Pushing the limits of RNN Compression

Urmish Thakker, Igor Fedorov, Jesse Beu, Dibakar Gope, Chu Zhou
Ganesh Dasika[*]**,Matthew Mattina**
Arm ML Research Lab

Abstract

Recurrent Neural Networks (RNN) can be difficult to deploy on resource constrained devices due to their size. As a result, there is a need for compression techniques that can significantly compress RNNs without negatively impacting task accuracy. This paper introduces a method to compress RNNs for resource constrained environments using Kronecker product (KP). KPs can compress RNN layers by $16 - 38\times$ with minimal accuracy loss. We show that KP can beat the task accuracy achieved by other state-of-the-art compression techniques across 4 benchmarks spanning 3 different applications, while simultaneously improving inference run-time.

1 Introduction

Recurrent Neural Networks (RNNs) achieve state-of-the-art (SOTA) accuracy for many applications that use time-series data. As a result, RNNs can benefit important Internet-of-Things (IoT) applications like wake-word detection [19], human activity recognition [4, 13], and predictive maintenance. IoT applications typically run on highly constrained devices. Due to their energy, power, and cost constraints, IoT devices frequently use low-bandwidth memory technologies and smaller caches compared to desktop and server processors. For example, some IoT devices have 2KB of RAM and 32 KB of Flash Memory. The size of typical RNN layers can prohibit their deployment on IoT devices or reduce execution efficiency [15]. Thus, there is a need for a compression technique that can drastically compress RNN layers without sacrificing the task accuracy.

First, we study the efficacy of traditional compression techniques like pruning [21] and low-rank matrix factorization (LMF) [3, 8]. We set a compression target of $15\times$ or more and observe that neither pruning nor LMF can achieve the target compression without significant loss in accuracy. We then investigate why traditional techniques fail, focusing on their influence on the rank and condition number of the compressed RNN matrices. We observe that pruning and LMF tend to either decrease matrix rank or lead to ill-condition matrices and matrices with large singular values.

To remedy the drawbacks of existing compression methods, we propose to use Kronecker Products (KPs) to compress RNN layers. We refer to the resulting models as KPRNNs. We are able to show that our approach achieves SOTA compression on IoT-targeted benchmarks without sacrificing wall clock inference time and accuracy.

2 Related work

KPs have been used in the deep learning community in the past [7, 20]. For example, [20] use KPs to compress fully connected (FC) layers in AlexNet. We deviate from [20] by using KPs to compress RNNs and, instead of learning the decomposition for fixed RNN layers, we learn the KP factors directly. Additionally, [20] does not examine the impact of compression on inference run-time. In

---

[*]Currently at AMD Research

[7], KPs are used to stabilize RNN training through a unitary constraint. A detailed discussion of how the present work differs from [7] can be found in Section 3.

The research in neural network (NN) compression can be roughly categorized into 4 topics: pruning [21], structured matrix based techniques [1], quantization [2, 5] and tensor decomposition [8, 14]. Compression using structured matrices translates into inference speed-up, but only for matrices of size 2048×2048 and larger [16] on CPUs or when using specialized hardware [1]. As such, we restrict our comparisons to pruning and tensor decomposition.

3 Kronecker Product Recurrent Neural Networks

3.1 Background

Let $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m_1 \times n_1}$ and $C \in \mathbb{R}^{m_2 \times n_2}$. Then, the KP between B and C is given by

$$A = B \otimes C \tag{1}$$

$$A = \begin{bmatrix} b_{1,1} \circ C & b_{1,2} \circ C & ... & b_{1,n_1} \circ C \\ b_{2,1} \circ C & b_{2,2} \circ C & ... & b_{2,n_1} \circ C \\ . & . & . & . \\ b_{m_1,1} \circ C & b_{1,2} \circ C & ... & b_{m_1,n_1} \circ C \end{bmatrix}$$

where, $m = m_1 \times m_2$, $n = n_1 \times n_2$, and $\circ$ is the hadamard product. The variables B and C are referred to as the Kronecker factors of A. The number of such Kronecker factors can be 2 or more. If the number of factors is more than 2, we can use (1) recursively to calculate the resultant larger matrix. For example, in the following equation -

$$W = W1 \otimes W2 \otimes W3 \tag{2}$$

W can be evaluated by first evaluating $W2 \otimes W3$ to a partial result, say R, and then evaluating $W = W1 \otimes R$.

Expressing a large matrix A as a KP of two or more smaller Kronecker factors can lead to significant compression. For example, $A \in \mathbb{R}^{154 \times 164}$ can be decomposed into Kronecker factors $B \in \mathbb{R}^{11 \times 41}$ and $C \in \mathbb{R}^{14 \times 4}$. The result is a $50\times$ reduction in the number of parameters required to store A. Of course, compression can lead to accuracy degradation, which motivates the present work.

3.2 Prior work on using KP to stabilize RNN training flow

Jose et al. [7] used KP to stabilize the training of vanilla RNN. An RNN layer has two sets of weight matrices - input-hidden and hidden-hidden (also known as recurrent). Jose et al. [7] use Kronecker factors of size 2×2 to replace the hidden-hidden matrices of every RNN layer. Thus a traditional RNN cell, represented by:

$$h_t = f([W_x \quad W_h] * [x_t; h_{t-1}]) \tag{3}$$

is replaced by,

$$h_t = f([W_x \quad W_0 \otimes W_1... \otimes W_{F-1}] * [x_t; h_{t-1}]) \tag{4}$$

where W_x (input-hidden matrix) $\in \mathbb{R}^{m \times n}$, W_h (hidden-hidden or recurrent matrix) $\in \mathbb{R}^{m \times m}$, $W_i \in \mathbb{R}^{2 \times 2}$ for $i \in \{0, \cdots, F-1\}$, $x_t \in \mathbb{R}^{n \times 1}$, $h_t \in \mathbb{R}^{m \times 1}$, and $F = log_2(m) = log_2(n)$. Thus a 256×256 sized matrix is expressed as a KP of 8 matrices of size 2×2. For an RNN layer with input and hidden vectors of size 256, this can potentially lead to $\sim 2\times$ compression (as we only compress the W_h matrix). The aim of Jose et al. [7] was to stabilize RNN training to avoid vanishing and exploding gradients. They add a unitary constraint to these 2×2 matrices, stabilizing RNN training. However, in order to regain baseline accuracy, they needed to increase the size of the RNN layers significantly, leading to more parameters being accumulated in the W_x matrix in (4). Thus, while they achieve their objective of stabilizing vanilla RNN training, they achieve only minor compression ($< 2\times$). In this paper, we show how to use KP to compress both the input-hidden and hidden-hidden matrices of vanilla RNN, LSTM and GRU cells and achieve significant compression ($> 16\times$). We show how to choose the size and the number of Kronecker factor matrices to ensure high compression rates , minor impact on accuracy, and inference speed-up over baseline on an embedded CPU.

3.3 KPRNN Layer

Choosing the number of Kronecker factors: A matrix expressed as a KP of multiple Kronecker factors can lead to significant compression. However, deciding the number of factors is not obvious. We started by exploring the framework of [7]. We used 2×2 Kronecker factor matrices for hidden-hidden/recurrent matrices of LSTM layers of the key-word spotting network [19]. This resulted in an approximately $2\times$ reduction in the number of parameters. However, the accuracy dropped by 4% relative to the baseline. When we examined the 2×2 matrices, we observed that, during training, the values of some of the matrices hardly changed after initialization. This behavior may be explained by the fact that the gradient flowing back into the Kronecker factors vanishes as it gets multiplied with the chain of 2×2 matrices during back-propagation. In general, our observations indicated that as the number of Kronecker factors increased, training became harder, leading to significant accuracy loss when compared to baseline.

Algorithm 1 Implementation of matrix vector product, when matrix is expressed as a KP of two matrices

---

Input: Matrices B of dimension $m_1 \times n_1$, C of dimension $m_2 \times n_2$ and x of dimension $n \times 1$. $m = m_1 \times m_2, n = n_1 \times n_2$
Output: Matrix y of dimension $m \times 1$
 1: $X = reshape(x, n_2, n_1)$ {reshapes the x vector to a matrix of dimension $n_2 \times n_1$}
 2: $Bt = B.transpose()$
 3: $Y = C \times X \times Bt$
 4: $y = reshape(Y, m, 1)$ {reshapes the y vector to a matrix of dimension $m \times 1$}

Additionally, using a chain of 2×2 matrices leads to significant slow-down during inference on a CPU. For inference on IoT devices, it is safe to assume that the batch size will be one. When the batch size is one, the RNN cells compute matrix vector products during inference. To calculate the matrix-vector product, we need to multiply and expand all of the 2×2 to calculate the resultant larger matrix, before executing the matrix vector multiplication. Referring to (4), we need to multiply $W_0, .., W_F$ to create W_h before executing the operation $W_h * h_{t-1}$. The process of expanding the Kronecker factors to a larger matrix, followed by matrix-vector products, leads to a slower inference than the original uncompressed baseline. Thus, inference for RNNs represented using (3) is faster than the compressed RNN represented using (4). The same observation is applicable anytime the number of Kronecker factors is greater than 2. The slowdown with respect to baseline increases with the number of factors and can be anywhere between $2 - 4\times$.

However, if the number of Kronecker factors is restricted to two, we can avoid expanding the Kronecker factors into the larger matrix and achieve speed-up during inference. Algorithm 1 shows how to calculate the matrix vector product when the matrix is expressed as a KP of two Kronecker factors. The derivation of this algorithm can be found in [12].

Choosing the dimensions of Kronecker factors: A matrix can be expressed as a KP of two Kronecker factors of varying sizes. The compression factor is a function of the size of the Kronecker factors. For example, a 256×256 matrix can be expressed as a KP of 2×2 and 128×128 matrices,

---

Algorithm 2 Finding dimension of Kronecker Factors for a matrix of dimension $m \times n$

Input: $list1$ is the sorted list of prime factors of m, $list2$ is the sorted list of prime factors of n
Output: $listA$ - Dimension of the first Kronecker factor. $listB$ - Dimension of the second Kronecker factor
 1: function reduceList (inputList)
 2:     temp1 = inputList[0]
 3:     inputList.del(0) //Delete the element at position zero
 4:     inputList[0] = inputList[0]*temp1
 5:     inputList.sort('ascending')
 6:     return inputList
 7: list2, list1 = reduceList(list2), reduceList(list1).sort('descending')
 8: listA, listB = [list1[0],list2[0]], listB = [list1[1],list2[1]]

Table 1: Benchmarks evaluated in this paper. These benchmarks represent some of the key applications in the IoT domain - Image Classification, Key-word spotting, Bidirectional LSTM. We cover a wide variety of applications and RNN cell types.

	MNIST-LSTM	USPS-FastRNN	KWS-LSTM	HAR1-BiLSTM
Reference Paper		[9]	[19]	[4]
Cell Type	LSTM	FastRNN	LSTM	Bi-LSTM
Dataset	[11]	[6]	[18]	[13]

Table 2: Model accuracy and runtime for our benchmarks before and after compression. The baseline networks are compared to networks with RNN layers in the baseline compressed using KPs, magnitude pruning, LMF, or by scaling the network size (Small Baseline). Each compressed network has fewer RNN parameters than the baseline (size indicated). For each row, best results are indicated in bold. The KP-based networks are consistently the most accurate alternative while still having speed-up over the baseline.

Benchmark Name	Parameter measured		Compression Technique			
		Baseline	Small Baseline	Magnitude Pruning	LMF	KP
MNIST-LSTM	Model Size (KB)[1]	*44.73*	4.51	4.19	4.9	**4.05**
	Accuracy (%)	*99.40*	87.50	96.49	97.40	**98.44**
	Compression factor [2]	*1×*	10×	16.7×	13.08×	**17.6×**
	Runtime (ms)	*6.3*	0.7	**0.66**	1.8	4.6
HAR1-BiLSTM	Model Size (KB)[1]	*1462.3*	75.9	75.55	76.39	**74.90**
	Accuracy (%)	*91.90*	88.84	82.97	89.94	**91.14**
	Compression factor [2]	*1×*	19.8×	28.6×	28.1×	**29.7×**
	Runtime (ms)	*470*	**29.92**	98.2	64.12	157
KWS-LSTM	Model Size (KB)[1]	*243.4*	16.3	15.56	16.79	**15.30**
	Accuracy (%)	*92.5*	89.70	84.91	89.13	**91.2**
	Compression factor [2]	*1×*	15.8×	23.81×	21.2×	**24.47×**
	Runtime (ms)	*26.8*	**2.01**	5.9	4.1	17.5
USPS-FastRNN	Model Size (KB)[1]	*7.25*	1.98	1.92	2.04	**1.63**
	Accuracy (%)	*93.77*	91.23	88.52	89.56	**93.20**
	Compression factor [2]	*1×*	4.4×	8.94×	8×	**16×**
	Runtime (ms)	*1.17*	0.4	**0.375**	0.28	0.6

[1] Model size is calculated assuming 32-bit weights. Further opportunities exist to compress the network via quantization and compressing the fully connected softmax layer.
[2] We measure the amount of compression of the LSTM/FastRNN layer of the network

leading to a $4\times$ reduction in the number of parameters used to store the matrix. However, if we use Kronecker factors of size 32×8 and 8×32, we achieve a compression factor of 128. In this paper, we choose the dimensions of the factors to achieve maximum compression using Algorithm 2.

Compressing LSTMs, GRUs and RNNs using the KP: KPRNN cells are RNN, LSTM and GRU cells with all of the matrices compressed by replacing them with KPs of two smaller matrices. For example, the RNN cell depicted in (3) is replaced by:

$$KPRNN \ \ cell : h_t = f((W_1 \otimes W_2) * [x_t; h_{t-1}]) \tag{5}$$

where $x_t \in \mathbb{R}^{n \times 1}$, $h_t \in \mathbb{R}^{m \times 1}$, $W_1 \in \mathbb{R}^{m_1 \times n1}$, $W_2 \in \mathbb{R}^{m_2 \times n2}$, $m_1 \times m_2 = m$ and $n_1 \times n_2 = (m + n)$. LSTM, GRU and FastRNN cells are compressed in a similar fashion. Instead of starting with a trained network and decomposing its matrices into Kronecker factors, we replace the RNN/LSTM/GRU cells in a NN with its KP equivalent and train the entire model from the beginning.

4   Results

Other compression techniques evaluated: We compare networks compressed using KPRNN with three techniques - pruning, LMF and Small Baseline.

Training platform, infrastructure, and inference run-time measurement: We use Tensorflow 1.12 as the training platform and 4 Nvidia RTX 2080 GPUs to train our benchmarks. To measure

the inference run-time, we implement the baseline and the compressed cells in C++ using the Eigen library and run them on the Arm Cortex-A73 core of a Hikey 960 development board.

Dataset and benchmarks: We evaluate the impact of compression using the techniques discussed in Section 3 on a wide variety of benchmarks.Table 1 shows the benchmarks used in this work.

4.1   KPRNN networks

Table 2 shows the results of applying the KP compression technique across a wide variety of applications and RNN cells. As mentioned in Section 3, we target the point of maximum compression using two matrix factors.

4.2   Possible explanation for the accuracy difference between KPRNN, pruning, and LMF

In general, the poor accuracy of LMF can be attributed to significant reduction in the rank of the matrix (generally < 10). KPs, on the other hand, will create a full rank matrix if the Kronecker factors are fully ranked [10]. We observe that, Kronecker factors of all the compressed benchmarks are fully-ranked. A full-rank matrix can also lead to poor accuracy if it is ill-conditioned. However, the condition numbers of the matrices of the best-performing KP compressed networks discussed in this paper are in the range of 1.2 to 7.3. To prune a network to the same compression factor as KP, networks need to be pruned to 94% sparsity or above. Pruning FastRNN cells to the required compression factor leads to an ill-conditioned matrix. This may explain the poor accuracy of sparse FastRNN networks. However, for other pruned networks, the resultant sparse matrices have a condition number less than 20 and are fully-ranked. Thus, condition number does not explain the loss in accuracy for these benchmarks. To further understand the loss in accuracy of pruned LSTM networks, we looked at the singular values of the resultant sparse matrices in the KWS-LSTM network. Let $y = Ax$. The largest singular value of A upper-bounds $\|y\|_2$, i.e. the amplification applied by A. Thus, a matrix with larger singular value can lead to an output with larger norm [17]. Since RNNs execute a matrix-vector product followed by a non-linear sigmoid or tanh layer, the output will saturate if the value is large. The matrix in the LSTM layer of the best-performing pruned KWS-LSTM network has its largest singular value in the range of 48 to 52 while the baseline KWS-LSTM network learns a LSTM layer matrix with largest singular value of 19 and the Kronecker product compressed KWS-LSTM network learns LSTM layers with singular values less than 15. This might explain the especially poor results achieved after pruning this benchmark. Similar observations can be made for the pruned HAR1 network.

5   Conclusion

We show how to compress RNN Cells by $15\times$ to $38\times$ using Kronecker products. We call the cells compressed using Kronecker products as KPRNNs. KPRNNs can act as a drop in replacement for most RNN layers and provide the benefit of significant compression with marginal impact on accuracy. None of the other compression techniques (pruning, LMF) match the accuracy of the Kronecker compressed networks. We show that this compression technique works across 5 benchmarks that represent key applications in the IoT domain.

References

[1] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S. Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2857–2865, Dec 2015.

[2] Dibakar Gope, Ganesh Dasika, and Matthew Mattina. Ternary hybrid neural-tree networks for highly constrained iot applications. *CoRR*, abs/1903.01531, 2019.

[3] Artem M. Grachev, Dmitry I. Ignatov, and Andrey V. Savchenko. Neural networks compression for language modeling. In B. Uma Shankar, Kuntal Ghosh, Deba Prasad Mandal, Shubhra Sankar Ray, David Zhang, and Sankar K. Pal, editors, *Pattern Recognition and Machine Intelligence*, pages 351–357, Cham, 2017. Springer International Publishing.

[4] Nils Y Hammerla, Shane Halloran, and Thomas Ploetz. Deep, convolutional, and recurrent models for human activity recognition using wearables. *IJCAI 2016*, 2016.

[5] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061, 2016.

[6] J. J. Hull. A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554, May 1994.

[7] Cijo Jose, Moustapha Cissé, and François Fleuret. Kronecker recurrent units. *CoRR*, abs/1705.10142, 2017.

[8] Oleksii Kuchaiev and Boris Ginsburg. Factorization tricks for LSTM networks. *CoRR*, abs/1703.10722, 2017.

[9] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. *CoRR*, abs/1901.02358, 2019.

[10] Alan J Laub. *Matrix analysis for scientists and engineers*, volume 91. Siam, 2005.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[12] James Nagy. Introduction to kronecker products. `http://www.mathcs.emory.edu/~nagy/courses/fall10/515/KroneckerIntro.pdf`, 2010. Accessed: 2019-05-20.

[13] D. Roggen, A. Calatroni, M. Rossi, T. Holleczek, K. Förster, G. Tröster, P. Lukowicz, D. Bannach, G. Pirkl, A. Ferscha, J. Doppler, C. Holzmann, M. Kurz, G. Holl, R. Chavarriaga, H. Sagha, H. Bayati, M. Creatura, and J. d. R. Millàn. Collecting complex activity datasets in highly rich networked sensor environments. In *2010 Seventh International Conference on Networked Sensing Systems (INSS)*, pages 233–240, June 2010.

[14] Urmish Thakker, Jesse G. Beu, Dibakar Gope, Ganesh Dasika, and Matthew Mattina. Run-time efficient RNN compression for inference on edge devices. *CoRR*, abs/1906.04886, 2019.

[15] Urmish Thakker, Ganesh Dasika, Jesse G. Beu, and Matthew Mattina. Measuring scheduling efficiency of rnns for NLP applications. *CoRR*, abs/1904.03302, 2019.

[16] Anna Thomas, Albert Gu, Tri Dao, Atri Rudra, and Christopher Ré. Learning compressed transforms with low displacement rank. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9066–9078. Curran Associates, Inc., 2018.

[17] Lloyd Trefethen and David Bau. *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[18] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. *CoRR*, abs/1804.03209, 2018.

[19] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *CoRR*, abs/1711.07128, 2017.

[20] Shuchang Zhou and Jia-Nan Wu. Compression of fully-connected layer in neural network by kronecker product. *CoRR*, abs/1507.05775, 2015.

[21] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv e-prints*, page arXiv:1710.01878, October 2017.