Recurrent Neural Networks for Edge Intelligence: A Survey

VARSHA S. LALAPURA and J. AMUDHA, Amrita School of Engineering, Bengaluru, India
HARIRAM SELVAMURUGAN SATHEESH, ABB GISPL, Bengaluru, India

Recurrent Neural Networks are ubiquitous and pervasive in many artificial intelligence applications such as speech recognition, predictive healthcare, creative art, and so on. Although they provide accurate superior solutions, they pose a massive challenge *"training havoc."* Current expansion of IoT demands intelligent models to be deployed at the edge. This is precisely to handle increasing model sizes and complex network architectures. Design efforts to meet these for greater performance have had inverse effects on portability on edge devices with real-time constraints of memory, latency, and energy. This article provides a detailed insight into various compression techniques widely disseminated in the deep learning regime. They have become key in mapping powerful RNNs onto resource-constrained devices. While compression of RNNs is the main focus of the survey, it also highlights challenges encountered while training. The training procedure directly influences model performance and compression alongside. Recent advancements to overcome the training challenges with their strengths and drawbacks are discussed. In short, the survey covers the three-step process, namely, architecture selection, efficient training process, and suitable compression technique applicable to a resource-constrained environment. It is thus one of the comprehensive survey guides a developer can adapt for a time-series problem context and an RNN solution for the edge.

CCS Concepts: • **Computing methodologies** → *Artificial intelligence*; *Distributed artificial intelligence*; **Intelligent agents**

Additional Key Words and Phrases: Recurrent neural networks (RNNs), artificial intelligence (AI), edge intelligence (EI), training, compression, resource constrained modeling, sequence modeling, sparsity, low-rank

ACM Reference format:
Varsha S. Lalapura, J. Amudha, and Hariram Selvamurugan Satheesh. 2021. Recurrent Neural Networks for Edge Intelligence: A Survey. *ACM Comput. Surv.* 54, 4, Article 91 (May 2021), 38 pages.
https://doi.org/10.1145/3448974

1 INTRODUCTION

Recurrent Neural Networks (RNNs) are state-of-the-art solutions for modeling time-series sequences such as speech signals, language texts, video frames, body sensor information, weather data, and so on. While **Convolutional Neural Networks (CNNs)** are most suitable to capture spatial information, they lack the ability to capture the temporal dimension and hence powerful RNNs were proposed in due course of time. Various complex tasks are now performed by RNNs [3, 79, 131, 139, 154, 155] but they are extremely difficult to train [8, 20, 68, 82, 104].

Authors' addresses: V. S. Lalapura and J. Amudha, Amrita School of Engineering, Bengaluru, Karnataka, 560036, India; emails: {s_varshalalapura, j_amudha}@blr.amrita.edu; H. S. Satheesh, ABB GISPL, Bengaluru, India; email: hariramsatheesh@gmail.com.

0360-0300/2021/05-ART91 $15.00
https://doi.org/10.1145/3448974

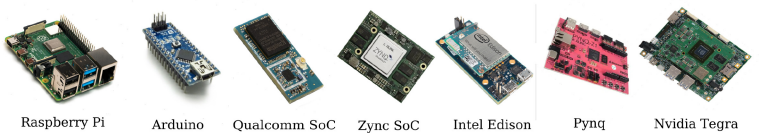

Fig. 1. A few candidate edge platforms for deep learning.

Another major challenge w.r.t. RNNs are that they are memory- and compute-intensive. Trained models generally have huge number of weight parameters that demands enough dedicated memory. Processing the input and hidden features at every timestep demands enough computing resources as well. Furthermore, the learning process involving such memory and compute needs is through **Back propagation through time (BPTT)** [33, 115, 137] algorithm. This algorithm incorporates learning w.r.t. time, hence RNNs are recurrent in nature. Learning is usually carried out on **GPU (Graphics Processing Unit)** systems that are a rich pool of **SIMD (Single Instruction Multiple Data)** compute cores but are limited in other processing resources, particularly memory and memory bandwidth.

1.1 Motivation and Inquisition

Compressing neural networks has obvious direct advantages of reduced computations and memory requirements compared to uncompressed full models. More importantly, if models successfully fit on low resource hardware, then data need not be communicated to the cloud (a remote server hosted on the Internet), fog (an intermediate remote server), or the near-mobile devices in the **Internet of Things (IoT)** per se. Such *"edge"*-based models are more fast, reliable, timely, secure, private from a user's perspective, and, most importantly, energy saving from a device perspective. In this context, we pose the following research questions:

Q1. *Are there simple yet efficient methods that bring down a given RNN model onto an off-the-shelf device with small die area, maybe, say, an Arduino?*
Q2. *How feasible is it to map complex RNNs onto small footprint devices without compromising for their performance?*
Q3. *Finally, how is AI transforming and where is it leading to in the supervised learning regime?*

1.2 Focus of the Survey and Organization

We traverse the literature to find answers for the feasibility of such application-to-device mapping in resource-constrained environments. From a **deep learning (DL)** foray, RNNs are worth investigating [77]. Also, we find that *training* and *compression* routines are the driving forces. Training process targets to meet desired performance levels. Compression targets model fitting onto the destined processing space. The two are generally inter-leaved and iterative (dotted rectangular box in Figure 2). This is because the objective of such a resource-constrained model design is achieving *both performance and compaction* without compromising one for the other. Strictly speaking, a certain compression scheme is said to be efficient if it is capable of simplifying a trained RNN model to fit the chosen target device without losing its model performance objective, say, accuracy. The scope of this survey is the progress of research in this direction.

There are a plethora of training strategies and compression techniques in the RNN literature. Likewise, there are a wide variety of hardware architectures and target devices that are potential candidates for an edge. Few of them are depicted in Figure 1. We find that training strategies such as gradient clipping [104] and inclusion of gating structures [16, 48] are some of the commonly implemented methods for efficient training. We discuss many of these training strategies in detail

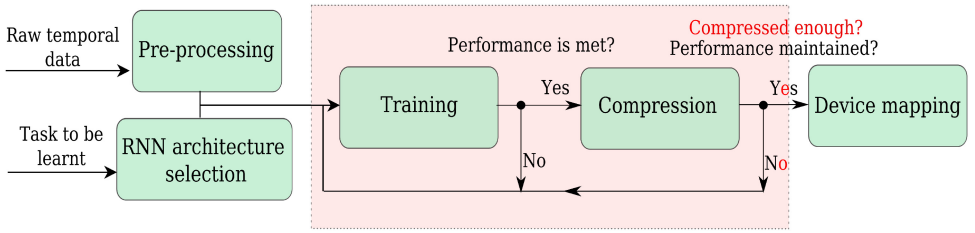

Fig. 2. A typical time-series supervised learning-based model-to-device mapping pipeline for a real-time application targeting a resource-constrained environment. The dotted rectangle represents resource-constrained neural modeling and that training and compression procedures are coupled and iterative.

in Section 4. For compression, we find techniques such as iterative pruning and quantization or iterative hard thresholding have been effectively applied to CNN [43] and **Deep Neural Networks (DNNs)** [57] architectures, respectively. Techniques like adaptive low-rank scheme are applicable to both RNNs and CNNs [14]. Interestingly, we find a few that are hardware-friendly. We discuss all these compression techniques among many others in Section 5. An important point to note here is that (to the best of our knowledge) no training methods in specific cater to edge devices alone. Memory and compute boundaries on the edge device limit direct application of GPU-based training procedures (see Table 12 for memory bounds of baseline RNN models). Training followed by compression is the typical procedure followed to limit the direct application. However, if we find a training method and a compression technique are tightly coupled, we mention them as practical caveats for implementation under each compression technique (5.1–5.10).

A typical workflow of an RNN model to edge mapping is as shown in Figure 2. The figure portrays the problem context end-to-end. This is true for any supervised model-to-device mapping study. The input (can be any image for a CNN modeling of a computer vision problem, for example) and the network selection (any CNN architecture for learning) would vary.

1.2.1 Problem Formulation. The RNN-based model-to-device pipeline (Figure 2) is the backbone of this survey. This can be viewed as a software-hardware co-design problem starting from the application to custom hardware or silicon bare metal [42]. But we focus on the aspects that will fit this context for any off-the-shelf device. Thus, it becomes a *neural modeling problem* constrained to the edge device. The objective of this survey is to provide a deeper end-to-end perspective of the problem context starting from scratch with the roadblocks involved. Throughout the article, we look for aspects and solutions that bear in mind RNN modeling for the *edge*. Another objective of the survey is to present the knowledge base in a handy, simplified, and usable manner. We find that the information on this topic is quite elaborate, intense, and spread out. Several aspects are already implemented and are open source. One could readily try many aspects from the survey for their application at hand. We organize the survey into four parts to throw light on every block of the pipeline (Figure 2) and show the significance of each block in terms of research advancements.

(1) *Basic premise:* This includes RNN history and evolution, basic training, and evaluation with implementation nuances. This is crucial for any level of research in this area (Section 2, Section 3).
(2) *Intermediate study:* This includes training challenges and how research has progressed to tackle them leading to novel ideas and structures. These directly affect the next step (Section 4).
(3) *Model-to-device mapping study:* This includes various RNN compression techniques for the edge. They are compared and analyzed from a given set of applications (Section 5 to Section 6).

(4) *Conclusion and Future scope:* This concludes the survey by finding research gaps and listing possible research directions (Section 7, Section 8).

1.3 Contributions through the Survey

(1) We identify a typical workflow for an RNN model mapping onto a resource-constrained edge.
(2) We investigate the heart of an RNN to device mapping problem, which is a combination of *efficient training* and a suitable *compression strategy* (dotted rectangle in Figure 2).
   (a) We identify key challenges in training RNNs and provide an elaborate investigation of the same. We believe that this will be an important reference section (Section 3, Section 4) for those who intend to carefully train RNNs independent of the pipeline workflow (Figure 2).
   (b) We present a detailed description and comparison of state-of-the-art compression techniques that apply to different RNN architectures and time-series/sequence applications. In the context of an application-to-device mapping (Figure 2) as a machine learning problem with scantily available processing room, we believe that this survey will enable and guide model developers to make prudent choices in terms of choosing a good network structure, a better training strategy, and a suitable compression scheme (with their dos and don'ts) for their application task and dataset.
(3) We believe to have marked important references and online open-source material in the edge context for implementation purposes (e.g., footnotes and tables). Solutions to training challenges (Tables 3, 4), model complexity analysis versus various compression choices (Section 6.7), TensorFlow-based open compression tools (Table 5), and concluding remarks based on the survey (Section 7) will be immediately useful for edge developers.
(4) We attempt to propose an alternative hypothesis/approach for such a model mapping challenge. Motivation for this comes from a quest for model mapping approaches on *"a significantly constrained die area"* with *"simplified training"* (Section 8, Figure 9).

2 RNN EVOLUTION

Neural Networks with recurrent connections to learn time correlations were proposed during the 1980s. There are a variety of recurrent neural network structures that were being developed there on. Hopfield networks [60], Jordon [61] nets, Ellman [28] nets, Timing and counting nets [32], Echo State networks [53], Gated structures like LSTM [48], GRUs [16], and feedback networks like Reference [19] are a few types of RNNs (see Table 2 for more). Apart from applications such as health care diagnosis [78], weather predictions [145], and computer-human interactions [29, 71]), these powerful architectures can be applicable in *agriculture*, *environment*, *sustainability*, and so on, for greener AI. We believe progress in this direction is the need of the hour.

$$h_t = f1\,[U_{hh}h_{t-1} + W_{hx}x_t + b_h], \tag{1}$$

$$y_t = f2\,[W_{yh}h_t + b_y]. \tag{2}$$

Temporal and recurrent behavior of RNNs are expressed mathematically using Equation (1) and Equation (2). Current time input features x_t and previous time hidden features h_{t-1} are learned by corresponding weight matrices W_{hx} and U_{hh} with *f1*, *f2* nonlinear function mapping. Learning is through a process called *training*. The training procedure maps the temporal complex relationships that exist between the input features x_t and the output label y_t associated with weight matrix W_{yh}. Once trained and fine-tuned for optimal weights, the learned *model* is ported to edge device for *inference*. Porting these RNNs on edge devices is challenging, since RNNs are memory- and compute-intensive. To illustrate, in Equation (1), if an input sample vector(x_t) is a 153 long feature

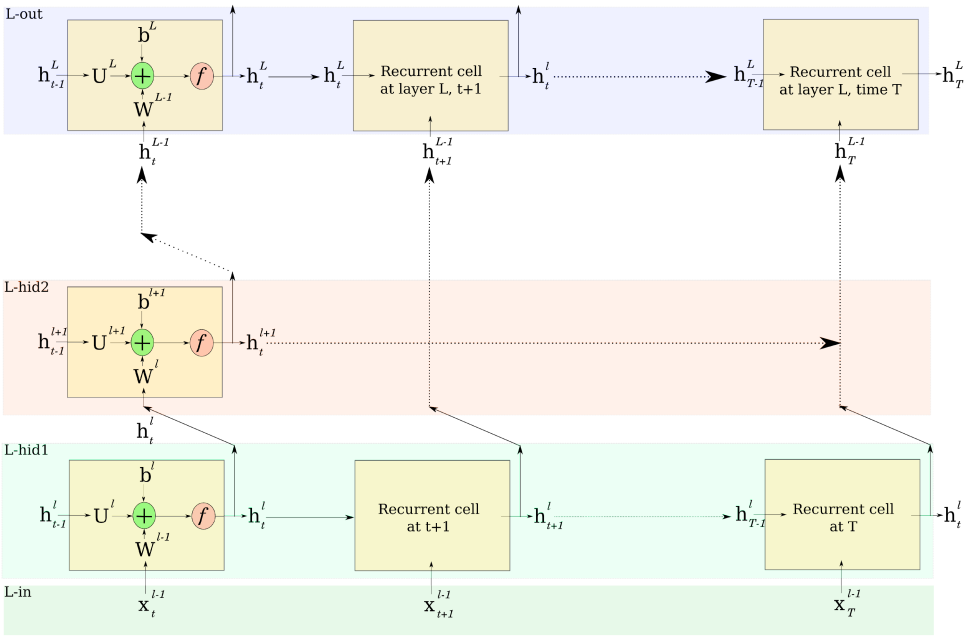

Fig. 3. A schematic of a generic vanilla RNN architecture depicting time unrolling characteristics (horizontal axis) and layered network topology (vertical axis). The first layer (L-in) represents input layer. The intermediate layers (L-hid) represent hidden layers(mostly 2–5 and not beyond) and the last layer (L-out) is output layer.

vector of a speech recognition engine [42] and the hidden state vector (h_t) size is 1,024, then the number of elements in the input-to-hidden weight matrix (W_{hx}) and recurrent or hidden-to-hidden weight matrix (U_{hh}) are 156,672 and 1,048,576, respectively. The number of **MAC (Multiply and Accumulate)** operations involved are proportional (ignoring the bias term (b_h)).

All of the history and evolution of RNNs have been precisely to tackle the most important challenge, i.e., training them (see Section 4 for details). Structurally, RNNs have varied w.r.t. the core cell component and the type of connectivity pattern. There are also a few nets that derive or adapt Convnet like structures in them [10]. They are beneficial in image applications, say, action recognition [101]. There are bi-directional structures as well for better translation machines [139].

Concerning the RNN network topology, there can be many variants, such as one-input to one-input, one-input to many-output, many-input to one-output, and many-input to many-output sequence structures, based on the task for which the model is designed for [65]. For a given task, there can be T_x input timesteps, T_y output steps, L layers, n_h hidden units, n_x input features, n_y output features. Correspondingly, recurrent matrix U_{hh} is of shape $[n_h,n_h]$ and non-recurrent matrix W_{hx} is of shape $[n_h,n_x]$. In general, for a layer l at time t (Figure 3), the hidden state description is given by Equation (3), where f can be $sigmoid(\sigma)$, $tanh$ non-linearity. The key feature is that the same set of weight parameters are updated across every timestamp, through the BPTT [137] algorithm. However, if a layered structure(L-hid1, L-hid2, and so on in Figure 3) is chosen, weight matrices implying memory requirements would multiply accordingly:

$$h_t^l = f[U_{hh}^l h_{t-1}^l + W_{hx}^{l-1} h_t^{l-1} + b^l]. \tag{3}$$

For a core RNN cell, an LSTM RNN is an important and popular variant. Reference [39] gives an interesting wide-angle description of LSTMs including its history and evolution. For this gated structure, the total number of weight parameters will be as in Equation (4):

$$4 \times n_h \times n_h \; + \; 4 \times n_x \times n_h \; + \; n_h \times n_y \; + \; 3 \times n_h. \tag{4}$$

However, one cannot disregard other elegant cells and structures like the GRUs with fewer params [18, 24], or Unitary [4, 59] or Orthogonal RNNs [87] that have evolved to address the training issues (we discuss later in Section 4). Perhaps, GRUs are preferred in many contexts [63] as a potential replacement for LSTMs, since they require less compute and memory to process. Reference [20] provides evidence that the observed superiority of gated models over vanilla RNN models is almost exclusively driven by *trainability*. Nonetheless, we still find that structures apart from LSTMs are yet to become fundamental backbones of sequence modeling applications especially in a constrained hardware setup. In brief, the type of cell, connecting pattern that voices the architecture, and the type of activation nonlinearities among other characterizations are widely different for different tasks. Below are a few critical challenges in modeling RNNs.

(1) It is seldom trivial nor generic to set RNN configurations for a given task that best captures the input dynamics and the output class label. In the deep learning jargon, this is called an *architecture selection* and *hyperparameter tuning* search. There have been advanced efforts to automate this step for a better neural network architecture selection [12, 35].
(2) Furthermore, having an RNN model trained well enough to result in low generalization/test error needs paramount effort, skill, time, and intuition (see Section 3.2 for generalization characteristics). If neural network handcrafting and training is easy, there cannot be research toward automation. To draw a parallel, there now exists a reliable **High-Level Synthesis (HLS)** version of a hardware design versus its hand-coded implementation.
(3) And items (1) and (2) influence each other significantly. This is true for any deep learning model and training workflow.

3 RNN TRAINING AND EVALUATION

Training RNNs is through two phases—forward pass or forward propagation of inputs through the network and backward pass or backpropagation *in time* of the error incurred between the actual and predicted class. Recent advances in machine learning have sophisticated tools to perform the forward and backward passes in an optimized manner. These are C/C++-based frameworks with Python wrappers around them. Specific to LSTMs, Reference [11] provides a recent study of LSTM benchmarking on different frameworks such as PyTorch, TensorFlow, Lasagne, and Keras. It gives a comparative study of LSTMs on these frameworks. Complementary to these tools, Edward [129] and Gen [22] are a few recent interesting advancements that use probabilistic languages for modeling. However, lack of consensus and diverse preferences [96] have resulted in confusion and difficulty to make a particular choice of tool/framework in the deep learning community.

3.1 RNN Training

Here, we cover aspects of RNN model selection, training, and testing if one prefers to design an RNN for a new application and work the problem from scratch.

(1) *Data and Task definition:* Temporally correlated data and a concrete task definition is first and foremost. Based on this, an architecture choice is made. For example, to generate the remainder of a piano lesson as musical sound, a mapping from a many-input to many-output architecture can be chosen [65]. The task definition will impact model complexity.

(2) *Dataset and pre-processing:* Well-annotated/labeled data is the next requirement. For example, the sample points of the weather (x_t) and say its label is "cloudy" (y). There needs to be enough of such labeled data pairs [$x_t : y$]. The input data points are collected over a while and binned into T timestamps $t1, t2, \ldots T$. This collection of rows(samples) and columns(data points over T timesteps plus label) are bifurcated as training, hold-out set, and test set. Training set is used for temporal pattern learning, hold-out set for tuning, and test set to measure the model's performance on a new unseen sample. The ratio choice between training, hold-out, and test sets affects the model's performance.
Pre-processing – This step is crucial for the model to learn better and show up in performance. This can be through providing indications of observation window between data points (say, pay attention here) or even weigh the recent data with a flag to indicate its relevance in time. It can also be shuffling the examples to prevent biases toward a certain set of examples. Handling missing data and variable-length input sequence [78], data normalization, and scaling are again critical [76]. Normalization means recalculating params (parameters) w.r.t. their mean and variance. Scaling means to choose a range for those params. In Reference [56], the speaker emphasizes the importance of understanding the data and organizing them before training.

(3) *RNN architecture selection:* A plausible list of variables in the RNN architecture selection are:
   i. The number of input data points per sample, the number of hidden units, and number of timesteps.
   ii. The core cell: This can be a vanilla-RNN/GRU/LSTM/any other from Tables 2, 3.
   iii. The number of layers: More the layers, larger the hypotheses space of mapping functions but also larger the memory requirement and longer training convergence. This can be overcome by truncating the BPTT [137] algorithm.
   iv. The activation function*: This can be a tanh/sigmoid/ReLu/modReLu/softmax function definition for input, hidden, and output neurons.

(4) *Training:* A plausible list of hyperparameter in the training context are:
   i. The cost function definition: This can be a cross-entropy loss, **CTC (Connectionist Temporal Classification)** loss [37], and so on, tightly coupled to the output function unit*.
   ii. The optimizer: The gradient descent optimization algorithm is a first-order solution most commonly used to reach the minimization point. There are higher-order solutions as well*.
   iii. Learning rate: This number ranging between 0 and 1 determines to what extent newly acquired information overrides old information. This is a very important hyperparameter that is correlated to the dataset [39]. The choice of the learning rate impacts training time and even convergence. The learning rate can be tuned first using a fairly small network.
   iv. The number of epochs: This number indicated the number of times the entire training dataset is shown to an RNN model. This is important, because the optimization cannot be completed within one epoch (entire dataset shown to the model once) and needs to be repeated many times to learn the representations and reach the minimization criteria.
   v. Batch size: Total number of samples in a batch of training usually dictated by GPU memory.
   vi. The number of iterations: This is total number of mini-batches to complete one epoch.
   vii. The initialization strategy*: This significantly impacts training, precisely the time for the network to converge. A good initialization leads to efficient training and lesser training time. References [34, 76] give the details for * marked.

Table 1. A Verbatim Example Analysis of a Neural Network's Behavior after a Training Process Observing the Training Set Error and Hold-out Set Error [95]

Error type	Case1	Case2	Case3	Case4
Training set error	1%	15%	15%	0.5%
Hold-out set error	11%	16%	30%	1%
Implication	↑ variance	↑bias	↑bias,↑variance	↓bias,↓variance

This is a fundamental training analysis procedure.

(5) *Model Tuning:* This step is fundamental to find the best combination of hyper-parameters to meet the desired performance objective and then fit onto the target device. Ideally, when compression is applied as the next step of tuning, there needs a lesser tradeoff between performance and model fitting. There are exclusive tools to track the DL experimentation and gain tuning of hyperparameters. But all need a certain level of expertise to use them. If compression is accompanied, then there are automated tools that do that [51] but less explored for the RNN context and sequence modeling applications.
(6) *Testing:* This is designed to evaluate the model's performance and behavior for unseen case.

3.2 RNN Evaluation

In Reference [88], the author stresses the importance and necessity of defining methods to evaluate new techniques so scientific progress would be measurable and accurate. We looked for a well-defined *goodness of fit* criteria to evaluate a model's generalization ability. We found the following content for conducting model evaluation:

3.2.1 Evaluation Characteristics. For a neural network model, characteristics of evaluation are under-fitting, generalization, and over-fitting. This is illustrated in Table 1 [95]. If a network performs well on the training set but relatively under-performs on the hold-out set, the behavior is implied as over-fitting and is categorized as a high-variance problem (Col. 2 of Table 1). Now, if the network is returning a high training error and also a high hold-out set error, then the behavior is implied as under-fitting and is categorized as a high-bias problem (Col. 1 of Table 1). A good training routine ideally should reflect the last column and a poor training routine or a heavily erroneous one will lead to Col. 3 in the table. Reference [127] aids this understanding of over-fitting and under-fitting with a DNN example in Keras. Methods to deal with bias and variance issues are discussed in Reference [93].

Another characterization of evaluation is determining the model's capacity. The model's capacity can be viewed as its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set (high bias problem). Models with high capacity can over-fit by memorizing properties of the training set that do not serve them well on the test set (high variance problem) [34]. In the RNN context, capacity is closely related to the expressiveness of the network towards its learning task. Reference [20] describes two types of capacity bottlenecks that various RNN architectures might be expected to suffer from: one, parameter efficiency related to learning the task, and two, the ability to remember input history. They conduct two separate experiments to understand these aspects of expressiveness.

3.2.2 Evaluation Metrics. There are a few evaluation metrics to measure the performance of a trained RNN model. They are Accuracy, Confidence Score, Precision, Recall, F1 measure, **Receiver Operating Characteristics (ROC), Area Under the Curve (AUC)**. To understand these metrics, it is necessary to know **True positive (TP), True Negative (TN), False Positive (FP)**, and **False Negative (FN)** categorizations. Further, *training time* and *inference time* are important metrics.

Train time is time taken for forward and backward passes of the training algorithm to reach the minimum criteria. If compression is tightly coupled to the training procedure, then this figure should relatively increase. Inference time is time taken for the forward pass only. If compression is accompanied, then this figure should significantly reduce.

For single number evaluation metric, Reference [94] provides a good illustration. Also, the author points out useful strategies of error analysis for the model w.r.t. the training/hold-out/test set as a means to improve performance. For a multi-class RNN context, an evaluation methodology is briefly discussed in Reference [78] under its experiments section.

3.2.3 Application Baselines. For performance evaluation and fair comparison, strong baseline models are essential. With this notion, we attempt to find the best model on a given dataset for a specific task. We find that this a herculean task as the search space is too vast. For example, The DeepSpeech2 [3] model comprising 2–3 CNN layers, 6–7 BiRNN (GRU or LSTM) layers is the baseline in References [91, 92, 152]. This, however, is "a" state-of-the-art model on the *Wall Street Journal* dataset. For speech recognition task alone, there exists a large number of models tuned to various dataset and pre-processing strategies. State-of-the-art results for speech recognition on various datasets and also other applications can be traced here.[1]

Due to the transition period from evolving ideas to efficient deep learning implementations, there are many faulty comparisons and conclusions drawn and have motivated many researchers to re-evaluate the claims [83]. We also find that the exact details (precisely the hyperparameters of the model and training configuration) for reproducing results were not clearly provided by model developers in the early phase of research, which might be a fundamental reason for propagating humane errors. Hence, strong baselines for specific applications need clear reproducing procedures with the associated datasets, pre-processing steps, network, training and hardware configurations, deep learning frameworks with exact version matches. All these, when encompassed, will lead to accurate reporting of results and benchmarking. Some implementations provide their pre-trained models that can be considered as baselines.

4 RNN TRAINING CHALLENGES

The most important challenge that RNNs pose is how to train them, and we discuss them in this section. Intuitively, training RNNs for simple tasks should address generalization and their ability to capture long-term dependencies. Training them for complex tasks should address expressiveness as well (see Section 3.2). There are indeed four major challenges that need to be tackled while training them. They are (1) Vanishing gradients problem, (2) Exploding gradients problem, (3) Handling long-range dependencies, (4) Model-fitting and generalization issues. We simply explain these four training related issues in Section 4.1 and present how research has made strides to understand and answer them in Section 4.2. We try to cover as many answers as possible. Further, we summarize the solutions (Table 3) and present a short table (Table 4) for quick reference and adaptation.

4.1 Challenges

The first two challenges (notated as $C1$ and $C2$) are influenced by the training aspect of the problem. Backpropagation through time is the standard way to train a recurrent neural network [47]. During backpropagation, gradients are expected to be *large and predictable enough* to serve as a good guide for the learning algorithm [34]. But the gradients either grow exponentially upward or they travel to zero very quickly. This is $C1$ and $C2$ [8, 68, 103, 104].

[1] https://github.com/syhw/wer_are_we.

The next 2 challenges $C3$ and $C4$ are influenced by data and training aspects of the problem. Time intricacies in the dataset could be highly correlated or span large intervals. The training procedure and the chosen RNN architecture should be able to capture whatever are the time relations. This is commonly known as handling the long-range dependency challenge ($C3$). Sufficiently enough weights and recurrent connections are required to store task information and data dependency. Little less will under-fit or be biased to some data samples, little more will over-fit. Further, training convergence should be fast and lead to least generalization errors. Training procedure should balance all of these in a fine manner. This is $C4$.

4.2 Proposed Solutions

The above-mentioned challenges have been pinned and addressed in length in literature ever since they have been identified. A set of solutions have stemmed from structural modifications on the vanilla RNN. Solutions have also been motivated by the kind of task the RNN is expected to learn. Another set of solutions have been driven by plainly observing the parameters of the RNN and then dealing with the training process efficiently.

4.2.1 Structure and Application-driven Solutions. From structure-driven thinking, research has provided the most elegant and powerful RNNs in the DL era. Echo State Networks are a special class of RNNs that handle $C1$ and $C2$ and modify only the weights of output units to achieve the learning task [52]. To reduce the effect of vanishing gradients, shorter paths between timestamps, either via connections with longer time delays or inertia (slow-changing units) in some of the hidden units are explored [7]. These leaky integrator units that behave as residual connections [55] are a key factor for the recent miniature RNN structure in Reference [72]. The residual modification is precisely to address the vanishing and exploding gradients problem. Long short term memory in ESNs [54] is the next advancement for handling $C3$ as well. Another type of structure is the **Temporal Kernel RNN (TKRNN)** [124] that are connected through time. The connections are such that the gradient flow through fewer nonlinearities, oriented well in space and time.[2]

The LSTM structures [48] are another power-house that handle $C1$ and $C3$. The three gated structures along with a memory cell introduced to the vanilla cell takes care of these issues. Authors in Reference [39] conclude *forget gate* and *output activation function* to be its most critical components after 5,400 experimental runs with LSTMs. LSTMs are ubiquitous in edge modeling [14, 108] (see tables in Section 6). However, few authors readdress its strength compared to other structures. To handle $C1$ and $C3$, Reference [89] proposed context memory for simple RNN structures and named them structurally constrained recurrent networks. From task-driven thinking, the answers are elegant and mighty as well. For machine translation problems in language texts, authors in Reference [16] came up with two structures within a single network. An encoder-decoder setup that was individually simpler was proposed to handle $C4$. This encoding and decoding network and the simplified RNN inside became a popular choice for many applications thereon [3, 139]. The following popular GRU structure [18] handled long and short term behavior through their update and reset gates. The choice between LSTM and GRU is still debatable.

For the phoneme classification problem in speech data, Reference [119] proposed training the RNN with connected hidden layers in opposite directions to the same output to deal $C3$. The two directions include one from the positive time direction (forward hidden states) and the other from the negative time direction (backward hidden states) to enable learning from the dynamics of the entire sequence. This approach is counter to learning only from past history. This again is

[2] www.cs.utoronto.ca/~ilya/code/TKRNN.tar.

Table 2. Timeline of Structure and Application-driven RNNs

1982	Hopfield Net [134]
1986	Jorden Net [61]
1990	Elman Net [28]
1995	Leaky Integrator Neuron(LIN) [7, 27]
1997	Long Short Term Memory [48]
2000	Time and Count Net [32]
2001	Echo State Net [52]
2007	Echo State Net with LIN [53, 55]
2010	Temporal Kernel RNN [124]
2012	Memory in ESN [54], Context Dependent RNN [90]
2014	Encoder Decoder RNN [16], Gated Recurrent Unit [18]
2015	Gated Feedback Net [19], Attention-based [140]
2017	Transformer net [130]

commonly found topology in speech recognition and machine translation applications. Attention and transformer additions are recent advancements to the LSTM, GRU, and BiRNN to improve performance or to deal $C4$ intelligently [130, 134, 140]. There may be many other structures and topology, but we believe to have covered the significant ones. As seen in the timeline Table 2, we now have a plethora of structures, additions, and variants of the vanilla RNN.

4.2.2 Training Driven Solutions. While training, observing the network parameters is a natural thing to do. One can observe if the params are of a certain distribution after some training or could be if they do not belong to any certain distribution at all. Further, what initialization of the network will yield faster convergence is another angle. These are non-trivial, because the weight numbers are too large and complexity is tied to the data and network structure intricately.

Finding bifurcation points [26] and limiting the params to be inside them is one way to deal $C2$. A small change in parameters can result in a drastic change in the behavior of the state, which causes the bifurcation. This method of observing the bifurcation and preventing gradients to cross them has been implemented on GRU-based language and music modeling experiments [64]. The method also addresses $C4$ reporting performance improvement. The method is robust to random fluctuations in inputs due to the choice of GRU for the RNN. A caveat, network initialization has a significant impact on this method. Last, the authors compare the method to a hard constraining counterpart called gradient clipping, a popular technique to handle $C2$ [104]. However, the clipping method introduces an additional threshold hyperparameter for training.

During training, specifying the initial state of all the hidden units of the RNN can be through some random initialization. Another way to do this is to learn the initialization. This idea is elaborated in Reference [95]. The speaker stresses on imposing initialization on the same subset of the weights over timestamps and specifies the final states of the hidden units closer to the output. This is also called the teacher forcing method [104]. This helps learning to stabilize and not wobble. We observe that almost all implementations mention their initialization criteria for the network and mention its significance for efficient training.

It is possible to train RNNs using non-linearities other than tanh and sigmoid functions. Reference [75] explores this by finding the identity initialization with biases set to zero as the right scheme in conjunction with the ReLu function. Incorporating ReLu makes it easier to train avoiding $C1$ and $C2$ issues. Initialization is again prime for authors in Reference [125] along with optimization through **stochastic gradient descent (SGD)** algorithm [9] and momentum [109]. Momentum helps accelerate SGD in the relevant direction and dampens oscillations. A detailed overview of stochastic gradient descent variants is presented in Reference [114]. Mini-batch gradient descent is said to be the most popular choice. In Reference [117], orthogonal initialization is explored and proved that it helps learning to converge faster.

To deal with $C3$, Reference [82] utilizes a second-order Hessian-free optimization with a linear conjugate gradient algorithm. It incorporates a structural damping strategy in which it makes use of a specific structure of the objective function. This is a detailed Ph.D. work dedicated to training RNNs efficiently from an atypical optimization foray. If one identifies an RNN as a network of a set of temporal transitions being captured and learned, then the vanishing and exploding gradients problem can be traced looking at the state transition matrices. The gradients vanish if the spectral norm of the transition matrix is less than one, and like-wise, gradients explode if the spectral norm of the transition matrix is greater than one. It is best to have the norm exactly equal to one. Spectral norm is the maximum singular value of a matrix. All the unitary and orthogonal RNNs have been proposed from this perspective [4, 46, 50, 58, 59, 62, 86, 128, 132, 138, 150] and all address $C3$ and $C4$ to a large extent. A common method to improving the performance of models after training or even compression can be through *regularization*. This is defined as any modification we make to a learning algorithm that is intended to reduce its generalization/test error but not its training error [34]. Methods like early stopping (a technique to terminate training before overfitting occurs) are also common to reach desired performance. To express regularization, an epsilon regularization term is generally added to the loss function definition. This imposes parameter values and gradients to be constrained in a given range. In L1 regularization, the cost added is proportional to the absolute value of the weights coefficients. In L2 regularization, the cost added is proportional to the square of the value of the weights coefficients. L2 regularization is also called weight decay. Vanishing gradient regularization as a norm (absolute value) preserving of the gradients during back prop is implemented in Reference [104]. In contrast, authors in Reference [70] stabilize by norm preserving of gradients during forward prop.

Regularization of language models is studied through a method of DropConnect [84, 133]. In this method, a randomly selected subset of weights within the network are set to zero. The method is applied only once before the forward and backward pass and thus has no much impact on training time. The method is also extended to other regularization variations such as **Activation Regularization (AR)** and **Temporal Activation Regularization (TAR)** [85]. AR penalizes activations that are significantly larger than zero and TAR penalizes the model from producing large changes in the hidden state. These methods are particularly coupled to certain compression techniques such as iterative hard thresholding (see Section 5.2). A caveat, the choice of the optimizer is crucial for a regularized model and can significantly impact the training process. The authors point to the SGD over adaptive for language models in conjunction with regularization.

Drop-out is another regularization technique applied to only non-recurrent matrices of the RNN [147]. By not using drop-out on the recurrent connections, the LSTM can benefit without sacrificing its valuable memorization ability. However, drop-out on recurrent connections and yet no loss in memory is explored in References [30, 120]. The key difference between drop-out and drop-connect is that a randomly selected subset of activations is set to zero within each layer in the former. The latter instead knocks off random weights as per its name. Both of these methods address $C4$. Further, variational RNNs incorporate drop-out on the same hidden units at each timestep as opposed to drop-out at different network units at each timestep [30]. Surprisingly, Reference [143] applies drop-out as a compression strategy. Their DeepIoT automation tool is based on this for neural compression.

Zoneout is a complementary regularization approach that forces some hidden units to maintain their previous values at each timestep [69]. This approach is useful to solve $C1$ and $C2$ through the norm stabilization of activations [70]. The method involves the activations to be stabilized by penalizing the squared distance between successive hidden states' norms. Batch normalization and layer normalization are another set of methods to train neural nets efficiently. In the context of RNNs, these were successfully applied to language modeling tasks [5, 21]. All of these methods help deal with $C4$. Many of these are incorporated in TF's TensorLayer DL library designed for simplicity, flexibility, and high performance [25]. Also, TensorLayer is awarded the 2017 Best Open Source Software by the ACM Multimedia Society.

In short, we have summarized most of the structures, approaches, and solutions to the four challenges in Table 3. We have listed online open-source codebases for a few of the methods in Table 4. Many of the techniques in Table 4 are mostly tested for LSTMs. Authors mention that many of their methods are very much applicable to any chosen RNN.

5 RNN MODEL COMPRESSION

It has been stark and distinct that neural networks are prone to redundancy and over-fitting. Compressing neural nets is extremely beneficial. It reduces processing both in terms of memory accesses and computations. Also, storage is smaller and easier. The direct implications of these are speed-up and reduced energy usages. These are highly desirable qualities from the edge perspective. A typical compression methodology is a two-step process. Step 1 is to reduce the redundancy in the net. Step 2 is to reduce the redundancy in the bits representing the net. The process of converting the number of bits from a large precision complex representation to a small precision simpler representation is called *quantization* in modern terms or *information latching* from the old school of thought [8]. Both the steps are crucial for model-to-device fitting translation. This two-step compression strategy is the crux of Reference [41], a dedicated Ph.D. that has been in the limelight and gaining momentum ever since then for its profound applicability in mobile and small footprint devices.

We classify the compression techniques into three categories, i.e., structure-driven methods, matrix-driven methods, and tensor-driven methods, as shown in Figure 4. We add the bit reduction scheme (quantization) and hashing to the structure-driven category. We present each compression scheme as a subsection. However, we discuss low-rank factorization and adaptive low-rank scheme together. Similarly, we discuss projection compression scheme along with localized group projection scheme and Toeplitz and block-circulant scheme together. We discuss tensor-train, tensor ring, and block-tensor decomposition under tensor decomposition methods. For each compression technique, we discuss (1) What is the compression technique? (2) How is the technique beneficial? (3) Are there specific parameters the technique is affected by? (4) How have they been addressed in literature?

Table 3. RNN Training Challenges and Existing Approaches and Solutions

Training Challenge	Proposed Approach/Solution	Ref.
Exploding Gradients	Finding bifurcation points and limiting the params within them	[26, 64]
	Gradient Clipping	[88, 104]
Vanishing gradients and long-term dependency	Gating operations with memory inside the RNN cell (LSTMs)	[48]
	Update and Reset gate structures (GRUs)	[16]
	Leaky Integrator Neurons	[7]
	Temporal Kernel RNN	[124]
Vanishing and exploding gradients and long term dependency	Long-short term memory in ESNs	[54]
	Residual connection in the basic RNN cell structure	[72]
	Hessian-free second order optimization	[82]
	ReLu non-linearity and identity matrix initialization	[75]
	Unitary matrices, parameterization in the complex domain	[4]
	Unitary matrices and optimizing along the Stiefel manifold and Cayley transformation	[138]
	Orthogonal matrices, identity initialization, unitary constraints	[46]
	Unitary matrices and parameterization using Lie algebra and Lie groups	[50]
	Unitary matrix, parameterization, and FFT approximations	[59]
	Orthogonal matrices, parameterization with householder reflections	[86]
	Hard or soft orthogonal constraints on weight matrices	[132]
	SVD parameterization	[150]
	Kronecker parameterization in complex domain, soft unitary constraints	[62, 128]
	uRNN(remembering) + gated RNN(forgetting)	[58]
Long-term memory problem	Stochastic gradient decent with momentum and proper initialization	[125]
	Orthogonal matrices	[117]
	Separate contextual memory	[89]
Vanishing gradients, over-fitting, generalization	Drop-out regularization on non-recurrent connections	[147]
	Drop-out regularization on recurrent connections	[120]
	Drop-the same network units at each timestep	[30])
	DropConnect	[84, 133]
	Activation regularization, temporal activation regularization	[85]
	Norm preserving of gradients during forward prop	[70]
	Norm preserving of gradients during back prop	[104]
	Several advanced optimization techniques	[7]

5.1 Iterative Pruning and Retraining

Network pruning or sparsification is a method of removing redundant nodes and connections [44]. The weight distribution in a given neural network closely resembles a Gaussian-like distribution

Table 4. Source Codebases for a Few Methods Proposed in Table 3 to Beat RNN Training Challenges [1, 40]

Method	Dealing issue	Model	Application	code	Framework	Ref.
Gradient Clipping(GC)	$C2$	Vanilla-RNN	Language Modeling Music Modeling	GC	Theano	[104]
Context Memory(CM)	$C1$, $C3$	Vanilla-RNN	Language Modeling	CM	Torch	[89]
L2 Regularization(R1)	$C1$	Vanilla-RNN	Language Modeling Music Modeling	R1	Theano	[104]
Dropout Regularization(R2)	$C4$	LSTM	Language Modeling Sentiment Analysis Name Entity Recognition	R2	Theano	[120]
Dropconnect(R3)	$C4$	LSTM	Language Modeling	R3	TensorFlow	[84]
Zoneout Regularization(R4)	$C1$, $C2$ $C4$	LSTM	permuted MNIST Classification Language Modeling	R4	Theano, TensorFlow	[69]
Batch Normalization(BN)	$C4$	LSTM	MNIST Classification Language Modeling	BN	Theano, Torch, Keras	[21]

[81]. Hence there are a large number of small weights (less significant) and few weights with larger values (salient ones). The shape of the distribution can be of varying peak based on the problem. The pruning method drops the insignificant ones in an iterative manner.

The idea of removing redundancy is extended in Reference [31], as removing those input units whose difference between two consecutive inputs is lesser than a certain threshold. Some sparsifying techniques are hardware-friendly, which result in block-sparse matrices. Reference [136] proposes **Intrinsic Sparse Structures (ISS)** with an optimization goal to remove as many ISS weight groups as possible without impacting accuracy. The pruning methods thus can be structured and hardware-friendly [91, 92], unstructured [43], gradual [153], or aggressive. Below are certain precautions while pruning:

(1) Finding the right threshold value is non-trivial. The threshold value is thus a hyperparameter. Reference [91] provides six different heuristics to find the threshold value. However, their pruning algorithm is said to be friendly in terms of finding the threshold.
(2) Pruning can be a performance overhead (second bar group/50% pruning in Figure 5) if pruning is relatively low. For example, storing an 80% dense matrix can be expensive in terms of storing their row column indices and their value. We conducted a pilot study of the pruning method on an LSTM with 30 input units, 30 hidden units, and one layer trained for sleep apnea diagnosis [105]. Multiplications, additions, and memory are calculated for Column Compressed Stored weight format as in Reference [42]. Figure 5 shows the impact of the sparse routine on the processing requirements of the model.
(3) Pruning method is quite empirical, i.e., the degree to which the network is pruned matters [42]. Less pruning may affect desired memory fitting, high pruning may affect performance.
(4) Choice of regularization impacts pruning. L1 and L2 regularizations are coupled with unstructured pruning [41] and Group Lazzo regularization is coupled with structured pruning [92].
(5) Learning rate schedule impacts pruning [153].

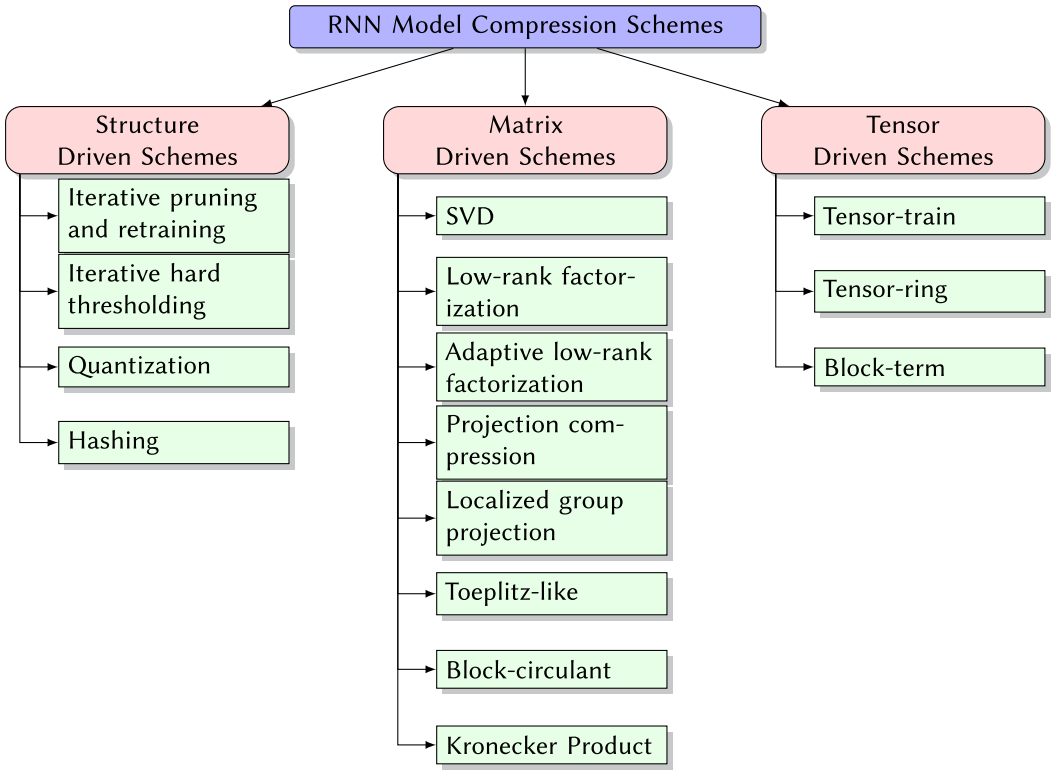

Fig. 4. Classification of RNN model compression schemes. The compression methods are either driven by network structure simplification thinking (branch-left) or weight matrix simplification thinking (mid-branch) or tensor simplification thinking (branch- right).

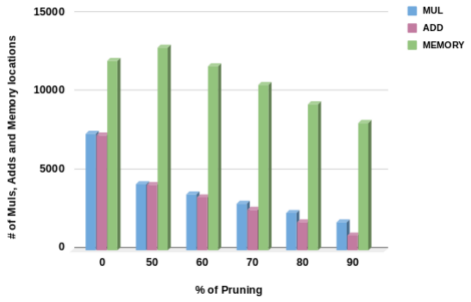

Fig. 5. Impact of pruning on the number of computations and memory requirements of a 1-Layer sleep apnea diagnostic LSTM model whose input length is 30, memory units is 30 [105].

5.2 Iterative Hard Thresholding

Abbreviated as ITH, this compression routine [146] provides an upper bound to the number of non-zero entries of a given weight matrix by a sparsity index k. This is as expressed in Equation (5). The index value is between 0 and 1; 1 indicates completely dense or a full weight matrix, 0 indicates complete sparsity. The sparsity index is a hyper-parameter. Both the sparsifying methods,

i.e., iterative pruning method and iterative hard thresholding method, target on minimizing the number of weights. Fewer weights imply lesser memory, lesser computations, and thus efficient device mapping:

$$\min_{w} f(w); \text{subject to } \|w\|_0 \leqslant k. \tag{5}$$

In References [57, 72], the authors chose a stochastic gradient descent optimizer to reduce the cost function constrained to this bound. A caveat, for such sparsification that is tightly coupled to the training procedure, techniques such as drop-out, drop-Connect, AR, TAR regularization are introduced to enhance the performance of the compressed model if degraded.

5.3 Quantization

This is a default step after reduction of network weights by any of the compression methods mentioned in Figure 4. Working on the remaining weights that are salient, the quantization scheme reduces the redundancy in the bits that represent them. Deep Compression [43] performs quantization using k-means clustering mechanism and a codebook generation. During forward prop, same value weights are binned and only their indices are stored, which is memory-friendly. During backprop algorithm, gradients are binned, added together, multiplied by the learning rate, and subtracted from the shared centroids from the previous iteration. k-means clustering is used to identify the shared weights *for each layer* and not between layers. Conversely, weight sharing can be between layers, as in the projection compression scheme discussed in Section 5.7.

From the implementation aspect, TensorFlow has a lite mode in which quantization to 8-bit integer is provided. However, TFlite does not support RNNs fully yet. Two-step compression scheme (pruning and quantization) is a part of the Xilinx tool flow called **Deep Neural Network Development Kit (DNNDK)** for neural network compression [51]. But they target **Computer Vision (CV)** applications and FPGAs primarily and are less explored for RNN modeling.

5.4 Hashing

Hashing technique is similar to weight quantization and clustering, except the assignment of weight to a cluster is determined according to a hash function [122]. Hashnet [15] computes hash functions on a cluster of weights grouped in buckets. The groups represent the network connections with the same weight value. This is beneficial for the back prop algorithm [76] to stabilize them in groups. The method is computationally expensive. And memory operations involved are not organized, which is again not desirable. Hashing is compared to other compression techniques in Reference [80].

5.5 Singular Value Decomposition

Abbreviated as SVD, this is a simple matrix decomposition method. However, the challenge lies when applied to trained weights of a neural net. The matrix is decomposed to three blocks of left singular vectors, diagonal matrix, and the right singular vectors, as seen in Figure 6. Generally, the SVD method has shown to degrade the performance when implemented as a singleton compression strategy after training an RNN [43, 108]. However, SVD in conjunction with a projection compression scheme is more promising (see Table 7).

Reference [108] incorporates SVD along with other compression schemes and analyzes performance for the speech recognition LSTM model. Unique to other explorations, we find an SVD-based software accelerator for deep learning execution called DeepX [73]. It performs the model mapping in two steps, namely, SVD-based Layer Compression *at runtime* [74] and Deep Architecture Decomposition. It is unique in the sense that it decouples SVD from the training routine. But

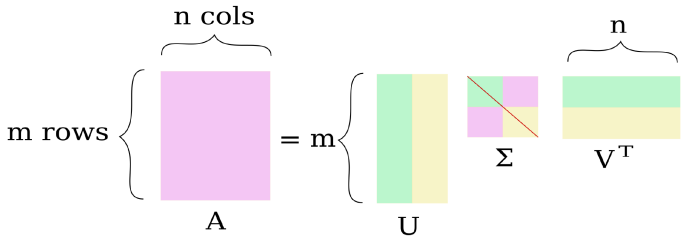

Fig. 6. Singular value decomposition of a matrix A into left singular vectors (U), diagonal matrix (Σ), and right singular vectors (V^T) [56].

experiments are w.r.t. DNNs and CNNs. The performance, though not up to the mark, is within a 5% error limit. They target mobile SoCs.

5.6 Low-rank Factorization

Low-rank factorization (LRF) [123] is a method where a matrix A $m \times n$ is factorized into matrix B $m \times r$ and matrix C $r \times n$ such that Equation (6) holds where p is the fraction by which the number of parameters needs to be reduced. Rearranging and solving for r gives Equation (7). For a given network, r is thus a hyper-parameter that is dependent on p. The key implication on the edge device is that, since both matrices B and C are dense, there is no need for a sparse matrix handling library or a customized memory structure. The scheme simply suits any of the off-the-shelf DL tools and hardware platforms. It suits GPU architectures also.

$$r < \frac{pmn}{m+n} \tag{6}$$

$$mr + rn \leqslant pmn \tag{7}$$

Any factorization for edge fitting affects expressivity. In other words, low-rank factorization accompanying training would hurt the model performance [142] of edge design. We find alternatives for this problem in literature. Authors in Reference [14] tweak the low-rank method very carefully. An adaptive input dependent parameter ($\pi(h)$) gently introduced handles the expressiveness issue. Their raw proposal is an unnormalized *learned mixture* of low-rank factorizations whose mixing weights are computed adaptively based on the input. This is illustrated in Equation (8). With input h and K mixture components, W $\in R^{n \times m}$ is decomposed as Equation (8). $\pi(\cdot) : R^n \rightarrow R^K$ is the function that maps each input to its mixture coefficients, and $U(k) \in R^{m \times d/K}, V(k) \in R^{n \times d/K}$, U and V being small rank (d) matrices. Additionally, they add two more strategies of pooling before projection and random projection to their method to balance both expressivity as well as reduced computation. Authors of DeepThin [122] use an auxiliary intermediate matrix and an efficient re-layout operation. This results in efficient compression and speedup. For practitioners, DeepThin is a compression library for neural nets that readily integrates with TensorFlow.

$$W(h) = \sum_{k=1}^{K} \pi_k(h) U^{(k)} (V^{(k)})^T \tag{8}$$

We observe that low-rank factorization is a common method and powerful as well. But they need careful design and intelligent adaptation for the edge context.

5.7 Projection Compression Scheme

RNN weights in time are inherently shared. What can complicate the edge model is adding more layers implying more weights. Weight sharing across layers is one way to overcome this. Projection compression simply does this. The method also inherits matrix decomposition into sub-matrices

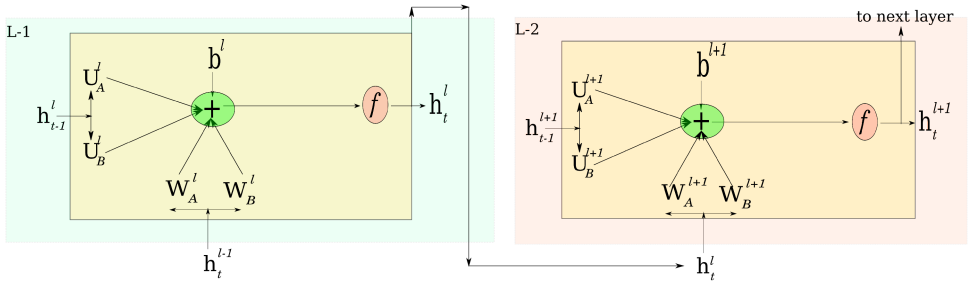

Fig. 7. Low-rank projection scheme of a two-layer (L1 and L2 boxes) RNN sharing a low-rank projection matrix across the two layers. The weight matrix is first decomposed into A and B sub-matrices using SVD or low-rank factorization and then shared, i.e., W_A^{l+1} is U_A^l and W_B^{l+1} is U_B^l.

across layers using truncated SVD [108] or a simple low-rank factorization scheme [80]. Figure 7 illustrates the projection scheme between two layers of an RNN.

References [80] and [108] discuss in detail how projections are created using truncated SVD and using low-rank factorization, respectively.

W.r.t. the number of parameters of the projection model, the third term in Equation (4) (in Section 2) will now be replaced by the recurrent and projection layers parameters resulting in Equation (9), where n_r is the number of units in the recurrent projection layer. One can reduce the number of parameters by controlling the ratio n_r/n_h, where n_h is the number of hidden units. When $n_r < n_h$, one can increase the model memory($n_h \equiv n_c$) and still be able to control the number of parameters in the recurrent connections and output layer [116].

$$4 \times n_h \times n_h + 4 \times n_x \times n_h + n_h \times n_r + n_r \times n_y + 3 \times n_h \tag{9}$$

TFlite incorporates the projection scheme in its design for neural compression. ProjectionNet [111] is a recent advancement that uses the projection approach uniquely. It works out on-device model fitting in two parallel architectures and training them jointly. One is a full network version and the other is a projected version that transforms inputs or intermediate representations into bits. The latter learns from the former or otherwise called as a teacher-student training method. It uses binary hashing and applies it to the projection vectors to transform the input into a binary hash representation. The compact network representation is desirable for device fitting. More interestingly, it is computed on the fly at inference time. Additionally, quantization and weight sharing can be accompanied for further shrinking and optimization.

The idea of projecting the hidden states onto low-dimensional space is further extended as group projections for generating sparse structured weight matrices in Reference [110]. Primarily, the output vectors that are broken into local groups are being represented as linear projections of corresponding local input group vectors. The corresponding weight matrices with linear projections are block diagonal matrices of g blocks or **LGPs (Localized Group Projections)**. This method has a disadvantage of restricting the passage of correlations among groups to subsequent timesteps. A mixing trick helps overcome this. Mixing allows information to flow in between different groups. Authors provide two ways of mixing. One is using permutation matrices (shuffle mixing). The other is using dense matrices (dense mixing) along with localized group projection matrices.

5.8 Toeplitz and Block Circulant Structures

Structured matrices are highly beneficial for parameter sharing schemes in RNNs. Toeplitz and block circulant structures are good examples for these. A Toeplitz matrix [38] is an $n \times n$ matrix

$T_n = [t_{k,j}; k, j = 0, 1, \ldots, n-1]$ where $t_{k,j} = t_{k-j}$. To simply state, it is a descending, diagonal constant matrix, i.e., elements of each diagonal are same, and number of elements decreases we move from the main diagonal.

$$T_n = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \ldots & t_{-(n-1)} \\ t_1 & t_0 & t_{-1} & & \\ t_2 & t_1 & t_0 & & \vdots \\ \vdots & & \ddots & & \\ t_{-(n-1)} & & \ldots & & t_0 \end{bmatrix}$$

The structure is beneficial for fast matrix-vector multiplication during forward pass and also gradient computations during backpropagation using Fast Fourier Transform like operations [80]. Also, Toeplitz matrices can be linearly transformed into matrices of rank less than or equal to 2 using certain shift and scale operations along with specific displacement operators. Reference [121] gives details of displacement operators associated with structured matrices. We note that displacement rank affects modeling capacity.

A special case of Toeplitz matrices is C_n when every row of the matrix is a right cyclic shift of the row above it so $t_k = t_{-(n-k)} = t_{k-n}$ for $k = 1, 2, \ldots, n-1$. Since each row is a reformat of the first row, storage is thus reduced to one row of this matrix. The model compression ratio is determined by the block size of the circulant submatrix [135]. The authors replace matrix-vector multiplication with **Fast Fourier Transform (FFT)** operations to speed up computations. The structure of block-circulant matrix highly enables this operation.

$$C_n = \begin{bmatrix} t_0 & t_{-1} & t_{-2} & \ldots & t_{-(n-1)} \\ t_{-(n-1)} & t_0 & t_{-1} & & \\ t_{-(n-2)} & t_{-(n-1)} & t_0 & & \vdots \\ \vdots & & \ddots & & \\ t_{-1} & t_{-2} & \ldots & & t_0 \end{bmatrix}$$

5.9 Kronecker Product

Ill-conditioned and highly parameterized recurrent matrices of RNNs are handled using Kronecker product matrices. Kronecker product of a matrix is given by $A = B \otimes C$ where, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{m1 \times n1}$, $C \in \mathbb{R}^{m2 \times n2}$, $m = m1 \times m2$, $n = n1 \times n2$, $\otimes$ denotes the Kronecker Product, and $\odot$ is the Hadamard or entry-wise product [62, 128].

$$A = B \otimes C \tag{10}$$

$$A = \begin{bmatrix} b_{1,1} \odot C & b_{1,2} \odot C & \ldots & b_{1,n1} \odot C \\ b_{2,1} \odot C & b_{2,2} \odot C & \ldots & b_{2,n1} \odot C \\ \vdots & & \ddots & \\ b_{m1,1} \odot C & b_{m1,2} \odot C & \ldots & b_{m1,n1} \odot C \end{bmatrix}$$

In Reference [62], authors use Kronecker factors of size 2×2 to replace the hidden-hidden matrices of every RNN layer. A 256×256 sized matrix is expressed as a product of eight matrices (Kronecker Product factors) of size 2×2. A compression of up to $2\times$ is achieved. However, a unitary constraint is added to these 2×2 matrices to circumvent the vanishing exploding gradients problem

and stabilize training. Reference [128] proposes an algorithm to find the dimensions of the KP factors. The resulting factors lead to about 50× reduction in weights.

Thus, a traditional RNN Cell, represented by Equation (11) is replaced by Equation (12):

$$h_t = f([W_x W_h] * [x_t; h_{t-1}]), \tag{11}$$

$$h_t = f([W_x W_0 \otimes W_1 \ldots \otimes W_{F-1}] * [x_t; h_{t-1}]), \tag{12}$$

where W_x (input-hidden matrix) $\in \mathbb{R}^{m \times n}$, W_h (hidden-hidden or recurrent matrix) $\in \mathbb{R}^{m \times m}$, $W0 \ldots W_{F-1} \in \mathbb{R}^{2 \times 2}$, $x_t \in \mathbb{R}^{n \times 1}$, $h_t \in \mathbb{R}^{m \times 1}$, and $F = log2(m) = log2(n)$.

5.10 Tensor Decomposition Methods

Decomposition of higher-dimensional arrays or "tensors" have been explored for applications such as video processing, identifying people in video frames, and so on, using recurrent models. Equation (13) represents a tensor A of order d and each of certain dimension $n_1 \times n_2 \times \cdots \times n_k$. When $d = 1$, tensor is a vector and when $d = 2$, tensor is a matrix. Number of elements when each $n = 2$ and order of the tensor $d = 20$ is $n^d = 2^{20} = 1M$, which is expensive. Tensor decomposition methods deal with this memory expense. Reference [67] has an interesting history and details of various tensor decomposition methods. It also provides the basic operations on tensors, how to convert a tensor to a matrix (matricization), the notion of rank and uniqueness of tensors, and so on. For implementation of tensor decomposition methods, Reference [45] is a useful open-source library in TensorFlow that includes major tensor decomposition methods such as **CANDECOMP (Canonical Decomposition)/PARAFAC Analysis (Parallel Factors)/CP (Canonical Polyadic)**, Tucker decomposition, and so on, and Reference [107] is software developed for decomposing sparse tensors that fits a variety of multicore, manycore, and GPU computing architectures. Tensorlab, a Matlab package for tensor computation, gives details of various tensor factorization schemes.

$$\mathcal{A} = [\mathcal{A}(i_1, \ldots, i_d)], \; i_k \in 1, \ldots, n_k \tag{13}$$

5.10.1 Tensor Train Decomposition. One can reshape a fully connected layer into a high-dimensional tensor and then factorize it using **Tensor-Train (TT)** [98]. Tensor train factorization layer replaces the input to hidden weight matrices [97]. A tensor train decomposition breaks a tensor of order d into two 2d tensors, $\mathcal{G}(1)$ and $\mathcal{G}(d)$ and (d - 2) 3d tensors, $\mathcal{G}(k)$ as seen in Equation (14):

$$\mathcal{A} = \mathcal{G}^{(1)} \otimes \mathcal{G}^{(2)} \otimes \cdots \otimes \mathcal{G}^{(d)}. \tag{14}$$

$\mathcal{G}(k) \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ is the kth *core* tensor. $r_0 = r_d = 1$ such that $\mathcal{G}(1)$ and $\mathcal{G}(d)$ are matrices. The dimensions $r_0, r_1, \ldots, r_d$ of the auxiliary indices are called the TT ranks. When all ranks have the same value, then it is simply its TT rank. Reference [66] is an interesting study between recurrent neural networks and TT decomposition. It claims a shallow network of exponentially large width is required to mimic a recurrent neural network or otherwise stated as the *expressive power theorem*. TF library has tensor-train incorporated known as T3F [97]. It is an explicit TT decomposition implementation in TF with GPU support, batch processing, automatic differentiation, and versatile functionality for Riemannian optimization framework, which takes into account the underlying manifold (group of interconnected points in space) structure to construct efficient optimization methods. Also, Kronecker product is a special case of a TT-object [99], and T3F library accommodates Kronecker products.

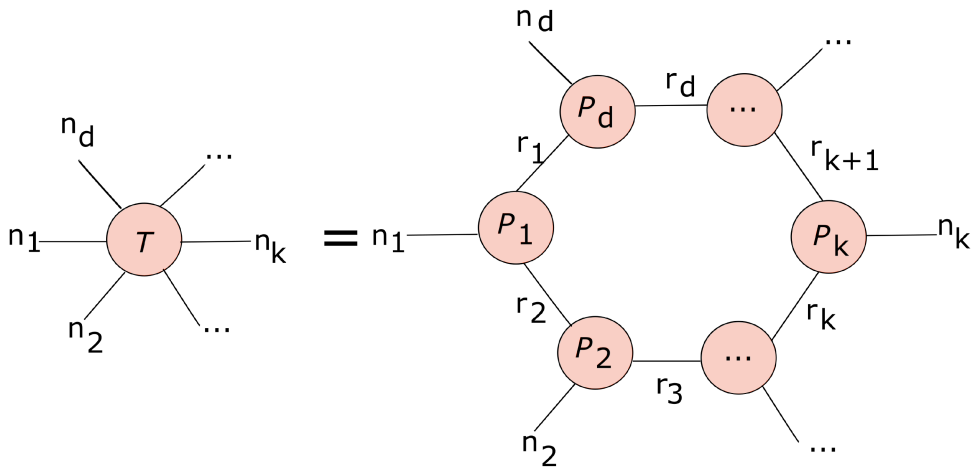

Fig. 8. Graphical representation of Tensor Ring decomposition of a d order tensor into latent cores- $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_d$ [151].

5.10.2 Tensor Ring Decomposition. TT decomposition highly depends on the permutations of tensor dimensions. Optimizing the representation over multilinear product over latent cores is difficult. Hence, a variant called the tensor ring decomposition [151] is proposed. The representation is circular multilinear products over a sequence of low-dimensional cores. This is graphically interpreted as a cyclic interconnection of 3rd-order tensors.

In Equation (15), $\mathcal{T} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is a tensor of order d and size $n_1 \times n_2 \times \cdots \times n_d$. Tensor Ring decomposes this into a sequence of latent tensors $\mathcal{P}_k \in \mathbb{R}^{r_k \times n_k \times r_{k+1}}$, k = 1,2,... ,d. $\mathcal{T}(i_1, i_2, \ldots, i_d)$ denotes $(i_1, i_2, \ldots, i_d)^{th}$ element of the tensor. $P_k(i_k)$ denotes the i_k^{th} lateral slice matrix of the latent tensor P_k of size $r_k \times r_{k+1}$. The latent tensor P_k is also called kth-core. The size of cores, $r_k, k = 1, 2, \ldots, d$, collected and denoted by a vector $r = [r_1, r_2, \ldots, r_d]^T$ are called TR-ranks.

$$\mathcal{T}(i_1, i_2, \ldots, i_d) = \mathcal{T}_r\Big\{\mathcal{P}_1(i_1)\mathcal{P}_2(i_2) \ldots \mathcal{P}_d(i_d)\Big\}, = \mathcal{T}_r\Big\{\prod_{k=1}^{d} \mathcal{P}_k(i_k)\Big\} \tag{15}$$

Trace operations (for a square matrix, it is the sum of the main diagonal elements) and latent cores treated equivalently makes the method invariant to circular dimensional permutation. TR model can be viewed as a linear combination of TT decompositions. Most importantly, the decomposition still captures the desired representational capability of the network.

5.10.3 Block Term Decomposition. Abbreviated as BTD, Block term decomposition approximates a tensor by a sum of low multilinear rank Tucker terms [100]. 3rd-order tensor $\mathcal{A} \in \mathbb{R}^{I,J,K}$ is decomposed as Equation (16), where $\mathcal{S}_\nabla$ is of size $M_r \times N_r \times P_r$, $\mathcal{U}_\nabla$ is of size $I \times M_r$, $\mathcal{V}_\nabla$ is of size $J \times N_r$, and $\mathcal{W}_\nabla$ is of size $K \times P_r$ for $r = 1, \ldots, R$ [67]. The drawback of BTD is that it is designed to compress only the input vector and not the hidden vectors [110]. So block tensor decomposition is suitable for high-dimensional input applications and compression at the input side of the RNN.

$$\mathcal{A} = \sum_{r=1}^{R} [[\mathcal{S}_\nabla; \mathcal{U}_\nabla, \mathcal{V}_\nabla, \mathcal{W}_\nabla]] \tag{16}$$

Finally, we also find other methods of compression like the teacher-student method and knowledge distillation [110] in the context of edge-based modeling. Skip RNNs [13] and Delta RNNs [31]

are based on skipping a few inputs/activations than a threshold. We believe to have covered important ones in detail. To summarize, we provide a concise list of compression techniques with their strengths and drawbacks in Table 5. Each technique is linked to a resource link given in the last column of the table. Mostly the resources are TF-based, expect Hashing, which is implemented in Torch. It is important to note the version of TF before its download and usage for these links. The recent TF version incorporates *eager execution*. Some of the resources are just one line function calls, some are tools in the software library, and some are a process in themselves. The developer needs to adopt from any of these resource links for his/her use case discretely for an RNN edge modeling context.

6 RNN COMPRESSION EVALUATION

Model evaluation after training and again after compression is essential for any RNN to device mapping study. Characteristics of evaluation of the model after compression is its performance with the reduced set of weights and connections after compression. Also, it is its performance after a reduced precision representation of the weights and inputs. Both need to be evaluated (this split is clear in most of the comparison tables in this section). Additionally, if the performance is retained after compression, such a technique is said to be efficient ($\mathcal{E}$). Sometimes the performance exceeds the baseline and they are said to be superior ($\mathcal{S}$).

Here, we present a comparative study of the various compression methods implemented in literature. They are applied to time-series application contexts and RNN-based model designs. We list the application and set of evaluation metrics in Table 6. The second column of the table gives the application context and metric details. Under each application, we present a comparison table listing the compression method applied and baseline details. The first column in each table (Table 7 to Table 11) is the list of articles that have explored RNN compression. We take the model baselines as mentioned in these articles as the baseline for comparison. While implementation, repeating/using a baseline model needs prior knowledge of how to either *freeze* a model using given hyperparameters or how to use a *frozen* model already open-sourced. Each framework has its own way to deal with the two. We look at two evaluation metrics after a compression scheme is applied for the edge context:

(1) Relative performance: This is the ratio of the difference in model performance before and after compression, to performance of the uncompressed model in percentage. This metric helps to gauge the technique's performance impact over the baseline. This could be efficient (performance is maintained) or superior (performance is improved) or degraded (hurting the baseline performance). Higher the better.
(2) Memory savings: This is a memory ratio before and after compression. This metric depicts the technique's ability in device-memory-fitting capacity. Higher the better.

There are two other important metrics such as compute savings (ratio of MACs performed before and after compression) and speedup due to reduced memory and compute figures. Not all authors provide these metrics as a part of their analysis, so we observe them as and when they appear in the reference implementation articles. We also see the impact of quantization in many implementations across tables. Finally, we flag the compression techniques as efficient ($\mathcal{E}$) if the performance is retained and superior ($\mathcal{S}$) if the performance is higher than the baseline after compression.

6.1 Application1: Large-scale Vocabulary Speech Recognition

The first application is a speech-processing application, i.e., converting speech signals to text, abbreviated as LVCSR. Deep Speech is the baseline for many of the references. The model is a combination of CNN, RNN (either an LSTM or GRU) and a CTC layer [37]. To explain Table 7, we expand

Table 5. RNN Compression Techniques and TensorFlow-based Implementation Resources

Compression Technique	Strength	Drawback	Paper	Helpful Resource
Iterative Pruning and Retraining	Weight reduction up to 95%	Pruning threshold is a hyperparameter, Can lead to sparse matrix if unstructured	[43, 44], [41, 91], [92, 153]	IPR
Iterative Hard Thresholding	Relatively simple technique	Sparsity index is a hyperparameter	[146], [57], [72]	HT
Quantization	Applicable after any compression technique	Activation quantization is difficult than input quantization	[43], [51]	DNNDK
Hashing	Backprop algorithm can easily handle hash params	compute-expensive method	[15, 122]	Hash
SVD	Can be decoupled from training process	Degrades performance, if singly applied	[108], [73], [43]	SVD
Low-rank Factorization	Produces only dense matrices, memory-friendly	Rank is a hyperparameter	[14], [122]	LRF
Projection Compression	Suitable for large models as weight sharing is across layers	Involves matrix decomposition needing regularization	[116], [80, 108],	Proj
Localized Group Projection	Compression is in groups, less storage requirement	Method restricts passage of information amongst groups	[110]	-
Toeplitz Structure	Fast matrix-vector calculation with FFT	Displacement operators involved affect model capacity	[80], [121]	Toep
Block Circulant Structure	Matrix storage is reduced to only one row	Compute-expensive method	[135]	Blk
Kronecker Product	Enables faster matrix-vector product	Complex field operations to circumvent ill-posed gradients	[62], [128]	KPF
Tensor Decomposition Techniques	Suitable for high-dimensionality problems and strongly captures representational capacity of RNNs	Involves TT, TR, rank finding, designed to compress only input vectors and not hidden feature vectors	[101, 141] [144]	TD

Some of the resources are just one-line function calls, some are tools in the software library, and some are a process in themselves.

Table 6. RNN-based Application Baselines and Performance Evaluation Metrics

Application	Ref.	Evaluation metric	Indication
Speech Recognition (SR)	[148]	Word Error Rate (WER), Character Error Rate (CER)	↓ the better ↓ the better
Key Word Spotting (KWS)	[126]	Accuracy (Acc)	↑ the better
Human activity Recognition (HaR)	[2]	Accuracy/F1 score	↑ the better
Human Action Recognition (HAR)	[149]	Accuracy/F1 score	↑ the better
Language Modeling (LM)	[88]	Perplexity (PPL)	↓ the better
Machine Translation (MT)	[102]	Bilingual Evaluation Understudy (BLEU) score	↑ the better

Table 7. A Comparison of RNN Compression Techniques for LVCSR Models

Ref.	Compression Technique	Model	Hidden Units	Params Before (M)	Params After (M)	Performance Before (WER/CER)	Performance After (WER/CER)	Relative Performance(%)	Memory Savings
[108]	SVD, projection	LSTM5	500	9.7	3.1	12.4	12.9	−4.03	3.13×
[80]	low-rank	RNN3	600	1.85	0.79	43.5	54.6	−25.52	2.34×
	Hashed Nets						49.2	−13.1	
	Toeplizt-like						48.4	−11.264	
	low-rank + projection				0.635		43.5	0, $\mathcal{E}$	2.91×
	projection-top Toeplitz-bottom	LSTM5	500	9.12	2.23	33.1	33.4	−0.91	4.09×
[91]	Pruning,	BiRNN7, Medium	1,760 2,560*	67	11.1	10.67	10.59	0.75, $\mathcal{S}$	6.03×
		BiRNN7, Large	1,760, 3,072*		16.7		10.25	3.94, $\mathcal{S}$	4.01×
		GRU3, Medium	2,560, 3,568*	115	17.8	9.55	9.76	−2.12	6.46×
[92]	Blocked Pruning	BiRNN7	1,760	67	7.3	15.36	17.93	−16.73	9.18×
			1,760, 3,072*		25.8		15.66	−1.95	3×
		GRU3	2,560	115	10.8	15.42	16.78	−8.81	10.65×
			2,560, 3,584*		25.6		16.23	−5.35	4.5×

LSTM5 represents a five-layer LSTM, BiRNN7 is a seven-layer bi-directional RNN. Medium (for example 2,560) represents a moderately large number.

on one row of the table. In row 1, SVD and projection compression scheme are explored and evaluated by Reference [108]. Their baseline is the LSTM model of five layers. The total number of hidden units in each layer is 500. Before a compression method is applied, the number of params of the model is 9.7M. After the compression is applied, the parameters are reduced to 3.1M, saving up to 3.13× memory requirements. The performance before the compression is 12.4 **WER (Word Error Rate)** and after compression is 12.9 WER. Relative performance w.r.t baseline is −4.04%. Hence, this compression does not fall under $\mathcal{E}$ or $\mathcal{S}$ class. The remaining rows of Table 7 can be understood similarly

In brief, Reference [80] explored low rank, hashing, Toeplitz-like, and projection compression scheme. References [91] and [92] experimented with pruning and block pruning techniques, respectively. Overall, we find that low rank with projection and pruning compression methods are Efficient and Superior compared to other methods for this speech application. In terms of memory savings alone, blocked pruning stands out from all of the methods.

Table 8. A Comparison of RNN Compression Techniques for Key Word Spotting on Peter Warden's Google 30 Voice Dataset

Ref.	Cell	Technique	Acc (%)	Relative Performance(%)	Model Size(KB)	Memory Savings
[72]	Base-RNN	-	80.05	-	63	-
	FGRNN	-	92.03	14.97	45	1.4×
		low rank	91.99	14.92, $\mathcal{S}$	38	1.66×
		low rank, sparse ITH,	91.18	13.9, $\mathcal{S}$	25	2.52×
		low rank, sparse ITH, quantization	90.78	13.4, $\mathcal{S}$	6.25	10.08×
[128]	Base-LSTM	-	92.5	-	243.42	-
	LSTM	pruning	84.91	−8.20	15.57	15.64×
		low rank	89.13	−3.64	16.8	14.5×
		KP	91.2	−1.41	15.3	15.91×
		KP, quantization	91.04	−1.58	8.01	30.39×
	Base-GRU	-	93.5	-	305.04	-
	GRU	low rank	90.88	−2.80	24.5	12.45×
		KP	92.3	−1.28	22.23	13.72×

6.2 Application2: Key Word Spotting

The second application is also another speech application, i.e., identifying **keywords (KW)** or wake words in a spoken sentence, abbreviated as KWS. Many articles worked a DNN, CNN, RNN or a combination for this task. We find Peter Warden's Google 30 voice dataset is common to a few important contributions in the context of RNN and edge modeling.

As seen in Table 8, Reference [72] adopted a simple RNN and not LSTM RNN for both baseline and implementation. This itself is a significant choice for the number of parameters consumed by the neural network affecting memory. However, methods to deal with the challenges pertaining to the vanilla RNN are hard. They handle the training challenges (see Section 4) by structurally modifying the vanilla RNN (see Section 4.2.1). The resulting RNN is named as FGRNN, fast-gated RNN. They apply low rank, sparsification using Iterative Hard Thresholding, and batch-SGD optimization algorithm to reach the desired compression. They incorporate quantization as a next step. Impact of step one (low rank, ITH, b-SGD) and step two (quantization) are reported separately. As in row 4, step one compression leads to a significant improvement in model performance (13.9). Thus, the method falls under the $\mathcal{S}$ class. For the edge, the memory savings are also quite large (10.08×) after quantization (row 5).

Reference [128] adapts LSTM and GRU as baselines individually. Applying pruning or low-rank compression, the performance is largely degraded compared to **Kronecker product (KP)** factorization. Further, the impact on memory savings is relatively similar. When quantized, the performance degrades further, but with obvious significant benefits (up to 30.39×) on memory savings. To conclude the table, for this application, techniques in Reference [72] are far superior to those explored in Reference [128] using the same dataset.

6.3 Application3: Human Activity Recognition

The third application is recognizing human activity in a given image or video frame, abbreviated as HaR. The two references in Table 9 use three different datasets for this task. UCI machine learning

Table 9. A Comparison of RNN Compression Techniques for Human Activity Recognition

Ref.	Dataset	Cell	Technique	Acc (%)	Relative Performance(%)	Model Size(KB)	Memory Savings
[72]	HAR2	Base-RNN	-	91.31	-	29	-
		FGRNN	-	95.38	4.46, $\mathcal{S}$	29	1×
			low rank	96.81	6.02, $\mathcal{S}$	28	1.03×
			low rank, sparse projection, ITH	96.37	5.54, $\mathcal{S}$	17	1.7×
			low rank, sparse ITH, quantized	95.59	4.68, $\mathcal{S}$	3	9.66×
	DSA	Base-RNN	-	71.68	-	20	-
		FGRNN	-	85	18.58, $\mathcal{S}$	207	0.096×
			low rank	85.27	18.96, $\mathcal{S}$	22	0.909×
			low rank, sparse ITH	83.93	17.09, $\mathcal{S}$	13	1.54×
			low rank, sparse ITH quantized	83.73	16.81, $\mathcal{S}$	3.25	6.15×
[128]	Sensor	Base-LSTM	-	91.9	-	1462.84	-
		LSTM	pruning	84.91	−2.13	75.55	19.36×
			low rank	89.94	−2.13	76.4	19.15×
			Kronecker product	91.14	−0.83	74.91	19.52×
			KP, quantized	90.9	−1.088	28.22	51.84×

Table 10. A Comparison of RNN Compression Techniques for Human Action Recognition on UCF11 Dataset

Ref.	Cell	Tensor Decomposition Technique	Acc (%)	Relative Performance(%)	Params	Memory Savings
	Base LSTM	-	69.7	-	59M	-
[141]	LSTM	Tensor train	79.6	14.203, $\mathcal{S}$	3360	17,600×
[144]		Block Term	85.3	22.381, $\mathcal{S}$	3387	17,420×
[101]		Tensor Ring	86.9	24.68, $\mathcal{S}$	1725	34,203×

repository having human activity information from smartphone dataset, UCI's Daily Sports Activity dataset, and Complex 72 sensor-based dataset [113] are the three dataset sources. Reference [72] uses RNN as the baseline, while Reference [128] uses LSTM as the baseline. As seen in the table, low rank, sparse ITH, and quantization are again superior compared to pruning or low rank or Kronecker Product factorized compression scheme for this application also. In terms of memory savings, KP with quantization is significantly beneficial. An LSTM of 1,462.84 KB is reduced to 28.22 KB, implying ≈ 52× savings.

6.4 Application4: Human Action Recognition

The fourth application is recognizing human action in a given image or video frame, abbreviated as HAR. This is more coarse grain in terms of the entire sequence of actions compared to recognizing a single activity. UCF11 is common to the three references in Table 10.

Table 11. A Comparison of RNN Compression Techniques for Language Models on PTB Dataset

Ref.	Technique	Hidden Units	Performance (Perplexity)	Performance after	Relative perf.	Params before(M)	Params after(M)	Memory Savings
[153]	Gradual Pruning (85% sparse)	1,500	78.45	78.31	0.18, $\mathcal{S}$	66	9.9	6.67×
[136]	ISS, Group Lazzo regularization	1,500	78.57	78.65	−0.1	36	3.66	9.84×
[110]	LGP-Shuffle	1,500	77.51	77.38	0.17, $\mathcal{S}$	36	0.72	50×
[72]	Low rank, Sparse ITH, Quantize	256	144	116.11	19.36, $\mathcal{S}$	129	39	3.30×
[14]	Adaptive, Low rank factorization	650	83.6	81.9	2.03, $\mathcal{S}$	6.9	4.1	1.68×

All show superior performance compared to baseline and all show significant memory savings. TensorRing decomposition stands out in both the categories of evaluation with 2,468% relative performance and 34, 203× memory savings. In brief, tensor decomposition methods significantly improve baseline performance and impact memory savings largely. One can note the degree of difference in figures (relative performance and memory saving figures) compared to other applications. They are very large compared to the other tables. However tensor decomposition methods are rarely adapted to other application domains.

6.5 Application5: Language Modeling

The fifth application we capture is an **NLP (Natural Language Processing)** application. Language modeling is an important aspect and subtask of NLP. Penn Tree Bank is the most commonly found dataset here. As seen in Table 11, almost all compression schemes chosen (such as gradual pruning, ISS, localized group projection, or adaptive low-rank) have shown superior performance to baseline. Looking for performance, low-rank with sparsification and quantization is more promising but for a smaller model size. For fairly larger hidden sizes of 1,500 units, the gradual pruning method proves better. Looking for memory benefits on a moderately sized model, adaptive low-rank factorization stands out. To note that ISS (a compression scheme) along with group Lazzo method (a regularization scheme) is adapted as their compression strategy in Reference [136] (row 2 of Table 11). Reference [136] mentions 7.48× and ≈ 11× speedup, Reference [110] mentions 5.0× compute savings, and Reference [72] mentions ≈ 0.67× speedup in their implementations after the compression techniques they used. These are a few references that provided these metric figures apart from memory savings and relative performance numbers.

Besides the table, we find another reference for this application. Reference [36] experimented pruning and quantization (branch one of Figure 4), low-rank factorization (branch two of Figure 4), and tensor train decomposition method (branch three of Figure 4). Their conclusions are noteworthy. For this application alone, a wide variety of compression schemes have been explored.

6.6 Application6: Machine Translation

The final application we look at is **Neural Machine Translation (NMT)**, i.e., speech/text translation from one language to another. We do not provide a comparison table for this application alone, as we find only one reference that implements RNN compression for MT.

Reference [153] uses a baseline following the Google Neural Machine Translation architecture [139]. They incorporate gradual pruning as the compression strategy. For a sparsity of 80%,

Table 12. Compression Techniques Applicable to RNNs That Vary in Complexity w.r.t. Core Cell, Number of Layers, Hidden Units, and Parameters

RNN type	Vanilla RNN	GRU/LSTM	GRU/LSTM/Bidirection
Layers	1–3	3–5	>5
Hid. units	500–600	600–2,000	>2,000
Params	<2M / up to 50KB	<10M / up to 500KB	>50M / >1000KB
Applied Compression Technique	SVD, Projection [108], Low-rank, Hashing, Projection, Toeplitz [80], Adaptive low-rank [14], Sparse ITH [72]	Toeplitz, Projection [80], ISS [136], LGP-shuffle [110], Sparse ITH [72], Kronecker Product [128]	Pruning [91], Blocked Pruning [92], Gradual Pruning [153], Tensor Decomposition [101, 141, 144],

Model complexity and task complexity progressively increase moving from left to right columns of the table.

the performance is superior w.r.t. baseline. But when sparsity is increased to 90%, performance drops by 2.16% but with significant memory savings up to 9.17×. Not to one's surprise, Reference [147] uses drop-out, a regularization method to improve the baseline four-layer LSTM (1,000 units) performance, i.e., from 25.9 to 29.03 BLEU score (see Table 6) or 5.8 to 5.0 Perplexity. This is for English-to-French translation. If one chooses to adapt drop-out as a compression scheme as in Reference [143], then drop-out technique can be considered beneficial for both training as well as compression.

6.7 Compression and Model Complexity Analysis

Here, we present Table 12 based on applications 1 through 6 and compression analysis tables 7–11. The table briefly describes the model complexity in terms of the RNN cell, number of hidden units, number of layers, and number of parameters of the baseline models for all the applications listed in the above subsections. Based on the model complexity, the compression techniques applied are listed. This can be a quick reference for edge developers to select compression methods based on their model complexity and also their task complexity. To explain one column of the table, a vanilla recurrent structure with 1 to 3 layers, 500–600 hidden units, less than 2M parameters in total, compression techniques such as SVD, projection, low-rank, adaptive low-rank schemes, hashing, Toeplitz, and sparse **Iterative Hard Thresholding (ITH)** methods are applied.

It is observed that for complicated AI tasks (action recognition in streaming video) and thus high model complexities (more layers, hidden units, or even bidirectional architectures), compression methods pruning and tensor decomposition methods are effective. For relatively low-complex tasks (keyword spotting) with thus less complex structures, compression methods such as low-rank, ITH, and projection schemes are commonly applicable. Special gating structures called Fast Cells [72] are effectively applicable for simpler tasks and simpler models. For intermediate complexities in models and tasks, projection schemes, LGP-shuffle, **ISS (Intrinsic Sparse Structuring)** are applicable. Another important observation is that many different compression methods are tried on language models (refer to Table 11)modeled with varying degrees of low, medium to high model complexities.

7 CONCLUSION

From this survey, we observe the following items for edge-based RNN modeling and design:

(1) The key objective of an edge-based neural modeling is to reach the best model's performance objective (or at least maintain) and still be able to achieve the desired compaction. Compute reduction and inference speedup are subsequent positive byproducts.

(2) Training dictates compression. A model developer's fine experience in training RNNs will imply efficiency in compression. Efficient training and efficient compression are key for edge-based neural modeling. Both involve many nuances and an intuitive approach to solve. We believe to have covered them in the survey.
(3) Very few compression techniques have been experimented as an independent step after training. Also, very few compression methods are incorporated within the training process with minimal fine-tuning procedures.
(4) Regularization methods (e.g., learning rate schedule) are important aspects of training and are very closely associated with compression and resulting efficiency.
(5) Sometimes a regularization method like drop-out is adapted for both training improvement and as a compression strategy.
(6) Usually, a combination of compression schemes is a better choice than a single compression scheme applied.
(7) The ease of applicability of the pruning compression scheme is vividly clear in any application domain and chosen RNN architecture. Almost all comparison tables (Tables 7 to 11 of Section 6) have pruning as one compression method.

And making strides, we find the answers to the research questions posed at the beginning of our survey.

Q1. *Are there simple yet efficient methods that bring down a given RNN model onto off-the-shelf devices with small die area, maybe, say, an Arduino?*
Yes, there are a few methods that tackle this. Simplicity and efficiency are highly desirable features. State-of-the-art techniques are aiming at efficiency. Simplicity would result from refined RNN architecture designs that are still evolving (see Table 2). Efficiency is intricately coupled to how training, compression, and optimizations are handled in a balanced manner.
Q2. *How feasible is it to map complex RNNs onto small footprint devices without compromising for their performance?*
Complex RNNs have been mapped to edge devices. From the survey, we find that few implementations have maintained baseline performance, few have superseded the baseline, and a few have under-performed (see Table 7–Table 11).
Q3. *Finally, how is AI transforming and where is it leading to in the supervised learning regime?*
Artificial intelligence is replaced with *"edge intelligence"* at this point in time. Thanks to deep learning-based modeling and compression, few off-the-shelf, small-footprint devices are now becoming the potential edge intelligent cores of the future smart systems. Simply without exaggeration, even an Arduino Nano would be quite a processing space for efficient edge intelligent models to come.

In short, this survey is a shred of evidence that neural network design for the edge is an intelligent amalgamation of choices starting from the network structure and topology, its configuration, training and compression strategies to be able to fit into off-the-shelf, low-footprint devices. Edge developers need to pay attention to all the blocks in the back-bone structure, i.e., Figure 2. This survey can be considered as a single point reference for beginners in supervised RNN-based time series modeling (Section 2 to 4). Finally, this is an edge-focused survey starting from basic to advanced understanding and implementation of edge RNNs.

8 ROAD MAP AHEAD

In this final section, we present the following research gaps and future directions:

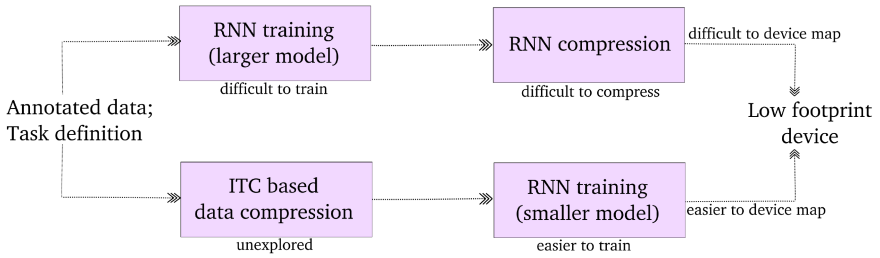

Fig. 9. A relook of RNN-based application mapping onto low-footprint device. A classical method is the upper half of the diagram; an unexplored path is the bottom half of the diagram.

(1) Feature engineering versus neural net modeling: If there is a methodology to pre-process and organize data and present it to the network, then we believe half the problem is solved. On the flip side, ideally, neural nets are expected to learn from any data shown to them without human intervention. Do we handle data more precisely for better feature extraction or design networks that handle data themselves are two diverse routes?
(2) Task handling: If the sequence task is too complex, then one can define smaller sub-tasks and work out an incremental learning procedure [23] just as human beings have evolved to perform complex tasks. Like learning letters, then words, then sentences, and then understand the context and dependency. This approach is yet to evolve to its full potential.
(3) RNN training methodology: If there could be a well-formulated training methodology, then it would reduce the burden of training RNNs to a certain degree like hyper-parameter search space, sensitivity to them [83], and so on. Exact hyper-parameters for reproducing/analysis of existing methods and cross-verification of open source codes is essential. Automation of network design is another route.
(4) RNN training for the edge: During the entire survey, we could hardly find training methods that are suitable specifically for edge modeling and mapping. It has always been training RNNs in general with no bearing toward the end device and later compress them to fit them on the chosen edge. We believe that future scope for training practitioners is to find methods that prepare RNN training solely for edge mapping.
(5) RNN exploration: Many RNNs are yet unexplored for real-time applications. RNNs such as URNNs and ORNNs that beat the training challenge and are yet simpler by structure than LSTMs can be more universal. Compressing them can thus be relatively simpler.
(6) Independent Compression: Can a compression scheme be totally decoupled from training or can it impose fewest retraining iterations? This will result in working the problem independent of training and with an implication of pre-trained RNN models ready to become more *"edge-friendly."*
(7) Transfer and active learning for the edge: Can transfer learning and active learning provide more foundation to edge intelligence and future deep learning expansion?

We believe modeling the RNNs bearing the edge in mind can be redefined. In the above research gaps and directions, we elaborate on item one. While neural network selection is getting automatic [49, 106, 118], so is feature extraction [17]. In this context, we propose an alternative hypothesis/approach. A better way to handle training data that imposes very less redundancy during training will lead to better model design and easier training procedures. This can be a plausible way to deal with edge modeling. A re-look at the device constrained deep learning modeling approach is presented in Figure 9. **ITC (Information Theory Coding)**-based minimal data description [6, 112] can prove powerful rather than terse RNN training and compression. Techniques

to convert sequences to minimal description length data formats and structures will be key. We conclude the survey with this road unexplored for edge intelligent RNNs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for providing valuable feedback on this survey.

REFERENCES

[1] Adrian. 2018. A review of Dropout as applied to RNNs. Blog post. Retrieved from https://medium.com/@bingobee01/a-review-of-dropout-as-applied-to-rnns-72e79ecd5b7b

[2] J. K. Aggarwal and Lu Xia. 2014. Human activity recognition from 3D data: A review. *Pattern Recog. Lett.* 48 (15 10 2014), 70–80. DOI: https://doi.org/10.1016/j.patrec.2014.04.011

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen et al. 2016. Deep Speech 2: End-to-end speech recognition in English and Mandarin. In *Proceedings of the International Conference on Machine Learning*. 173–182.

[4] Martin Arjovsky, Amar Shah, and Yoshua Bengio. 2016. Unitary evolution recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 1120–1128.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. arXiv:stat.ML/1607.06450 (2016).

[6] Yoshua Bengio. [n.d.]. Learning deep architectures for AI. ([n.d.]).

[7] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. 2012. Advances in optimizing recurrent networks. *CoRR* abs/1212.0901 (2012).

[8] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* 5, 2 (Mar. 1994), 157–166. DOI: https://doi.org/10.1109/72.279181

[9] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the International Conference on Computational Statistics*. Springer, 177–186.

[10] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576* (2016).

[11] Stefan Braun. 2018. LSTM benchmarks for deep learning frameworks. *CoRR* abs/1806.01818 (2018).

[12] Han Cai, Ligeng Zhu, and Song Han. 2019. Proxylessnas: Direct neural architecture search on target task and hardware. In *Proceedings of the International Conference on Learning Representations (ICLR'19)*.

[13] Victor Campos, Brendan Jou, Xavier Giró i Nieto, Jordi Torres, and Shih-Fu Chang. 2017. Skip RNN: Learning to skip state updates in recurrent neural networks. *CoRR* abs/1708.06834 (2017).

[14] Ting Chen, Ji Lin, Tian Lin, Song Han, Chong Wang, and Denny Zhou. 2018. Adaptive mixture of low-rank factorizations for compact neural modeling. (2018).

[15] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. 2015. Compressing neural networks with the hashing trick. In *Proceedings of the International Conference on Machine Learning*. 2285–2294.

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

[17] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W. Kempa-Liehr. 2018. Time series feature extraction on basis of scalable hypothesis tests (tsfresh–a python package). *Neurocomputing* 307 (2018), 72–77.

[18] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR* abs/1412.3555 (2014).

[19] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated feedback recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 2067–2075.

[20] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. 2016. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913* (2016).

[21] Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. 2016. Recurrent batch normalization. *CoRR* abs/1603.09025 (2016).

[22] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. Association for Computing Machinery, New York, NY, 221–236. DOI: https://doi.org/10.1145/3314221.3314642

[23] Edwin D. de Jong. 2016. Incremental sequence learning. *CoRR* abs/1611.03068 (2016).

[24] R. Dey and F. M. Salemt. 2017. Gate-variants of gated recurrent unit (GRU) neural networks. In *Proceedings of the IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS'17)*. 1597–1600. DOI: https://doi.org/10.1109/MWSCAS.2017.8053243

[25] Hao Dong, Akara Supratak, Luo Mai, Fangde Liu, Axel Oehmichen, Simiao Yu, and Yike Guo. 2017. TensorLayer: A versatile library for efficient deep learning development. *ACM Multimedia* (2017). Retrieved from http://tensorlayer.org.
[26] Kenji Doya. 1993. Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Trans. Neural Netw.* 1, 75 (1993), 164.
[27] Salah El Hihi and Yoshua Bengio. 1996. Hierarchical recurrent neural networks for long-term dependencies. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 493–499.
[28] Jeffrey L. Elman. 1990. Finding structure in time. *Cog. Sci.* 14, 2 (1990), 179–211.
[29] Pascale Fung, Dario Bertero, Yan Wan, Anik Dey, Ricky Ho Yin Chan, Farhad Bin Siddique, Yang Yang, Chien-Sheng Wu, and Ruixi Lin. 2016. Towards empathetic human-robot interactions. In *Proceedings of the International Conference on Intelligent Text Processing and Computational Linguistics*. Springer, 173–193.
[30] Yarin Gal and Zoubin Ghahramani. 2016. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 1019–1027.
[31] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. 2018. DeltaRNN: A power-efficient recurrent neural network accelerator. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. Association for Computing Machinery, New York, NY, 21–30. DOI: https://doi.org/10.1145/3174243.3174261
[32] Felix A. Gers and Jürgen Schmidhuber. 2000. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN'00) (Neural Computing: New Challenges and Perspectives for the New Millennium*, Vol. 3). IEEE, 189–194.
[33] C. Goller and A. Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of the International Conference on Neural Networks (ICNN'96)*. 347–352. DOI: https://doi.org/10.1109/ICNN.1996.548916
[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press. Retrieved from http://www.deeplearningbook.org.
[35] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. 2018. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1586–1595.
[36] Artem M. Grachev, Dmitry I. Ignatov, and Andrey V. Savchenko. 2017. Neural networks compression for language modeling. In *Proceedings of the International Conference on Pattern Recognition and Machine Intelligence*. Springer, 351–357.
[37] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*. ACM, 369–376.
[38] Robert M. Gray et al. 2006. Toeplitz and circulant matrices: A review. *Found. Trends® Commun. Inf. Theor* 2, 3 (2006), 155–239.
[39] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2016. LSTM: A search space odyssey. *IEEE Trans. Neural Netw. Learn. Syst.* 28, 10 (2016), 2222–2232.
[40] Danijar Hafner. 2017. Tips for Training Recurrent Neural Networks. Blog post. Retrieved from https://danijar.com/tips-for-training-recurrent-neural-networks/.
[41] Song Han and B. Dally. 2017. Efficient methods and hardware for deep learning. *University Lecture* (2017).
[42] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang et al. 2017. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA'17)*. Association for Computing Machinery, New York, NY, 75–84. DOI: https://doi.org/10.1145/3020078.3021745
[43] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[44] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 1135–1143.
[45] Liyang Hao, Siqi Liang, Jinmian Ye, and Zenglin Xu. 2018. TensorD: A tensor decomposition library in TensorFlow. *Neurocomputing* 318 (2018), 196–200.
[46] Mikael Henaff, Arthur Szlam, and Yann LeCun. 2016. Recurrent orthogonal networks and long-memory tasks. *arXiv preprint arXiv:1602.06662* (2016).
[47] Geoffrey Hinton. 2016. Training RNNs using Back propogation through time. Retrieved from https://www.youtube.com/watch?v=hTcm8AJjvfE.
[48] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[49] Zhiheng Huang and Bing Xiang. 2019. WeNet: Weighted networks for recurrent network architecture search. *CoRR* abs/1904.03819 (2019).
[50] Stephanie L. Hyland and Gunnar Rätsch. 2017. Learning unitary operators with help from u (n). In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*.
[51] Xilinx Inc. 2019. UG1327, DNNDK User Guide. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug1327-dnndk-user-guide.pdf.
[52] Herbert Jaeger. 2001. *The "Echo State" Approach to Analysing and Training Recurrent Neural Networks—with an Erratum Note*. German National Research Center for Information Technology GMD Technical Report, Bonn, Germany, 148, 34 (2001), 13.
[53] H. Jaeger. 2007. Echo state network. *Scholarpedia* 2, 9 (2007), 2330. DOI: https://doi.org/10.4249/scholarpedia.2330 revision #189893.
[54] Herbert Jaeger. 2012. *Long Short-term Memory in Echo State Networks: Details of a Simulation Study*. Technical Report. Jacobs University Bremen.
[55] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, and Udo Siewert. 2007. Optimization and applications of echo state networks with leaky-integrator neurons. *Neural Netw.* 20, 3 (2007), 335–352.
[56] Joe Jevnik. 2017. A Worked Example of Using Neural Networks for Time Series Prediction. Retrieved from https://www.youtube.com/watch?v=hAlGqT3Xpus.
[57] Xiaojie Jin, Xiaotong Yuan, Jiashi Feng, and Shuicheng Yan. 2016. Training skinny deep neural networks with iterative hard thresholding methods. *arXiv preprint arXiv:1607.05423* (2016).
[58] Li Jing, Caglar Gulcehre, John Peurifoy, Yichen Shen, Max Tegmark, Marin Soljacic, and Yoshua Bengio. 2019. Gated orthogonal recurrent units: On learning to forget. *Neural Comput.* 31, 4 (2019), 765–783.
[59] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. 2017. Tunable efficient unitary neural networks (EUNN) and their application to RNNs. In *Proceedings of the 34th International Conference on Machine Learning*. JMLR.org, 1733–1741.
[60] John J. Hopfield. 1982. Neural network and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci. United States Amer.* 79 (1982), 2554–2558.
[61] Michael Jordan. 1986. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the 8th Conference of the Cognitive Science Society*.
[62] Cijo Jose, Moustapha Cissé, and François Fleuret. 2017. Kronecker recurrent units. *CoRR* abs/1705.10142 (2017).
[63] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In *Proceedings of the International Conference on Machine Learning*. 2342–2350.
[64] Sekitoshi Kanai, Yasuhiro Fujiwara, and Sotetsu Iwamura. 2017. Preventing gradient explosions in gated recurrent units. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 435–444.
[65] Andrej Karpathy. 2015. Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
[66] Valentin Khrulkov, Alexander Novikov, and Ivan V. Oseledets. 2017. Expressive power of recurrent neural networks. *CoRR* abs/1711.00811 (2017).
[67] Tamara G. Kolda and Brett W. Bader. 2009. Tensor decompositions and applications. *SIAM Rev.* 51, 3 (2009), 455–500.
[68] J. F. Kolen and S. C. Kremer. 2001. *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*. IEEE. DOI: https://doi.org/10.1109/9780470544037.ch14
[69] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo Larochelle, Aaron C. Courville, and Chris Pal. 2016. Zoneout: Regularizing RNNs by randomly preserving hidden activations. *CoRR* abs/1606.01305 (2016).
[70] David Krueger and Roland Memisevic. 2015. Regularizing rnns by stabilizing activations. *arXiv preprint arXiv:1511.08400* (2015).
[71] Ankit Kumar, Ozan Irsoy, Peter Ondruska, Mohit Iyyer, James Bradbury, Ishaan Gulrajani, Victor Zhong, Romain Paulus, and Richard Socher. 2016. Ask me anything: Dynamic memory networks for natural language processing. In *Proceedings of the International Conference on Machine Learning*. 1378–1387.
[72] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 9017–9028.
[73] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. 2016. DeepX: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'16)*. 1–12. DOI: https://doi.org/10.1109/IPSN.2016.7460664
[74] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar. 2017. Squeezing deep learning into mobile and embedded devices. *IEEE Pervas. Comput.* 16, 3 (2017), 82–88. DOI: https://doi.org/10.1109/MPRV.2017.2940968

[75] Quoc V. Le, Navdeep Jaitly, and Geoffrey E. Hinton. 2015. A simple way to initialize recurrent networks of rectified linear units. *CoRR* abs/1504.00941 (2015).
[76] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 2012. Efficient backprop. In *Neural Networks: Tricks of the Trade*. Springer, 9–48.
[77] Zachary C. Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019* (2015).
[78] Zachary C. Lipton, David C. Kale, Charles Elkan, and Randall Wetzel. 2015. Learning to diagnose with LSTM recurrent neural networks. *arXiv preprint arXiv:1511.03677* (2015).
[79] Jiasen Lu, Caiming Xiong, Devi Parikh, and Richard Socher. 2017. Knowing when to look: Adaptive attention via a visual sentinel for image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.
[80] Zhiyun Lu, Vikas Sindhwani, and Tara N. Sainath. 2016. Learning compact recurrent neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'16)*. IEEE, 5960–5964.
[81] Tomas Lundin and Perry Moerland. 1997. *Quantization and Pruning of Multilayer Perceptrons: Towards Compact Neural Networks*. Technical Report. IDIAP.
[82] James Martens and Ilya Sutskever. 2011. Learning recurrent neural networks with Hessian-free optimization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning (ICML'11)*. Omnipress, 1033–1040. DOI: https://doi.org/10.5555/3104482.3104612
[83] Gábor Melis, Chris Dyer, and Phil Blunsom. 2018. On the state of the art of evaluation in neural language models. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=ByJHuTgA-.
[84] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).
[85] Stephen Merity, Bryan McCann, and Richard Socher. 2017. Revisiting activation regularization for language RNNs. *arXiv preprint arXiv:1708.01009* (2017).
[86] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. 2016. Efficient Orthogonal Parametrisation of Recurrent Neural Networks Using Householder Reflections. arXiv:cs.LG/1612.00188 (2016).
[87] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. 2017. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning*. JMLR.org, 2401–2409.
[88] Tomáš Mikolov. 2012. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April* 80 (2012).
[89] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. 2014. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753* (2014).
[90] Tomas Mikolov and Geoffrey Zweig. 2012. Context dependent recurrent neural network language model. In *Proceedings of the IEEE Spoken Language Technology Workshop (SLT'12)*. IEEE, 234–239.
[91] Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. 2017. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119* (2017).
[92] Sharan Narang, Eric Undersander, and Gregory Diamos. 2017. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782* (2017).
[93] Andrew Ng. 2011. Advice for applying machine learning. In *Mach. Learn.*
[94] Andrew Ng. 2017. *Machine Learning Yearning*. Online Draft. Retrieved from http://www.mlyearning.org/,/bib/ng/ng2017mlyearning/Ng_MLY01_13.pdf.
[95] Andrew Ng. 2017. Bias Variance Tradeoff. Retrieved from https://www.youtube.com/watch?v=SjQyLhQIXSM.
[96] Linh Nguyen, Peifeng Yu, and Mosharaf Chowdhury. 2017. No! Not another deep learning framework. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*. Association for Computing Machinery, New York, NY, 88–93. DOI: https://doi.org/10.1145/3102980.3102995
[97] Alexander Novikov, Pavel Izmailov, Valentin Khrulkov, Michael Figurnov, and Ivan V. Oseledets. 2018. Tensor train decomposition on TensorFlow (T3F). *CoRR* abs/1801.01928 (2018).
[98] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. 2015. Tensorizing neural networks. *CoRR* abs/1509.06569 (2015).
[99] Alexander Novikov, Anton Rodomanov, Anton Osokin, and Dmitry Vetrov. 2014. Putting MRFs on a tensor train. In *Proceedings of the International Conference on Machine Learning*. 811–819.
[100] Guillaume Olikier, Pierre-Antoine Absil, Lieven De Lathauwer, and Yurii NESTEROV. [n.d.]. *Tensor Approximation by Block Term Decomposition*. Ph.D. Dissertation. Master's thesis, Ecole Polytechnique de Louvain, Université catholique de Louvain.

[101] Yu Pan, Jing Xu, Maolin Wang, Jinmian Ye, Fei Wang, Kun Bai, and Zenglin Xu. 2019. Compressing recurrent neural networks with tensor ring for action recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 4683–4690.

[102] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 311–318.

[103] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2012. Understanding the exploding gradient problem. *ArXiv* abs/1211.5063 (2012).

[104] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML'13)*. JMLR.org, 1310–1318. DOI: https://doi.org/10.5555/3042817.3043083

[105] R. K. Pathinarupothi, D. P. J, E. S. Rangan, G. E. A, V. R, and K. P. Soman. 2017. Single sensor techniques for sleep apnea diagnosis using deep learning. In *Proceedings of the IEEE International Conference on Healthcare Informatics (ICHI'17)*. 524–529. DOI: https://doi.org/10.1109/ICHI.2017.37

[106] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. *CoRR* abs/1802.03268 (2018).

[107] Eric T. Phipps and Tamara G. Kolda. 2018. Software for sparse tensor decomposition on emerging computing architectures. *CoRR* abs/1809.09175 (2018).

[108] Rohit Prabhavalkar, Ouais Alsharif, Antoine Bruguier, and Lan McGraw. 2016. On the compression of recurrent neural networks with an application to LVCSR acoustic modeling for embedded speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'16)*. IEEE, 5970–5974.

[109] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural Netw.* 12, 1 (Jan. 1999), 145–151. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6

[110] Samyam Rajbhandari, Harsh Shrivastava, and Yuxiong He. 2018. AntMan: Sparse low-rank compression to accelerate RNN inference. (2018).

[111] Sujith Ravi. 2017. ProjectionNet: Learning efficient on-device deep networks using neural projections. *CoRR* abs/1708.00630 (2017).

[112] Jorma Rissanen. 1978. Modeling by shortest data description. *Automatica* 14, 5 (1978), 465–471.

[113] 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. Millán. 2010. Collecting complex activity datasets in highly rich networked sensor environments. In *Proceedings of the 7th International Conference on Networked Sensing Systems (INSS'10)*. 233–240. DOI: https://doi.org/10.1109/INSS.2010.5573462

[114] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *CoRR* abs/1609.04747 (2016).

[115] David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams et al. 1988. Learning representations by back-propagating errors. *Cog. Model.* 5, 3 (1988), 1.

[116] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the 15th Conference of the International Speech Communication Association*.

[117] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. 2013. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120* (2013).

[118] Martin Schrimpf, Stephen Merity, James Bradbury, and Richard Socher. 2017. A flexible approach to automated RNN architecture generation. *CoRR* abs/1712.07316 (2017).

[119] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Trans. Sig. Proc.* 45, 11 (1997), 2673–2681.

[120] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. 2016. Recurrent dropout without memory loss. *CoRR* abs/1603.05118 (2016).

[121] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. 2015. Structured transforms for small-footprint deep learning. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 3088–3096.

[122] Matthew Sotoudeh and Sara S. Baghsorkhi. 2018. DeepThin: A self-compressing library for deep neural networks. *CoRR* abs/1802.06944 (2018).

[123] Gilbert Strang. 2009. *Introduction to Linear Algebra* (4 ed.). Wellesley-Cambridge Press, Wellesley, MA.

[124] Ilya Sutskever and Geoffrey Hinton. 2010. Temporal-kernel recurrent neural networks. *Neural Netw.* 23, 2 (2010), 239–243.

[125] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *Proceedings of the International Conference on Machine Learning*. 1139–1147.

[126] Igor Szoke, Petr Schwarz, Pavel Matejka, Lukás Burget, Martin Karafiát, Michal Fapso, and Jan Cernocky. 2005. Comparison of keyword spotting approaches for informal continuous speech. In *Proceedings of the 9th European Conference on Speech Communication and Technology*.

[127] TensorFlow. 2018. Explore overfit and underfit. Tutorial. Retrieved from https://www.tensorflow.org/tutorials/keras/overfit_and_underfit.

[128] Urmish Thakker, Jesse Beu, Dibakar Gope, Chu Zhou, Igor Fedorov, Ganesh Dasika, and Matthew Mattina. 2019. Compressing RNNs for IoT devices by 15–38x using Kronecker Products. *arXiv preprint arXiv:1906.02876* (2019).

[129] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

[130] Jakob Uszkoreit. 2017. Transformer: A novel neural network architecture for language understanding. *Google Research Blog* (2017).

[131] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. 2017. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *IEEE Trans. Pattern Anal. Mach. Intell.* 39, 4 (Apr. 2017), 652–663. DOI: https://doi.org/10.1109/TPAMI.2016.2587640

[132] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. 2017. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning*. JMLR.org, 3570–3578.

[133] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of neural networks using dropconnect. In *Proceedings of the International Conference on Machine Learning*. 1058–1066.

[134] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. 2017. Residual attention network for image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[135] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Yanzhi Wang, Qinru Qiu, and Yun Liang. 2018. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. *CoRR* abs/1803.06305 (2018).

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

[137] P. J. Werbos. 1990. Backpropagation through time: What it does and how to do it. *Proc. IEEE* 78, 10 (Oct. 1990), 1550–1560. DOI: https://doi.org/10.1109/5.58337

[138] Scott Wisdom, Thomas Powers, John Hershey, Jonathan Le Roux, and Les Atlas. 2016. Full-capacity unitary recurrent neural networks. In *Proceedings of the International Conference on Advances in Neural Information Processing Systems*. 4880–4888.

[139] 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, Łukasz 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. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv:cs.CL/1609.08144 (2016).

[140] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International Conference on Machine Learning*. 2048–2057.

[141] Yinchong Yang, Denis Krompass, and Volker Tresp. 2017. Tensor-train recurrent neural networks for video classification. In *Proceedings of the 34th International Conference on Machine Learning*. JMLR.org, 3891–3900.

[142] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. 2017. Breaking the Softmax Bottleneck: A high-rank RNN language model. *CoRR* abs/1711.03953 (2017).

[143] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek F. Abdelzaher. 2017. Compressing deep neural network structures for sensing systems with a compressor-critic framework. *CoRR* abs/1706.01215 (2017).

[144] Jinmian Ye, Linnan Wang, Guangxi Li, Di Chen, Shandian Zhe, Xinqi Chu, and Zenglin Xu. 2018. Learning compact recurrent neural networks with block-term tensor decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 9378–9387.

[145] Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. 2017. Long-term forecasting using tensor-train RNNs. *arXiv preprint arXiv:1711.00073* (2017).

[146] Xiaotong Yuan, Ping Li, and Tong Zhang. 2014. Gradient hard thresholding pursuit for sparsity-constrained optimization. In *Proceedings of the International Conference on Machine Learning*. 127–135.

[147] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[148] Klaus Zechner and Alex Waibel. 2000. Minimizing word error rate in textual summaries of spoken language. In *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics*.

[149] H. B. Zhang et al. 2019. A comprehensive survey of vision-based human action recognition methods. *Sensors* 19 5 (2019).

[150] Jiong Zhang, Qi Lei, and Inderjit S. Dhillon. 2018. Stabilizing gradients for deep neural networks via efficient SVD parameterization. *arXiv preprint arXiv:1803.09327* (2018).

[151] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. 2016. Tensor ring decomposition. *CoRR* abs/1606.05535 (2016).

[152] Feiwen Zhu, Jeff Pool, Michael Andersch, Jeremy Appleyard, and Fung Xie. 2018. Sparse persistent RNNs: Squeezing large recurrent networks on-chip. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=HkxF5RgC-

[153] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: Exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).

[154] Anupriya Shrivastava, J. Amudha, Deepa Gupta, and Kshitij Sharma. 2019. Deep learning model for text recognition in images. In *10th International Conference on Computing, Communication and Networking Technologies (ICCCNT'19)*. 1–6. DOI: 10.1109/ICCCNT45670.2019.8944593

[155] C. V. Amrutha, C. Jyotsna, and J. Amudha. 2020. Deep learning approach for suspicious activity detection from surveillance video. In *2nd International Conference on Innovative Mechanisms for Industry Applications (ICIMIA'20)*. 335–339. DOI: 10.1109/ICIMIA48430.2020.9074920

Received December 2019; revised December 2020; accepted February 2021