Appendices

A. Hyperparameters

In this section, we share the hyperparameters used in thie paper. For fair comparison, we use the exact same hyperparameters and regularization introduced in ON-LSTM (Shen et al., 2019) and AWD-LSTM-MoS (Yang et al., 2018). We then extend the similar settings to stacked LSTMs and RHNs. No hyperparameter tuning techniques such as Melis et al. (2017) are involved in our experiments. No need of finetuning the original hyperparameters of the dense model is another advantage of our method. For all DST methods, the hyperparameters are the same, as shared in Table 4.

B. Ablation Study

To analyze the influence of cell gate redistribution and Sparse NT-ASGD on the performance of sparse RNN training, we conduct an ablation study for all architectures. All models use the same hyperparameters with the ones reported in the main paper. We present the validation and testing perplexity for variants of our model without these two contributions, as shown in Table 5. Not surprisingly, removing either of these two novelties degrades the performance. There is a significant degradation in the performance for all models, up to 13 perplexity point, if the optimizer switches to the standard NT-ASGD. This stands as empirical evidence regarding the benefit of SNT-ASGD. Without cell gate redistribution, the testing perplexity of all models degrades except for RHNs whose number of redistributed weights in each layer is only two. This indicates that cell gate redistribution is more effective for the models with more cell gates.

C. Comparison of Different Cell Gate Redistribution Methods

In Table 6, we conduct a small experiment to compare different methods of cell gate redistribution with stacked LSTMs. We consider weight redistribution based on the mean value of the magnitude of nonzero weights and the mean value of the gradient magnitude of nonzero weights. Our method can achieve the lowest perplexity.

D. Experimental Results with ON-LSTM

Proposed by Shen et al. (2019) recently, ON-LSTM can learn the latent tree structure of natural language by learning the order of neurons. For a fair comparison, we use exactly the same model hyper-parameters and regularization used in ON-LSTM. We set the sparsity of each layer to 55% and the initial pruning rate to 0.5. Same as ON-LSTM, we train the model for 1000 epochs and restart SNT-ASGD as a fine-tuning step once at the 500th epoch, dubbed as Selfish-RNN$_{1000}$. As shown in Table 7, Selfish-RNN outperforms the dense model while reducing the model size to 11.3M. Without SNT-ASGD, sparse training techniques can not reduce the test perplexity to 60. SNT-ASGD is able to improve the performance of RigL by 5 perplexity. Moreover, one interesting observation is that one of the regularizations used in the standard ON-LSTM, DropConnect, is perfectly compatible with our method, although it also drops the hidden-to-hidden weights out randomly during training.

In our experiments we observe that Selfish-RNN benefits significantly from the second fine-tuning operation. We scale the learning schedule to 1300 epochs with two fine-tuning operations at epoch 500 and 1000, respectively, dubbed as Selfish-RNN$_{1300}$. It is interesting that Selfish-RNN$_{1300}$ can achieve lower testing perplexity after the second fine-tuning step, whereas the dense model Dense$_{1300}$ can not even reach again the perplexity that it had before the second fine-tuning. The heuristic explanation here is that our method helps the optimization escape the local optima or a local saddle point by optimizing the sparse structure, while for dense models whose energy landscape is fixed, it is very difficult for the optimizer to find its way off the saddle point or the local optima.

E. Experimental Results with AWD-LSTM-MoS

We also evaluate Selfish-RNN on the WikiText-2 dataset. The model we choose is AWD-LSTM-MoS (Yang et al., 2018), which is the state-of-the-art RNN-based language model. It replaces Softmax with *Mixture of Softmaxes* (MoS) to alleviate the Softmax bottleneck issue in modeling natural language. For a fair comparison, we exactly follow the model hyper-parameters and regularization used in AWD-LSTM-MoS. We sparsify all layers with 55% sparsity except for the prior layer as its number of parameters is negligible. We train our model for 1000 epochs without finetuning or dynamical evaluation (Krause et al., 2018) to simply show the effectiveness of our method. As demonstrated in Table 8, Selfish AWD-LSTM-MoS can match the performance of the corresponding dense model with 15.6M parameters.

F. Effect of Sparsity

There is a trade-off between the sparsity level S and the test perplexity of Selfish-RNN. When there are too few parameters, the sparse neural network will not have enough capacity to fit the data. Here, we analyze this trade-off by training all models with Selfish-RNN at various sparsity levels $S \in [0.50, 0.55, 0.60, 0.70, 0.80, 0.90]$, reported in Figure 6a. We find that Selfish Stacked LSTMs, RHNs,

Table 4. Experiment hyperparameters including Optimizer (Opt), Learning rate (Lr), Batch size (Bs), Backpropagation through time (BPTT), Clip norm (Clip), Non-monotone interval for SNT-ASGD (Nonmono), Initial pruning rate (P); Lr Drop with (A, B) refers to B epochs with no improvement after which learning rate will be reduced by a factor of A; Dropout refers to the word-level dropout, embedding dropout, hidden layer dropout, and output dropout, respectively; Coupled means that the carry gate and the transform gate are coupled in RHNs; Tied means reusing the input word embedding matrix as the output matrix.

Model	Data	Opt	Lr	Lr Drop	Bs	BPTT	Dropout	Epochs	Tied	Coupled	Clip	Nonmono	P
Stacked LSTMs	PTB	Adam SNT-ASGD Momentum SGD	0.001 40 2	(2x, 2) - (1.33x, 1)	20	35	(0, 0, 0.65, 0)	100	no	no	0.25	5	0.7
RHNs	PTB	Adam SNT-ASGD	0.001 15	(2x, 2) -	20	35	(0.2, 0.65, 0.25, 0.65)	500	yes	yes	0.25	5	0.5
ON-LSTM	PTB	Adam SNT-ASGD	0.001 30	(2x, 2) -	20	70	(0.1, 0.5, 0.3, 0.45)	1000	yes	no	0.25	5	0.5
AWD-LSTM-MoS	WikiText-2	Adam SNT-ASGD	0.001 15	(2x, 2) -	15	70	(0.1, 0.55, 0.2, 0.4)	1000	yes	no	0.25	5	0.5

Table 5. Ablation study of Selfish-RNN with stacked LSTMs, RHNs, ON-LSTM on Penn Treebank and AWD-LSTM-MoS on WikiText-2.

Methods	Stacked LSTMs	RHNs	ON-LSTM	AWD-LSTM-MoS
Selfish-RNN	71.65	60.35	55.68	63.05
w/o cell gate redistribution	72.89	60.26	57.48	65.27
w/o Sparse NT-ASGD	73.74	69.70	69.28	71.65

ON-LSTM, and AWD-LSTM-MoS need around 25%, 40%, 45%, and 40% parameters to reach the performance of their dense counterparts, respectively.

G. Effect of Initial pruning rate

The initial pruning rate p determines how many weights would be removed at each connectivity update. We analyze the performance sensitivity of our algorithm to the initial pruning rate p by varying it $\in [0.3, 0.5, 0.7]$. We set the sparsity level of each model as the one having the best performance in Figure 6a. Results are shown in Figure 6b. We can clearly see that our method is very robust to the choice of the initial pruning rate.

H. Difference Among SET, Selfish-RNN and Iterative Pruning Methods

The topology update strategy of Selfish-RNN differs from SET in several important features (1) we automatically redistribute weights across cell gates for better regularization, (2) we use magnitude-based removal instead of removing a fraction of the smallest positive weights and the largest negative weights, (3) we use uniform initialization rather than non-uniform sparse distribution like ER or ERK. Additionally, the optimizer proposed in this work, SNT-ASGD, brings substantial perplexity improvement to the sparse RNN training.

Iterative pruning and retraining techniques (Han et al., 2016; Zhu & Gupta, 2017; Frankle & Carbin, 2019) usually involve three steps (1) pre-training a dense model, (2) pruning the unimportant based on some criteria, and (3) re-training the pruned model to improve performance. The pruning and re-training cycle is required at least once, but may many times depending on the specific algorithms used. Therefore, the computational costs required by iterative pruning and retraining is at least the same as fully training a dense model. Different from iterative pruning and retraining, FLOPs required by Selfish-RNN is proportional to the density of the model, as it allows us to train a sparse network with a fixed number of parameters throughout training in one single run, without any retraining phases. Moreover, the overhead caused by the dynamic sparse connectivity operation is negligible, as it performs only once per epoch.

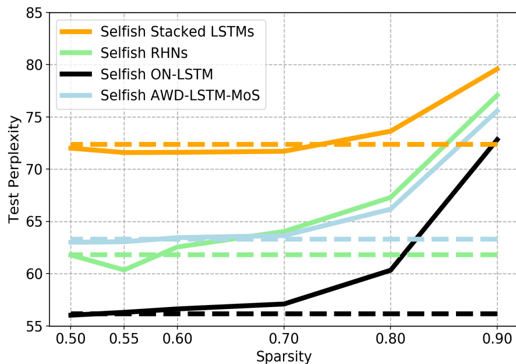
I. Comparison between Selfish-RNN and Pruning

It has been shown by Evci et al. (2020) that while state-of-the-art sparse training method (RigL) achieves promising performance with various CNN models, it fails to match the performance of pruning in RNNs. Given the fact that magnitude pruning has become a widely-used and strong baseline for model compression, we also report a comparison between Selfish-RNN and iterative magnitude pruning with stacked LSTMs. The pruning baseline is obtained from Zhu & Gupta (2017). The results are demonstrated in Figure 7-right.

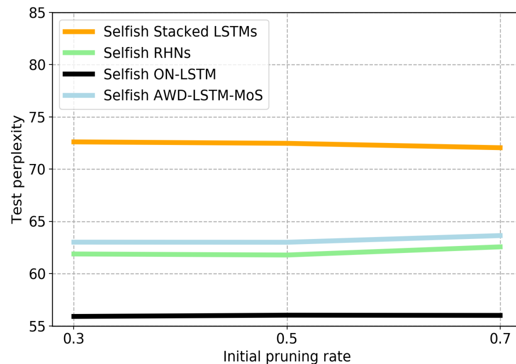
We can see that Selfish-RNN exceeds the performance of pruning in most cases. An interesting phenomenon is that,

Table 6. A small experiment about the comparison among different cell gate redistribution methods. The experiment is evaluated with stacked LSTMs on Penn Treebank.

cell gate redistribution	#Param	Validation	Test
Mean of the magnitude of nonzero weights	21.8M	74.04	72.40
Mean of the gradient magnitude of nonzero weights	21.8M	74.54	72.31
Ours	21.8M	**73.76**	**71.65**



(a) Performance Sensitivity of Sparsity



(b) Performance Sensitivity of Pruning Rate

Figure 6. Sensitivity analysis of sparsity levels S and initial pruning rates p with Selfish stacked LSTMs, RHNs, ON-LSTM, and AWD-LSTM-MoS. (a) Test perplexity of all models with various sparsity levels. The initial pruning rate is 0.7 for stacked LSTMs, and 0.5 for the rest models. The dashed lines represent the performance of the corresponding dense models. (b) Test perplexity of all models with different initial pruning rates. The sparsity level is 67%, 52.8%, 55% and 55% for Selfish stacked LSTMs, RHNs, ON-LSTM, and AWD-LSTM-MoS, respectively.

Table 7. Single model perplexity on validation and test sets for the Penn Treebank language modeling task with ON-LSTM. Methods indicated with "ASGD" are trained with SNT-ASGD. The numbers reported are averaged over five runs.

Models	#Param	Val	Test
Dense_{1000} (NT-ASGD)	25M	58.29 ± 0.10	56.17 ± 0.12
Dense_{1300} (NT-ASGD)	25M	58.55 ± 0.11	56.28 ± 0.19
SET (Adam)	11.3M	65.90 ± 0.08	63.56 ± 0.14
DSR (Adam)	11.3M	65.22 ± 0.07	62.55 ± 0.06
SNFS (Adam)	11.3M	68.00 ± 0.10	65.52 ± 0.15
RigL (Adam)	11.3M	64.41 ± 0.05	62.01 ± 0.13
RigL_{1000} (ASGD)	11.3M	59.17 ± 0.08	57.23 ± 0.09
RigL_{1300} (ASGD)	11.3M	59.10 ± 0.05	57.44 ± 0.15
$\text{Selfish-RNN}_{1000}$ (ASGD)	11.3M	58.17 ± 0.06	56.31 ± 0.10
$\text{Selfish-RNN}_{1300}$ (ASGD)	11.3M	$\mathbf{57.67 \pm 0.03}$	$\mathbf{55.82 \pm 0.11}$

with increased sparsity, we see a decreased performance gap between Selfish-RNN and pruning. Especially, Selfish-RNN performs worse than pruning when the sparsity level is 95%. This can be attributed to the poor trainability problem of sparse models with extreme sparsity levels. Noted in Lee et al. (2020), the extreme sparse structure can break dynamical isometry (Saxe et al., 2014) of sparse networks, which subsequently degrades the trainability of sparse neural networks. Different from sparse training methods, pruning operates from a dense network and thus, does not have this problem.

Table 8. Single model perplexity on validation and test sets for the WikiText-2 language modeling task with AWD-LSTM-MoS. Baseline is AWD-LSTM-MoS obtained from Yang et al. (2018). Methods with "ASGD" are trained with SNT-ASGD.

Models	#Param	Val	Test
Dense (NT-ASGD)	35M	66.01	63.33
SET (Adam)	15.6M	72.82	69.61
DSR (Adam)	15.6M	69.95	66.93
SNFS (Adam)	15.6M	79.97	76.18
RigL (Adam)	15.6M	71.36	68.52
RigL (ASGD)	15.6M	68.84	65.18
Selfish-RNN (ASGD)	15.6M	**65.96**	**63.05**

J. Sparse Connectivity Distance Measurement

Our sparse connectivity distance measurement considers the unit alignment based on a *semi-matching* technique introduced by Li et al. (2016) and a graph distance measurement based on graph edit distance (GED) (Sanfeliu & Fu, 1983). More specifically, our measurement includes the following

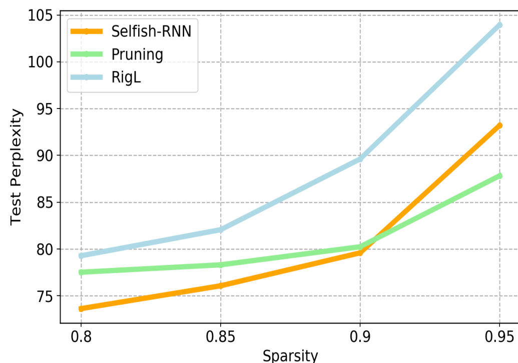


Figure 7. Comparison among Selfish-RNN, RigL and iterative magnitude pruning with stacked LSTMs on PTB. The pruning baseline is obtained from Zhu & Gupta (2017).

steps:

1. We train two sparse networks with dynamic sparse training on the training dataset and store the sparse topology after each epoch. Let $\boldsymbol{W}_l^i$ be the set of sparse topologies for the l-th layer of network i.

2. Using the saved model, we compute the activity output on the test data, $\boldsymbol{O}_l^i \in \mathbb{R}^{n \times m}$, where n is the number of hidden units and m is the number of samples.

3. We pair-wisely match two topologies obtained from different networks $\boldsymbol{W}_l^i$ and $\boldsymbol{W}_l^j$ by the semi-matching method introduced in Li et al. (2016) based on their activity units. The semi-matching step is achieved by finding the a pair of units from different networks with the maximum correlation.

4. After alignment, we apply graph edit distance (GED) to measure the similarity between $\boldsymbol{W}_l^i$ and $\boldsymbol{W}_l^j$. Eventually, the distance is scaled to lie between 0 and 1. The smaller the distance is, the more similar the two sparse topologies are.

Here, we choose stacked LSTMs on PTB dataset as a specific case to analyze. Specifically, we train two stacked LSTMs for 100 epochs with different random seeds. We choose a relatively small pruning rate of 0.1. We start alignment at the 5th epoch to ensure a good alignment, as the model does not learn very well at the very beginning of training.

K. Sparse Connectivity Distance Comparison between Different Growth Methods

In this section, we investigate the topological distance between sparse connectivities learned by gradient weight growth and random weight growth. We empirically illustrate that gradient growth drives different networks into some similar connectivity patterns based on the proposed distance measurement between sparse connectivities. The initial pruning rates are set as 0.1 for all training runs in this section.

First, we measure the sparse connectivity distance between two different training runs trained with gradient growth and random growth, respectively, as shown in Figure 8. We can see that, starting with very different sparse connectivity topologies, two networks trained with random growth end up at the same distance, whereas the distance between two networks trained with gradient growth is continuously decreasing and this tendency is likely to continue as the training goes on. We further report the distance between two networks with same initialization (same sparse connectivity and same weight values) but different training seeds in Figure 9. We can see that the distance between sparse connectivities optimized by gradient growth is smaller than the ones optimized by random growth.

These observations are in line with our hypothesis and indicate that gradient growth drives networks into some similar structures, whereas random growth allows models to explore more sparse structures spanned over the dense network, and thus has a better chance to find a better sparse connectivity.

L. FLOPs Analysis of Different Approaches

Although different DST methods maintain a fix parameter count throughout training, their training costs can be very different since different sparse distributions lead to different computational costs. Hence, we also report the estimated training and inference FLOPs for all methods in this section.

We follow the way of calculating training FLOPs proposed by Evci et al. (2020). The perplexity and the corresponding training and inference FLOPs of different methods are given in Table 9. We split the process of training a sparse recurrent neural network into two steps: *forward pass* and *backward pass*.

Forward pass In order to calculate the loss of the current models given a batch of input data, the output of each layer is needed to be calculated based on a linear transformation and a non-linear activation function. Within each RNN layer, different cell gates are used to regulate information in sequence using the output of the previous time step and the input of this time step.

Backward pass In order to update weights, during the backward pass, each layer calculates 2 quantities: the gradient of the loss function with respect to the activations of the previous layer and the gradient of the loss function with

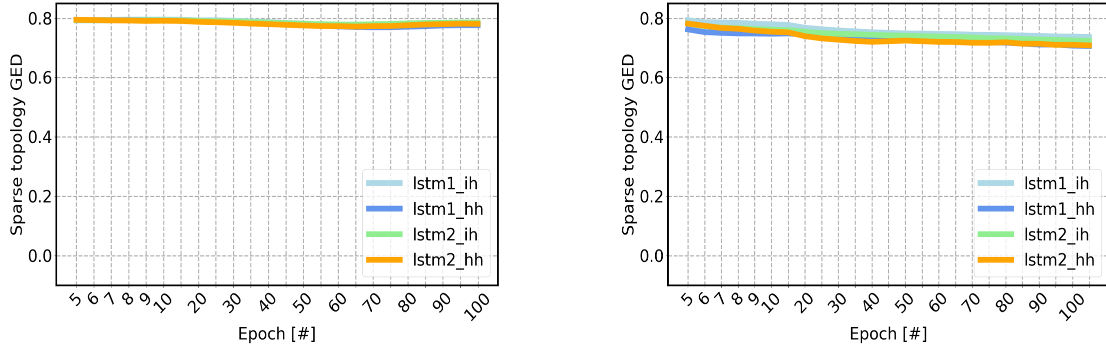


Figure 8. (**left**) The sparse connectivity distance between two different training runs of stacked LSTMs trained with random growth. (**right**) The sparse connectivity distance between two different training runs of stacked LSTMs trained with gradient growth.

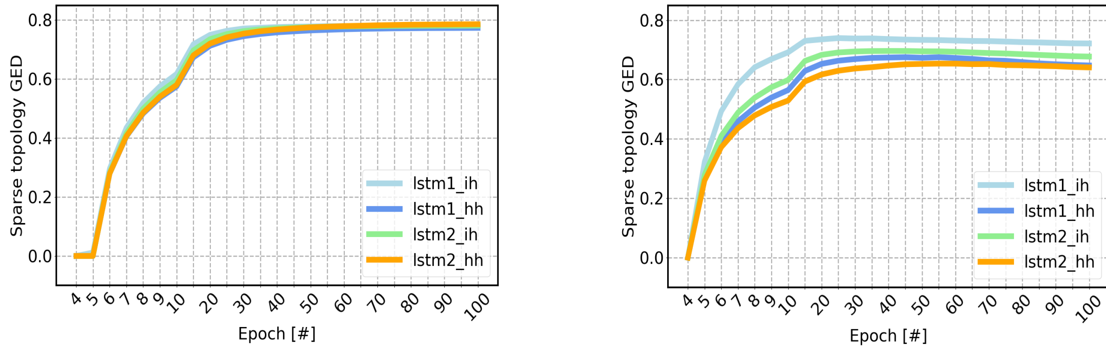


Figure 9. (**left**) The sparse connectivity distance between two stacked LSTMs with same initialization but different training seeds trained with random growth. (**right**) The sparse connectivity distance between two networks with same initialization but different training seeds trained with gradient growth. ih is the input weight tensor comprising four cell gates and hh is the hidden state weight tensor comprising four cell gates.

respect to its own weights. Therefore, the computational expense of *backward pass* is twice that of *forward pass*. Given that RNN models usually contain an embedding layer from which it is very efficient to pick a word vector, for models not using weight tying, we only count the computations to calculate the gradient of its parameters as the training FLOPs and we omit its inference FLOPs. For models using weight tying, both the training FLOPs and the inference FLOPs are omitted.

Given a specific architecture, we denote f_D as dense FLOPs required to finish one training iteration and f_S as the corresponding sparse FLOPs ($f_S \approx (1 - S)f_D$), where S is the sparsity level. Thus $f_S \ll f_D$ for very sparse networks. Since different sparse training methods use different sparse distribution, their FLOPs f_S are also different from each other. We omit the FLOPs used to update the sparse connectivity, as it is only performed once per epoch. Overall, the total FLOPs required for one training update on one single sample are given in Table 10. The training FLOPs of dense-to-sparse methods like, ISS and pruning, are $3f_D * s_t$, where s_t is the sparsity of the model at iteration t. Since dense-to-sparse methods require to train a dense model for a while, their training FLOPs and memory requirement are higher than our method. For methods that allow the sparsity of each layer dynamically changing e.g., DSR and SNFS, we approximate their training FLOPs via their final distribution, as their sparse distribution converges to the final distribution in the first few epochs. ER distribution causes a bit more inference FLOPs than uniform distribution because it allocates more weights to the RNN layers than other layers. SNFS requires extra FLOPs to calculate dense momentum during the backward pass. Although RigL also uses the dense gradients to assist weight growth, it only needs to calculate dense gradients every ΔT iterations, thus with a smaller number of FLOPs given by $\frac{3f_S \Delta T + 2f_S + f_D}{\Delta T + 1}$. Here, we simply omit the extra FLOPs required by the full gradient calculation as it is negligible compared with the whole training FLOPs. Moreover, the inference FLOPs are calculated with the final sparse distribution learned by different methods.

Table 9. Single model perplexity on validation and test sets for the Penn Treebank language modeling task with stacked LSTMs and RHNs. FLOPs required to train the entire model and to test on single sample are reported. '*' indicates the reported results from the original papers: "Dense" is obtained from Zaremba et al. (2014) and ISS is obtained from Wen et al. (2018). "Static-ER" and "Static-uni" are the static sparse network trained from scratch with ER distribution and uniform distribution, respectively. "Small" refers the small-dense network.

	Stacked LSTMs				RHNs			
Models	FLOPs (Train)	FLOPs (Test)	Val	Test	FLOPs (Train)	FLOPs (Test)	Val	Test
Dense*	1x(3.1e16)	1x(7.2e10)	82.57	78.57	1x(6.5e16)	1x(3.3e10)	67.90	65.40
Dense (NT-ASGD)	1x	1x	74.51	72.40	1x	1x	63.44	61.84
	S=0.67				S=0.53			
Small (NT-ASGD)	0.33x	0.33x	88.67	86.33	0.47x	0.47x	70.10	68.40
Static-ER (SNT-ASGD)	0.33x	0.34x	81.02	79.30	0.47x	0.47x	75.74	73.21
Static-uni (SNT-ASGD)	0.33x	0.33x	80.37	78.61	0.47x	0.47x	74.11	71.83
ISS*	0.28x	0.20x	82.59	78.65	0.50x	0.47x	68.10	65.40
GMP (Adam)	0.63x	0.33x	89.47	87.97	0.62x	0.47x	63.21	61.55
SET (Adam)	0.33x	0.34x	87.30	85.49	0.47x	0.47x	63.66	61.08
DSR (Adam)	0.38x	0.40x	89.95	88.16	0.47x	0.47x	65.38	63.19
SNFS (Adam)	0.63x	0.38x	88.31	86.28	0.63x	0.45x	74.02	70.99
RigL (Adam)	0.33x	0.34x	88.39	85.61	0.47x	0.47x	67.43	64.41
Selfish-RNN (Adam)	0.33x	0.33x	85.70	82.85	0.47x	0.47x	63.28	60.75
GMP (SNT-ASGD)	0.63x	0.33x	76.78	74.84	0.62x	0.47x	65.63	63.96
RigL (SNT-ASGD)	0.33x	0.34x	78.31	75.90	0.47x	0.47x	64.82	62.47
Selfish-RNN (SNT-ASGD)	0.33x	0.33x	**73.76**	**71.65**	0.47x	0.47x	**62.10**	**60.35**
	S=0.62				S=0.68			
ISS*	0.32x	0.23x	80.24	76.03	0.34x	0.32x	70.30	67.70
GMP (SNT-ASGD)	0.63x	0.38x	74.86	73.03	0.51x	0.32x	66.61	64.98
RigL (SNT-ASGD)	0.38x	0.39x	77.16	74.76	0.32x	0.32x	69.32	66.64
Selfish-RNN (SNT-ASGD)	0.38x	0.38x	**73.50**	**71.42**	0.32x	0.32x	**66.35**	**64.03**

Table 10. Training FLOPs analysis of different sparse training approaches. f_D refers to the training FLOPs for a dense model to compute one single prediction in the *forward pass* and f_S refers to the training FLOPs for a sparse model. ΔT is the number of iterations used by RigL to update sparse connectivity. s_t is the sparsity level of the model at iteration t.

Method	Forward Pass	Backward Pass	Total
Dense	f_D	$2f_D$	$3f_D$
ISS	$f_D * s_t$	$2f_D * s_t$	$3f_D * s_t$
Pruning	$f_D * s_t$	$2f_D * s_t$	$3f_D * s_t$
SET	f_S	$2f_S$	$3f_S$
DSR	f_S	$2f_S$	$3f_S$
SNFS	f_S	$f_S + f_D$	$2f_S + f_D$
RigL	f_S	$\frac{(2\Delta T+1)f_S+f_D}{\Delta T+1}$	$\frac{3f_S\Delta T+2f_S+f_D}{\Delta T+1}$
Selfish-RNN (ours)	f_S	$2f_S$	$3f_S$

M. Final Cell Gate Sparsity Breakdown

We further investigate the final sparsity level of different cell gates learned automatically by our method in Figure 10. We find a consistent observation existing in all models, i.e., the weight of the forget gates, either the forget gate in the standard LSTM or the master forget gate in ON-LSTM, tend to be sparser than other gate weights. The weight of the cell gates and output gates are denser than the average. However, there is no big difference between the gates in RHNs, even

although the H nonlinear transform gate is slightly sparser than the T gate weight in most RHNs layers.

N. Limitation

The aforementioned training benefits have not been fully explored, as off-the-shelf software and hardware have limited support for sparse operations. The unstructured sparsity is difficult to be efficiently mapped to the existing parallel processors. The results of our paper provide motivation for new types of hardware accelerators and libraries with better support for sparse neural networks. Nevertheless, many recent works have been developed to accelerate sparse neural networks including Gray et al. (2017); Moradi et al. (2019); Ma et al. (2019); Yang & Ma (2019); Liu et al. (2020b). For instance, NVIDIA develops the A100 GPU enabling the Fine-Grained Structured Sparsity (NVIDIA, 2020). The sparse structure is enforced by allowing two nonzero values in every four-entry vector to reduce memory storage and bandwidth by almost $2\times$. We hope that our results will pile up on other researchers results in sparse training and soon there will be a change of perspective in such a way that the developers of deep learning software and hardware will start considering including real sparsity support in their solutions.

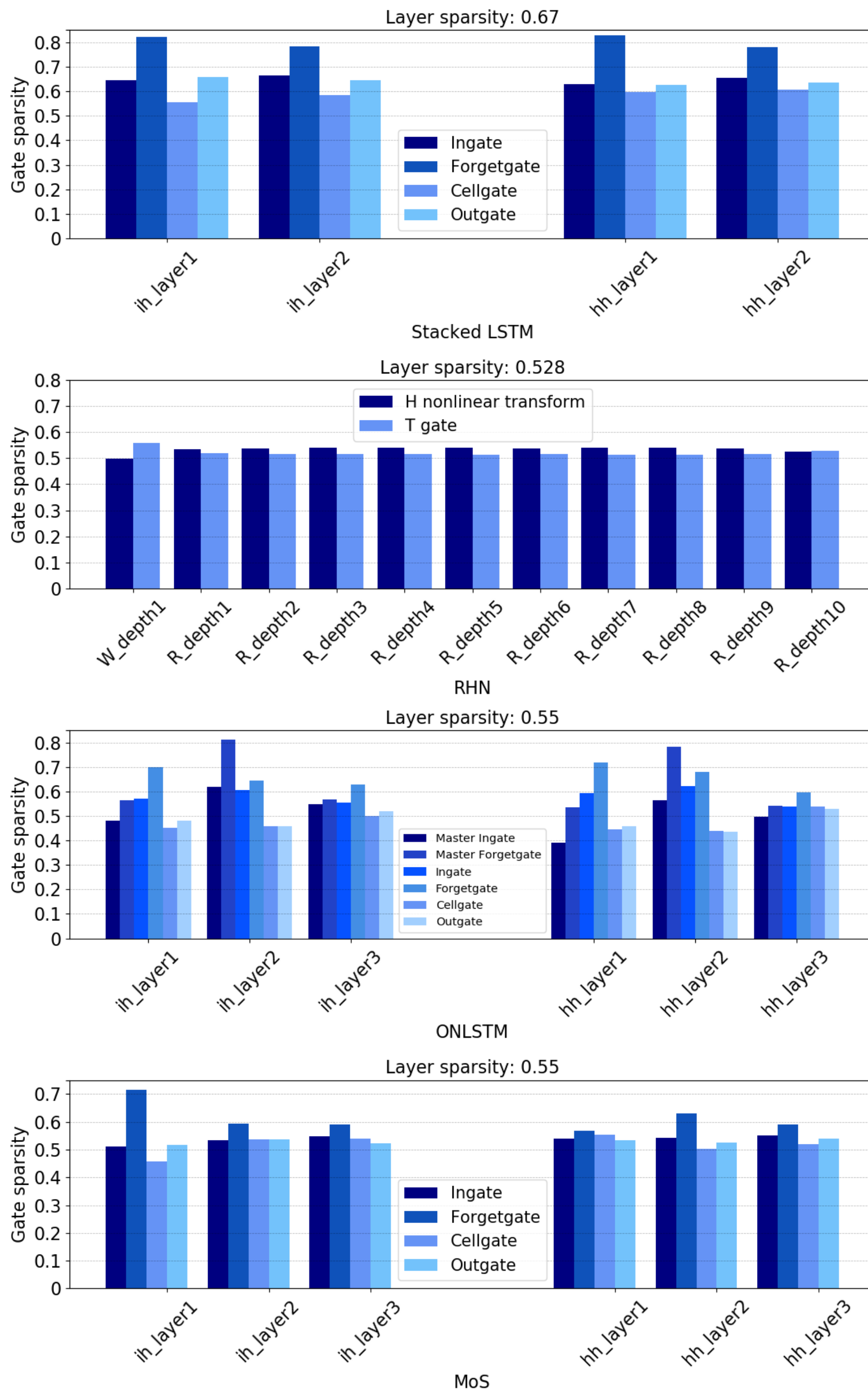


Figure 10. Breakdown of the final sparsity level of cell gates with stacked LSTMs, RHNs, ON-LSTM on PTB, AWD-LSTM-MoS on Wikitext-2. W and R is the weight of the H nonlinear transform and the T gate in RHNs, respectively; ih and hh refer to the input weight and the hidden weight of each LSTM layer, respectively.