UNDER REVIEW

RESCIENCE C

Editorial / ML Reproducibility Challenge 2020

[Re] Rigging the Lottery: Making All Tickets Winners

Varun Sundar[1, ID] and Rajat Vadiraj Dwaraknath[2, ID]

[1]University of Wisconsin Madison, Wisconsin, USA – [2]Stanford University, California, USA

Edited by
(Editor)

Reviewed by
Anonymous Reviewers

Received
01 November 2018

Published
–

DOI
–

Reproducibility Summary

Scope of Reproducibility

For a fixed parameter count and compute budget, the proposed algorithm (*RigL*) claims to directly train sparse networks that match or exceed the performance of existing dense-to-sparse training techniques (such as pruning). *RigL* does so while requiring constant Floating Point Operations (FLOPs) throughout training. The technique obtains state-of-the-art performance on a variety of tasks, including image classification and character-level language-modelling.

Methodology

We implement *RigL* from scratch in Pytorch using boolean masks to simulate unstructured sparsity. We rely on the description provided in the original paper, and referred to the authors' code for only specific implementation detail such as handling overflow in ERK initialization. We evaluate sparse training using *RigL* for WideResNet-22-2 on CIFAR-10 and ResNet-50 on CIFAR-100, requiring 2 hours and 6 hours respectively per training run on a GTX 1080 GPU.

Results

We reproduce *RigL*'s performance on CIFAR-10 within 0.1% of the reported value. On both CIFAR-10/100, the central claim holds—given a fixed training budget, *RigL* surpasses existing dynamic-sparse training methods over a range of target sparsities. By training longer, the performance can match or exceed iterative pruning, while consuming constant FLOPs throughout training. We also show that there is little benefit in tuning *RigL*'s hyper-parameters for every sparsity, initialization pair—the reference choice of hyperparameters is often close to optimal performance.

Going beyond the original paper, we find that the optimal initialization scheme depends on the training constraint. While the Erdos-Renyi-Kernel distribution outperforms Random distribution for a fixed parameter count, for a fixed FLOP count, the latter performs better. Finally, redistributing layer-wise sparsity while training can bridge the performance gap between the two initialization schemes, but increases computational cost.

Correspondence should be addressed to Varun Sundar (vsundar4@wisc.edu)
The authors have declared that no competing interests exist.
Code is available at https://github.com/varun19299/rigl-reproducibility. – SWH swh:1:dir:0707870fafa16ef60dc64071e1aed482e373e75f.
Open peer review is available at https://openreview.net/forum?id=riCIeP6LzEE.

What was easy

The authors provide code for most of the experiments presented in the paper. The code was easy to run and allowed us to verify the correctness of our re-implementation. The paper also provided a thorough and clear description of the proposed algorithm without any obvious errors or confusing exposition.

What was difficult

Tuning hyperparameters involved multiple random seeds and took longer than anticipated. Verifying the correctness of a few baselines was tricky and required ensuring that the optimizer's gradient (or momentum) buffers were sparse (or dense) as specified by the algorithm. Compute limits restricted us from evaluating on larger datasets such as Imagenet.

Communication with original authors

We had responsive communication with the original authors, which helped clarify a few implementation and evaluation details, particularly regarding the FLOP counting procedure.

1 Introduction

Sparse neural networks are a promising alternative to conventional dense networks—having comparatively greater parameter efficiency and lesser floating-point operations (FLOPs) (Han et al., Ashby et al., Srinivas, Subramanya, and Venkatesh Babu[1,2,3]). Unfortunately, present techniques to produce sparse networks of commensurate accuracy involve multiple cycles of training dense networks and subsequent pruning. Consequently, such techniques offer no advantage over training dense networks, either computationally or memory-wise.

In the paper Evci et al.[4], the authors propose *RigL*, an algorithm for training sparse networks from scratch. The proposed method outperforms both prior art in training sparse networks, as well as existing dense-to-sparse training algorithms. By utilising dense gradients only during connectivity updates and avoiding any global sparsity redistribution, *RigL* can maintain a fixed computational cost and parameter count throughout training.

As a part of the ML Reproducibility Challenge, we replicate *RigL* from scratch and investigate if dynamic-sparse training confers significant practical benefits compared to existing sparsifying techniques.

2 Scope of reproducibility

In order to verify the central claims presented in the paper we focus on the following target questions:

- Does *RigL* outperform existing sparse-to-sparse training techniques—such as SET (Mocanu et al.[5]) and SNFS (Dettmers and Zettlemoyer[6])—and match the accuracy of dense-to-sparse training methods such as iterative pruning (Zhu and Gupta[7])?
- *RigL* requires two additional hyperparameters to tune. We investigate the sensitivity of final performance to these hyperparameters across a variety of target sparsities (Section 5.3).
- How does the choice of sparsity initialization affect the final performance for a fixed parameter count and a fixed training budget (Section 6.1)?
- Does redistributing layer-wise sparsity during connection updates (Dettmers and Zettlemoyer[6]) improve *RigL*'s performance? Can the final layer-wise distribution serve as a good sparsity initialization scheme (Section 6.2)?

3 Methodology

The authors provide publicly accessible code[1] written in Tensorflow (Abadi et al.[8]). To gain a better understanding of various implementation aspects, we opt to replicate *RigL* in Pytorch (Paszke et al.[9]). Our implementation extends the open-source code[2] of Dettmers and Zettlemoyer[6] which uses a boolean mask to simulate unstructured sparsity. Our source code is publicly accessible on Github[3] with training plots available on WandB[4] (Biewald[10]).

[1] https://github.com/google-research/rigl
[2] https://github.com/TimDettmers/sparse_learning
[3] https://github.com/varun19299/rigl-reproducibility
[4] https://wandb.ai/ml-reprod-2020

Mask Initialization — For a network with L layers and total parameters N, we associate each layer with a random boolean mask of sparsity s_l, $l \in [L]$. The overall sparsity of the network is given by $S = \frac{\sum_l s_l N_l}{N}$, where N_l is the parameter count of layer l. Sparsities s_l are determined by the one of the following mask initialization strategies:

- **Uniform:** Each layer has the same sparsity, i.e., $s_l = S \; \forall l$. Similar to the original authors, we keep the first layer dense in this initialization.
- **Erdos-Renyi (ER):** Following Mocanu et al.[5], we set $s_l \propto \left(1 - \frac{C_{\text{in}} + C_{\text{out}}}{C_{\text{in}} \times C_{\text{out}}}\right)$, where $C_{\text{in}}, C_{\text{out}}$ are the in and out channels for a convolutional layer and input and output dimensions for a fully-connected layer.
- **Erdos-Renyi-Kernel (ERK):** Modifies the sparsity rule of convolutional layers in ER initialization to include kernel height and width, i.e., $s_l \propto \left(1 - \frac{C_{\text{in}} + C_{\text{out}} + w + h}{C_{\text{in}} \times C_{\text{out}} \times w \times h}\right)$, for a convolutional layer with $C_{\text{in}} \times C_{\text{out}} \times w \times h$ parameters.

We do not sparsify either bias or normalization layers, since these have a negligible effect on total parameter count.

Mask Updates — Every ΔT training steps, certain connections are discarded, and an equal number are grown. Unlike SNFS (Dettmers and Zettlemoyer[6]), there is no redistribution of layer-wise sparsity, resulting in constant FLOPs throughout training.

Pruning Strategy — Similar to SET and SNFS, *RigL* prunes f fraction of smallest magnitude weights in each layer. As detailed below, the fraction f is decayed across mask update steps, by cosine annealing:

$$f(t) = \frac{\alpha}{2}\left(1 + \cos\left(\frac{t\pi}{T_{\text{end}}}\right)\right) \tag{1}$$

where, α is the initial pruning rate and T_{end} is the training step after which mask updates are ceased.

Growth Strategy — *RigL*'s novelty lies in how connections are grown: during every mask update, k connections having the largest absolute gradients among current inactive weights (previously zero + pruned) are activated. Here, k is chosen to be the number of connections dropped in the prune step. This requires access to dense gradients at each mask update step. Since gradients are not accumulated (unlike SNFS), *RigL* does not require access to dense gradients at *every* step. Following the paper, we initialize newly activated weights to zero.

4 Experimental Settings

4.1 Model descriptions

For experiments on CIFAR-10 (Alex Krizhevsky[11]), we use a Wide Residual Network (Zagoruyko and Komodakis[12]) with depth 22 and width multiplier 2, abbreviated as WRN-22-2. For experiments on CIFAR-100 (Alex Krizhevsky[11]), we use a modified variant of ResNet-50 (He et al.[13]), with the initial 7×7 convolution replaced by two 3×3 convolutions (architecture details provided in the supplementary material).

Table 1. Test accuracy of reference and our implementations on CIFAR-10, tabulated for three different sparsities. Note that the runs listed here do not use a separate validation set while training.

Method	Ours			Original		
Dense	94.6			94.1		
	$1-s=0.1$	$1-s=0.2$	$1-s=0.5$	$1-s=0.1$	$1-s=0.2$	$1-s=0.5$
Static (ERK)	91.6	93.2	94.3	91.6	92.9	94.2
Pruning	93.2	93.6	94.3	93.3	93.5	94.1
RigL (ERK)	93.2	93.8	94.4	93.1	93.8	94.3

4.2 Datasets and Training descriptions

We conduct our experiments on the CIFAR-10 and CIFAR-100 image classification datasets. For CIFAR-10, we use a train/val/test split of 45k/5k/10k samples. In comparison, the authors use no dedicated validation set, with 50k samples and 10k samples comprising the train set and test set, respectively. This causes a slight performance discrepancy between our reproduction and the metrics reported by the authors (dense baseline has a test accuracy of 93.4% vs 94.1% reported). However, our replication matches the paper's performance when 50k samples are used for the train set (Table 4). We use a validation split of 10k samples for CIFAR-100 as well.

On both datasets, we train models for 250 epochs each, optimized by SGD with momentum. Our training pipeline uses standard data augmentation, which includes random flips and crops. When training on CIFAR-100, we additionally include a learning rate warmup for 2 epochs and label smoothening of 0.1 (Goyal et al.[14]). We also initialize the last batch normalization layer (Ioffe and Szegedy[15]) in each BottleNeck block to 0, following He et al.[16].

4.3 Hyperparameters

RigL includes two additional hyperparameters ($\alpha, \Delta T$) in comparison to regular dense network training. In Sections 5.1 and 5.2, we set $\alpha = 0.3, \Delta T = 100$, based on the original paper. Optimizer specific hyperparameters—learning rate, learning rate schedule, and momentum—are also set according to the original paper. In Section 5.3, we tune these hyperparameters with Optuna (Akiba et al.[17]). We also examine whether indivdually tuning the learning rate for each sparsity value offers any significant benefit.

4.4 Baseline implementations

We compare *RigL* against various baselines in our experiments: SET (Mocanu et al.[5]), SNFS (Dettmers and Zettlemoyer[6]), and Magnitude-based Iterative-pruning (Zhu and Gupta[7]). We also compare against two weaker baselines, viz., *Static Sparse* training and *Small-Dense* networks. The latter has the same structure as the dense model but uses fewer channels in convolutional layers to lower parameter count. We implement iterative pruning with the pruning interval kept same as the masking interval for a fair comparison.

4.5 Computational requirements

We run our experiments on a SLURM cluster node—equipped with 4 NVIDIA GTX1080 GPUs and a 32 core Intel CPU. Each experiment on CIFAR-10 and CIFAR-100 consumes about 1.6 GB and 7 GB of VRAM respectively and is run for 3 random seeds to capture performance variance. We require about 6 and 8 days of total compute time to produce

Table 2. WideResNet-22-2 on CIFAR10, tabulated for two density ($1 - s$) values. We group methods by their FLOP requirement and in each group, we mark the best accuracy in bold. Similar to Evci et al.[4], we assume that algorithms utilize sparsity during training. All results are obtained by methods implemented in our unified codebase.

Method	$1 - s = 0.1$		$1 - s = 0.2$	
	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)
Small Dense	89.0 ± 0.35	0.11x, 0.11x	91.0 ± 0.07	0.20x, 0.20x
Static	89.1 ± 0.17	0.10x, 0.10x	91.2 ± 0.16	0.20x,0.20x
SET	91.3 ± 0.47	0.10x, 0.10x	**92.7 ± 0.28**	0.20x, 0.20x
RigL	**91.7 ± 0.18**	0.10x, 0.10x	92.6 ± 0.10	0.20x, 0.20x
SET (ERK)	92.2 ± 0.04	0.17x, 0.17x	92.9 ± 0.16	0.35x, 0.35x
RigL (ERK)	**92.4 ± 0.06**	0.17x, 0.17x	**93.1 ± 0.09**	0.35x, 0.35x
$\text{Static}_{2\times}$	89.15 ± 0.17	0.20x, 0.10x	91.2 ± 0.16	0.40x, 0.20x
Lottery	90.4 ± 0.09	0.45x, 0.13x	92.0 ± 0.31	0.68x,0.27x
$\text{SET}_{2\times}$	83.3 ± 15.33	0.20x, 0.10x	93.0 ± 0.22	0.41x, 0.20x
SNFS	92.4 ± 0.43	0.51x, 0.27x	92.7 ± 0.20	0.66x, 0.49x
SNFS (ERK)	92.2 ± 0.2	0.52x, 0.28x	92.8 ± 0.07	0.66x, 0.49x
$\text{SNFS}_{2\times}$	92.3 ± 0.33	1.02x, 0.27x	93.2 ± 0.14	1.32x, 0.98x
$\text{RigL}_{2\times}$	92.3 ± 0.25	0.20x, 0.10x	93.0 ± 0.21	0.41x, 0.20x
Pruning	92.6 ± 0.08	0.32x,0.13x	93.2 ± 0.27	0.41x,0.27x
$\text{RigL}_{2\times}$ (ERK)	**92.7 ± 0.37**	0.34x, 0.17x	**93.3 ± 0.09**	0.70x, 0.35x
Dense Baseline	**93.4 ± 0.07**	9.45e8, 3.15e8	-	-

all results, including hyper-parameter sweeps and extended experiments, on CIFAR-10 and CIFAR-100 respectively.

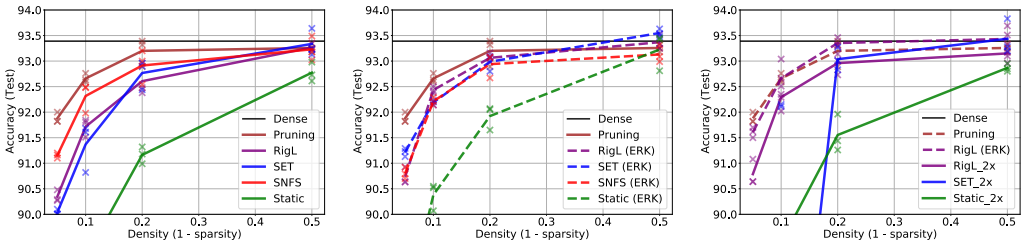
5 Results

Given a fixed training FLOP budget, *RigL* surpasses existing dynamic sparse training methods over a range of target sparsities, on both CIFAR-10 and 100 (Sections 5.1, 5.2). By training longer, *RigL* matches or marginally outperforms iterative pruning. However, unlike pruning, its FLOP consumption is constant throughout. This a prime reason for using sparse networks, and makes training larger networks feasible. Finally, as evaluated on CIFAR-10, the original authors' choice of hyper-parameters are close to optimal for multiple target sparsities and initialization schemes (Section 5.3).

5.1 WideResNet-22 on CIFAR-10

Results on the CIFAR-10 dataset are provided in Table 2. Tabulated metrics are averaged across 3 random seeds and reported with their standard deviation. All sparse networks use random initialization, unless indicated otherwise.

While SET improves over the performance of static sparse networks and small-dense networks, methods utilizing gradient information (SNFS, *RigL*) obtain better test accuracies. SNFS can outperform *RigL*, but requires a much larger training budget, since it (a) requires dense gradients at each training step, (b) redistributes layer-wise sparsity during mask updates. For all sparse methods, excluding SNFS, using ERK initialization improves performance, but with increased FLOP consumption. We calculate theoretical FLOP requirements in a manner similar to Evci et al.[4] (exact procedure is described



(a) Random initialization (b) ERK initialization (c) 2x longer

Figure 1. Test Accuracy vs Sparsity on CIFAR-10, plotted for Random initialization **(left)**, ERK initialization **(center)**, and for training $2\times$ longer **(right)**. Owing to random growth, SET can be unstable when training for longer durations with higher sparsities. Overall, $RigL_{2\times}$ (ERK) achieves highest test accuracy.

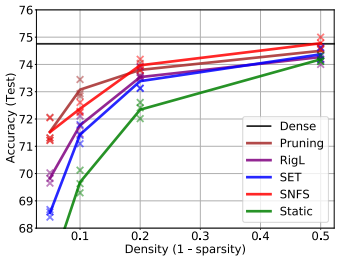
in the appendix).

Figure 1 contains test accuracies of select methods across two additional sparsity values: $(0.5, 0.95)$. At lower sparsities (higher densities), *RigL* matches the performance of the dense baseline. Performance further improves by training for longer durations. Particularly, training *RigL* (ERK) twice as long at 90% sparsity exceeds the performance of iterative pruning while requiring similar theoretical FLOPs. This validates the original authors' claim that *RigL* (a sparse-to-sparse training method) outperforms pruning (a dense-to-sparse training method).

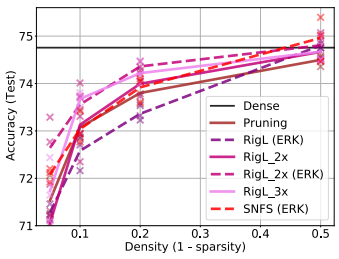
5.2 ResNet-50 on CIFAR100

Table 3 & Figure 2. Benchmarking sparse ResNet-50s on CIFAR-100, tabulated by performance and cost **(below)**, and plotted across densities **(right)**. In each group below, *RigL* outperforms or matches existing sparse-to-sparse and dense-to-sparse methods. Notably, $RigL_{3\times}$ at 90% sparsity and $RigL_{2\times}$ at 80% sparsity surpass iterative pruning with similar FLOP consumption. $RigL_{2\times}$ (ERK) further improves performance but requires a larger training budget.

Method	$1-s=0.1$		$1-s=0.2$	
	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)
Static	69.7 ± 0.42	0.10x, 0.10x	72.3 ± 0.30	0.20x,0.20x
Small Dense	70.8 ± 0.22	0.11x, 0.11x	72.6 ± 0.93	0.20x, 0.20x
SET	71.4 ± 0.35	0.10x, 0.10x	73.4 ± 0.45	0.20x, 0.20x
RigL	$\mathbf{71.8 \pm 0.33}$	0.10x, 0.10x	$\mathbf{73.5 \pm 0.04}$	0.20x, 0.20x
Static (ERK)	71.5 ± 0.18	0.22x, 0.22x	73.2 ± 0.39	0.38x, 0.38x
SET (ERK)	72.3 ± 0.39	0.22x, 0.22x	$\mathbf{73.5 \pm 0.25}$	0.38x, 0.38x
RigL (ERK)	$\mathbf{72.6 \pm 0.37}$	0.23x, 0.22x	73.4 ± 0.15	0.38x, 0.38x
SNFS	72.3 ± 0.20	0.58x, 0.37x	73.9 ± 0.20	0.70x, 0.55x
SNFS (ERK)	73.0 ± 0.33	0.59x, 0.38x	73.9 ± 0.27	0.69x, 0.54x
Pruning	73.1 ± 0.32	0.36x,0.11x	73.8 ± 0.23	0.45x,0.25x
$RigL_{2\times}$	73.1 ± 0.71	0.20x, 0.10x	74.0 ± 0.24	0.41x, 0.20x
Lottery	73.6 ± 0.32	0.62x,0.11x	74.2 ± 0.41	0.81x,0.25x
$\mathbf{RigL_{3\times}}$	$\mathbf{73.7 \pm 0.16}$	0.30x, 0.10x	74.2 ± 0.23	0.61x, 0.20x
$\mathbf{RigL_{2\times}}$ **(ERK)**	73.6 ± 0.05	0.46x, 0.22x	$\mathbf{74.4 \pm 0.10}$	0.76x, 0.38x
Dense Baseline	$\mathbf{74.7 \pm 0.38}$	7.77e9, 2.59e9	-	-



(a) Random initialization



(b) ERK initialization, Extended training

We see similar trends when training sparse variants of ResNet-50 on the CIFAR-100 dataset (Table 3, metrics reported as in Section 5.1). We also include a comparison against sparse networks trained with the Lottery Ticket Hypothesis (Frankle and Carbin[18]) in Table 3—we obtain tickets with a commensurate performance for sparsities lower than 80%.

Finally, the choice of initialization scheme affects the performance and FLOP consumption by a greater extent than the method used itself, with the exception of SNFS (groups 1 and 2 in Table 3).

5.3 Hyperparameter Tuning

Table 4. Reference vs Optimal $(\alpha, \Delta T)$ on CIFAR-10. Optimal hyperparameters are obtained by tuning with a TPE sampler in Optuna. The difference between the reference and optimal performance is small, indicating that there is not a significant benefit in tuning $(\alpha, \Delta T)$ individually for each initialization and sparsity configuration.

Initialization	Density	Reference		Optimal	
	$(1-s)$	$(\alpha, \Delta T)$	Accuracy ↑ (Test)	$(\alpha, \Delta T)$	Accuracy ↑ (Test)
Random	0.1	0.3, 100	91.7 ± 0.18	0.197, 50	$\mathbf{91.8 \pm 0.17}$
Random	0.2	0.3, 100	92.6 ± 0.10	0.448, 150	$\mathbf{92.8 \pm 0.16}$
Random	0.5	0.3, 100	$\mathbf{93.3 \pm 0.07}$	0.459, 550	$\mathbf{93.3 \pm 0.18}$
ERK	0.1	0.3, 100	$\mathbf{92.4 \pm 0.06}$	0.416, 200	$\mathbf{92.4 \pm 0.23}$
ERK	0.2	0.3, 100	$\mathbf{93.1 \pm 0.09}$	0.381, 950	$\mathbf{93.1 \pm 0.21}$
ERK	0.5	0.3, 100	93.4 ± 0.14	0.287, 500	$\mathbf{93.8 \pm 0.06}$

$(\alpha, \Delta T)$ vs Sparsities – To understand the impact of the two additional hyperparameters included in *RigL*, we use a Tree of Parzen Estimator (TPE sampler, Bergstra et al.[19]) via Optuna to tune $(\alpha, \Delta T)$. We do this for sparsities $(1-s) \in \{0.1, 0.2, 0.5\}$, and a fixed learning rate of 0.1. Additionally, we set the sampling domain for α and ΔT as $[0.1, 0.6]$ and $\{50, 100, 150, ..., 1000\}$ respectively. We use 15 trials for each sparsity value, with our objective function as the validation accuracy averaged across 3 random seeds.

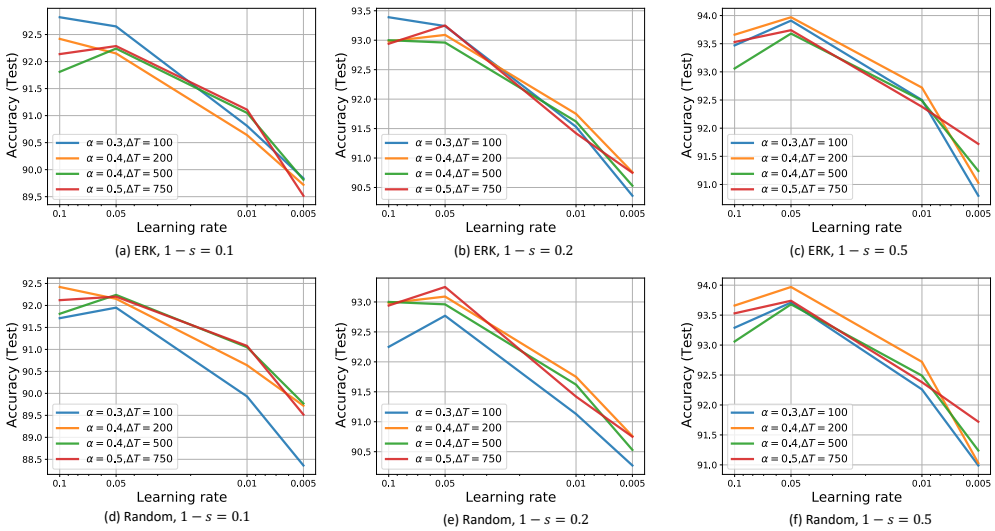


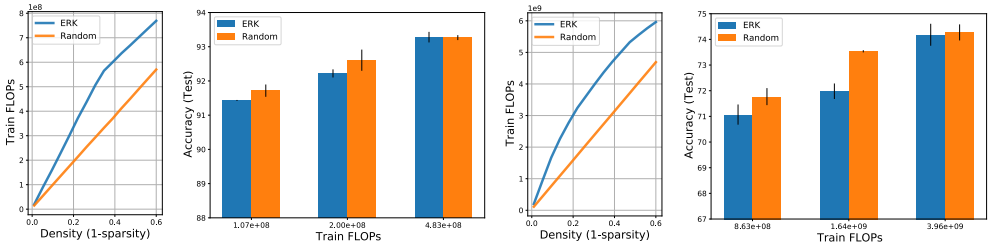
Figure 3. Learning Rate vs Sparsity on CIFAR-10. Runs using a learning rate > 0.1 do not converge and are not plotted here. There is little benefit in tuning the learning rate for each sparsity, and $0.1, 0.05$ are good choices overall.

Table 4 shows the test accuracies of tuned hyperparameters. While the reference hyperparameters (original authors, $\alpha = 0.3, \Delta T = 100$) differ from the obtained optimal hyperparameters, the difference in performance is marginal, especially for ERK initialization. This in agreement with the original paper, which finds $\alpha \in \{0.3, 0.5\}, \Delta T = 100$ to be suitable choices. We include contour plots detailing the hyperparameter trial space in the supplementary material.

Learning Rate vs Sparsities — We further examine if the final performance improves by tuning the learning rate (η) individually for each sparsity-initialization pair. We employ a grid search over $\eta \in \{0.1, 0.05, 0.01, 0.005\}$ and $(\alpha, \Delta T) \in \{(0.3, 100), (0.4, 200), (0.4, 500), (0.5, 750)\}$. As seen in Figure 3, $\eta = 0.1$ and $\eta = 0.05$ are close to optimal values for a wide range of sparsities and initializations. Since these learning rates also correspond to good choices for the Dense baseline, one can employ similar values when training with *RigL*.

6 Results beyond Original Paper

6.1 Sparsity Distribution vs FLOP Consumption



(a) WRN-22-2 on CIFAR 10 (b) ResNet-50 on CIFAR100

Figure 4. Test Accuracy vs FLOP consumption of WideResNet-22-2 on CIFAR-10 and ResNet-50 on CIFAR-100, compared for Random and ERK initializations. For the same FLOP budget, models trained with ERK initialization must be more sparse, resulting in inferior performance.

While ERK initialization outperforms Random initialization consistently for a given target parameter count, it requires a higher FLOP budget. Figure 4 compares the two initialization schemes across fixed training FLOPs. Theoretical FLOP requirement for Random initialization scales linearly with density $(1 - s)$, and is significantly lesser than ERK's FLOP requirements. Consequently, Random initialization outperforms ERK initialization for a given training budget.

6.2 Effect of Redistribution

One of the main differences of *RigL* over SNFS is the lack of layer-wise redistribution during training. We examine if using a redistribution criterion can be beneficial and bridge the performance gap between Random and ERK initialization. Following Dettmers and Zettlemoyer[6], during every mask update, we reallocate layer-wise density proportional to its average sparse gradient or momentum (*RigL*-SG, *RigL*-SM).

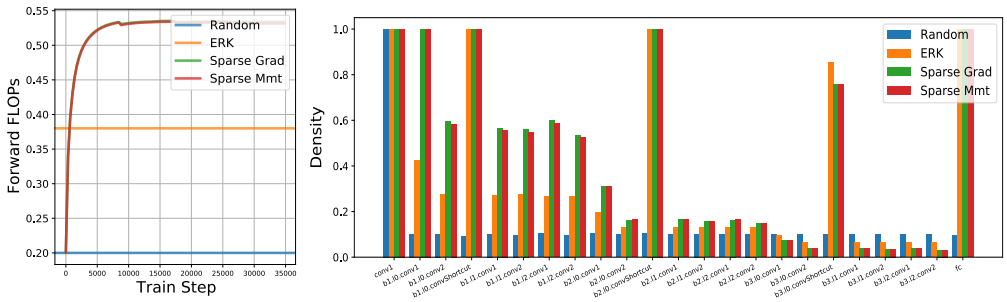
Table 5 shows that redistribution significantly improves *RigL* (Random), but not *RigL* (ERK). We additionally plot the FLOP requirement against training steps and the final sparsity distribution in Figure 5. The layer-wise sparsity distribution largely becomes constant within a few epochs. The final distribution is similar, but more "extreme" than ERK—wherever ERK exceeds/falls short of Random, redistribution does so by a greater

Table 5. Effect of redistribution during *RigL* updates, evaluated on CIFAR10 and CIFAR100. By utilising sparse gradient or sparse momentum based redistribution, *RigL* (Random) matches *RigL* (ERK)'s performance. Among Random and ERK initialized experiments, we mark the best metrics under each sparsity and dataset in bold.

Method	Redistribution	CIFAR-10				CIFAR-100			
		$1-s=0.1$		$1-s=0.2$		$1-s=0.1$		$1-s=0.2$	
		Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)	Accuracy ↑ (Test)	FLOPs ↓ (Train, Test)
Random Initialization									
RigL	-	91.7 ± 0.18	0.10x, 0.10x	92.9 ± 0.10	0.20x, 0.20x	71.8 ± 0.33	0.10x, 0.10x	73.5 ± 0.04	0.20x, 0.20x
RigL-SG	Sparse Grad	**92.2 ± 0.17**	0.28x, 0.28x	92.7 ± 0.25	0.49x, 0.49x	72.3 ± 0.12	0.36x,0.35x	**73.7 ± 0.15**	0.53x, 0.53x
RigL-SM	Sparse Mmt	**92.2 ± 0.20**	0.28x, 0.28x	**92.9 ± 0.21**	0.50x, 0.49x	**72.6 ± 0.27**	0.36x,0.36x	**73.7 ± 0.35**	0.53x, 0.53x
ERK Initialization									
RigL	-	**92.4 ± 0.06**	0.17x, 0.17x	**93.1 ± 0.09**	0.35x, 0.35x	72.6 ± 0.37	0.23x, 0.22x	73.4 ± 0.15	0.38x, 0.38x
RigL-SG	Sparse Grad	92.1 ± 0.19	0.28x, 0.28x	92.7 ± 0.19	0.49x, 0.49x	**73.0 ± 0.13**	0.37x,0.36x	74.2 ± 0.26	0.53x, 0.53x
RigL-SM	Sparse Mmt	**92.27 ± 0.01**	0.28x, 0.28x	**93.0 ± 0.13**	0.50x, 0.49x	72.6 ± 0.27	0.37x, 0.37x	**74.2 ± 0.13**	0.53x, 0.53x
Re-Initialization with *RigL*-SM (Random, ERK)									
RigL	-	90.3 ± 0.34	0.28x, 0.28x	91.0 ± 0.38	0.50x, 0.49x	67.6 ± 0.28	0.36x, 0.36x	68.9 ± 0.65	0.53x, 0.53x
RigL (ERK)	-	90.2 ± 0.57	0.28x, 0.28x	90.6 ± 0.56	0.50x, 0.49x	67.8 ± 0.73	0.37x, 0.37x	68.9 ± 0.47	0.53x, 0.53x

extent.

By allocating higher densities to 1×1 convolutions (*convShortcut* in Figure 5), redistribution significantly increases the FLOP requirement—and hence, is not a preferred alternative to ERK. Surprisingly, initializing *RigL* with the final sparsity distribution in a manner similar to the Lottery Ticket Hypothesis results in subpar performance (group 3, Table 5).



(a) FLOP consumption vs train steps (b) Layer-wise density distribution

Figure 5. Effect of redistribution on *RigL*'s performance, evaluated using WideResNet-22-2 on CIFAR10 at 80% sparsity. **(left)** FLOPs required per forward pass, shown relative to the dense baseline, rises quickly and saturates within a few epochs (~10k steps) for both sparse gradient and sparse momentum based redistribution. **(right)** Comparison of the final density distribution against Random and ERK counterparts. "b" refers to block and "l" layer here.

7 Discussion

Evaluated on image classification, the central claims of Evci et al.[4] hold true—*RigL* outperforms existing sparse-to-sparse training methods and can also surpass other dense-to-sparse training methods with extended training. *RigL* is fairly robust to its choice of hyperparameters, as they can be set independent of sparsity or initialization. We find that the choice of initialization has a greater impact on the final performance and compute requirement than the method itself. Considering the performance boost obtained by redistribution, proposing distributions that attain maximum performance given a FLOP budget could be an interesting future direction.

For computational reasons, our scope is restricted to small datasets such as CIFAR-10/100. *RigL*'s applicability outside image classification—in Computer Vision and beyond (machine translation etc.) is not covered here.

What was easy — The authors' code covered most of the experiments in their paper and helped us validate the correctness of our replicated codebase. Additionally, the original paper is quite complete, straightforward to follow, and lacked any major errors.

What was difficult — Implementation details such as whether momentum buffers were accumulated sparsely or densely had a substantial impact on the performance of SNFS. Finding the right ϵ for ERK initialization required handling of edge cases—when a layer's capacity is exceeded. Hyperparameter tuning (α, ΔT) involved multiple seeds and was compute-intensive.

Communication with original authors — We acknowledge and thank the original authors for their responsive communication, which helped clarify a great deal of implementation and evaluation specifics. Particularly, FLOP counting for various methods while taking into account the changing sparsity distribution. We also discussed experiments extending the original paper—as to whether the authors had carried out a similar study before.

References

1. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. "EIE: efficient inference engine on compressed deep neural network." In: **ACM SIGARCH Computer Architecture News** 44.3 (2016), pp. 243–254.
2. M. Ashby, C. Baaij, P. Baldwin, M. Bastiaan, O. Bunting, A. Cairncross, C. Chalmers, L. Corrigan, S. Davis, N. van Doorn, et al. "Exploiting Unstructured Sparsity on Next-Generation Datacenter Hardware." In: (2017).
3. S. Srinivas, A. Subramanya, and R. Venkatesh Babu. "Training Sparse Neural Networks." In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops**. July 2017.
4. U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen. "Rigging the Lottery: Making All Tickets Winners." In: **Proceedings of Machine Learning and Systems (ICML)**. July 2020.
5. D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta. "Scalable Training of Artificial Neural Networks with Adaptive Sparse Connectivity inspired by Network Science." In: **Nature Communications** (2018). DOI: 10.1038/s41467-018-04316-3.
6. T. Dettmers and L. Zettlemoyer. **Sparse Networks from Scratch: Faster Training without Losing Performance**. 2020. URL: https://openreview.net/forum?id=ByeSYa4KPS.
7. M. Zhu and S. Gupta. "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression." In: **Proceedings of the International Conference on Learning Representations (ICLR)**. Apr. 2018.
8. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "Tensorflow: A system for large-scale machine learning." In: **12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)**. 2016, pp. 265–283.
9. A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: **Advances in Neural Information Processing Systems**. Dec. 2019.
10. L. Biewald. **Experiment Tracking with Weights and Biases**. Software available from wandb.com. 2020. URL: https://www.wandb.com/.
11. G. H. Alex Krizhevsky. **Learning multiple layers of features from tiny images**. Tech. rep. 2009.
12. S. Zagoruyko and N. Komodakis. "Wide Residual Networks." In: **Proceedings of the British Machine Vision Conference (BMVC)**. Sept. 2016.
13. K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. June 2016.
14. P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. "Accurate, large minibatch sgd: Training imagenet in 1 hour." In: **arXiv preprint arXiv:1706.02677** (2017).
15. S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: **International Conference on Machine Learning (ICML)**. July 2015.

16. T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li. "Bag of Tricks for Image Classification with Convolutional Neural Networks." In: **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)**. June 2019.
17. T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework." In: **Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. Aug. 2019.
18. J. Frankle and M. Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." In: **Proceedings of the International Conference on Learning Representations (ICLR)**. Apr. 2018.
19. J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. "Algorithms for Hyper-Parameter Optimization." In: **Advances in Neural Information Processing Systems**. Dec. 2011.
20. O. Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge." In: **International Journal of Computer Vision (IJCV)** 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.
21. S. Gray, A. Radford, and D. P. Kingma. "Gpu kernels for block-sparse weights." In: **arXiv preprint arXiv:1711.09224** 3 (2017).
22. D. Teja Vooturi, G. Varma, and K. Kothapalli. "Dynamic Block Sparse Reparameterization of Convolutional Neural Networks." In: **Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops**. Oct. 2019.

A Architecture Specific Details—ResNet-50 on CIFAR100

Table 6. ResNet-50 architecture used on CIFAR100. Building blocks are shown in brackets, with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

Layer Name	Output Size	ResNet-50
conv1	32×32	3×3, 64, no stride
conv2_x	32×32	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times3$
conv3_x	16×16	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times4$
conv4_x	8×8	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times6$
conv5_x	4×4	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times3$
	1×1	average pool, 100-d fc, softmax
FLOPs		2.59e9

We use a variant of the originally proposed ResNet architecture (He et al.[13]). Particularly, we replace the initial 7×7 conv layer with a 3×3 conv layer. Here, "conv layer" refers to convolution followed by batchnorm (Ioffe and Szegedy[15]) and ReLU activation. This is intended to not excessively downsample the image—CIFAR-100 (Alex Krizhevsky[11]) has images of dimensions 32×32, compared to Imagenet's (Russakovsky et al.[20]) 224×224. Each block used (conv2_x, conv3_x, etc.) is a bottleneck block, and uses the conv-batchnorm-ReLU ordering.

B FLOP Counting Procedure

Following Evci et al.[4], we base our counting procedure on the Micronet Challenge[5], which was conducted as a part of NeurIPS 2019. Support for unstructured sparsity is assumed while computing the number of additions and multiplication operations. The sum of these two gives us the theoretical FLOPs for a single forward pass through the model.

Concretely, let the FLOPs required for a forward pass through a dense model be f_d and the corresponding for a sparse model (or small-dense model) be f_s. Then, the FLOPs for training a dense model are $3f_d$—since the backward pass involves computing gradients with respect to each weight and activation. f_s can be computed for a model given its sparsity distribution via the counting procedure. The FLOPs required to train a sparse model depend on the technique used, as detailed below.

[5] https://micronet-challenge.github.io/

B.1 Inference FLOPs

Small-Dense, RigL, SET, Static — These methods involve constant layer-wise sparsity throughout training, hence the FLOP count can be determined during any step. The FLOP count for Random initialized models are $(1 - s)$ times the Dense FLOPs.

SNFS, Pruning — Both methods involve varying layer-wise sparsity during training, and hence non-constant FLOP consumption. The final weights are used to determine inference FLOPs in this case.

B.2 Train FLOPs

Small-Dense, Static — Dense gradients are not required by these models, and hence have a train FLOP count of $3f_s$.

SET — Dense gradients are not required, and random growth can be implemented quite efficiently. Thus, the train FLOP count is $3f_s$.

RigL — Dense gradients are required only every ΔT steps, hence the corresponding train FLOP count is: $\frac{3\Delta T f_s + 2f_s + f_d}{\Delta T + 1}$. We note that since ΔT is typically set between 100–1000, the preceding expression is quite close to $3f_s$.

SNFS — Dense gradients are required at each training step, resulting in $2f_s + f_d$ FLOPs consumed at each step. Since the sparse FLOP count varies as we train, the average FLOP count is: $2\mathbb{E}[f_{s,t}] + f_d$, where $f_{s,t}$ is the sparse inference FLOPs at train step t.

Pruning — Does not require dense gradients, but the sparsity increases smoothly from 0% to the target value as we train. The FLOP consumption here is $3\mathbb{E}[f_{s,t}]$, , where $f_{s,t}$ is the sparse inference FLOPs at train step t.

To determine $\mathbb{E}[f_{s,t}]$, we compute a running average of the FLOP consumption after every epoch. Notably, we find that the inference cost of Pruning is often close to a Random initialized sparse network, while SNFS, regardless of initiazation, is compute-intensive.

C Trial Space of Hyperparameter Tuning

Figure 6 shows the hyper-parameter study for tuning $(\alpha, \Delta T)$ as a contour plot. We observe that for multiple initialization-density configurations, the reference choice $(\alpha = 0.3, \Delta T = 100)$, is quite close to the optimal hyper-parameters. Furthermore, where they differ, the difference is within standard deviation bounds (Table 4 of the main report).

D Dynamic Structured Sparsity

Present hardware accelerators lack efficient implementations for unstructured sparsity. As a result, in practice, the reduced FLOP requirement of sparse methods rarely translate to wall-clock improvements. In comparison, there are efficient implementations available for structured (or block) sparsity which reach theoretical speedups (Gray, Radford, and Kingma, Teja Vooturi, Varma, and Kothapalli[21,22]).

Motivated by this, we try modifying *RigL* to explicitly work on structured sparsity. We promote channel sparsity for convolutional layers and keep fully connected layers dense.

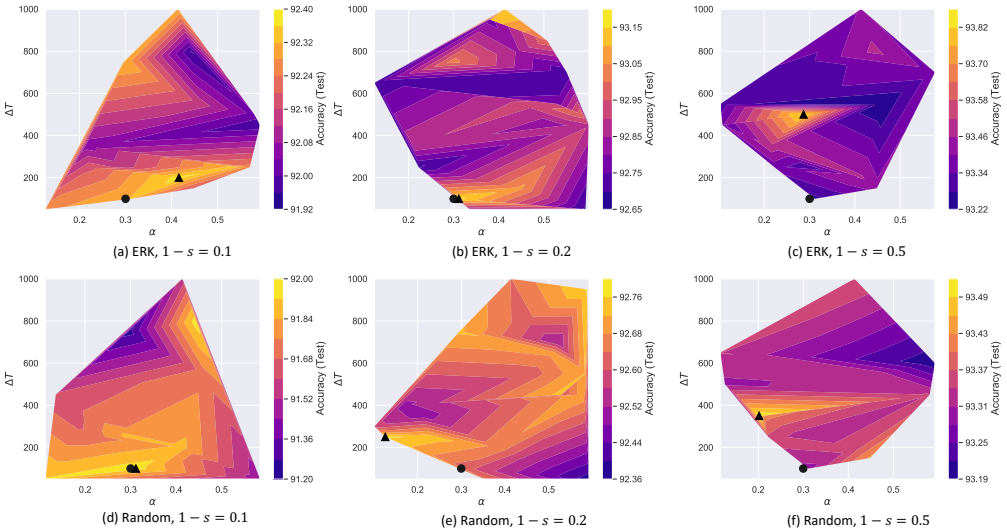


Figure 6. **Trial space of tuning** $(\alpha, \Delta T)$, shown as a countor plot. Here, black circle corresponds to $(\alpha = 0.3, \Delta T = 100)$, while black triangle corresponds to the optimal hyper-parameter pair found. We plot the convex hull of the trial space, so in a few cases the reference point lies on the border of this space.

Table 7. **Modifying *RigL* for structured sparsity, compared on CIFAR-10 and CIFAR-100 datasets.** *RigL*-struct fails to match the accuracy of *RigL* and just matches Small-Dense in performance.

Method	CIFAR-10				CIFAR-100			
	$1-s=0.1$		$1-s=0.2$		$1-s=0.1$		$1-s=0.2$	
	Accuracy ↑ (Test)	Wall Time ↓	Accuracy ↑ (Test)	Wall Time ↓	Accuracy ↑ (Test)	Wall Time ↓	Accuracy ↑ (Test)	Wall Time ↓
Small-Dense	89.0 ± 0.35	0.11x	91.0 ± 0.07	0.20x	70.8 ± 0.22	0.11x	72.6± 0.93	0.20x
Random Initialization								
RigL	91.7 ± 0.18	1.0x	92.9 ± 0.10	1.0x	71.8 ± 0.33	1.0x	73.5 ± 0.04	1.0x
RigL-Struct	87.0 ± 0.09	0.10x	90.4 ± 0.27	0.20x	69.1 ± 0.11	0.10x	71.9 ± 0.13	0.20x
ERK Initialization								
RigL	92.4 ± 0.06	1.0x	93.1 ± 0.09	1.0x	72.6 ± 0.37	1.0x	73.4 ± 0.15	1.0x
RigL-Struct	89.6 ± 0.16	0.17x	91.3 ± 0.18	0.35x	71.1 ± 0.15	0.23x	72.9 ± 0.08	0.38x

Mask update steps also operate at the channel level, based on *RigL*'s growth and pruning criterion. We name this method as *RigL*-struct. Such an approach is enticing, as we can remove masked-out channels, and obtain practical speedups on accelerators without needing support for unstructured sparsity.

Unfortunately, *RigL*-struct does not preserve the performance of originally proposed *RigL* (Table 7). In fact, it performs only as good as Small-Dense models, which negates the motivation behind such an experiment—Small-Dense models already achieve the intended speedups.