by $\left\{\left(\boldsymbol{\phi}_l^{(t)}, \boldsymbol{\psi}_l^{(t)}\right)\right\}$) for P batch iterations;
2. Reallocate free parameters within and across weight tensors following Algorithm 2 to arrive at new reparameterization $\left\{\left(\boldsymbol{\phi}_l^{(t+1)}, \boldsymbol{\psi}_l^{(t+1)}\right)\right\}$.

The adaptive reallocation is in essence a two-step procedure: a global pruning followed by a tensor-wise growth. Specifically our algorithm has the following key features:

1. Pruning was based on magnitude of weights, by comparing all parameters to a global threshold H, making the algorithm much more scalable than methods relying on layer-specific pruning.
2. We made H adaptive, subject to a simple setpoint control dynamics that ensured roughly N_p weights to be pruned globally per iteration. This is computationally cheaper than pruning exactly N_p smallest weights, which requires sorting all weights in the network.

?

3. Growth was by uniformly sampling zero weights and tensor-specific, thereby achieving a reallocation of parameters across layers. The heuristic guiding growth is

$$G_l^{(t)} = \left\lfloor \frac{R_l^{(t)}}{\sum_l R_l^{(t)}} \sum_l K_l^{(t)} \right\rceil, \quad (2)$$

pruned surviving non-0

where $K_l^{(t)}$ and $R_l^{(t)} = M_l^{(t)} - K_l^{(t)}$ are the pruned and surviving parameter counts, respectively. This rule allocated more free parameters to weight tensors with more surviving entries, while keeping the global sparsity the same by balancing numbers of parameters pruned and grown [††].

The entire procedure can be fully specified by hyperparameters $(s, P, N_p, \delta, H^{(0)})$.

?. Same?

B. Details of implementation

We implemented all models and reparameterization mechanisms using `pytorch`. Experiments were run on GPUs, and all sparse tensors were represented as dense tensors filtered by a binary mask [‡‡]. Source code to reproduce all experiments is available in the anonymous repository: `https://github.com/IntelAI/dynamic-reparameterization`.

Training Hyperparameter settings for training are listed in the first block of Table 4. Standard mild data augmentation was used in all experiments for CIFAR10 (random translation, cropping and horizontal flipping) and for Imagenet (random cropping and horizontal flipping). The last linear layer of WRN-28-2 was always kept dense as it has a negligible number of parameters. The number of training epochs for the *thin dense* and *static sparse* baselines are double the number of training epochs shown in Table 4.

Sparse compression baseline We compared our method against iterative pruning methods (Han et al., 2015b; Zhu & Gupta, 2017). We start from a full dense model trained with hyperparameters provided in the first block of Table 4 and then gradually prune the network to a target sparsity in T steps. As in (Zhu & Gupta, 2017), the pruning schedule

[**]By $\boldsymbol{\Psi}_p(Q) \triangleq \left\{\sigma(\Psi) : \Psi \in 2^Q, |\Psi| = p, \sigma \in \mathrm{S}_p\right\}$ we denote the set of all cardinality p ordered subsets of finite set Q.

[††]Note that an exact match is not guanranteed due to rounding errors in Eq. 2 and the possibility that $M_l^{(t)} - K_l^{(t)} + G_l^{(t)} > N_l$, i.e. free parameters in a weight tensor exceeding its dense size after reallocation. We added an extra step to redistribute parameters randomly to other tensors in these cases, thereby assuring an exact global sparsity.

[‡‡]This is a mere implementational choice for ease of experimentation given available hardware and software, which did not save memory because of sparsity. With computing substrate optimized for sparse linear algebra, our method is duly expected to realize the promised memory efficiency.

Algorithm 2: Reallocate free parameters within and across weight tensors

```
Input: $\left\{\left(\boldsymbol{\phi}_l^{(t)}, \boldsymbol{\psi}_l^{(t)}\right)\right\}$, $M^{(t)}$, $H^{(t)}$        ▷ From step $t$
Output: $\left\{\left(\boldsymbol{\phi}_l^{(t+1)}, \boldsymbol{\psi}_l^{(t+1)}\right)\right\}$, $M^{(t+1)}$, $H^{(t+1)}$        ▷ To step $t+1$
Need: $N_p, \delta$        ▷ Target number of parameters to be pruned and its fractional tolerance
for $l \in \{1, \cdots, L\}$ do        ▷ For each reparameterized weight tensor
    $\Pi_l^{(t)} \leftarrow \left\{i : \left|\phi_{l,i}^{(t)}\right| < H^{(t)}\right\}$        ▷ Indices of subthreshold components of $\boldsymbol{\phi}_l^{(t)}$ to be pruned
    $\left(K_l^{(t)}, R_l^{(t)}\right) \leftarrow \left(\left|\Pi_l^{(t)}\right|, M_l^{(t)} - \left|\Pi_l^{(t)}\right|\right)$        ▷ Numbers of pruned and surviving weights
if $\sum_l K_l^{(t)} < (1-\delta)N_p$ then        ▷ Too few parameters pruned
    $H^{(t+1)} \leftarrow 2H^{(t)}$        ▷ Increase pruning threshold
else if $\sum_l K_l^{(t)} > (1+\delta)N_p$ then        ▷ Too many parameters pruned
    $H^{(t+1)} \leftarrow \frac{1}{2}H^{(t)}$        ▷ Decrease pruning threshold
else        ▷ A proper number of parameters pruned
    $H^{(t+1)} \leftarrow H^{(t)}$        ▷ Maintain pruning threshold
for $l \in \{1, \cdots, L\}$ do        ▷ For each reparameterized weight tensor
    $G_l^{(t)} \leftarrow \left\lfloor \frac{R_l^{(t)}}{\sum_l R_l^{(t)}} \sum_l K_l^{(t)} \right\rceil$        ▷ Redistribute parameters for growth
    $\tilde{\boldsymbol{\psi}}_l^{(t)} \sim \mathcal{U}\left[\boldsymbol{\Psi}_{G_l^{(t)}}\left(\{1, \cdots, N_l\} \setminus \left\{\psi_{l,i}^{(t)}\right\}\right)\right]$        ▷ Sample zero positions to grow new weights
    $M_l^{(t+1)} \leftarrow M_l^{(t)} - K_l^{(t)} + G_l^{(t)}$        ▷ New parameter count
    $\left(\boldsymbol{\phi}_l^{(t+1)}, \boldsymbol{\psi}_l^{(t+1)}\right) \leftarrow \left(\left[\boldsymbol{\phi}_{l,i\notin\Pi_l^{(t)}}^{(t)}, \mathbf{0}\right], \left[\boldsymbol{\psi}_{l,i\notin\Pi_l^{(t)}}^{(t)}, \tilde{\boldsymbol{\psi}}_l^{(t)}\right]\right)$        ▷ New reparameterization
```

we used was

$$s^{(t)} = s + (1-s)\left(1 - \frac{t}{T}\right)^3, \quad (3)$$

where $t = 0, 1, \cdots, T$ indexes pruning steps, and s the target sparsity reached at the end of training. Thus, this baseline (labeled as *compressed sparse* in the paper) was effectively trained for more iterations (original training phase plus compression phase) than our *dynamic sparse* method. Hyperparameter settings for sparse compression are listed in the second block of Table 4.

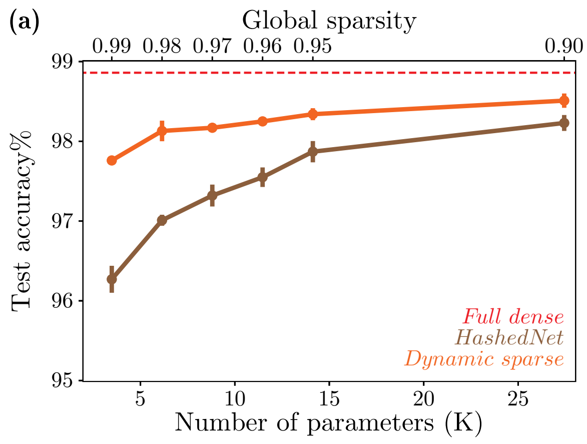
Dynamic reparameterization (ours) Hyperparameter settings for dynamic sparse reparameterization (Algorithm 1) are listed in the third block of Table 4.

Sparse Evolutionary Training (SET) Because the larger-scale experiments here (WRN-28-2 on CIFAR10 and Resnet-50 on Imagenet) were not attempted by (Mocanu et al., 2018), no specific settings for reparameterization in these cases were available in the original paper. In order to make a fair comparison, we used the same hyperparameters as those used in our dynamic reparameterization scheme (third block in Table 4). At each reparameterization step, the weights in each layer were sorted by magnitude and the smallest fraction was pruned. An equal number of parameters were then randomly allocated in the same layer and initialized to zero. For control, the total number of reallocated weights at each step was chosen to be the same as our dynamic reparameterization method, as was the schedule for reparameterization.

Deep Rewiring (DeepR) The fourth block in Table 4 contain hyperparameters for the DeepR experiments. We refer the reader to (Bellec et al., 2017) for details of the deep rewiring algorithm and for explanation of the hyperparameters. We chose the DeepR hyperparameters for the different networks based on a parameter sweep.

C. Comparison to dense reparameterization method *HashedNet*

We also compared our dynamic sparse reparameterization method to a number of static dense reparameterization techniques, e.g. (Denil et al., 2013; Yang et al., 2014; Moczulski et al., 2015; Sindhwani et al., 2015; Chen et al., 2015; Treister et al., 2018). Instead of sparsification, these methods impose structure on large parameter tensors by parameter sharing. Most of these methods have not been used for convolutional layers except for recent ones (Chen et al., 2015; Treister et al., 2018). We found that *HashedNet* (Chen et al., 2015) had the best performance over other static dense reparameterization methods, and also benchmarked our method against it. Instead of reparameterizing a parameter tensor with N entries to a sparse one with $M < N$ non-zero components, *HashedNet*'s reparameterization is to put M free parameters into N positions in



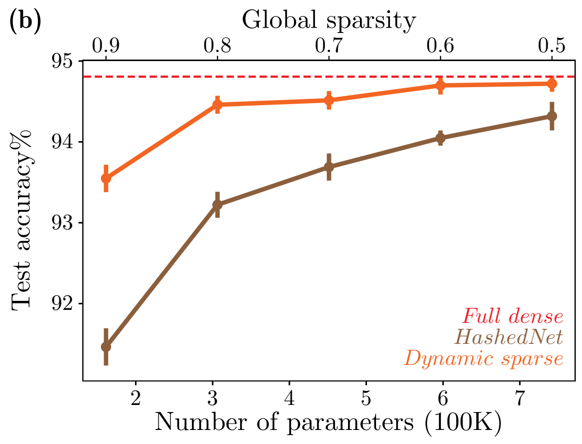


Figure 5: Comparison to *HashedNet*. (a) Test accuracy for LeNet-300-100-10 trained on MNIST. (b) Test accuracy for WRN-28-2 trained on CIFAR10. Conventions same as in Figure 7a.

the parameter through a random mapping from $\{1, \cdots, N\}$ to $\{1, \cdots, M\}$ computed by cheap hashing, resulting in a dense parameter tensor with shared components.

Results of LeNet-300-100-10 on MNIST are presented in Figure 5a, those of WRN-28-2 on CIFAR10 in Figure 5b, and those of Resnet-50 on Imagenet in Table 5. For a certain global sparsity s of our method, we compare it against a *HashedNet* with all reparameterized tensors hashed such that each had a fraction $1 - s$ of unique parameters. We found that our method *dynamic sparse* significantly outperformed *HashedNet*.

D. A taxonomy of training methods that yield "sparse" deep CNNs

As an extension to Section 2 of the main text, here we elaborate on existing methods related to ours, how they compare with and contrast to each other, and what features, apart from effectiveness, distinguished our approach from all previous ones. We confine the scope of comparison to training methods that produce smaller versions (i.e. ones with fewer parameters) of a given modern (i.e. post-AlexNet) deep convolutional neural network model. We list representative methods in Table 6. We classify these methods by three key features.

Strict parameter budget throughout training and inference This feature was discussed in depth in the main text. Most of the methods to date are *compression* techniques, i.e. they start training with a fully parameterized, dense model, and then reduce parameter counts. To the best of our knowledge, only three methods, namely DeepR (Bellec et al., 2017), SET (Mocanu et al., 2018) and ours, *strictly* impose, throughout the entire course of training, a fixed small parameter budget, one that is equal to the size of the final sparse model for inference. We make a distinction between these *direct training* methods (first block) and *compression* methods (second and third blocks of Table 6) [§§].

This distinction is meaningful in two ways: (a) practically, *direct training* methods are more memory-efficient on appropriate computing substrate by requiring parameter storage of no more than the final compressed model size; (b) theoretically, these methods, if performing on par with or better than *compression* methods (as this work suggests), shed light on an important question: whether gross overparameterization during training is necessary for good generalization performance?

Granularity of sparsity The *granularity* of sparsity refers to the additional structure imposed on the placement of the non-zero entries of a sparsified parameter tensor. The finest-grained case, namely *non-structured*, allows each individual weight in a parameter tensor to be zero or non-zero independently. Early compression techniques, e.g. (Han et al., 2015b), and more recent pruning-based compression methods based thereon, e.g. (Zhu & Gupta, 2017), are non-structured (second block of Table 6). So are all direct training methods like ours (first block of Table 6).

Non-structured sparsity can not be fully exploited by mainstream compute devices such as GPUs. To tackle this problem, a class of compression methods, *structured pruning* methods (third block in Table 6), constrain "sparsity" to a much coarser granularity. Typically, pruning is performed at the level of an entire feature map, e.g. ThiNet (Luo et al., 2017), whole layers, or even entire residual blocks (Huang & Wang, 2017). This way, the compressed "sparse" model

[§§]Note that an intermediate case is NeST (Dai et al., 2017; 2018), which starts training with a small network, grows it to a large size, and finally prunes it down again. Thus, a fixed parameter footprint is not strictly imposed throughout training, so we list NeST in the second block of Table 6.

has essentially smaller and/or fewer *dense* parameter tensors, and computation can thus be accelerated on GPUs the same way as dense neural networks.

These *structured compression* methods, however, did not make a useful baseline in this work, for the following reasons. First, because they produce dense models, their relevance to our method (non-structured, non-compression) is far more remote than non-structured compression techniques yielding sparse models, for a meaningful comparison. Second, typical structured pruning methods substantially underperformed non-structured ones (see Table 2 in the main text for two examples, ThiNet and SSS), and emerging evidence has called into question the fundamental value of structured pruning: (Mittal et al., 2018) found that the channel pruning criteria used in a number of state-of-the-art structured pruning methods performed no better than random channel elimination, and (Liu et al., 2018) found that fine-tuning in a number of state-of-the-art pruning methods fared no better than direct training of a randomly initialized pruned model which, in the case of channel/layer pruning, is simply a less wide and/or less deep dense model (see Table 2 in the main text for comparison of ThiNet and SSS against *thin dense*).

In addition, we performed extra experiments in which we constrained our method to operate on networks with structured sparsity and obtained significantly worse results, see Appendix E.

Predefined versus automatically discovered sparsity levels across layers The last key feature (rightmost column of Table 6) for our classification of methods is whether the sparsity levels of different layers of the network is automatically discovered during training or predefined by manual configuration. The value of automatic sparsification, e.g. ours, is twofold. First, it is conceptually more general because parameter reallocation heuristics can be applied to diverse model architectures, whereas layer-specific configuration has to be cognizant of network architecture, and at times also of the task to learn. Second, it is practically more scalable because it obviates the need for manual configuration of layer-wise sparsity, keeping the overhead of hyperparameter tuning constant rather than scaling with model depth/size. In addition to efficiency, we also show in Appendix F extra experiments on how automatic parameter reallocation across layers contributed to its effectiveness.

In conclusion, our method is unique in that it:

1. strictly maintains a fixed parameter footprint throughout the entire course of training.
2. automatically discovers layer-wise sparsity levels during training.

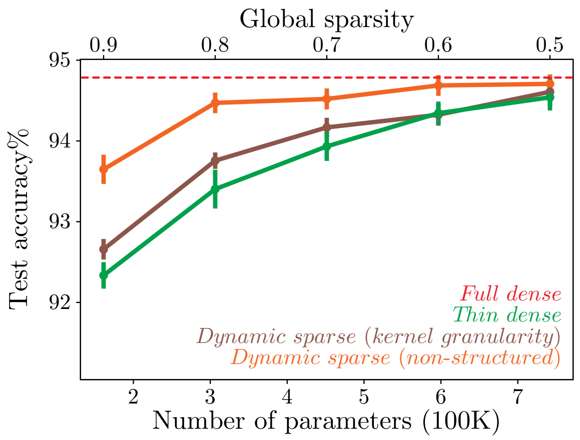


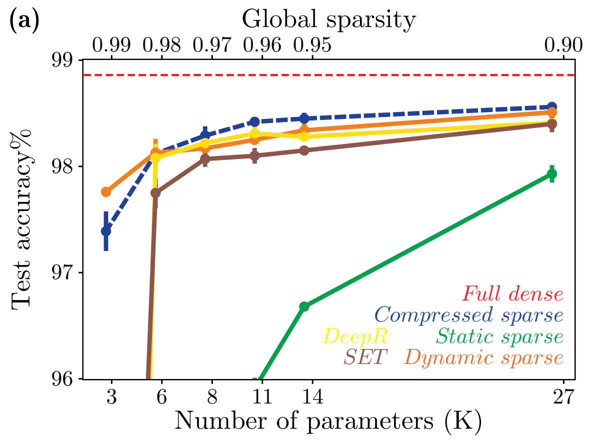
Figure 6: Test accuracy for WRN-28-2 trained on CIFAR10 for two variants of *dynamic sparse*, i.e. kernel-level granularity of sparsity and non-structured (same as *dynamic sparse* in the main text), as well as the *thin dense* baseline. Conventions same as in Figure 7a.

E. Structured versus non-structured sparsity

We investigated how our method performs if it were constrained to training sparse models at a coarser granularity. Consider a weight tensor of a convolution layer, of size $C_{\text{out}} \times C_{\text{in}} \times 3 \times 3$, where C_{out} and C_{in} are the number of output and input channels, respectively. Our method performed dynamic sparse reparameterization by pruning and reallocating individual weights of the 4-dimensional parameter tensor–the finest granularity. To adapt our procedure to coarse-grain sparsity on groups of parameters, we modified our algorithm (Algorithm 1 in the main text) in the following ways:

1. the pruning step now removed entire groups of weights by comparing their L^1-norms with the adaptive threshold.
2. the adaptive threshold was updated based on the difference between the target number and the actual number of groups to prune/grow at each step.
3. the growth step reallocated groups of weights within and across parameter tensors using the heuristic in Line 11 of Algorithm 2.

We show results at kernel-level granularity (i.e. groups are 3×3 kernels) in Figure 6 and Table 7, for WRN-28-2 on CIFAR10 and Resnet-50 on Imagenet, respectively. We observe that enforcing kernel-level sparsity leads to significantly worse accuracy compared to unstructured sparsity. For WRN-28-2, kernel-level parameter re-allocation still outperforms the *thin dense* baseline, though the performance advantage disappears as the level of sparsity decreases. Note that the *thin dense* baseline was always trained for double the number of epochs used to train the models with dynamic parameter re-allocation.



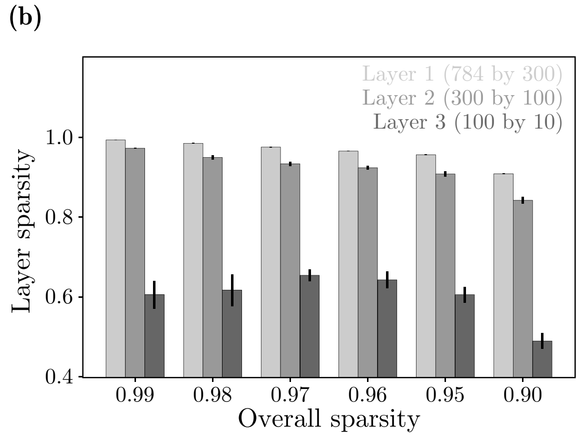


Figure 7: Test accuracy for LeNet-300-100-10 on MNIST for different training methods. Circular symbols mark the median of 5 runs, and error bars are the standard deviation. Parameter counts include all trainable parameters, i.e, parameters in sparse tensors plus all other dense tensors, such as those of batch normalization layers. Notice the failure of training at the highest sparsity level for *static sparse*, SET, and DeepR.

When we further coarsened the granularity of sparsity to channel level (i.e. groups are $C_{\text{in}} \times 3 \times 3$ slices that generate output feature maps), our method failed to produce performant models.

F. Multi-layer perceptrons and training at extreme sparsity levels

We carried out experiments on small multi-layer perceptrons to assess whether our dynamic parameter reallocation method can effectively distribute parameters in small networks at extreme sparsity levels. we experimented with a simple LeNet-300-100 trained on MNIST. Hyperparameters for the experiments are reported in appendix B. The results are shown in Fig. 7a. Our method is the only method, other than pruning from a large dense model, that is capable of effectively training the network at the highest sparsity setting by automatically moving parameters between layers to realize layer sparsities that can be effectively trained. The per-layer sparsities discovered by our method are shown in Fig. 7b. Our method automatically leads to a top layer with much lower sparsity than the two hidden layers. Similar sparsity patterns were found through hand-tuning to improve the performance of DeepR (Bellec et al., 2017). All layers were initialized at the same sparsity level (equal to the global sparsity level). While hand-tuning the per-layer sparsities should allow SET and DeepR to learn at the highest sparsity setting, our method automatically discovers the per-layer sparsities and allows us to dispense with such a tuning step.

Table 4: Hyperparameters for all experiments presented in the paper

Experiment	LeNet-300-100 on MNIST	WRN-28-2 on CIFAR10	Resnet-50 on Imagenet
Hyperparameters for training			
Number of training epochs	100	200	100
Mini-batch size	100	100	256
Learning rate schedule (epoch range: learning rate)	1 - 25: 0.100 26 - 50: 0.020 51 - 75: 0.040 76 - 100: 0.008	1 - 60: 0.100 61 - 120: 0.020 121 - 160: 0.040 161 - 200: 0.008	1 - 30: 0.1000 31 - 60: 0.0100 61 - 90: 0.0010 91 - 100: 0.0001
Momentum (Nesterov)	0.9	0.9	0.9
L^1 regularization multiplier	0.0001	0.0	0.0
L^2 regularization multiplier	0.0	0.0005	0.0001
Hyperparameters for sparse compression (*compressed sparse*) (Zhu & Gupta, 2017)			
Number of pruning iterations (T)	10	20	20
Number of training epochs between pruning iterations	2	2	2
Number of training epochs post-pruning	20	10	10
Total number of pruning epochs	40	50	50
Learning rate schedule during pruning (epoch range: learning rate)	1 - 20: 0.0200 21 - 30: 0.0040 31 - 40: 0.0008	1 - 25: 0.0200 25 - 35: 0.0040 36 - 50: 0.0008	1 - 25: 0.0100 26 - 35: 0.0010 36 - 50: 0.0001
Hyperparameters for dynamic sparse reparameterization (*dynamic sparse*) (ours)			
Number of parameters to prune (N_p)	600	20,000	200,000
Fractional tolerance of N_p (δ)	0.1	0.1	0.1
Initial pruning threshold ($H^{(0)}$)	0.001	0.001	0.001
Reparameterization period (P) schedule (epoch range: P)	1 - 25: 100 26 - 50: 200 51 - 75: 400 76 - 100: 800	1 - 25: 100 26 - 80: 200 81 - 140: 400 141 - 200: 800	1 - 25: 1000 26 - 50: 2000 51 - 75: 4000 76 - 100: 8000
Hyperparameters for Sparse Evolutionary Training (*SET*) (Mocanu et al., 2018)			
Number of parameters to prune at each re-parameterization step	600	20,000	200,000
Reparameterization period (P) schedule (epoch range: P)	1 - 25: 100 26 - 50: 200 51 - 75: 400 76 - 100: 800	1 - 25: 100 26 - 80: 200 81 - 140: 400 141 - 200: 800	1 - 25: 1000 26 - 50: 2000 51 - 75: 4000 76 - 100: 8000
Hyperparameters for Deep Rewiring (*DeepR*) (Bellec et al., 2017)			
L^1 regularization multiplier (α)	10^{-4}	10^{-5}	10^{-5}
Temperature (T) schedule (epoch range: T)	1 - 25: 10^{-3} 26 - 50: 10^{-4} 51 - 75: 10^{-5} 76 - 100: 10^{-6}	1 - 25: 10^{-5} 26 - 80: 10^{-8} 81 - 140: 10^{-12} 141 - 200: 10^{-15}	1 - 25: 10^{-5} 26 - 50: 10^{-8} 51 - 75: 10^{-12} 76 - 100: 10^{-15}

Table 5: Test accuracy% (top-1, top-5) of Resnet-50 on Imagenet for *dynamic sparse* vs. *HashedNet*. Numbers in square brackets are differences from the *full dense* baseline.

Final global sparsity (# Parameters)	0.8 (7.3M)		0.9 (5.1M)	
HashedNet	70.0 [-4.9]	89.6 [-2.8]	66.9 [-8.0]	87.4 [-5.0]
Dynamic sparse (ours)	**73.3** [**-1.6**]	**92.4** [**0.0**]	**71.6** [**-3.3**]	**90.5** [**-1.9**]

Table 6: Representative examples of training methods that yield "sparse" deep CNNs

Method	**Strict parameter budget throughout training and inference**	**Granularity of sparsity**	**Automatic layer sparsity**
Dynamic Sparse Reparameterization (Ours)	yes	non-structured	yes
Sparse Evolutionary Training (SET) (Mocanu et al., 2018)	yes	non-structured	no
Deep Rewiring (DeepR) (Bellec et al., 2017)	yes	non-structured	no
NN Synthesis Tool (NeST) (Dai et al., 2017; 2018)	no	non-structured	yes
`tf.contrib.model_pruning` (Zhu & Gupta, 2017)	no	non-structured	no
RNN Pruning (Narang et al., 2017)	no	non-structured	no
Deep Compression (Han et al., 2015b)	no	non-structured	no
Group-wise Brain Damage (Lebedev & Lempitsky, 2015)	no	channel	no
L^1-norm Channel Pruning (Li et al., 2016)	no	channel	no
Structured Sparsity Learning (SSL) (Wen et al., 2016)	no	channel/kernel/layer	yes
ThiNet (Luo et al., 2017)	no	channel	no
LASSO-regression Channel Pruning (He et al., 2017)	no	channel	no
Network Slimming (Liu et al., 2017)	no	channel	yes
Sparse Structure Selection (SSS) (Huang & Wang, 2017)	no	layer	yes
Principal Filter Analysis (PFA) (Suau et al., 2018)	no	channel	yes/no

We provide examples of different categories of methods. This is not a complete list of methods.

Table 7: Test accuracy% (top-1, top-5) of Resnet-50 on Imagenet for different levels of granularity of sparsity. Numbers in square brackets are differences from the *full dense* baseline.

Final overall sparsity (# Parameters)	0.8 (7.3M)		0.9 (5.1M)	
Thin dense	72.4 [-2.5]	90.9 [-1.5]	70.7 [-4.2]	89.9 [-2.5]
Dynamic sparse (kernel granularity)	72.6 [-2.3]	91.0 [-1.4]	70.2 [-4.7]	89.8 [-2.6]
Dynamic sparse (non-structured)	**73.3** [**-1.6**]	**92.4** [**0.0**]	**71.6** [**-3.3**]	**90.5** [**-1.9**]