BLOCK-SPARSE RECURRENT NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent Neural Networks (RNNs) are used in state-of-the-art models in domains such as speech recognition, machine translation, and language modelling. Sparsity is a technique to reduce compute and memory requirements of deep learning models. Sparse RNNs are easier to deploy on devices and high-end server processors. Even though sparse operations need less compute and memory relative to their dense counterparts, the speed-up observed by using sparse operations is less than expected on different hardware platforms. In order to address this issue, we investigate two different approaches to induce *block* sparsity in RNNs: pruning blocks of weights in a layer and using group lasso regularization with pruning to create blocks of weights with zeros. Using these techniques, we can create block-sparse RNNs with sparsity ranging from 80% to 90% with a small loss in accuracy. This technique allows us to reduce the model size by roughly 10×. Additionally, we can prune a larger dense network to recover this loss in accuracy while maintaining high block sparsity and reducing the overall parameter count. Our technique works with a variety of block sizes up to 32×32. Block-sparse RNNs eliminate overheads related to data storage and irregular memory accesses while increasing hardware efficiency compared to unstructured sparsity.

1 INTRODUCTION

Improvements in several applications such as speech recognition (Amodei et al., 2016), language modeling (Józefowicz et al., 2016), and machine translation (Wu et al., 2016) are a result of large Recurrent Neural Networks (RNNs) trained on large scale datasets. As the datasets available to train these models have grown, so have model sizes. Deployment of such large models is compute and memory intensive.

Pruning weights of deep neural networks is an effective strategy to reduce the overall memory and compute requirements of these models (Narang et al., 2017; Han et al., 2015). However, these approaches induce random, unstructured sparsity in the weight matrices. Speed-up obtained with unstructured sparsity on various hardware platforms are often lower than expected (as shown in Narang et al. (2017); Narang & Diamos (2017)). Sparse formats do not efficiently utilize the hardware resources due to storage overheads and irregular memory access. Block sparsity can address these issues. Saving indices of non-zero blocks instead of indices for non-zero elements reduces the storage overhead by a factor of block size. Block-sparse formats store blocks contiguously in memory reducing irregular memory accesses.

Another disadvantage of unstructured sparsity is that it cannot directly exploit array-data-paths in modern processors. These include the 16×16 TensorCore units in the Volta GPU (NVIDIA, 2017) or the 256×256 hardware units in the Tensor Processing Unit (TPU) (Jouppi et al., 2017). Structured sparsity in the form of two-dimensional blocks allows us to take advantage of these faster units.

In order to induce block sparsity in RNNs, we propose a block pruning approach that zeros out blocks of weights in the matrix while the network is training. At the end of training, the algorithm creates a block-sparse RNN. In addition to this pruning technique, we examine the efficacy of group lasso regularization (Yuan & Lin, 2006b) to induce block sparsity in the network. We also combine group lasso regularization with block pruning.

We demonstrate that block pruning and group lasso regularization with pruning are successful in creating block-sparse RNNs. Inducing block sparsity with 4×4 blocks in vanilla RNNs and Gated Recurrent Units (GRUs) (Cho et al., 2014) results in 9% to 17% loss in accuracy compared to the

dense baseline. Model size reduces by nearly 10×for speech recognition. Block sizes can be scaled up to 32×32 with our approach. We can also reduce accuracy loss by starting with a larger dense matrix than the baseline and then pruning it down while still reducing the number of parameters compared to the baseline. We demonstrate that this approach works with Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) cells for Language Modelling as well.

Our approach is agnostic to the optimization algorithm and does not require any hyper-parameter retuning (besides pruning and regularization hyper-parameters). Furthermore, since our approach does not require re-training the model, training time remains constant.

2 Related Work

There have been several approaches to reduce the network size by pruning the model. Hanson & Pratt (1989) use several bias techniques to decay weights in a network. LeCun et al. (1989) and Hassibi et al. (1993) both use Hessian-based approaches to prune weights below a certain threshold. Simpler approaches like sorting or thresholding can be used to prune a neural network. Han et al. (2015) and Liu et al. (2015) prune Convolution Neural Networks (CNNs) while maintaining high accuracy. Yu et al. (2012) use a hard threshold to prune deep learning models. Narang et al. (2017) and Zhu & Gupta (2017) prune recurrent neural networks during the initial training run with a small accuracy loss using gradual pruning. Unlike our technique, all of the above approaches induce random, unstructured sparsity in neural networks.

Several approaches exist to induce structured sparsity in neural networks. Mao et al. (2017) use a simple threshold based technique to create structurally sparse CNNs. Yu et al. (2017) propose Scalpel, which prunes CNNs taking into account the underlying target hardware architecture. Wen et al. (2017) alter the structure of LSTMs to create cells with smaller memory footprint. They demonstrate that this technique works for language modeling on the Penn Tree Bank dataset. Our approach works with both vanilla RNN and GRU models trained on a large-scale datasets for speech recognition.

Regularization is a known method to induce sparsity in deep neural networks (Faraone et al., 2017; Fan et al., 2016). Group lasso regularization has been used as an efficient method for generating sparse structures (Yuan & Lin, 2006b; Kim & Xing, 2010). Wen et al. (2016) use group lasso regularization to induce structured sparsity in CNNs. Scardapane et al. (2017) also use group lasso regularization to induce sparisty in fully connected networks. To the best of our knowledge, none of these approaches have been used with RNNs trained on large-scale datasets.

Other approaches to reduce compute and memory footprint for deep learning models include quantization (Micikevicius et al., 2017; Vanhoucke et al., 2011; Rastegari et al., 2016; Gupta et al., 2015) and low-rank factorization (Denil et al., 2013; Denton et al., 2014). Our approach is orthogonal to these methods and can be combined with them.

3 Implementation

3.1 Block Pruning

Our approach to pruning deep learning models builds on the work by Narang et al. (2017). They propose a weight pruning algorithm that introduces random, unstructured sparsity in RNNs. In their work, they propose pruning weights below a monotonically increasing threshold. Their pruning strategy does not impose any structure on the weights.

We extend this approach to prune blocks of a matrix instead of individual weights. We divide the weight matrix into a grid of two-dimensional blocks with a fixed block size. Block size ranges between 4×4 to 32×32 in our experiments. In order to prune blocks, we pick the weight with the maximum magnitude to represent the entire block. If this maximum magnitude of a block is below the threshold, we set all the weights in that block to zeros. Figure 1 depicts the process of generating a block-sparse mask from a weight matrix for a given threshold. The block-sparse mask is multiplied with the weights to generate block-sparse weight matrix. The monotonically growing threshold (ϵ) causes more blocks to be pruned as training progresses. We stop pruning more blocks after 40% of training has completed. All zeroed out blocks are held at zero until the end of training.

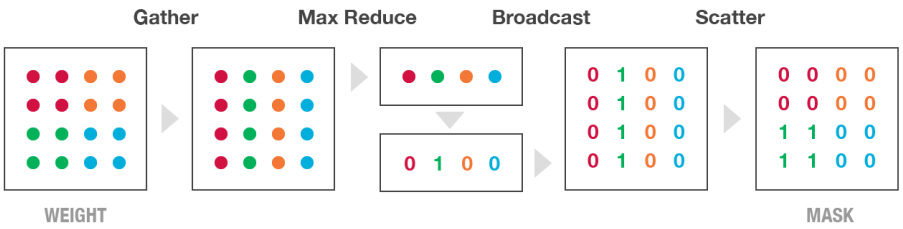


Figure 1: Generating block-sparse masks from a weight matrix

Table 1: Heuristics to pick hyper-parameters for block-pruning

HYPER-PARAM	DESCRIPTION	HEURISTIC VALUES
start_itr	Iteration to start pruning	Start of second epoch
ramp_itr	Iteration to increase the rate of pruning	Start of 20% of total epochs
end_itr	Iteration to stop pruning more parameters	Start of 40% of total epochs
start_slope (θ)	Initial rate of increasing the threshold	See Equation 2
ramp_slope (ϕ)	Rate of increasing threshold after ramp iteration	1.2θ to 1.7θ
freq	Number of iterations after which ϵ is updated	100

Narang et al. (2017) use six hyper-parameters to determine the threshold at a given iteration. Table 1 provides the description and heuristics (adapted for block pruning) for these hyper-parameters. The *start_slope* and *ramp_slope* determine the rate at which the threshold increases. In order to determine *start_slope*, they recommend using weights from an existing dense model. To achieve 90% sparsity, they assign q to the weight that is the 90th percentile of the absolute values in a weight matrix. Assuming ϕ is 1.5θ, they use Equation 1 to determine θ.

$$\theta = \frac{2 \times q \times freq}{2 \times (ramp_itr - start_itr) + 3 \times (end_itr - ramp_itr)} \tag{1}$$

For block pruning, we need to modify the *start_slope* to take into account the number of elements in a block (N_b). In order to calculate the *start_slope*, we first calculate *start_slope* for weight pruning (θ_w) using the Equation 1. Given θ_w, we suggest using Equation 2 to determine the initial slope (θ_b) for block pruning. Based on empirical results, we have found that using this approach allows us to achieve block sparsity ranging from 85% to 95%. Further tuning of these hyper-parameters is required to achieve desired block sparsity.

$$\theta_b = \theta_w \times \sqrt[4]{N_b} \tag{2}$$

We prune all the recurrent and fully connected layers in the network using the same block size. The pruning hyper-parameters are same for each type of layer in the network — recurrent weight layer and linear or fully connected layer.

3.2 Group Lasso Regularization

Group lasso is a type of weight regularization that works on groups of weights. For each group, we add a loss term proportional to the ℓ_2 norm of the group.

$$L = L_{\text{training}} + \lambda_g \sum_{g=1}^{G} \|w^{(g)}\|_2$$

where $w^{(g)}$ is a group of weights, $\|w^{(g)}\|_2$ is the ℓ_2 norm of the group, and G is the total number of groups. Our use of ℓ_2 norm is a variant of the more general group lasso defined in Yuan & Lin (2006a) as $\|n\|_K = (n'Kn)^{1/2}$.

A large enough λ_g will drive all weights within certain groups to zeros. The choice of grouping varies by application, and group lasso is widely-used to induce various kinds of structured sparsity (Wen et al., 2017; Scardapane et al., 2017). By choosing groups to exactly match our 2D grid of blocks, we can induce block-sparsity. Thus, group lasso is an existing sparsity algorithm that we can readily compare to our block pruning approach.

In addition, we extend group lasso to work with block pruning. The groups match the 2D grid of blocks used by the pruning algorithm. One interpretation of weight regularization is that less important weights are driven towards zero and more important weights retain large absolute values. Thus, group lasso guides the selection of blocks to prune. We apply group lasso to coincide with the pruning schedule. We use a relatively small λ_g to avoid underfitting due to excessive regularization. We turn off group lasso when the pruning schedule ends, which is typically after around 40% of training epochs. Weights that were already set to zero remain unchanged after this point.

Group lasso is related to the well-known ℓ_1 regularization. In Appendix A, we discuss exploration of ℓ_1 regularization combined with weight pruning.

4 Experiments

We present results on two different applications: Speech Recognition (Section 4.1) and Language Modelling (Section 4.2).

4.1 Speech Recognition

We run block sparsity experiments on two different speech recognition models from Amodei et al. (2016). The RNN model consists of seven bidirectional vanilla recurrent layers with 1760 hidden units for a total of 67 million parameters. The GRU model consists of three recurrent layers with GRU cells with 2560 hidden units for a total of 115 million parameters. Both models use the Connectionist Temporal Classification (CTC) (Graves et al., 2006) cost function. We use a training set of 2100 hours of speech and validation set of 3.46 hours. The Character Error Rate (CER) results are reported on an independent test set, consisting of 2.9 hours of English data.

In order to introduce block sparsity in these models, we run two different types of experiments — Block Pruning (BP) and Group Lasso with block pruning (GLP). We prune weights in the recurrent layers (both linear and recurrent weights) and fully connected layers. Biases, batch-normalization parameters and weights in the convolutional and CTC layers are not pruned since they account for a small portion of the total weights in the network. No existing hyper-parameter changes were required for sparse training runs. The models are trained using Nesterov Stochastic Gradient Descent (SGD) with momentum. All models are trained for 25 epochs.

In Section 4.1.1, we report results for different sparse models pruned with 4×4 blocks and compare these results with other pruning approaches. In Section 4.1.2, we discuss the impact of varying the block size on the accuracy of the model.

4.1.1 Block Sparsity

Initially, we prune the dense RNN model. Using BP, we are able to reduce the parameter count for both these models by nearly 10×. As shown in Table 2, the block-sparse RNN model with 1760 hidden units has an overall block sparsity of 89% with a CER of 17.93.

As mentioned in Section 3, group lasso by itself can induce block-sparsity. However, as shown in Table 2, group lasso results in significantly worse CER than our block pruning approach. In order to achieve high sparsity (80% or higher) with group lasso, we need to set λ_g to a relatively high value. This high regularization factor hurts the model accuracy. The dense baseline model is trained without any regularization. Therefore, group lasso results in underfitting the training data due to the high value of λ_g. Group lasso could be more successful in inducing block-sparsity where the dense model overfits the training dataset.

Table 2: Bidirectional RNN model results. Block-Sparse models are trained with 4x4 blocks

MODEL	LAYER SIZE	# PARAMS (in millions)	CER (% LOSS)	EPOCHS	PRUNING ALGORITHM
RNN Dense	1760	67	15.36 (0.0%)	25	N/A
RNN Block-Sparse	1760	10.9	30.14 (-96%)	25	Group lasso
RNN Sparse	1760	8.3	18.91 (-23%)	25	Yu et al. (2012)
RNN Sparse	1760	7.3	17.32 (-13%)	25	Narang et al. (2017)
RNN Sparse	1760	7.1	**15.41 (-0.3%)**	**60**	Han et al. (2015)
RNN Block-Sparse	1760	7.3	17.93 (-17%)	25	Ours (BP)
RNN Block-Sparse	2560	12.9	**15.89 (-3.4%)**	25	Ours (GLP)
RNN Block-Sparse	3072	25.8	**15.66 (-1.9%)**	25	Ours (BP)

Table 3: GRU model results with 4×4 blocks

MODEL	LAYER SIZE	# PARAMS (in millions)	CER (% LOSS)	EPOCHS	PRUNING ALGORITHM
GRU Dense	2560	115	15.42 (0.0%)	25	N/A
GRU Block-Sparse	2560	10.8	**16.78 (-8.8%)**	25	Ours (GLP)
GRU Block-Sparse	3584	25.6	**16.23 (-5.3%)**	25	Ours (BP)

Comparison to other pruning methods: In addition to group lasso, we compare our block pruning approach with three existing pruning methods. As shown in Table 2, our block-sparse model achieves better accuracy than the hard thresholding scheme in Yu et al. (2012). Sparse RNNs generated using Narang et al. (2017) is about 4% better than the block-sparse model. The sparse RNN model generated using iterative pruning (Han et al., 2015) is significantly better than than block-sparse model. However, this approach requires training the model for 60 epochs instead of 25 epochs for all other approaches. This results in 180 hours of additional training time for the RNN model. This 2-3×increase in training time may not be practical for state-of-the-art models trained on large datasets, which usually need weeks of training time. Additionally, all the above approaches generate random, unstructured sparsity in the model. In current hardware, the compute and memory savings with block sparsity are significantly higher than random sparsity. We discuss the performance aspect in more detail in Section 5.

Larger models: In order to recover the accuracy loss with our approach, we train sparse models with more hidden units in each recurrent layers. For RNN models, we increase the hidden layer size to 2560 and 3072. As shown in Table 2, the RNN sparse 3072 is only 1.9% worse than the dense baseline model. The 2560 and 3072 sparse RNN models reduce the overall parameter count by 5×and 2.5×respectively relative to the dense model with 1760 hidden units in each layer.

GRU model: Similar to the RNN models, the block-sparse GRU model can reduce the overall parameter count by 11×. As shown in Table 3, the block-sparse GRU model achieves slightly higher sparsity (90%) with a CER of 16.23 which is only 9% worse than the dense GRU model. This indicates that the block-sparse GRU model retains most of the capacity of the dense model. As demonstrated with the RNN model, pruning a larger GRU model with 3584 hidden nodes reduces the accuracy loss to about 5% while still shrinking the model by 4.5×relative to the dense model with 2560 hidden nodes in each layer.

4.1.2 Block Size Variation

Table 4 shows that block pruning works for block sizes upto 32×32. Increasing the block size to 16×16 and 32×32 requires reducing the sparsity to 83.6% and 79.1% respectively for RNN models

Table 4: Results for GRU model with 2560 layer size and bidirectional RNN model with 170 layer size pruned with different block sizes using BP.

MODEL	BLOCK SIZE	# PARAMS (in millions)	SPARSITY	CER (% LOSS)
RNN Block-Sparse	4x4	7.3	89.2%	17.93 (-17%)
RNN Block-Sparse	12x2	10.8	84.1%	16.96 (-10%)
RNN Block-Sparse	8x8	10.7	84.1%	17.66 (-15%)
RNN Block-Sparse	**16x16**	11.1	83.6%	17.10 (-11%)
RNN Block-Sparse	**32x32**	14.1	79.1%	16.67 (-8.5%)
GRU Block-Sparse	4x4	16.2	86.0%	16.97 (-10%)
GRU Block-Sparse	**16x16**	20.8	81.9%	16.84 (-9.2%)

Table 5: Word language modelling results on Penn Tree Bank using BP for 4×4 blocks.

MODEL	LAYER SIZE	# PARAMS (in millions)	PERPLEXITY (% LOSS)	EPOCHS	PRUNING ALGORITHM
LSTM Dense	1500	66.0	78.29 (0.0%)	55	N/A
LSTM Block-Sparse	1500	23.1	**77.04 (1.6%)**	55	Ours (BP)
LSTM Block-Sparse	1500	11.6	80.25 (-2.5%)	55	Ours (BP)
LSTM Block-Sparse	1500	**7.95**	82.72 (-5.7%)	55	Ours (BP)

to obtain good accuracy. Similar results hold true for the GRU model as well. Large sparse blocks reduce memory overhead for storing non zero values and can take advantage of array data-paths in modern processors. Therefore, even though large blocks achieve lower sparsity, they result in lower memory and compute requirements.

The exact choice of block size for a given application depends on the underlying hardware used for inference. For example, NVIDIA's Volta processor supports 16×16 blocks whereas ARM processors support blocks of 12x2. We demonstrate that our approach is agnostic to block size and can be used to generate block-sparse models for arbitrary blocks.

4.2 Neural Language Modelling

We conducted block pruning experiments on Penn Tree Bank (PTB) (Marcus et al., 1993) dataset using word level language models. For our experiments, we use the large LSTM model with 1500 hidden units from Zaremba et al. (2014). The hyperparameters are unchanged from the original model, except for slightly increased dropout keep probability which ranges from 0.4 to 0.52 for the sparse models. We prune weights in the embedding, LSTM and softmax layers of the model.

We report results on the test set using BP with 4×4 blocks in Table 5. Block pruning can reduce the parameter count by nearly 3×while retaining the accuracy of the dense model. With a 5% loss in accuracy, we can reduce the parameter count by 8.3×. There is a trade-off between sparsity and accuracy of the model. For inference, we would pick the model that meets the desired memory and compute budget.

5 Performance

Sparse formats incur at least three types of overhead: i) indexing overhead, ii) irregular memory accesses, and ii) incompatibility with array-data-paths, all of which are mitigated by using larger block sizes.

Indexing Overheads. Sparse formats use extra memory to track the location of each non-zero value. For example, the compressed-sparse-row (CSR) format uses between one and two extra index values for each non-zero value. Assuming that neural network weights and indices are represented with 16-bits as in Micikevicius et al. (2017), this is at least 100% overhead. Block sparsity reduces this overhead by a factor of the block size because the index is shared over the entire block. For example, using a block size of 4x4 reduces the memory bloat to 6.25%, and using a block size of 16x16 reduces it to less than 1%.

Irregular Memory Accesses. Caches lines, DRAM row buffers, and TLBs provide the best performance when memory is accessed in relatively large contiguous units (e.g. 64 bytes for cache lines, 4KB for a DRAM row) as opposed to in fine-grained random accesses. Block-sparse formats store blocks contiguously in memory, resulting in large coalesced accesses.

Array Data-Paths. Block-sparse models make it easier to exploit array-data-paths in modern processors. There are significant advantages of using these units, for example, on the Volta V100 GPU, they enable up to 8x higher throughput than the SIMD data-paths. In order to keep these units busy, the block size should be at least as large as the hardware data-path size (16×16 or larger on V100).

5.1 Inference Performance

Inference performance depends of three different factors: accuracy, latency of evaluation and memory requirements. In order to understand the trade-off between unstructured sparsity and block sparsity, we benchmark the General Matrix Multiply (GEMM) speed-up and memory reduction for a single layer in the speech recongition RNN model. We evaluate GEMM speed-up with batch size of 16 using NVIDIA's CuSparse and CuBLAS libraries on a TitanX Maxwell GPU. Sparse matrices are represented in CSR or Block-CSR format depending on the sparsity structure. Memory savings are calculated using CSR and Block Sparse Row (BSR) from Scipy module in Python. We evaluate a single layer with different block sizes. We also evaluate the best unstructured sparsity result obtained using iterative pruning from Han et al. (2015).

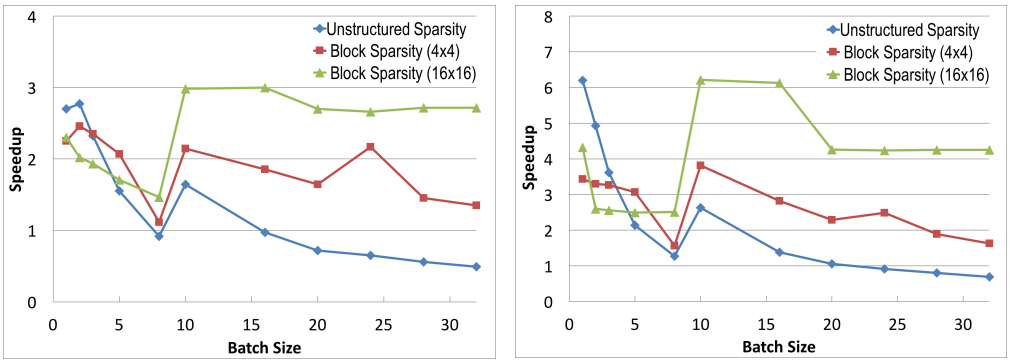
As shown in Table 6, the unstructured sparsity model (Han et al., 2015) achieves the best accuracy but requires much longer training time and does not improve the compute time relative to the dense model. For a small loss in accuracy, block-sparse models can significantly reduce both compute and memory requirements. For example, layers with 16x16 block sparsity reduce memory consumption by $11\times$and speedup compute by $3\times$with a 10% loss in accuracy. Additionally, Figure 2 shows that block-sparse matrices achieve higher speed-up than unstructured sparsity for large batch sizes for RNN and GRU layers. The speed-up is achieved due to reducing irregular memory accesses and improving load balance. Future work involves efficient implementation of block-sparse kernels to take advantage of array-data-paths in modern processors.

Table 6: Accuracy, speed-up and memory reduction for sparse layers. Block-sparse layers achieve higher speedup and memory reduction with some loss in accuracy.

MODEL	LAYER SIZE	GEMM SPEEDUP	MEMORY SAVINGS	CER (% LOSS)	ALGORITHM
RNN Sparse	1760	1.0×	5.0×	**15.41 (-0.3%)**	Han et al. (2015)
RNN Block-Sparse	2560	1.1×	6.2×	15.89 (-3.4%)	BP (4x4)
RNN Block-Sparse	1760	1.5×	7.1×	16.67 (-8.5%)	BP (32x32)
RNN Block-Sparse	1760	**3.0**×	11×	17.10 (-11%)	BP (16x16)
RNN Block-Sparse	1760	1.9×	**17**×	17.93 (-17%)	BP (4x4)

6 Impact of sparsity on accuracy

Using our baseline RNN model, we run many Weight Pruning (WP) (using (Narang et al., 2017)) and block pruning experiments, varying hyper-parameters to produce a spectrum of results ranging from 70% to 97% sparsity. For these experiments, the models are trained for 20 epochs and the accuracy is reported on the validation set. As shown in Figure 3, models pruned using WP with



(a) Speed-up for RNN 1760 layer matrix multiply

(b) Speed-up for GRU 2560 layer matrix multiply

Figure 2: Speed-up for sparse matrix multiply over GEMM. RNN matrix sizes are (1760,1760) with 90% sparsity and (1760, batch_size). GRU matrix sizes are (7680,2560) with 95% sparsity and (2560, batch_size). Block-sparse matrices achieve consistently good speedup across batch-sizes

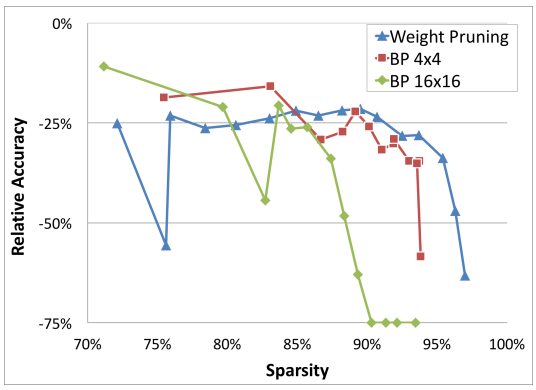


Figure 3: Relative accuracy for different block sizes (4x4, 16x16) and WP for varying sparsity on the RNN 1760 model. Any models with relative accuracy worse than -75% are capped at 75%.

sparsity less than 95% have relative accuracy ranging from -20% to -27%. Increasing the sparsity for the model beyond 95% results in 30% or more accuracy loss. This "accuracy cliff" is earlier for models pruned with block sparsity. For block size 4×4, models with sparsity greater 90% yield a relative accuracy loss of 30% or higher. Similarly, for blocks of 16×16, models with sparsity greater than 86% have 30% or more accuracy loss. A similar trend is observed for block size 32×32. This indicates that there is a trade-off between sparsity and block-size for a given accuracy. Larger blocks reach the "accuracy cliff" sooner.

7 Conclusion and Future Work

We have demonstrated that using block pruning and group lasso combined with pruning during training can build block-sparse RNNs that are about as accurate as the dense baseline models. The block-sparse models have significantly fewer parameters than the dense baselines reducing memory requirements. Block-sparse models can take advantage of the underlying hardware efficiently.

We would like to investigate if pruning can be performed even earlier in the training, thereby allowing us to train sparse models. Training sparse models would allow us to reap the benefits of sparsity during training resulting in lesser compute and memory demands. Further work remains to implement efficient block-sparse matrix multiplies for array-data-paths in modern processors that would provide increased speed-up during deployment.

REFERENCES

Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, JingDong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proceedings of The 33rd International Conference on Machine Learning*, pp. 173–182, 2016.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. *CoRR*, abs/1306.0543, 2013. URL http://arxiv.org/abs/1306.0543.

Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. *CoRR*, abs/1404.0736, 2014. URL http://arxiv.org/abs/1404.0736.

Qinwei Fan, Wei Wu, and Jacek M Zurada. Convergence of batch gradient learning with smoothing regularization and adaptive momentum for neural networks. *SpringerPlus*, 5(1):295, 2016.

Julian Faraone, Nicholas Fraser, Giulio Gamberdella, Michaela Blott, and Philip HW Leong. Compressing low precision deep neural networks using sparsity-induced regularization in ternary networks. *arXiv preprint arXiv:1709.06262*, 2017.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pp. 369–376. ACM, 2006.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746, 2015.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Stephen José Hanson and Lorien Pratt. Advances in neural information processing systems 1. chapter Comparing Biases for Minimal Network Construction with Back-propagation, pp. 177–185. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989. ISBN 1-558-60015-9. URL http://dl.acm.org/citation.cfm?id=89851.89872.

Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*, pp. 293–299. IEEE, 1993.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017. URL http://arxiv.org/abs/1704.04760.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016. URL http://arxiv.org/abs/1602.02410.

Seyoung Kim and Eric P Xing. Tree-guided group lasso for multi-task regression with structured sparsity. 2010.

Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 2, pp. 598–605, 1989.

Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 806–814, 2015.

Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William Dally. Exploring the regularity of sparse structure in convolutional neural networks. 05 2017.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. *ArXiv e-prints*, October 2017.

Sharan Narang and Gregory Diamos. Deepbench. https://svail.github.io/DeepBench-update/, 2017. Accessed: 2017-06-28.

Sharan Narang, Gregory Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

NVIDIA. NVIDIA Tesla V100 GPU Architecture. Technical report, 2017.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*, pp. 525–542. Springer International Publishing, Cham, 2016. ISBN 978-3-319-46493-0. doi: 10.1007/978-3-319-46493-0_32. URL https://doi.org/10.1007/978-3-319-46493-0_32.

Simone Scardapane, Danilo Comminiello, Amir Hussain, and Aurelio Uncini. Group sparse regularization for deep neural networks. *Neurocomputing*, 241:81–89, 2017.

Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.

Wei Wen, Yuxiong He, Samyam Rajbhandari, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. *arXiv preprint arXiv:1709.05027*, 2017.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Dong Yu, Frank Seide, Gang Li, and Li Deng. Exploiting sparseness in deep neural networks for large vocabulary speech recognition. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4409–4412. IEEE, 2012.

Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560. ACM, 2017.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. 2006a.

Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67, 2006b.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014. URL http://arxiv.org/abs/1409.2329.

Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

A ℓ_1 AND $\ell_{1/2}$ REGULARIZATION

Prior to our work with group lasso regularization, we considered ℓ_1 and $\ell_{1/2}$ regularizers to induce sparsity in the network. These regularizers act on individual weights and could aid in inducing unstructured sparsity in the network. ℓ_1 regularization is defined as:

$$L = L_{\text{training}} + \lambda \sum_{i=1}^{k} |w_i|$$

where $|w_i|$ is the absolute value of a weight and k is the total number of weights. Note the gradient expression for each weight w_j:

$$\frac{\partial}{\partial w_j} \sum_{i=1}^{k} |w_i| = sgn(w_j)$$

As with the group lasso experiments described in 3.2, we explore ℓ_1 regularization with and without pruning. The weight pruning (WP) algorithm from Narang et al. (2017) is used along with regularization. The motivation is the same as group lasso block sparsity experiments: either to guide pruning or to produce sparsity directly.

We also explore $\ell_{1/2}$ regularization which is defined as:

$$L = L_{\text{training}} + \lambda \sum_{i=1}^{k} |w_i|^{1/2}$$

Fan et al. (2016) uses $\ell_{1/2}$ regularization to produce sparsity directly. The gradient for $\ell_{1/2}$ regularization is $\frac{1}{2}|w_j|^{-1/2}$. This term is smaller for weights with larger magnitude. Our expectation is that $\ell_{1/2}$ will drive unimportant weights towards zero while leaving large weights relatively unaffected, thus avoiding the accuracy loss associated with excessive regularization.

For our ℓ_1 and $\ell_{1/2}$ experiments, we use the Deep Speech 2 Bidirectional RNN baseline model described in Section 4. These models are trained for 25 epochs on our internal training dataset of 2000 hours. The results are reported on a independent test set consisting of 2.9 hours.

Table 7: ℓ_1 and $\ell_{1/2}$ results with the bidirectional RNN model with 1760 hidden units

MODEL	# PARAMS (in millions)	SPARSITY	CER	RELATIVE PERF	PRUNING ALGORITHM
RNN Dense	67	0.0%	15.36	0.0%	N/A
RNN Sparse	7.3	89.2%	**17.32**	-12.8%	Weight pruning
RNN Sparse	11.2	83.6%	24.8	-61.5%	ℓ_1
RNN Sparse	7.4	89.1%	**17.28**	-12.5%	ℓ_1 with pruning
RNN Sparse	6.6	90.3%	18.50	-20.4%	$\ell_{1/2}$ with pruning

Without pruning, ℓ_1 model results in significantly worse accuracy compared to the dense baseline. Combining ℓ_1 with weight pruning allows us to recover the loss in accuracy with similar sparsity. The $\ell_{1/2}$ with pruning model performs worse than the ℓ_1 with pruning model. Comparing the two regularizers, this result indicates that ℓ_1 is better at guiding pruning than $\ell_{1/2}$, more suitable as a regularizer, or both.

Similar to group lasso experiments, ℓ_1 regularization experiments require a significantly higher λ to achieve high sparsity without any pruning. We suspect that these regularizers would be more successful in inducing sparsity for models that overfit the training training dataset.

B PRUNING CHARACTERISTICS

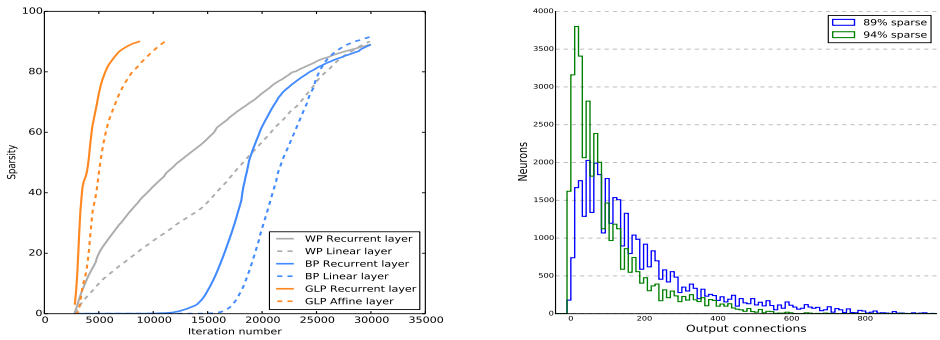
In this section, we discuss some pruning characteristics and how they relate to training and accuracy of the models.

B.1 PRUNING SCHEDULE

In Figure 4a, we plot the pruning schedule of a recurrent and linear layer of the bidirectional model trained with Block Pruning (BP) and Weight Pruning (WP) (Narang et al., 2017) and Group lasso with block pruning (GLP). For all three algorithms, pruning begins just after the first epoch at 2700 iterations. The BP and GLP models result in a sharper curve with more weights being set to zero in a short span of iterations. In these experiments, we use the *max* function to reduce the blocks to a single value which could be the cause of the sharpness in pruning. Also the GLP model reaches 90% sparsity just before 10,000 iterations which is significantly earlier than the BP model. GLP training encourages sparsity early on in the training run by pushing the blocks of weights towards zero.

B.2 OUTPUT CONNECTIONS

Figure 4b shows the histogram of the number of output connections for all the neurons in a network for two models with different sparsity pruned with BP. The 94% sparse model does significantly worse than the 89% sparse. For the model with 89% sparsity, only 180 neurons have all their output weights set to zero out of a total of 38270. This model produced good accuracy relative to the dense baseline. However, increasing the sparsity to 94% for the layer results in 1620 neurons having all zero output weights. Additionally, a lot more neurons have a smaller number of non-zero output weights.



(a) (b)

Figure 4: Figure 4a shows the pruning schedule for two layers in the network for WP, GLP and BP models. The GLP and BP models use block size of 4x4. Figure 4b plots the histogram of the number of output connections for all neurons in the network using block pruning with 4×4 blocks.

B.3 SPARSITY VS LAYERS

Figure 5 shows the sparsity of all the recurrent layers in the network using BP and WP. All recurrent layers have the same pruning hyper-parameters. Layer 1 is the first recurrent layer and layer 14 is the final recurrent layer before the CTC cost layer. For both block pruning and weight pruning, we see that the initial layers are pruned more aggressively compared to the final layers. Increasing sparsity in the layers closer to the output results in poor accuracy. Additionally, the variance in sparsity across the layers increases with the block size. This increasing variance makes it harder to increase the block size beyond 32×32 with the same pruning hyper-parameters for all recurrent layers.

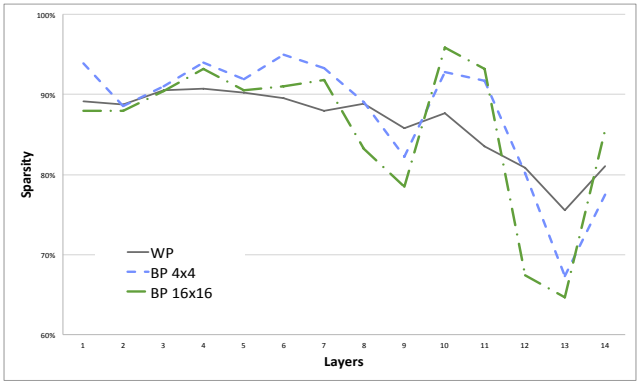


Figure 5: Sparsity of different recurrent layers in the network in the RNN model, pruned using BP and WP.