Dynamic Sparse Training with Structured Sparsity

Mike Lasby[1], **Anna Golubeva**[2,3], **Utku Evci**[4], **Mihai Nica**[5,6], **Yani A. Ioannou**[1]
[1]University of Calgary, [2]Massachusetts Institute of Technology, [3]IAIFI
[4]Google Research, [5]University of Guelph, [6]Vector Institute for AI
mklasby@ucalgary.ca, golubeva@mit.edu, utku@google.com,
nicam@uoguelph.ca, yani.ioannou@ucalgary.ca

Abstract

Dynamic Sparse Training (DST) methods achieve state-of-the-art results in sparse neural network training, matching the generalization of dense models while enabling sparse training and inference. Although the resulting models are highly sparse and theoretically cheaper to train, achieving speedups with unstructured sparsity on real-world hardware is challenging. In this work, we propose a sparse-to-sparse DST method to learn a variant of *structured* N:M sparsity by imposing a *constant fan-in* constraint. We demonstrate with both a theoretical analysis and empirical results: state-of-the-art spare-to-sparse structured DST performance on a variety of network architectures, a condensed representation with a reduced parameter and memory footprint, and reduced inference time compared to dense models with a naive PyTorch CPU implementation of the condensed representation. Our source code is available here.

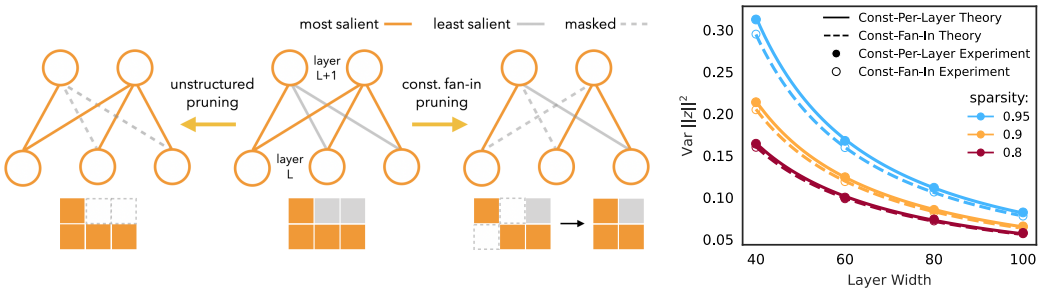
1 Introduction

Dynamic Sparse Training (DST) methods such as RigL (Evci et al., 2021) are the state-of-the-art in sparse training methods, learning *unstructured* Sparse Neural Networks (SNNs) with 85–95% fewer weights than dense models, while maintaining similar generalization. Furthermore, sparse training methods employ sparsity *both during training and inference*, unlike pruning and other methods (Zhou et al., 2021) that only exploit sparsity at inference time.

While models trained with DST methods are highly sparse and enable a large reduction in Floating Point Operations (FLOPs) in theory, realizing these speedups on hardware is challenging when the sparsity pattern is unstructured. Even considering recent advances in accelerating unstructured SNNs (Elsen et al., 2020; Gale et al., 2020), structured sparsity realizes much stronger acceleration on real-world hardware. On the other hand, structured sparse pruning often removes salient weights, resulting in worse generalization than comparable unstructured SNNs for the same sparsity level (Fig. 1a). Our work presents a best-of-both-worlds approach: we exploit the DST framework to learn *both* a highly-sparse *and* structured representation while maintaining the generalization performance of DST and dense baselines. In summary, our work makes the following contributions:

1. We propose a novel sparse-to-sparse DST method, Structured RigL (SRigL), based on RigL (Evci et al., 2021). SRigL learns a SNN with constant fan-in structured sparsity (Fig. 1a) while maintaining generalization comparable with RigL up to a high sparsity level (99%) for a variety of network architectures. This structure is a particular case of "N:M sparsity" which requires N out of M consecutive weights to be non-zero (Nvidia, 2020).
2. Our empirical analysis shows that at sparsity levels >90% RigL ablates whole neurons. By allowing ablation in SRigL, we match the generalization performance of RigL even in this high-sparsity regime.

Preprint. Under review.



(a) Constant fan-in pruning v.s. unstructured pruning.

(b) Output-norm variance analysis.

Figure 1: (a) **Constant fan-in** pruning keeps the most salient weights *per neuron*, while unstructured pruning keeps the most salient weights *per layer*. A constant fan-in weight matrix has the same number of non-zero elements (here 2) per column allowing condensed representation. While pruning may remove salient weights affecting generalization, with SRigL structure and weights are learned concurrently. (b) **Output-norm variance**: Theoretical predictions and simulation results demonstrating that sparse layers with constant fan-in have consistently smaller output-norm variance than layers with the same number of non-zero weights but without the constant fan-in constraint.

3. We motivate our choice of structure with a theoretical analysis of SNN output norm variance — a property related to training stability — and find that the constant fan-in constraint does not have a negative effect.
4. We demonstrate that, similar to other N:M sparsity variants (Nvidia, 2020), our constant fan-in sparsity enables a compact representation that is not only parameter- and memory-efficient, but also amenable to real-world acceleration. While hardware support for our specific form of N:M sparsity is not yet available, we observe increased performance with highly sparse networks over dense baseline at inference time even with only a naive PyTorch CPU implementation.

2 Related work

Dynamic Sparse Training Unlike with pruning, where weights are typically pruned after the dense network was trained (Han et al., 2015, 2016), or at initialization (Wang et al., 2020), DST methods learn the sparse connectivity during training by periodically adding and removing weights based on various saliency criteria. For instance, Sparse Evolutionary Training (SET) (Mocanu et al., 2018) removes weights with the smallest magnitude and adds weights randomly; similarly, RigL (Evci et al., 2021) prunes weights with the smallest magnitude and regrows weights that have large-magnitude gradients. RigL has been shown to learn models with 95% fewer parameters than dense baselines, while maintaining generalization performance. Liu et al. (2021b) further improved the original RigL results by increasing the extent of the parameter space explored by modifying the sparse connectivity update schedule and drop rate.

Many recent works have examined the effect of different grow and prune saliency criteria on unstructured DST approaches, including SET, Deep Rewiring (DeepR) (Bellec et al., 2023), Sparse Networks from Scratch (SNFS) (Dettmers & Zettlemoyer, 2019), Dynamic Sparse Reparameterization (DSR) (Mostafa & Wang, 2019),Top-K Always Sparse Training (Top-KAST) (Jayakumar et al., 2020), Memory-Economic Sparse Training (MEST) (Yuan et al., 2021). In Section 4 we compare SRigL to several of these methods. While the above-noted DST methods are highly effective at finding SNNs which reduce theoretical inference cost, they result in unstructured SNNs which are difficult to accelerate in practice due to various restrictions of common hardware architectures.

Accelerating Unstructured Sparse Neural Networks Elsen et al. (2020) proposed a method for accelerating unstructured SNNs based on one-dimensional tiling of non-zero elements, which demonstrated significant speedups on both Central Processing Unit (CPU) (Elsen et al., 2020) and Graphics Processing Unit (GPU) (Gale et al., 2020). However, like most approaches to accelerating unstructured SNNs, this method relies on imposing structure on an existing sparse weight matrix *after training*. Our method can be considered a way of adding structure to SNNs *during training*, allowing the model to maximally utilize non-zero weights since structure and weights are learned concurrently.

Learning Block Structured Sparsity From Scratch Block sparsity is a particular type of structured sparsity in which blocks of non-zero weights are grouped together in arrangements that reduce the memory overhead required to store the indices of the non-zero weights. Blocks can be generated out of contiguous weights in 1D (sometimes called tiles) or 2D or by utilizing a fixed number of non-zero weights per row or column group in the case of block-balanced sparsity (Hoefler et al., 2021). Spurred by the success of DST in learning unstructured sparse models, recent works have attempted to apply DST principles to learn block-structured sparsity. Jiang et al. (2022) introduced a novel block-aware DST algorithm known as Dynamic Shuffled Block (DSB). DSB reshuffles non-zero weights into a block sparsity pattern after sparse connectivity updates, thereby improving memory access efficiency. Wall-clock speed-ups of up to $4\times$ were reported with this method; however, generalization performance was reduced compared to RigL at comparable sparsities. Dietrich et al. (2022) applied a modified variant of RigL to BERT models (Devlin et al., 2019). The resulting method is capable of learning models with block-structured sparsity. The authors demonstrated improvements in generalization performance at reduced FLOPs compared to a dense network baseline.

Learning N:M Structured Sparsity from Scratch N:M sparsity is a specific form of block-balanced sparsity in which 1D blocks with M contiguous elements contain exactly N non-zero elements. N:M sparsity is particularly amenable to acceleration and several attempts have been made to train models with N:M fine-grained structure using DST methods.

Yang et al. (2022) extended the DST method proposed by Liu et al. (2021a) to train multiple sparse sub-networks sampled from a single dense super-network. Their proposed method, Alternating Sparse Training (AST), switches the network topology between sparse sub-networks after each mini-batch during training. Yang et al. (2022) demonstrated state-of-the-art performance on several typical sparse training benchmarks. However, the DST methodology used in (Liu et al., 2021a) gradually increases sparsity during training to yield the desired sparsity at the end of training. Therefore, the dense model weights and gradients are required throughout the majority of training, greatly increasing the overall compute and storage requirements. While AST demonstrated a tantalizing possibility of training multiple sparse sub-networks within a single training loop, the gradual dense-to-sparse training paradigm used by (Liu et al., 2021a) is not directly comparable to RigL or other similar end-to-end sparse DST methods.

Zhou et al. (2021) explored how N:M sparsity can be achieved during training using magnitude-based pruning during the forward pass and a Straight-Through Estimator (STE) (Bengio et al., 2013) on the backward pass. In their method, the dense network weights are projected into a sparse network during each training iteration. The sparse network is obtained by selecting the top-N out of every M contiguous weights and STE is used to propagate the approximated gradients through the projection function. However, in their initial experiments sparse networks trained with STE exhibited a significant performance drop compared to a dense benchmark. The authors conjectured that the reduced performance could be due the gradients approximation performed by STE resulting in sparse connectivity instabilities during training. To counteract this, Zhou et al. (2021) introduced a regularization term applied to the gradients of pruned weights and called their approach Sparse-Refined Straight-Through Estimator (SR-STE). Their results include N:M ratios of 1:4, 2:4, 2:8, 4:8, 1:16, corresponding to model sparsities of 25%, 50%, 75% and 93.75%. Although SR-STE utilizes sparse operations in the forward pass and can find sparse models optimized for inference, it does not reduce the training cost significantly. Specifically, SR-STE training requires (1) storing original parameters in their dense format, and (2) calculating dense gradients during each training iteration. This makes SR-STE training as expensive as the original dense training in terms of memory and compute cost[1]. On the other hand, DST methods such as RigL, and our proposed method SRigL, were developed for end-to-end sparse training and use sparse parameters and gradients throughout training.

Accelerating Fine-grained N:M Structured Sparsity Nvidia (2020) introduced the Ampere Tensor Core GPU architecture (e.g. A100 GPUs) and proposed the 2:4 fine-grained structured sparsity scheme that enables SNNs to be accelerated on this hardware *at inference time*. This scheme places a constraint on the allowed sparsity pattern: For every contiguous array of four weights, two are pruned, yielding a 50%-sparse net. The resulting regular structure of the weight matrix allows one to compress it efficiently and to reduce memory storage and bandwidth by operating on the nonzero weights only.

[1]To be precise, SR-STE can use some sparse operations and reduce training cost up to two thirds of the original dense training. However this is still far from fully sparse acceleration for training.

Since the focus is on acceleration at inference time, the authors proposed to use the standard method of magnitude-based pruning post training to achieve the 2:4 sparsity. Importantly, this work considered exclusively the 2:4 ratio; other N:M ratios cannot be accelerated on Ampere GPUs.

3 Method

Most existing sparse-to-sparse DST methods, including the state-of-the-art RigL, learn an *unstructured* sparse mask, and yet structured sparsity realizes substantially better acceleration in practice. Our goal in this work is to introduce structural constraints on the sparse mask learned by RigL, in order to make it more amenable to acceleration, at both training and inference time, while not affecting RigL's generalization performance. The constant fan-in constraint represents a special case of N:M sparsity where N is the number of non-zero weights per neuron and M is the dense fan-in for each neuron within a given layer. While current acceleration results exist only for 2:4 sparsity and rely on specialized hardware (e.g. Nvidia's Ampere GPU architecture), a constant fan-in constraint can also theoretically take advantage of the efficient memory access and throughput increase that N:M sparsity yields (Mishra et al., 2021).

We start with a theoretical analysis to explore the effect of various sparsity distributions with different degrees of structural constraints on the training dynamics of SNNs, motivating the particular structured sparsity we use.

3.1 Sparsity and Output-Norm Variance

Consider a SNN with ReLU activations, where each neuron has on average k connections to the previous layer (i.e., fan-in). It has been shown by Evci et al. (2022), that by normalizing the weights on initialization by a factor of $\sqrt{2/k}$, one achieves the following desirable normalization property for each layer ℓ with output z^ℓ:

$$\mathbb{E}\left(\frac{||z^{\ell+1}||^2}{||z^\ell||^2}\right)=1,$$

Meaning that on average the variance of the norm of each layer's output is constant. However, the variance of this ratio is non-trivial. In networks with large depth, it can accumulate, leading to exponentially large variance at the final layer (Li et al., 2021). Minimizing this variance on initialization has been shown to have a positive effect on training dynamics in some network models (Littwin et al., 2020), as it stabilizes the gradients. We therefore analyze the output norm variance as a guiding quantity for sparsity-type selection.

In the following, we consider three different types of sparsity distributions, which respectively correspond to different degrees of sparsity *structure* in the SNN, and derive analytic expressions for the behaviour of output norm variance in SNNs with the given sparsity type. The derivations for the following results can be found in Appendix A:

- **"Bernoulli sparsity"**: A connection between each neuron in layer $\ell+1$ and each neuron in layer ℓ appears *independently* with probability $p=\frac{k}{n}$, resulting in each neuron having k connections *on average* and each layer having nk connections *on average*. The variance is:

$$\mathbf{Var}_{\text{Bernoulli}}\left(\frac{||z^{\ell+1}||^2}{||z^\ell||^2}\right)=\frac{5n-8+18\frac{k}{n}}{n(n+2)}. \tag{1}$$

- **"Constant Per-Layer sparsity"**: Exactly kn connections are distributed at random in the layer connecting the n neurons in layer $\ell+1$ and the n neurons in layer ℓ, resulting in each neuron having k connections *on average*. The variance is:

$$\mathbf{Var}_{\text{Const-Per-Layer}}\left(\frac{||z^{\ell+1}||^2}{||z^\ell||^2}\right)=\frac{(n^2+7n-8)C_{n,k}+18\frac{k}{n}-n^2-2n}{n(n+2)}, \tag{2}$$

  where $C_{n,k}=\frac{n-1/k}{n-1/n}$. Note that when $n\gg 1$, $C_{n,k}\approx 1-\frac{n-k}{n^2k}$ is close to 1, and with $C_{n,k}=1$ we recover the formula for Bernoulli sparsity, meaning that this sparsity type and Bernoulli sparsity are very similar.
- **"Constant Fan-In sparsity"**: Each neuron in layer $\ell+1$ is connected to exactly k neurons from layer ℓ, chosen uniformly at random. In this case, the variance is:

$$\mathbf{Var}_{\text{Const-Fan-In}}\left(\frac{||z^{\ell+1}||^2}{||z^\ell||^2}\right)=\frac{5n-8+18\frac{k}{n}}{n(n+2)}-\frac{3(n-k)}{kn(n+2)}. \tag{3}$$

Figure 2: Neuron ablation. At sparsity levels over 90%, RigL learns to completely mask (ablate) a large number of neurons within each layer, effectively reducing layer width. Imposing a constant fan-in constraint requires all neurons to have the same number of (non-pruned) incoming weights and therefore inhibits ablation, which results in worse generalization performance than RigL. Allowing SRigL to ablate neurons restores RigL-level performance.

In deriving the above results we assumed that the direction of the layer output vector $\frac{z^\ell}{||z^\ell||}$ is uniformly distributed on the unit sphere. We compare our theoretical predictions with simulations in Fig. 1b and verify their accuracy. Bernoulli and constant-per-layer distribution result in unstructured sparsity, and most of the current DST approaches, including RigL, operate with constant-per-layer sparsity. In contrast, the constant-fan-in type imposes a strong structural constraint. Therefore we are somewhat surprised to find that, in fact, constant-fan-in sparsity always produces slightly smaller output-norm variance than the other types. The difference is larger when $k \ll n$, i.e., for very sparse networks. This indicates that, at the very least, the constant fan-in constraint should not impair SNN training dynamics and performance, motivating our method of maintaining the constant fan-in sparsity constraint within a DST approach.

3.2 Structured RigL

As motivated by Section 3.1, we propose to enforce the constant-fan-in constraint within a sparse-to-sparse DST method to learn structured sparse connectivity from scratch. Specifically, we use RigL by Evci et al. (2021), which can obtain highly sparse networks with generalization performance comparable to their dense baselines.

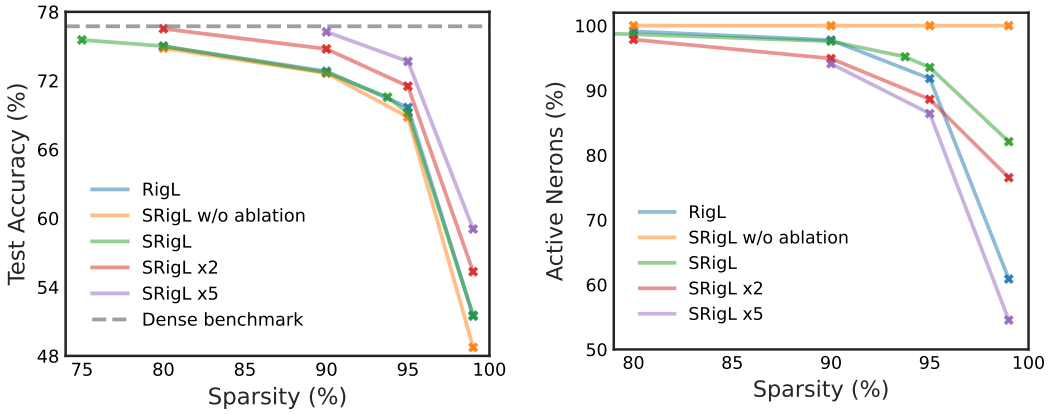
In brief, the methodology of RigL is to update the SNN connectivity during training by *pruning* weights with the smallest magnitude and *regrowing* those with the largest corresponding gradient magnitude in *each layer*. This occurs in periodic, but relatively infrequent mask update steps throughout most of training. In SRigL, weight saliency must be determined at the *neuron level* (in convolutional layers, at the level of each filter), since we enforce that every neuron (output channel) has the same number of unmasked incoming weights, thereby satisfying the constant fan-in constraint. (Fig. 1a).

However, this approach alone significantly lags behind RigL's generalization at very high sparsities (>90%) and with transformer architectures, as shown in Fig. 3a and Table 4. This is because the constant fan-in constraint has an important side-effect: under a strict constant fan-in constraint, neurons can never be entirely masked (ablated), as illustrated in Fig. 2. At very high sparsity levels this can lead to many neurons that have only 1–2 weights, limiting the capacity to learn complex features and consequently reducing generalization performance. Indeed, at high sparsities we observed empirically that RigL ablates large numbers of neurons (Figs. 3b, 10 and 11). Effectively, *RigL reduces the width of the model at high sparsities to maintain generalization performance*; we believe we are the first to explicitly identify this behaviour within a DST method.

To resolve this issue in SRigL, we implement a **neuron ablation method**, allowing SRigL to maintain both a constant fan-in constraint *and* to reduce layer width at high sparsities. We introduce a new hyperparameter, γ_{sal}, which defines the required minimum percentage of salient weights per neuron. Given a neuron with constant fan-in of k, if fewer than $\gamma_{sal} * k$ weights are considered salient by either the drop *or* grow criteria, then the neuron is ablated and its weights redistributed to other neurons within the same layer. The steps below outline our final SRigL method with neuron ablation.

In the following procedure, the first two steps are the same as in RigL, while the other steps are specific to SRigL, containing modifications to include the constant fan-in constraint and neuron ablation. We first set an ablation threshold γ_{sal}. Then, for each layer we do the following:

1. Obtain magnitudes of the active weights and gradient magnitudes of the pruned weights; these will serve as prune and growth criteria, respectively.
2. Compute K, the number of weights to be grown and pruned in the current step in this layer. We always grow the same number of connections as we prune.



(a) ResNet-50/ImageNet

(b) Neuron Ablation

Figure 3: (a) ResNet-50/ImageNet top-1 test accuracy when trained with SRigL for a range of sparsities is comparable to RigL. Extended training durations of $\times 2$ and $\times 5$ are also reported for SRigL. Results reported are single runs. **(b) Neuron ablation:** The percentage active neurons (i.e., not ablated) following RigL/SRigL training. RigL ablates a large number of neurons at high sparsities.

3. Count the number of salient weights per neuron. A weight is considered *salient* if it is in the top-K of either the largest-magnitude weights or the largest-magnitude gradients.
4. Ablate neurons that have fewer salient weights than $\gamma_{sal} * k$, where k is the fan-in. Ablation is done by pruning all incoming weights. These pruned weights are redistributed to the remaining neurons in the following steps.
5. Compute the new constant fan-in constraint, k', based on the number of ablated neurons.
6. Prune the K smallest-magnitude weights in the current layer. Note that this pruning criterion considers all weights within a layer rather than pruning only the smallest weights in each neuron.
7. For each active neuron, regrow as many weights as required, proceeding in order of decreasing gradient magnitude, until the target fan-in, k', is achieved.

4 Results

We implement SRigL in PyTorch by extending an existing implementation of RigL (McCreary, 2020). We evaluate our method empirically on image classification tasks, training the ResNet-18 (He et al., 2016) and Wide ResNet-22 (Zagoruyko & Komodakis, 2017) models on the CIFAR-10 dataset (Krizhevsky, 2009), and the ResNet-50 He et al. (2016) and Vision Transformer (ViT-B/16) Dosovitskiy et al. (2021) models on the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC-12) dataset (Russakovsky et al., 2015), commonly referred to as ImageNet. See Appendix B for Wide ResNet-22 experiment results.

Unless noted otherwise, we use the same hyperparameter configuration as the original RigL method. A detailed summary of our hyperparameter settings and training details can be found in Appendix C. The modified hyperparameters proposed by Liu et al. (2021b) may yield higher generalization performance, but a detailed investigation of the hyperparameters for SRigL is left to future work.

Unless noted otherwise, we set γ_{sal} to 30% for all SRigL experimental results. This value was selected based on a hyperparameter sweep performed by training ResNet-18 and Wide ResNet-22 on the CIFAR-10 dataset, see Appendix D.

4.1 ResNet-18 trained on CIFAR-10

Our training regimen generally follows Evci et al. (2021), see Appendix C.1 for more information. We repeat training with five different random seeds for both methods and report the mean and standard deviation compared to a densely-connected benchmark model in Table 2.

These results confirm that imposing a constant fan-in constraint during sparse training does not significantly degrade generalization performance of the SNN compared to the RigL method. We inspect the connectivity of ResNet models trained with the RigL method and find, as shown in Fig. 3b, that at 95% sparsity 10.9% of neurons are removed completely. Thus, RigL results in fewer, but more densely connected neurons, whereas the fan-in constraint enforces that all neurons are retained.

In Fig. 10 we plot the number of neurons ablated at ablation thresholds of 0, 30, and 50% to demonstrate how the γ_{sal} hyperparameter can be used to guide the final model width during training.

4.2 ResNet-50 trained on ImageNet

Our training regimen for the ImageNet dataset generally follows Evci et al. (2021), see Appendix C.2 for more details. We investigate the effect of extended training with $\times 2$ and $\times 5$ the original number of training epochs. We train each model with a single seed and report the results in Fig. 3a and Table 1.

SRigL yields similar generalization performance as RigL across each sparsity and training duration considered. At high sparsities, SRigL with ablation outperforms SRigL without ablation, highlighting the importance of neuron ablation as sparsity increases. Notably, RigL $\times 5$ results at 99% sparsity in Evci et al. (2021) used a dense first layer, unlike all other results reported in Table 1. Despite this difference, SRigL $\times 5$ at 99% sparsity is comparable to the RigL $\times 5$ results. We expect that the 99% sparse models would be improved by using a dense first layer for all SRigL results. Similar to RigL, we observe that SRigL generalization performance improves with increasing training time.

In Table 3 we compare SRigL to a variety of DST algorithms. SRigL performs comparably to other methods, even those which learn unstructured sparsity. Methods with a memory footprint listed as dense require training with the dense network and therefore are not directly comparable to other sparse-to-sparse DST methods. The most directly comparable method to ours is DSB; we note that SRigL outperforms DSB at all sparsity ratios reviewed.

4.3 Vision Transformer trained on ImageNet

We train the vision transformer variant ViT-B/16 on ImageNet generally following the original training recipe per Dosovitskiy et al. (2021) with several modifications, see Appendix C.3 for more information.

Unlike in our convolutional neural network experiments, RigL *does not* ablate neurons when applied to the ViT-B/16 architecture with sparsities between 80-90%. Instead, we find that RigL learns sparse connectivities with a high variance of fan-in between neurons (see Fig. 11). At 90% sparsity, some neurons are allocated up to $\times 10$ more active weights than the mean number of active weights in the same layer. We hypothesize that these more densely connected neurons found in our RigL experiments are important for generalization performance; therefore, a high γ_{sal} threshold should improve performance of SRigL by ablating neurons until a sufficient density of sparse fan-in is reached. Indeed, we find that SRigL's generalization performance is sensitive to γ_{sal} and that high γ_{sal} thresholds of 90% to 99% perform best. See Fig. 8 and Appendix D for more details on how γ_{sal} affects the generalization performance of ViT-B/16. For the following results, we used a γ_{sal} of 95%.

We train each model with a single seed and report the results in Table 4. SRigL without ablation is unable to match the generalization performance of RigL. However, with neuron ablation enabled, SRigL's performance greatly improves, outperforming RigL at the 90% sparsity level.

4.4 FLOPs analysis

In Fig. 4, we present an analysis of the FLOPs required during training and inference for SRigL and compare with SR-STE. We calculate FLOPs using the same methodology as Evci et al. (2021) by considering only operations induced by convolutional and linear layers and their activations. FLOPs for add and pooling operations are ignored. For training FLOPs, we also disregard FLOPs required for mask updates, as this step is amortized over ΔT steps and is negligible compared to the FLOPs required otherwise for training. The open-source code for counting operations is from the NeurIPS 2019 MicroNet Challenge and is available on GitHub[2].

[2] MicroNet Challenge Github Repository

Table 1: Top-1 ImageNet test accuracy of ResNet-50 trained with RigL or SRigL at high sparsities and with various training times (as in Evci et al. (2021)), e.g. 5× more training epochs than dense ResNet-50.

	RigL		SRigL			
sparsity			w/o	w/ ablation		
(%)	1×	5×[†]	1×	1×	2×	5×
80	74.9	77.1	74.8	75.0	76.5	–
90	72.8	76.6	72.6	72.7	74.7	76.2
95	69.6	74.6	68.8	69.1	71.5	73.6
99	51.4	61.9[‡]	48.7	51.5	55.3	59.0
0	*dense ResNet-50*: 76.7					

[†] 5× RigL results are from Evci et al. (2021)
[‡] uses a dense first layer, unlike other results

Table 2: Test accuracy for ResNet-18 on CIFAR-10 trained with RigL or SRigL with/without neuron ablation at varying sparsities repeated with five different random seeds.

	RigL	SRigL	
sparsity		w/o	w/ ablation
(%)			
80	95.2±0.1	95.2±0.1	95.2±0.0
90	95.1±0.1	95.0±0.1	95.1±0.1
95	94.6±0.2	94.5±0.3	94.7±0.2
99	92.9±0.1	91.5±0.3	92.8±0.1
0	*dense ResNet-18*: 95.5		

Table 3: Top-1 ImageNet test accuracy of ResNet-50 trained with a variety of DST methods, highlighting methods that both are sparse-to-sparse (i.e. sparse training) *and* learn structured sparsity similar to SRigL — only DSB-16 (2:4 and 1:4 sparsity) is directly comparable in this regard. RigL and SRigL results are from our experiments, other values are obtained from each method's corresponding paper, unless noted otherwise.

	training		sparsity				
method	method	structured	50%	75%	80%	90%	93.75%
Static[*]	sparse	no	–	–	70.6±0.06	65.8±0.04	–
SET[*]	sparse	no	–	–	72.9±0.39	69.6±0.23	–
DeepR[§]	sparse	no	–	–	71.7	70.2	–
DSR	sparse	no	–	–	73.3	71.6	–
Top-KAST[‡]	sparse	no	–	–	74.76	70.42	–
MEST[†]	sparse	no	–	–	**75.39**	72.58	–
RigL	sparse	no	–	–	74.98	**72.81**	–
DSB-16	**sparse**	**yes**	76.33	74.04	–	–	–
SRigL (Ours)	**sparse**	**yes**	**76.60**	**75.55**	75.01	72.71	70.56
SNFS (ERK)[*]	dense	no	–	–	75.2±0.11	73.0±0.04	–
SR-STE	dense	yes	–	76.2	–	–	71.5
dense ResNet-50:			76.7				

[*] values obtained from Evci et al. (2021)
[§] values obtained from Mostafa & Wang (2019)
[†] values for the MEST (x0.67+EM) variant, matched to the same number of training FLOPs as RigL
[‡] values tabulated for Top-KAST correspond to the *backwards sparsity* as Top-KAST uses different sparsities in the forward and backward passes. For 80% sparsity, the value reported is based on using a random regrow criteria, unlike the 90% sparsity value. For more information see Table 1 in Jayakumar et al. (2020)

Table 4: Top-1 test accuracy of ViT-B/16 trained on ImageNet with or without neuron ablation

	RigL	SRigL	
sparsity (%)		w/o	w/ ablation
80	**77.4**	72.8	77.0
90	75.0	69.5	**75.7**
0	*dense ViT-B/16*: 78.35		

Table 5: SRigL sparsity and FLOPs for ResNet-50/ImageNet training and inference.

	SRigL FLOPs	
sparsity (%)	training (×1e18)	inference (×1e9)
80	1.13	3.40
90	0.77	1.99
95	0.40	1.01
99	0.09	0.21
0	3.15	8.20

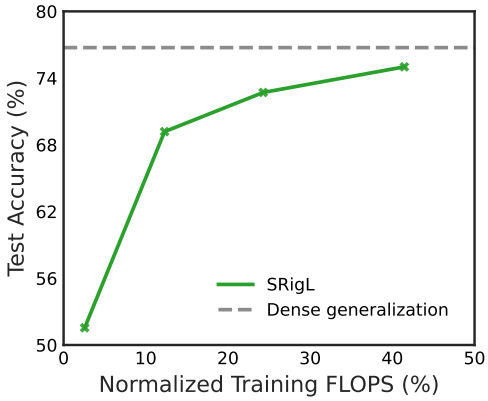


Figure 4: Training FLOPs for SRigL on ResNet-50/ImageNet at a variety of sparsities compared with dense generalization. FLOPs are normalized by dense training FLOPs.

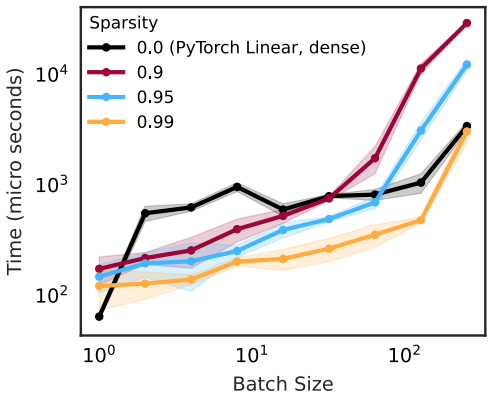


Figure 5: PyTorch timings. Benchmarking a naive (non-optimized) PyTorch implementation of our condensed representation on CPU.

Similar to other DST methods, SRigL obtains generalization performance comparable to a dense network benchmark at a fraction of the FLOPs required for both training and inference.

4.5 Acceleration of constant fan-in sparsity

Although commodity hardware can accelerate 2:4 sparsity (Mishra et al., 2021), the specific type of sparsity we propose to learn with SRigL, constant fan-in sparsity, has seen less attention. Theoretical speedups (i.e. FLOPs) are limited in demonstrating the real-world acceleration potential of a proposed sparse representation in general, and yet conversely creating a fully-optimized software or hardware implementation of a novel representation typically requires significant engineering effort outside of the scope of this paper.

Here we show that even a straight-forward PyTorch implementation of our proposed condensed neural network representation (see Appendix E) can demonstrate this real-world acceleration on dual Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz with 48 threads on 24 cores. In Fig. 5, we present real-world timings comparing the `torch.nn.Linear` layer and our condensed linear layer, which works as a drop-in replacement. We report mean timings and standard deviations across a minimum of five forward passes for a single condensed layer at sparsities of 99%, 95%, and 90% and a standard dense PyTorch Linear layer. For both the condensed and dense layer, we use a single layer with 10 neurons. We vary the batch size between 1 to 256 and use 65,536 features in our input tensor. The input tensors consist of 32 bit floating point values and the convolutional weight tensors are in a channel-first layout typical to PyTorch (batch size, channels, height, width). At high sparsity levels and moderate batch sizes, the condensed layer outperforms the dense `torch.nn.Linear` layer moderately. This result is highly promising, since it is reasonable to expect that a more optimized software implementation and/or explicit hardware support would improve upon these results significantly. For example, when we use the recently published SparseProp library (Nikdan et al., 2023) to accelerate networks trained with SRigL, we obtain accelerations that outperform our naive implementation showcased here. See Appendix F for more details.

5 Conclusion

In this work we present SRigL, a novel dynamic sparse training method that learns a variant of structured N:M sparsity. SRigL is capable of sparse-to-sparse training while maintaining generalization performance on par with state-of-the-art unstructured sparse training methods on a wide variety of network architectures. By enforcing a constant fan-in constraint on the learned topology of our networks, the models are amenable to hardware acceleration in a similar manner to N:M sparsity. Our observation that RigL ablates neurons at high sparsities inspires our neuron ablation method which enables SRigL to match the performance of RigL at high sparsities and on the ViT-B/16 network architecture. We present preliminary timings using an non-optimized implementation of our condensed

representation which compare well against a dense layer. We hope this work will motivate future work implementing additional fine-grained structured sparsity schemes within the DST framework.

Acknowledgments and Disclosure of Funding

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Innovates, the Digital Research Alliance of Canada, the NSF AI Institute for Artificial Intelligence and Fundamental Interactions (IAIFI). We are grateful for computational resources made available to us by Google and Denvr Dataworks. We also acknowledge the very helpful feedback of Trevor Gale.

References

Bellec, G., Kappel, D., Maass, W., and Legenstein, R. Deep Rewiring: Training very sparse deep networks. In *International Conference on Learning Representations*, January 2023.

Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv pre-print, 2013.

Cubuk, E. D., Zoph, B., Shlens, J., and Le, Q. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In *Advances in Neural Information Processing Systems*, volume 33, pp. 18613–18624. Curran Associates, Inc., 2020.

Dettmers, T. and Zettlemoyer, L. Sparse Networks from Scratch: Faster Training without Losing Performance. Technical Report arXiv:1907.04840, arXiv, August 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].

Dietrich, A. S. D., Gressmann, F., Orr, D., Chelombiev, I., Justus, D., and Luschi, C. Towards Structured Dynamic Sparse Pre-Training of BERT. Submitted to the International Conference on Learning Representations, January 2022.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*, January 2021.

Elsen, E., Dukhan, M., Gale, T., and Simonyan, K. Fast sparse convnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the Lottery: Making All Tickets Winners. Technical Report arXiv:1911.11134, arXiv, July 2021. arXiv:1911.11134.

Evci, U., Ioannou, Y., Keskin, c., and Dauphin, Y. Gradient Flow in Sparse Neural Networks and How Lottery Tickets Win - AAAI 2022 Poster, February 2022.

Gale, T., Zaharia, M., Young, C., and Elsen, E. Sparse gpu kernels for deep learning, 2020.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: Training imagenet in 1 hour, April 2018.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 2015.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, June 2016. IEEE. ISBN 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.90.

Hoefler, T., Alistarh, D., Ben-Nun, T., Dryden, N., and Peste, A. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv:2102.00554 [cs]*, January 2021. arXiv: 2102.00554.

Jayakumar, S., Pascanu, R., Rae, J., Osindero, S., and Elsen, E. Top-KAST: Top-K Always Sparse Training. In *Advances in Neural Information Processing Systems*, volume 33, pp. 20744–20754. Curran Associates, Inc., 2020.

Jiang, P., Hu, L., and Song, S. Exposing and Exploiting Fine-Grained Block Structures for Fast and Accurate Sparse Training. In *Proceedings of the Neural Information Processing Systems Conference (NeurIPS)*, October 2022.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Li, M., Nica, M., and Roy, D. M. The future is log-gaussian: Resnets and their infinite-depth-and-width limit at initialization. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021.

Littwin, E., Myara, B., Sabah, S., Susskind, J., Zhai, S., and Golan, O. Collegial ensembles. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 18738–18748. Curran Associates, Inc., 2020.

Liu, S., Chen, T., Chen, X., Atashgahi, Z., Yin, L., Kou, H., Shen, L., Pechenizkiy, M., Wang, Z., and Mocanu, D. C. Sparse Training via Boosting Pruning Plasticity with Neuroregeneration. In *Advances in Neural Information Processing Systems*, volume 34, pp. 9908–9922. Curran Associates, Inc., 2021a.

Liu, S., Yin, L., Mocanu, D. C., and Pechenizkiy, M. Do We Actually Need Dense Over-Parameterization? In-Time Over-Parameterization in Sparse Training. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 6989–7000. PMLR, July 2021b. ISSN: 2640-3498.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, December 2018.

Maintainers and Contributors. Torchvision: Pytorch's computer vision library. https://github.com/pytorch/vision, 2016.

McCreary, D. Pytorch implementation of rigging the lottery: Making all tickets winners, Nov 2020. Re-implementation/extension of the work done by Google Research: https://github.com/google-research/rigl.

Mishra, A., Latorre, J. A., Pool, J., Stosic, D., Stosic, D., Venkatesh, G., Yu, C., and Micikevicius, P. Accelerating Sparse Deep Neural Networks, April 2021. arXiv:2104.08378 [cs].

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, December 2018. ISSN 2041-1723. doi: 10.1038/s41467-018-04316-3.

Mostafa, H. and Wang, X. Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 4646–4655. PMLR, May 2019. ISSN: 2640-3498.

Nikdan, M., Pegolotti, T., Iofinova, E., Kurtic, E., and Alistarh, D. SparseProp: Efficient Sparse Backpropagation for Faster Training of Neural Networks, February 2023. arXiv:2302.04852 [cs].

Nvidia. Nvidia A100 Tensor Core GPU Architecture. Technical report, Nvidia, 2020.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision (IJCV)*, 2015.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56): 1929–1958, 2014. ISSN 1533-7928.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818–2826, June 2016. doi: 10.1109/CVPR.2016.308. ISSN: 1063-6919.

Wang, C., Zhang, G., and Grosse, R. Picking winning tickets before training by preserving gradient flow. In *ICLR*, 2020.

Yang, L., Meng, J., Seo, J.-s., and Fan, D. Get More at Once: Alternating Sparse Training with Gradient Correction. In *Proceedings of the Neural Information Processing Systems Conference (NeurIPS)*, October 2022.

Yuan, G., Ma, X., Niu, W., Li, Z., Kong, Z., Liu, N., Gong, Y., Zhan, Z., He, C., Jin, Q., Wang, S., Qin, M., Ren, B., Wang, Y., Liu, S., and Lin, X. MEST: Accurate and Fast Memory-Economic Sparse Training Framework on the Edge. In *Advances in Neural Information Processing Systems*, volume 34, pp. 20838–20850. Curran Associates, Inc., 2021.

Yun, S., Han, D., Chun, S., Oh, S. J., Yoo, Y., and Choe, J. CutMix: Regularization Strategy to Train Strong Classifiers With Localizable Features. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6022–6031, Seoul, Korea (South), October 2019. IEEE. ISBN 978-1-72814-803-8. doi: 10.1109/ICCV.2019.00612.

Zagoruyko, S. and Komodakis, N. Wide Residual Networks, 2017. arXiv:1605.07146.

Zhang, H., Cisse, M., Dauphin, Y. N., and Lopez-Paz, D. mixup: Beyond Empirical Risk Minimization. In *International Conference on Machine Learning*, May 2023.

Zhou, A., Ma, Y., Zhu, J., Liu, J., Zhang, Z., Yuan, K., Sun, W., and Li, H. Learning n:m fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations*, 2021.

Appendices

A Computing the output norm variance

Definition A.1. Let $\xi \in \{0,1\}^N$ be a binary vector. Let $I \in \{0,1\}^{N\times N}$ be an $N \times N$ binary matrix. Let $u \in \mathbb{R}^N$ be any vector. Let $W \in \mathbb{R}^{N\times N}$ be a matrix of iid $\mathcal{N}(0,1)$ random variables.

Define the vector z by:

$$z = \sqrt{\frac{2}{k}}(W \odot I)(\xi \odot u) \tag{4}$$

i.e. the entries z_i are given by:

$$z_i = \sqrt{\frac{2}{k}} \sum_{j=1}^{n} W_{ij} I_{ij} \xi_j u_j \tag{5}$$

Proposition A.2. *The variance of each entry z_i is:*

$$\mathbf{Var}(z_i) = \frac{2}{k} \sum_{j=1}^{n} I_{ij} \xi_j u_j^2 \tag{6}$$

and therefore the distribution of each z_i can be written as

$$z_i \overset{d}{=} g_i \sqrt{\frac{2}{k} \sum_{j=1}^{n} I_{ij} \xi_j u_j^2} \tag{7}$$

where g_i are N iid $\mathcal{N}(0,1)$ random variables.

Proof. By the properties of variance:

$$\mathbf{Var}(z_i) = \frac{2}{k} \sum_{j,j'} I_{ij} I_{ij'} \xi_j \xi_{j'} u_j u_j' \mathbf{Cov}(W_{ij}, W_{ij'}) \tag{8}$$

$$= \frac{2}{k} \sum_{j,j'} I_{ij} I_{ij'} \xi_j \xi_{j'} u_j u_j' \delta_{j=j'} \tag{9}$$

$$= \frac{2}{k} \sum_{j} I_{ij}^2 \xi_j^2 u_j^2 \tag{10}$$

$$= \frac{2}{k} \sum_{j} I_{ij} \xi_j u_j^2 \tag{11}$$

since $I_{ij}^2 = I_{ij}$ and $\xi_j^2 = \xi_j$ because they are binary valued. Once the variance is established, notice that z_i is a linear combination of Gaussians with $z_i \perp z_{i'}$, because the row $W_{ij} \perp W_{i'j}$. Hence the z_i are independent Gaussians, so the form $z_i \overset{d}{=} g_i \sqrt{\frac{2}{k} \sum_{j=1}^{n} I_{ij} \xi_j u_j^2}$ follows. □

Corollary A.3. *The norm $\|z\|^2$ can be written as:*

$$\|z\|^2 \overset{d}{=} \frac{2}{k} \sum_{i,j=1}^{n} g_i^2 I_{ij} \xi_j u_j^2 \tag{12}$$

Proposition A.4 ("Bernoulli Sparsity")**.** *Suppose that $u \in \mathbb{R}^n$ is uniform from the unit sphere, the entries $I_{ij} \sim Ber\left(\frac{k}{n}\right)$, $\xi_j \sim Ber(\frac{1}{2})$ all independent of each other. Then:*

$$\mathbb{E}\left(\|z\|^2\right) = 1 \tag{13}$$

$$\mathbf{Var}\left(\|z\|^2\right) = \frac{5n - 8 + 18\frac{n}{k}}{n(n+2)} \tag{14}$$

Case	Num. Terms	$\mathbb{E}\big[g_i^2 g_{i'}^2\big]$	$\mathbb{E}[I_{i'j'}I_{ij}]$	$\mathbb{E}[\xi_j\xi_{j'}]$	$\mathbb{E}\big[u_j^2 u_{j'}^2\big]$
$i=i',j=j'$	n^2	3	$\frac{k}{n}$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i\neq i',j=j'$	$n^2(n-1)$	1	$\big(\frac{k}{n}\big)^2$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i=i',j\neq j'$	$n^2(n-1)$	3	$\big(\frac{k}{n}\big)^2$	$\big(\frac{1}{2}\big)^2$	$\frac{1}{n(n+2)}$
$i\neq i',j\neq j'$	$n^2(n-1)^2$	1	$\big(\frac{k}{n}\big)^2$	$\big(\frac{1}{2}\big)^2$	$\frac{1}{n(n+2)}$

Table 6: Overview of terms for Bernoulli type sparsity.

Proof. We have

$$\mathbb{E}\Big(\|z\|^2\Big) = \frac{2}{k}\sum_{i,j=1}^{n}\mathbb{E}\big[g_i^2 I_{ij}\xi_j u_j^2\big] \tag{15}$$

$$= \frac{2}{k}\sum_{i,j=1}^{n}\mathbb{E}\big[g_i^2\big]\mathbb{E}[I_{ij}]\mathbb{E}[\xi_j]\mathbb{E}\big[u_j^2\big] \tag{16}$$

$$= \frac{2}{k}\sum_{i,j=1}^{n} 1\cdot\frac{k}{n}\cdot\frac{1}{2}\cdot\frac{1}{n} \tag{17}$$

$$= 1 \tag{18}$$

Similarly, we compute the 4-th moment as follows:

$$\mathbb{E}\Big(\|z\|^4\Big) = \left(\frac{2}{k}\right)^2\sum_{i,j,i',j'}^{n}\mathbb{E}\big[g_i^2 g_{i'}^2\big]\mathbb{E}[I_{i'j'}I_{ij}]\mathbb{E}[\xi_j\xi_{j'}]\mathbb{E}\big[u_j^2 u_{j'}^2\big] \tag{19}$$

We split this into four cases and evaluate these based on whether or not $i=i'$ and $j=j'$ in the following table.

Combining the value of each term with the number of terms gives the desired result for the variance. □

Proposition A.5 ("Constant-per-layer sparsity")**.** *Suppose that $u\in\mathbb{R}^n$ is uniform from the unit sphere and $\xi_j\sim Ber(\frac{1}{2})$ are independent of each other. Suppose the entries of the matrix I_{ij} are chosen such that:*

There are exactly kn ones and exactly n^2-nk zeros in the matrix I, and their positions in the matrix are chosen uniformly from the $\binom{n^2}{nk}$ possible configurations. Then:

$$\mathbb{E}\Big(\|z\|^2\Big) = 1 \tag{20}$$

$$\mathbf{Var}\Big(\|z\|^2\Big) = \frac{(n^2+7n-8)C_{n,k}+18\frac{k}{n}-n^2-2n}{n(n+2)} \tag{21}$$

Proof. Note that $\mathbb{E}(I_{ij})=k/n$ still holds, since there are kn ones distributed over n^2 locations. Thus the computation for $\mathbb{E}(\|z\|^2)$ is identical to the previous proposition. Note also that when there are two entries, we have:

$$\mathbb{E}[I_{ij}I_{i'j'}] = \begin{cases}\frac{k}{n} & \text{if } i=i' \text{ and } j=j'\\ \frac{k}{n}\cdot\frac{nk-1}{n^2-1} & \text{otherwise}\end{cases} \tag{22}$$

$$= \begin{cases}\frac{k}{n} & \text{if } i=i' \text{ and } j=j'\\ \big(\frac{k}{n}\big)^2\cdot C_{n,k} & \text{otherwise}\end{cases} \tag{23}$$

where $C_{n,k}=\frac{n-1/k}{n-1/n}$. The table with terms for computing $\mathbb{E}(\|z\|^4)$ becomes: The extra factor of $C_{n,k}$ in the entries leads to the stated result. □

Case	Num. Terms	$\mathbb{E}\left[g_i^2 g_{i'}^2\right]$	$\mathbb{E}[I_{i'j'}I_{ij}]$	$\mathbb{E}[\xi_j\xi_{j'}]$	$\mathbb{E}\left[u_j^2 u_{j'}^2\right]$
$i=i', j=j'$	n^2	3	$\frac{k}{n}$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i\neq i', j=j'$	$n^2(n-1)$	1	$\left(\frac{k}{n}\right)^2 C_{n,k}$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i=i', j\neq j'$	$n^2(n-1)$	3	$\left(\frac{k}{n}\right)^2 C_{n,k}$	$\left(\frac{1}{2}\right)^2$	$\frac{1}{n(n+2)}$
$i\neq i', j\neq j'$	$n^2(n-1)^2$	1	$\left(\frac{k}{n}\right)^2 C_{n,k}$	$\left(\frac{1}{2}\right)^2$	$\frac{1}{n(n+2)}$

Table 7: Overview of terms for Constant-per-layer type sparsity.

Case	Num. Terms	$\mathbb{E}\left[g_i^2 g_{i'}^2\right]$	$\mathbb{E}[I_{i'j'}I_{ij}]$	$\mathbb{E}[\xi_j\xi_{j'}]$	$\mathbb{E}\left[u_j^2 u_{j'}^2\right]$
$i=i', j=j'$	n^2	3	$\frac{k}{n}$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i\neq i', j=j'$	$n^2(n-1)$	1	$\left(\frac{k}{n}\right)^2$	$\frac{1}{2}$	$\frac{3}{n(n+2)}$
$i=i', j\neq j'$	$n^2(n-1)$	3	$\frac{k}{n}\cdot\frac{k-1}{n-1}$	$\left(\frac{1}{2}\right)^2$	$\frac{1}{n(n+2)}$
$i\neq i', j\neq j'$	$n^2(n-1)^2$	1	$\left(\frac{k}{n}\right)^2$	$\left(\frac{1}{2}\right)^2$	$\frac{1}{n(n+2)}$

Table 8: Overview of terms for Constant-fan-in type sparsity.

Proposition A.6 ("Constant Fan-In sparsity")**.** *Suppose that $u\in\mathbb{R}^n$ is uniform from the unit sphere, and $\xi_j\sim Ber(\frac{1}{2})$ all independent of each other. Suppose the entries of the matrix I_{ij} are chosen so that:*

1. *There are exactly k ones **in each row** of the matrix I and exactly $n-k$ zeros in the matrix I, chosen uniformly from the $\binom{n}{k}$ possible ways this can happen.*
2. *Different rows of I are independent.*

Then:

$$\mathbb{E}\left(\|z\|^2\right)=1 \tag{24}$$

$$\mathbf{Var}\left(\|z\|^2\right)=\frac{5n-8+18\frac{n}{k}}{n(n+2)}-\frac{3(n-k)}{kn(n+2)} \tag{25}$$

Proof. Same arguments as before apply, but now we have

$$\mathbb{E}[I_{ij}I_{i'j'}]=\begin{cases}\frac{k}{n} & \text{if } i=i' \text{ and } j=j'\\ \frac{k}{n}\frac{k-1}{n-1} & \text{if } i=i' \text{ and } j\neq j'\\ \left(\frac{n}{n}\right)^2 & \text{otherwise}\end{cases} \tag{26}$$

(27)

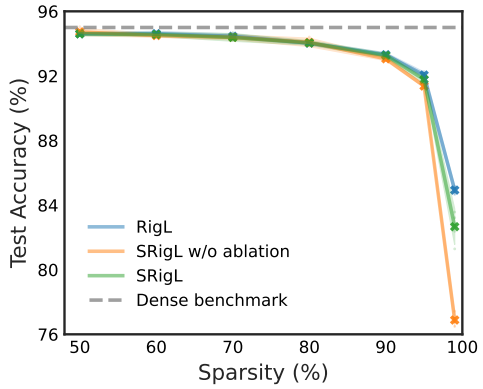
and the table for the variance computation becomes:

Which leads to the stated result. □

(28)

B Wide ResNet-22 trained on CIFAR-10

In Fig. 6 we present results of training Wide ResNet-22 (Zagoruyko & Komodakis, 2017) with RigL or SRigL on the CIFAR-10 dataset. The training details for this experiment are identical to those reported in Section 4.1. SRigL without ablation performs poorly at very high sparsities. With ablation, SRigL achieves generalization performance comparable to RigL.



sparsity (%)	RigL	SRigL w/o	SRigL w/ ablation
50	94.6±0.1	94.7±0.1	94.6±0.1
60	94.6±0.1	94.5±0.1	94.6±0.1
70	94.5±0.1	94.4±0.1	94.4±0.1
80	94.0±0.1	94.1±0.2	94.0±0.1
90	93.3±0.1	93.1±0.1	93.3±0.1
95	92.1±0.1	91.4±0.1	91.8±0.2
99	**84.9±0.2**	76.9±0.3	82.7±0.8
0	*dense Wide ResNet-22*: 95.0		

Figure 6 & Table 9: Test accuracy of Wide ResNet-22 trained on CIFAR-10. Mean and 95% confidence intervals are reported over five runs.

C Hyperparameter and training details

C.1 ResNet-18 trained on CIFAR-10

We train each network for 250 epochs (97,656 steps) using a batch size of 128. An initial learning rate of 0.1 is reduced by a factor of 5 every 77 epochs (about 30,000 steps). We use stochastic gradient descent (SGD) with momentum, with an L2 weight decay coefficient of 5e-4 and momentum coefficient of 0.9. We train each model using a single Nvidia V100 GPU.

We achieve the desired overall sparsity by distributing the per-layer sparsity according to the Erdős-Rényi-Kernel (ERK) (Evci et al., 2021; Mocanu et al., 2018) distribution, which scales the per-layer sparsity based on the number of neurons and the dimensions of the convolutional kernel, if present. We set the number of mini-batch steps between connectivity updates, ΔT, to 100. γ_{sal} is set at 30% based on the results of a small grid search performed on CIFAR-10 with ResNet-18 and Wide ResNet-22. See Fig. 7 for details.

For each trial, we select a desired sparsity in the range from 0.5 to 0.99. At each connectivity update, the portion of weights to be pruned or regrown is based on a cosine annealing schedule (Dettmers & Zettlemoyer, 2019) with an initial value $\alpha = 0.3$. The portion of weights to be updated decays from the initial value to zero once 75% of the total training steps have been completed, after which the weight mask remains constant.

C.2 ResNet-50 trained on Imagenet

We use a mini-batch size of 512 instead of 4096, We linearly scale the learning rate and ΔT to account for our smaller batch size. Linearly scaling the learning rate in this manner was included in the original RigL source code and is further motivated by Goyal et al. (2018). We increase ΔT to 800 and average the dense gradients over eight mini-batch steps to ensure that SRigL has the same quality of parameter saliency information available as RigL at each network connectivity update. We set γ_{sal} to 30% based on our grid search presented in Fig. 7.

Our learning rate uses a linear warm-up to reach a maximum value of 0.2 at epoch five and is reduced by a factor of 10 at epochs 30, 70, and 90. Using a mini-batch of 512, we train the networks for 256,000 steps to match RigL's training duration. We use a cosine connectivity update schedule with $\alpha = 0.3$. We initialize the sparse model weights per Evci et al. (2022). We train the networks using SGD with momentum, L2 weight decay, and label smoothing Szegedy et al. (2016) coefficients of 0.9, 1e-4 and 0.1, respectively.

We use the same standard data augmentation in our data preprocessing as RigL, including randomly resizing to 256×256 or 480×480 pixels, random crops to 224×224 pixels, random horizontal flips, and per-image normalization to zero mean and unit variance using identical per RGB channel mean and standard deviation values as RigL. We train each model using either four Nvidia V100 or A100 GPUs.

C.3 Vision Transformer trained on ImageNet

We add additional data augmentations following the standard TorchVision (Maintainers & Contributors, 2016) ViT-B/16 training procedure for ImageNet. These data augmentations applied include: random cropping, resizing the cropped image to 224 by 224 pixels, randomly horizontal flips, randomly augmenting with RandAugment algorithm (Cubuk et al., 2020), and normalizing with the typical RGB channel mean and standard deviation values. We also randomly choose one of random mixup (Zhang et al., 2023) or random cutmix (Yun et al., 2019) and add it to the above-noted augmentations. We use 0.2 and 1.0 for the alpha parameter values for mixup and cutmix, respectively.

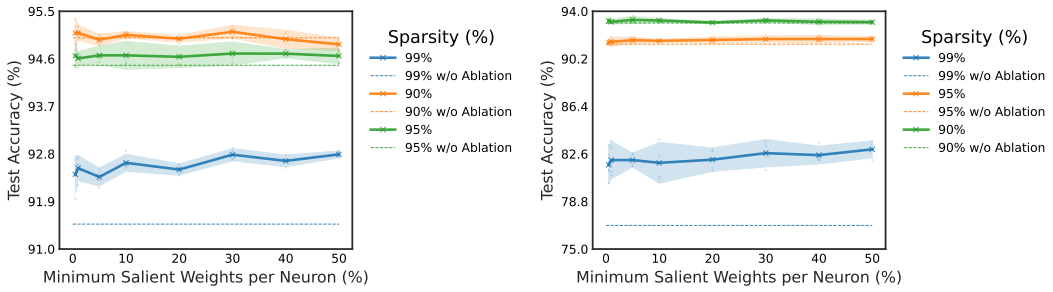
We omit Dropout (Srivastava et al., 2014) from the model entirely to avoid potential layer collapse in the case where all non-zero weights are dropped from a layer and to avoid any other unintended interference with SRigL's sparse training procedure.

We sample eight mini-batch steps with 512 samples per mini-batch and accumulate gradients before applying the optimizer, resulting in an effective mini-batch size of 4096. We train the model for 150 epochs using an AdamW (Loshchilov & Hutter, 2018) optimizer with weight decay, label smoothing, β_1, and β_2 coefficients of 0.3, 0.11, 0.9, and 0.999, respectively. We use cosine annealing with linear warm-up for our learning rate scheduler with an initial learning rate of 9.9e-5 that warms-up to a maximum value of 0.003 at epoch 16. We clip all parameter gradients to a max L2 norm of 1.0. We apply uniformly distributed sparsity across all layers in the model. ΔT is set to 100 to update network connectivity every 100 mini-batch steps. We train each model using either four Nvidia V100 or A100 GPUs.

D Tuning γ_{sal}, minimum percentage salient weights per neuron

Fig. 7 depicts the generalization performance of highly sparse ResNet-18 and Wide ResNet-22 models trained on the CIFAR-10 dataset. SRigL's generalization performance at high sparsities is improved with neuron ablation; however, the specific value selected for γ_{sal} does not have a significant effect on performance. Our experiments demonstrate that SRigL performs well with a variety of γ_{sal} values. In Section 4 we report the results of SRigL models trained with γ_{sal} set to 30%. With dynamic ablation enabled, we set the minimum salient weights per neuron to one if the user-defined threshold results in a value less than one. In Fig. 9, many layers in ResNet-50 are set to the minimum threshold of one when we apply a γ_{sal} of 30% for all model types other than ViT-B/16. This minimum threshold explains the invariance of the model's performance when comparing against multiple values for γ_{sal}.

Fig. 8 demonstrates how ViT-B/16's generalization performance is much more sensitive to γ_{sal}. We find that RigL learns a sparse connectivity pattern with a large variance in sparse fan-in between neurons within a given layer, with some neurons having an order of magnitude more fan-in connection than the mean fan-in.



(a) ResNet-18/CIFAR-10 **(b)** Wide ResNet-22/CIFAR-10

Figure 7: (a) ResNet18/CIFAR-10 Test Accuracy vs. γ_{sal} when trained with SRigL with and without ablation for a range of sparsities. The mean and 95% confidence intervals are shown for five different random seeds for the runs with ablation. For the runs without ablation, we report the mean of five different random seeds. **(b) Wide ResNet-22 Test Accuracy vs.** γ_{sal}. The mean and 95% confidence intervals are shown for five different random seeds.

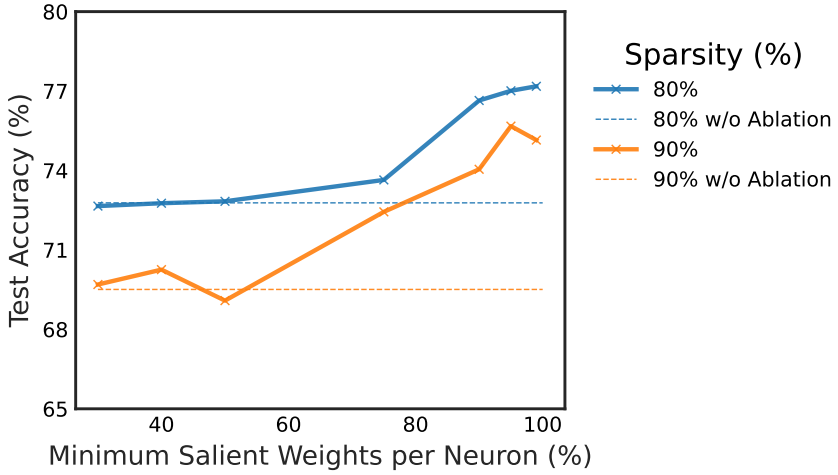


Figure 8: ViT-B/16/ImageNet Test Accuracy vs. γ_{sal} when trained with SRigL with and without ablation enabled for 80% and 90% sparsity. ViT-B/16's performance is much more sensitive to γ_{sal} and generally performs best with high ablation thresholds. Based on this data we set γ_{sal} to 95% for our results reported in Section 4.3.

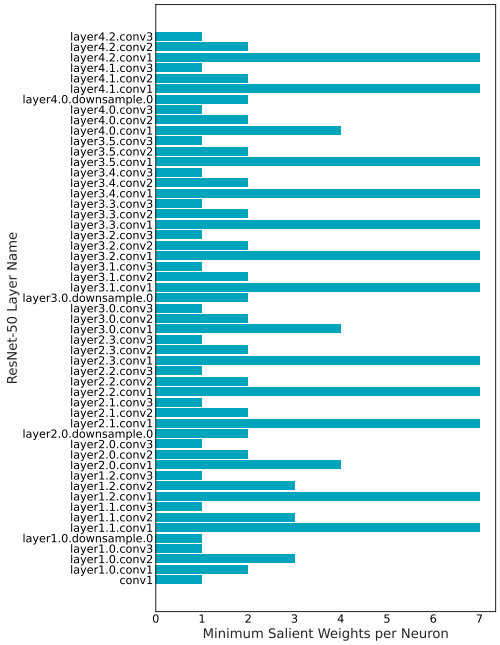


Figure 9: ResNet-50 Layer vs. Minimum salient weights per neuron. SRigL sets the minimum salient weight per neuron to 1 if the product between γ_{sal} and the sparse fan-in per neuron is less than 1. Therefore, even in a relatively large network such as ResNet50 many of the layers only require that a single weight be active to keep the neuron active. We believe this is why SRigL's performance is relatively invariant to various ablation thresholds when applied to convolutional networks.

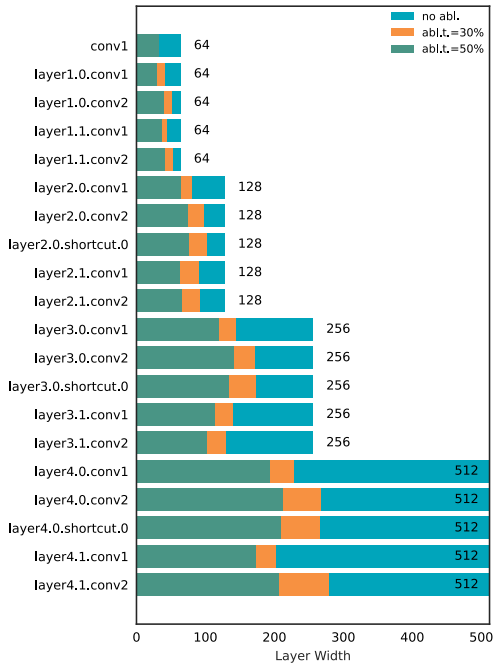


Figure 10: ResNet-18/CIFAR-10 layer widths at the end of training at 99% sparsity. Without ablation, constant fan-in constraint enforces that sparse layers retain their original width. When ablation is enabled, the γ_{sal} threshold (minimum percentage salient weights per neuron) is used to control the amount of ablation.

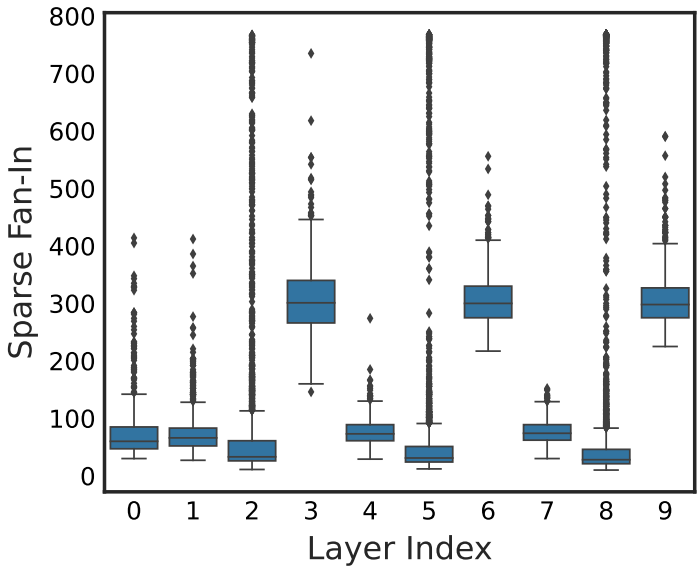


Figure 11: Sparse Fan-In vs. ViT-B/16 layer index at the end of training with RigL at 90% sparsity. Only the first 10 layers are shown for clarity. We find that RigL learns a sparse connectivity with large variance in fan-in between neurons within the same layer with some neurons receiving up to $\times 10$ the number of active connections than the mean for the same layer.

E Condensed matrix multiplication

Using a constant fan-in sparse representation presents an advantage compared to the general N:M sparse representation in that we can represent our weight matrices in a compact form, since every neuron/convolutional filter has the same number of non-zero weights. Here we demonstrate how this can be used to accelerate a fully-connected layer.

Consider the standard matrix-vector product:

$$Wv = \begin{pmatrix} W_{11} & W_{12} & \dots & W_{1d} \\ W_{21} & W_{22} & \dots & W_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ W_{n1} & W_{n2} & \dots & W_{nd} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_d \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{d} W_{1j} v_j \\ \sum_{j=1}^{d} W_{2j} v_j \\ \vdots \\ \sum_{j=1}^{d} W_{nj} v_j \end{pmatrix} = v^{\text{out}} \tag{29}$$

When $W \in \mathbb{R}^{n \times d}$ is sparse and has only k non-zero elements per row, the sums representing each element of v^{out} will be limited to k terms, i.e.:

$$v_i^{\text{out}} = \sum_{\alpha=1}^{k} W_{ij_\alpha} v_{j_\alpha} \quad \text{with } j_\alpha \in \{1,\dots,d\}, \quad j_\alpha \neq j_{\alpha'} \tag{30}$$

Note that the expression on the right-hand side of Eq. (29) can be represented as an operation between a dense matrix $W^c \in \mathbb{R}^{n \times k}$ (we call it "condensed W") and k vectors $v^{\pi_1},\dots,v^{\pi_k}$, $v^{\pi_i} \in \mathbb{R}^n$, whose elements are drawn from v with replacement (we call them "recombinations of v"). The operation is a sum over element-wise products between the i-th column of W^c and the i-th column vector v^{π_i}:

$$Wv = \sum_{i=1}^{k} W^c_{:,i} \odot v^{\pi_i} \tag{31}$$

Mathematically, these methods are equivalent for any matrices. Computationally, the condensed method can be more efficient, in particular for sparse matrices with constant small fan-in k. By

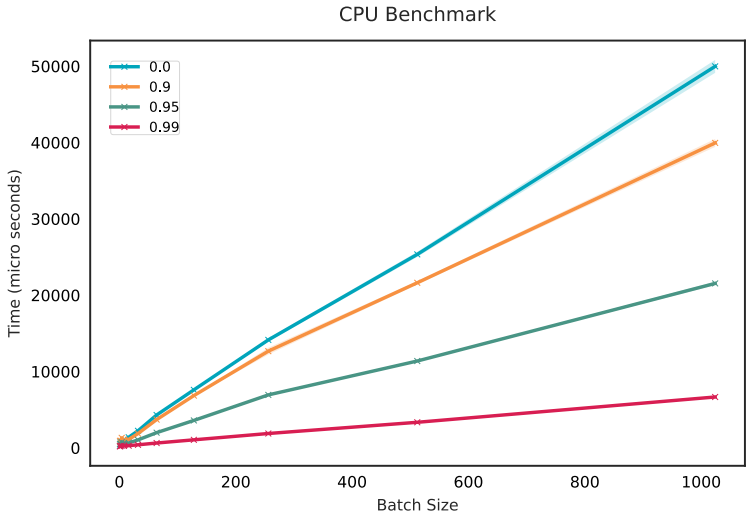


Figure 12: SparseProp Timings Benchmarking our condensed representation on CPU with the SparseProp backend (Intel(R) Xeon(R) E5-2683 v4 @ 2.10GHz).

construction, this method requires the sparse matrix W to be stored in dense representation which involves two 2D arrays of shape $n \times k$: One holds the *values* of the non-zero elements of W and the other one their respective *column indices*, which are used to generate input vector re-combinations. An efficient computational implementation of this method is subject of ongoing work on this project. Based on our results, the constant fan-in constraint does not appear to have a limiting effect on SNNs.

F SparseProp backend benchmarks

Our straight-forward PyTorch implementation is only faster than the linear layer benchmark at sparsities of 99% as demonstrated in Fig. 5. However, since our condensed representation GEMM algorithm was implemented in python directly, it suffers significant overhead compared to C++ or CUDA backends. Here we demonstrate that the SparseProp (Nikdan et al., 2023) C++ backend is capable of accelerating our sparse models in real-world settings. We report mean timings and standard deviations across a minimum of five forward passes for a single condensed layer with SparseProp backend at sparsities of 99%, 95%, and 90% and a standard dense PyTorch Linear layer. Both the condensed and linear layer are based on the linear classifier in a ResNet-50 model. The layers have 1000 neurons and we use an input tensor with 2048 features. We vary the batch size between 1 to 1024. The input tensors consist of 32 bit floating point values and the convolutional weight tensors are in a channel-first layout typical to PyTorch (batch size, channels, height, width). This result is highly promising since the SparseProp backend does not utilized the constant fan-in constraint. Additional acceleration is expected with a suitable backend designed to exploit the constant fan-in constraint.