A. Effect of Mask Updates on the Energy Landscape

To update the connectivity of our sparse network, we first need to drop a fraction d of the existing connections for each layer independently to create a budget for growing new connections. Like many prior works (Thimm & Fiesler, 1995; Ström, 1997; Narang et al., 2017; Han et al., 2015), we drop parameters with the smallest magnitude. The effectiveness of this simple criteria can be explained through the first order Taylor approximation of the loss L around the current set of parameters θ.

$$\Delta L = L(\theta + \Delta\theta) - L(\theta) = \nabla_\theta L(\theta)\Delta\theta + R(||\Delta\theta||_2^2)$$

The main goal of dropping connections is to remove parameters with minimal impact on the output of the neural network and therefore on its loss. Since removing the connection θ_i corresponds to setting it to zero, it incurs a change of $\Delta\theta = -\theta_i$ in that direction and a change of $\Delta L_i = -\nabla_{\theta_i} L(\theta)\theta_i + R(\theta_i^2)$ in the loss, where the first term is usually defined as the *saliency* of a connection. Saliency has been used as a criterion to remove connections (Molchanov et al., 2016), however it has been shown to produce inferior results compared to magnitude based removal, especially when used to remove multiple connections at once (Evci, 2018). In contrast, picking the lowest magnitude connections ensures a small remainder term in addition to a low saliency, limiting the damage we make when we drop connections. Additionally, we note that connections with small magnitude can only remain small if the gradient they receive during training is small, meaning that the saliency is likely small when the parameter itself is small.

After the removal of insignificant connections, we enable new connections that have the highest expected gradients. Since we initialize these new connections to zero, they are guaranteed to have high gradients in the proceeding iteration and therefore to reduce the loss quickly. Combining this observation with the fact that *RigL* is likely to remove low gradient directions,) and the results in Section 4.4, suggests that *RigL* improves the energy landscape of the optimization by replacing flat dimensions with ones with higher gradient. This helps the optimization procedure escape saddle points.

B. Comparison with Bayesian Structured Pruning Algorithms

Structured pruning algorithms aim to remove entire neurons (or channels) instead of individual connections either at the end of, or throughout training. The final pruned network is a smaller dense network. Liu et al. (2019) demonstrated that these smaller networks could themselves be successfully be trained from scratch. This recasts structured pruning approaches as a limited kind of architecture search, where the search space is the size of each hidden layer.

In this section we compare *RigL* with three different structured pruning algorithms: SBP (Neklyudov et al., 2017), L0 (Christos Louizos, 2018), and VIB (Dai et al., 2018). We show that starting from a random sparse network, *RigL* finds compact networks with fewer parameters, that require fewer FLOPs for inference *and* require fewer resources for training. This serves as a general demonstration of the effectiveness of unstructured sparsity.

For our setting we pick the standard LeNet 300-100 network with ReLU non-linearities trained on MNIST. In Table 2 we compare methods based on how many FLOPs they require for training and also how efficient the final architecture is. Unfortunately, none of the papers have released the code for reproducing the MLP results, so we therefore use the reported accuracies and calculate lower bounds for the the FLOPs used during training. For each method we assume that one training step takes as much as the dense 300-100 architecture and omit any additional operations each method introduces. We also consider training the pruned networks from scratch and report the training FLOPs required in parenthesis. In this setting, training FLOPs are significantly lower since the starting networks are have been significantly reduced in size. We assume that following (Liu et al., 2019) the final networks can be trained from scratch, but we cannot verify this for these MLP networks since it would require knowledge of which pixels were dropped from the input.

To compare, we train a sparse network starting from the original MLP architecture (**RigL**). At initialization, we randomly remove 99% and 89% of the connections in the first and second layer of the MLP. At the end of the training many of the neurons in the first 2 layers have no in-coming or out-going connections. We remove such neurons and use the resulting architecture to calculate the inference FLOPs and the size. We assume the sparse connectivity is stored as a bit-mask (We assume parameters are represented as floats, i.e. 4 bytes). In this setting, *RigL* finds smaller, more FLOP efficient networks with far less work than the Bayesian approaches.

Next, we train a sparse network starting from the architecture found by the first run (**RigL+**) (408-100-69) but with a new random initialization (both masks and the parameters). We reduce the sparsity of the first 2 layers to 96% and 86% respectively as the network is already much smaller. Repeating *RigL* training results in an even more compact architecture half the size and requiring only a third the FLOPs of the best architecture found by Dai et al. (2018).

Method	Final Architecture	Sparsity	Training Cost (GFLOPs)	Inference Cost (KFLOPs)	Size (bytes)	Error
SBP	245-160-55	0.000	13521.6 (2554.8)	97.1	195100	1.6
L0	266-88-33	0.000	13521.6 (1356.4)	53.3	107092	1.6
VIB	97-71-33	0.000	13521.6 (523)	19.1	38696	1.6
RigL	408-100-69	0.870	482.0	12.6	31914	1.44 (1.48)
RigL+	375-62-51	0.886	**206.3**	**6.2**	**16113**	1.57 (1.69)

Table 2. Performance of various structured pruning algorithms on compressing three layer MLP on MNIST task. Cost of training the final architectures found by SBP, L0 and VIB are reported in parenthesis. *RigL* finds more compact networks compared to structured pruning approaches.

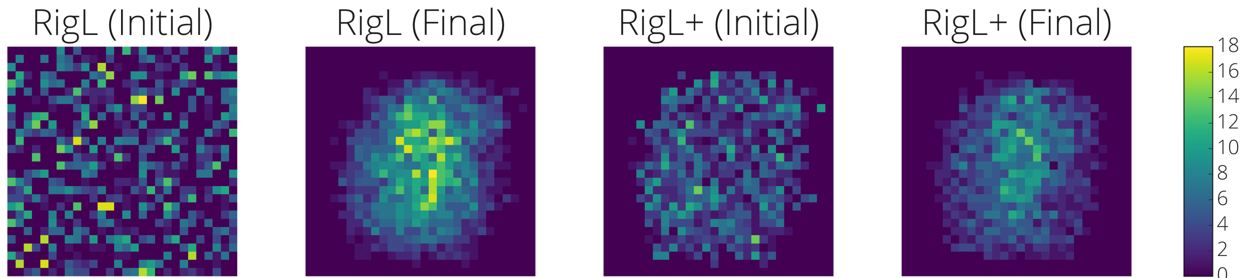

Figure 7. Number of connections that originate from the pixels of MNIST images at the beginning and end of the training. RigL+ starts from a smaller architecture (408-100-69) that has already removed some of the input pixels near the edges. Starting from an initially random distribution, *RigL* converges on the most relevant dimensions. See main text for further details.

Examination of the open-sourced code for the methods considered here made us aware that all of them keep track of the test error during training and report the best error ever observed during training as the final error. We generally would not encourage such overfitting to the test/validation set, however to make the comparisons with these results fair we report both the lowest error observed during training and the error at the end of training (reported in parenthesis). All hyper-parameter tuning was done using only the final test error.

In Figure 7 we visualize how RigL chooses to connect to the input and how this evolves from the beginning to the end of training. The heatmap shows the number of outgoing connections from each input pixels at the beginning (RigL Initial) and at the end (RigL (Final)) of the training. The left two images are for the initial network and the right two images are for **RigL+** training. *RigL* automatically discards uninformative pixels and allocates the connections towards the center highlighting the potential of *RigL* on model compression and feature selection.

C. Effect of Sparsity Distribution on Other Methods

In Figure 8-left we show the effect of sparsity distribution choice on 4 different sparse training methods. ERK distribution performs better than other distributions for each training method.

D. Effect of Momentum Coefficient for SNFS

In Figure 8 right we show the effect of the momentum coefficient on the performance of SNFS. Our results shows that using a coefficient of 0.99 brings the best performance. On the other hand using the most recent gradient only (coefficient of 0) performs as good as using a coefficient of 0.9. This result might be due to the large batch size we are using (4096), but it still motivates using *RigL* and instantaneous gradient information only when needed, instead of accumulating them.

E. (Non)-Existence of Lottery Tickets

We perform the following experiment to see whether *Lottery Tickets* exist in our setting. We take the sparse network found by *RigL* and restart training using original initialization, both with RigL and with fixed topology as in the original Lottery

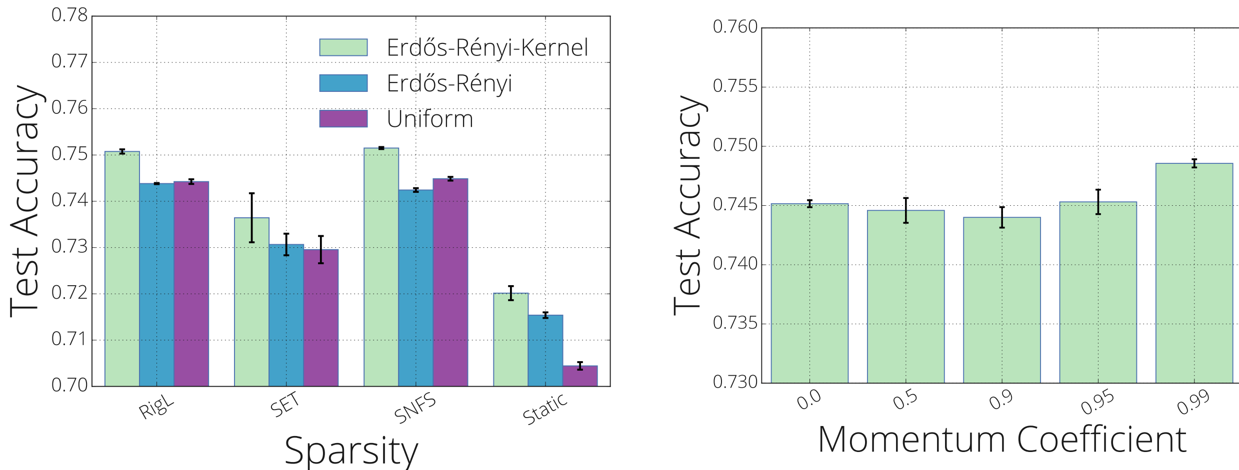

Figure 8. **(left)** Effect of sparsity distribution choice on sparse training methods at different sparsity levels. We average over 3 runs and report the standard deviations for each. **(right)** Effect of momentum value on the performance of SNFS algorithm. Momentum does not become helpful until it reaches extremely large values.

Ticket Hypothesis. Results in table 3 demonstrate that training with a fixed topology is significantly worse than training with *RigL* and that *RigL* does not benefit from starting again with the final topology and the original initialization - training for twice as long instead of rewiring is more effective. In short, there are no special tickets, with *RigL* all tickets seems to win.

F. Effect of Update Schedules on Other Dynamic Sparse Methods

In Figure 9 we repeat the hyper-parameter sweep done for *RigL* in Figure 5-right, using SET and SNFS. Cosine schedule with $\Delta T = 50$ and $\alpha = 0.1$ seems to work best across all methods. An interesting observation is that higher drop fractions (α) seem to work better with longer intervals ΔT. For example, SET with $\Delta T = 1000$ seems to work best with $\alpha = 0.5$.

G. Alternative Update Schedules

In Figure 10, we share the performance of two alternative annealing functions:

1. *Constant:* $f_{decay}(t) = \alpha$.
2. *Inverse Power:* The fraction of weights updated decreases similarly to the schedule used in (Zhu & Gupta, 2018) for iterative pruning: $f_{decay}(t) = \alpha(1 - \frac{t}{T_{end}})^k$. In our experiments we tried $k = 1$ which is the linear decay and their default $k = 3$.

Constant seems to perform well with low initial drop fractions like $\alpha = 0.1$, but it starts to perform worse with increasing α. *Inverse Power* for k=3 and k=1 (*Linear*) seems to perform similarly for low α values. However the performance drops noticeably for k=3 when we increase the update interval. As reported by (Dettmers & Zettlemoyer, 2019) linear (k=1) seems to provide similar results as the cosine schedule.

H. Calculating FLOPs of models and methods

In order to calculate FLOPs needed for a single forward pass of a sparse model, we count the total number of multiplications and additions layer by layer for a given layer sparsity s^l. The total FLOPs is then obtained by summing up all of these multiply and adds. Different sparsity distributions require different number of FLOPs to compute a single prediction. For example *Erdős-Renyi-Kernel* distributions usually cause 1x1 convolutions to be less sparse than the 3x3 bottleneck layers (see Appendix K). The number of input/output channels of 1x1 convolutional layers are greater and therefore require more FLOPs to compute the output features compared to 3x3 layers of the ResNet blocks. Thus, allocating smaller sparsities to 1x1 convolutional layers results in a higher overall FLOPs than a sparse network with uniform sparsity.

Training a neural network consists of 2 main steps:

1. *forward pass:* Calculating the loss of the current set of parameters on a given batch of data. During this process layer activations are calculated in sequence using the previous activations and the parameters of the layer. Activation of layers are stored in memory for the backward pass.

Initialization	Training Method	Test Accuracy	Training FLOPs
Lottery	Static	70.82±0.07	0.46x
Lottery	RigL	73.93±0.09	0.46x
Random	RigL	74.55±0.06	0.23x
Random	$\text{RigL}_{2\times}$	76.06±0.09	0.46x

Table 3. Effect of lottery ticket initialization on the final performance. There are no special tickets and the dynamic connectivity provided by *RigL* is critical for good performance.

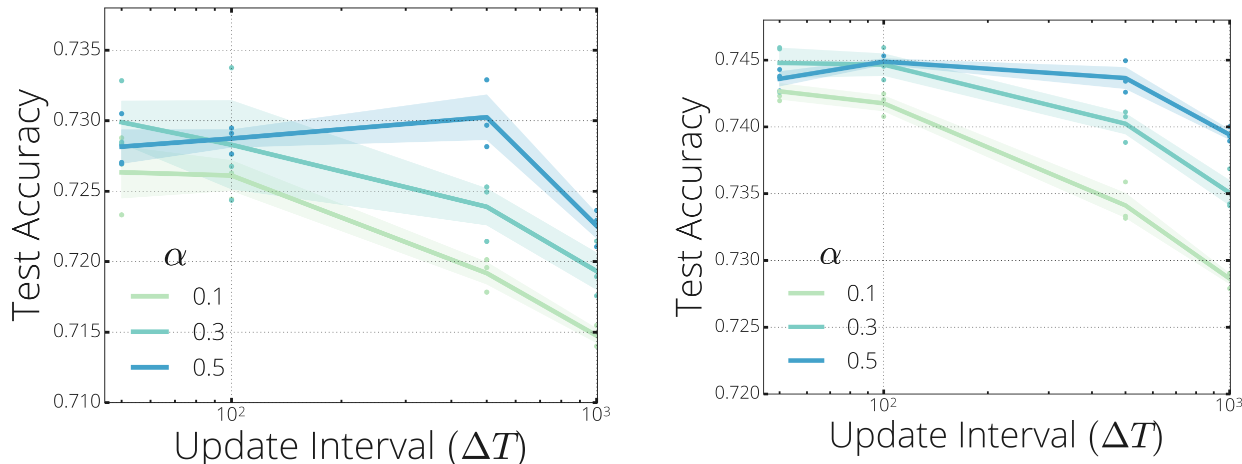

Figure 9. Cosine update schedule hyper-parameter sweep done using dynamic sparse training methods SET **(left)** and SNFS **(right)**.

2. *backward pass:* Using the loss value as the initial error signal, we back-propagate the error signal while calculating the gradient of parameters. During the backward pass each layer calculates 2 quantities: the gradient of the activations of the previous layer and the gradient of its parameters. Therefore in our calculations we count backward passes as two times the computational expense of the forward pass. We omit the FLOPs needed for batch normalization and cross entropy.

Dynamic sparse training methods require some extra FLOPs to update the connectivity of the neural network. We omit FLOPs needed for dropping the lowest magnitude connections in our calculations. For a given dense architecture with FLOPs f_D and a sparse version with FLOPs f_S, the total FLOPs required to calculate the gradient on a single sample is computed as follows:

- **Static Sparse and Dense.** Scales with $3{*}f_S$ and $3{*}f_D$ FLOPs, respectively.
- **Pruning.** $\mathbb{E}_t[3{*}f_D{*}s_t]$ FLOPs where s_t is the sparsity of the model at iteration t.
- **Snip.** We omit the initial dense gradient calculation since it is negligible, which means Snip scales in the same way as Static methods: $3 * f_S$ FLOPs.
- **SET.** We omit the extra FLOPs needed for growing random connections, since this operation can be done on chip efficiently. Therefore, the total FLOPs for SET scales with $3 * f_S$.
- **SNFS.** Forward pass and back-propagating the error signal needs $2{*}f_S$ FLOPs. However, the dense gradient needs to be calculated at every iteration. Thus, the total number of FLOPs scales with $2 * f_S + f_D$.
- **RigL.** Iterations with no connection updates need $3{*}f_S$ FLOPs. However, at every ΔT iteration we need to calculate the dense gradients. This results in the average FLOPs for *RigL* given by $\frac{(3*f_S*\Delta T+2*f_S+f_D)}{(\Delta T+1)}$.

I. Hyper-parameters used in Charachter Level Language Modeling Experiments

As stated in the main text, our network consists of a shared embedding with dimensionality 128, a vocabulary size of 256, a GRU with a state size of 512, a readout from the GRU state consisting of two linear layers with width 256 and 128 respectively. We train the next step prediction task with the cross entropy loss using the Adam optimizer. We set the learning rate to $7e-4$ and L2 regularization coefficient to $5e-4$. We use a sequence length of 512 and a batch size of 32. Gradients are clipped when their magnitudes

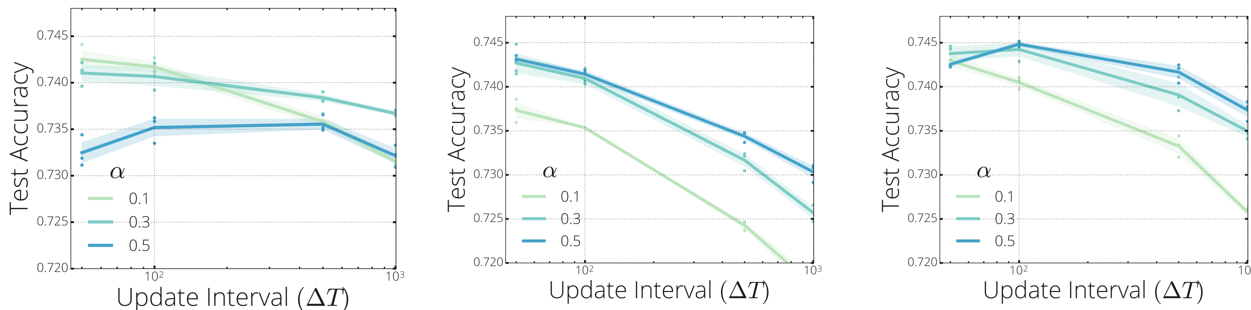

Figure 10. Using other update schedules with *RigL*: **(left)** Constant **(middle)** Exponential (k=3) and **(right)** Linear

exceed 10. We set the sparsity to 75% for all models and run 200,000 iterations. When inducing sparsity with magnitude pruning (Zhu & Gupta, 2018), we perform pruning between iterations 50,000 and 150,000 with a frequency of 1,000. We initialize sparse networks with a uniform sparsity distribution and use a cosine update schedule with $\alpha = 0.1$ and $\Delta T = 100$. Unlike the previous experiments we keep updating the mask until the end of the training since we observed this performed slightly better than stopping at iteration 150,000.

J. Additional Plots and Experiments for CIFAR-10

In Figure 11-left, we plot the final training loss of experiments presented in Section 4.3 to investigate the generalization properties of the algorithms considered. Poor performance of *Static* reflects itself in training loss clearly across all sparsity levels. *RigL* achieves similar final loss as the pruning, despite having around half percent less accuracy. Training longer with *RigL* decreases the final loss further and the test accuracies start matching pruning (see Figure 4-right) performance. These results show that *RigL* improves the optimization as promised, however generalizes slightly worse than pruning.

In Figure 11-right, we sweep mask update interval ΔT and plot the final test accuracies. We fix initial drop fraction α to 0.3 and evaluate two different sparsity distributions: *Uniform* and *ERK*. Both curves follow a similar pattern as in Imagenet-2012 sweeps (see Figure 9) and best results are obtained when $\Delta T = 100$.

K. Sparsity of Individual Layers for Sparse ResNet-50

Sparsity of ResNet-50 layers given by the Erdős-Rényi-Kernel sparsity distribution plotted in Figure 12.

L. Performance of Algortihms at Training 95 and 96.5% Sparse ResNet-50

In this section we share results of algorithms at training ResNet-50s with higher sparsities. Results in Table 4 indicate *RigL* achieves higher performance than the pruning algorithm even without extending training length.

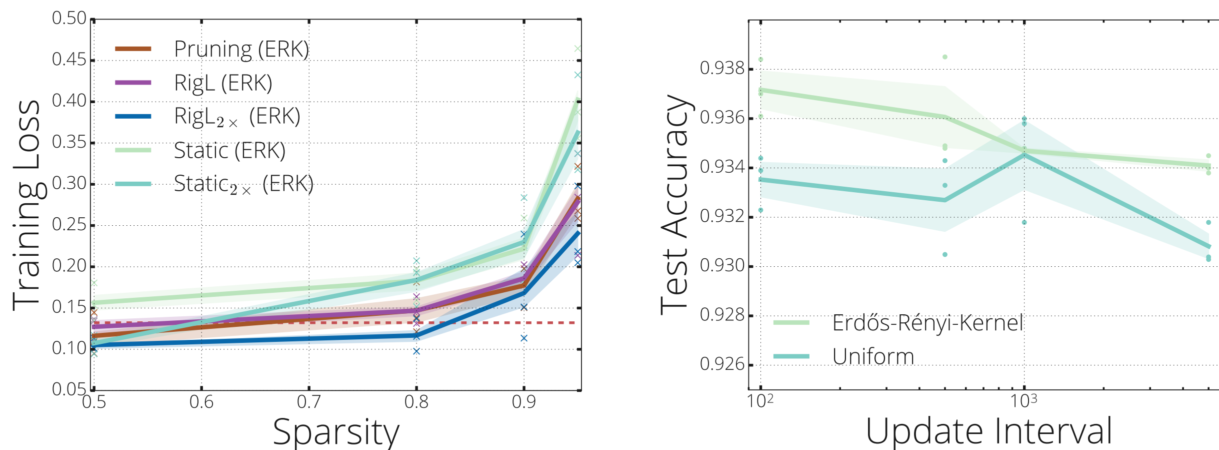

Figure 11. Final training loss of sparse models **(left)** and performance of *RigL* at different mask update intervals **(right)**.

Method	Top-1 Accuracy	FLOPs (Train)	FLOPs (Test)	Top-1 Accuracy	FLOPs (Train)	FLOPs (Test)
Dense	76.8±0.09	1x (3.2e18)	1x (8.2e9)			
	S=0.95			S=0.965		
Static	59.5+-0.11	0.23x	0.08x	55.4+-0.06	0.13x	0.07x
Snip	57.8+-0.40	0.23x	0.08x	52.0+-0.20	0.13x	0.07x
SET	64.4+-0.77	0.23x	0.08x	60.8+-0.45	0.13x	0.07x
RigL	67.5+-0.10	0.23x	0.08x	65.0+-0.28	0.13x	0.07x
$\text{RigL}_{5\times}$	73.1+-0.12	1.14x	0.08x	71.1+-0.20	0.66x	0.07x
Static (ERK)	72.1±0.04	0.42x	0.42x	67.7±0.12	0.24x	0.24x
RigL (ERK)	69.7+-0.17	0.42x	0.12x	67.2+-0.06	0.25x	0.11x
$\text{RigL}_{5\times}$ (ERK)	74.5+-0.09	2.09x	0.12x	72.7+-0.02	1.23x	0.11x
SNFS (ERK)	70.0+-0.04	0.61x	0.12x	67.1+-0.72	0.50x	0.11x
Pruning* (Gale)	70.6	0.56x	0.08x	n/a	0.51x	0.07x
$\text{Pruning}_{1.5\times}$ (Gale)	72.7	0.84x	0.08x	69.26	0.76x	0.07x

Table 4. Results with increased sparsity on ResNet-50/ImageNet-2012.

M. Bugs Discovered During Experiments

Our initial implementations contained some subtle bugs, which while not affecting the general conclusion that *RigL* is more effective than other techniques, did result in lower accuracy for all sparse training techniques. We detail these issues here with the hope that others may learn from our mistakes.

1. **Random operations on multiple replicas.** We use data parallelism to split a mini-batch among multiple replicas. Each replica independently calculates the gradients using a different sub-mini-batch of data. The gradients are aggregated using an ALL-REDUCE operation before the optimizer update. Our implementation of SET, SNFS and *RigL* depended on each replica independently choosing to drop and grow the same connections. However, due to the nature of random operations in Tensorflow, this did not happen. Instead, different replicas diverged after the first drop/grow step. This was most pronounced in SET where each replica chose at random and much less so for SNFS and *RigL* where randomness is only needed to break ties. If left unchecked this might be expected to be catastrophic, but due to the behavior of Estimators and/or TF-replicator, the values on the first replica are broadcast to the others periodically (every approximately 1000 steps in our case).

   We fixed this bug by using stateless random operations. As a result the performance of SET improved slightly (0.1-0.3 % higher on Figure 2-left).

2. **Synchronization between replicas.** *RigL* and SNFS depend on calculating dense gradients with respect to the masked parameters. However, as explained above, in the multiple replica setting these gradients need to be aggregated. Normally this aggregation is automatically done by the optimizer, but in our case, this does not happen (only the gradients with respect to the *unmasked* parameters are aggregated automatically). This bug affected SNFS and *RigL*, but not SET since SET does not rely on the gradients to grow connections. Again, the synchronization of the parameters from the first replica every approximately 1000 steps masked this bug.

   We fixed this bug by explicitly calling ALL-REDUCE on the gradients with respect to the masked parameters. With this fix, the performance of *RigL* and SNFS improved significantly, particularly for default training lengths (around 0.5-1% improvement).

3. **SNIP Experiments.** Our first implementation of SNIP used the gradient magnitudes to decide which connections to keep causing its performance to be worse than static. Upon our discussions with the authors of SNIP, we realized that the correct metric is the saliency (gradient times parameter magnitude). With this correction SNIP performance improved dramatically to better than random (Static) even at Resnet-50/ImageNet scale. It is surprising that picking connections with the highest gradient magnitudes can be so detrimental to training (it resulted in much worse than random performance).

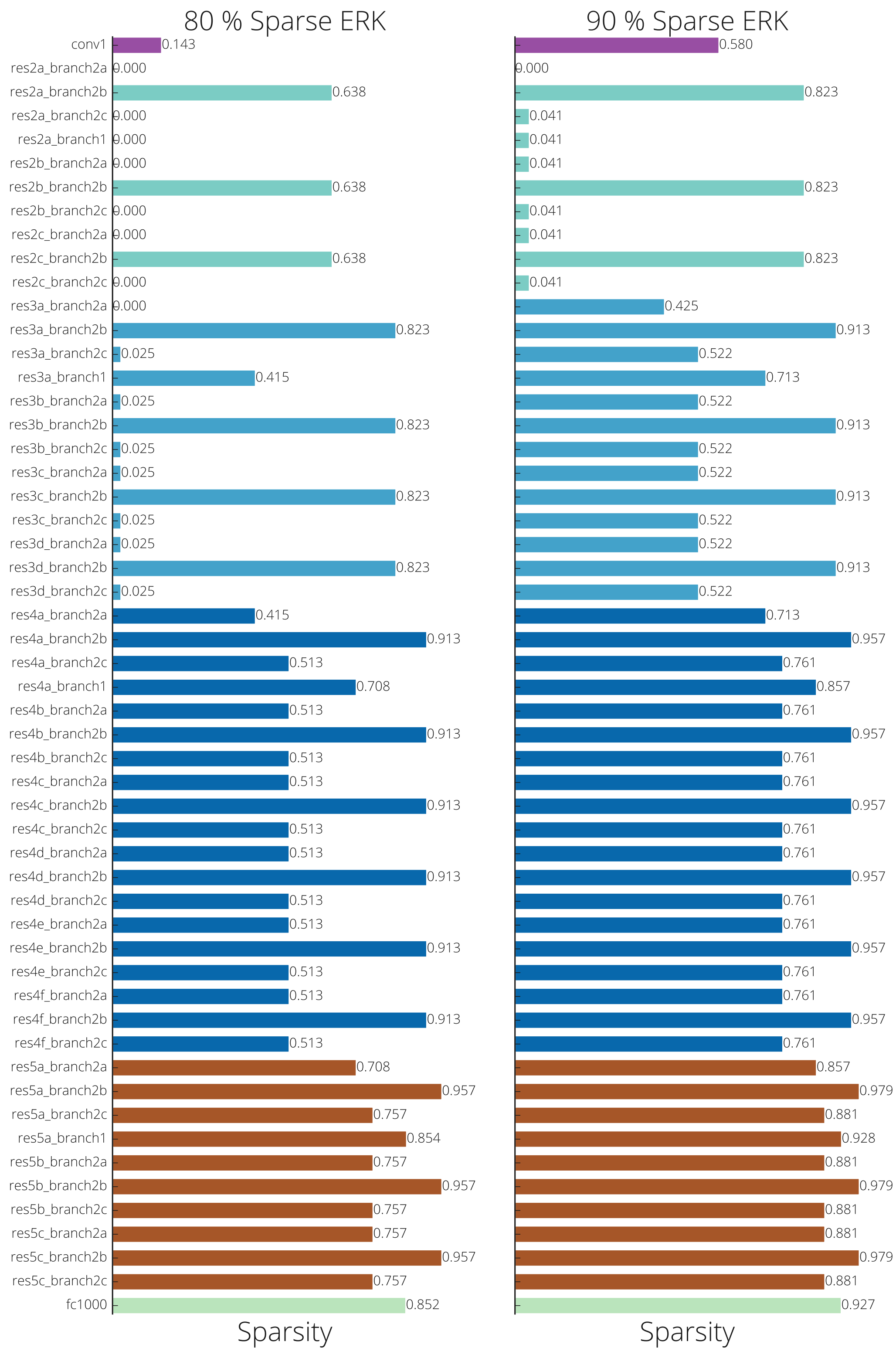

Figure 12. Sparsities of individual layers of the ResNet-50.