algorithms

MDPI

Article

A Systematic Evaluation of Recurrent Neural Network Models for Edge Intelligence and Human Activity Recognition Applications

Varsha S. Lalapura [1,*], Veerender Reddy Bhimavarapu [1], J. Amudha [2] and Hariram Selvamurugan Satheesh [3]

[1] Department of Electronics and Communication Engineering, Amrita School of Engineering, Amrita Vishwa Vidyapeetham, Bengaluru 560035, Karnataka, India; b_veerender@blr.amrita.edu
[2] Department of Computer Science and Engineering, Amrita School of Computing, Amrita Vishwa Vidyapeetham, Bengaluru 560035, Karnataka, India; j_amudha@blr.amrita.edu
[3] ABB Global Industries and Services Private Limited, Bengaluru 560048, Karnataka, India; hariram.satheesh@in.abb.com
* Correspondence: s_varshalalapura@blr.amrita.edu

Abstract: The Recurrent Neural Networks (RNNs) are an essential class of supervised learning algorithms. Complex tasks like speech recognition, machine translation, sentiment classification, weather prediction, etc., are now performed by well-trained RNNs. Local or cloud-based GPU machines are used to train them. However, inference is now shifting to miniature, mobile, IoT devices and even micro-controllers. Due to their colossal memory and computing requirements, mapping RNNs directly onto resource-constrained platforms is arcane and challenging. The efficacy of *edge-intelligent* RNNs (EI-RNNs) must satisfy both performance and memory-fitting requirements at the same time without compromising one for the other. This study's aim was to provide an empirical evaluation and optimization of historic as well as recent RNN architectures for high-performance and low-memory footprint goals. We focused on Human Activity Recognition (HAR) tasks based on wearable sensor data for embedded healthcare applications. We evaluated and optimized six different recurrent units, namely Vanilla RNNs, Long Short-Term Memory (LSTM) units, Gated Recurrent Units (GRUs), Fast Gated Recurrent Neural Networks (FGRNNs), Fast Recurrent Neural Networks (FRNNs), and Unitary Gated Recurrent Neural Networks (UGRNNs) on eight publicly available time-series HAR datasets. We used the hold-out and cross-validation protocols for training the RNNs. We used low-rank parameterization, iterative hard thresholding, and spare retraining compression for RNNs. We found that efficient training (i.e., dataset handling and preprocessing procedures, hyperparameter tuning, and so on, and suitable compression methods (like low-rank parameterization and iterative pruning) are critical in optimizing RNNs for performance and memory efficiency. We implemented the inference of the optimized models on Raspberry Pi.

Keywords: Recurrent Neural Networks; hyperparameter tuning; compression; weights and biases (wandb); sparsity; low rank

Citation: Lalapura, V.S.; Bhimavarapu, V.R.; Amudha, J.; Satheesh, H.S. A Systematic Evaluation of Recurrent Neural Network Models for Edge Intelligence and Human Activity Recognition Applications. *Algorithms* **2024**, *17*, 104. https://doi.org/10.3390/a17030104

Academic Editor: Frank Werner

Received: 5 January 2024
Revised: 30 January 2024
Accepted: 12 February 2024
Published: 28 February 2024

1. Introduction

Nowadays, data-dominated technologies like Deep Learning (DL) are thriving and profoundly crossing technological barriers in the era of Artificial Intelligence (AI). Their application domains are large (speech processing, natural language processing, creative and art, AI for Earth, healthcare, human–computer interaction) and so are their architectural complexities and following training and inference procedures [1–4].

Sequence modeling is an important subclass of machine learning problem. Sequence data involve a notion of time, and the learning of models such as RNNs that incorporate this aspect are far more complex than feed-forward, parallelizable, and spatial counterparts like Convolutional Neural Networks (CNNs) [5–7]. RNNs can process input data one at a

time and remember information through their structure and hidden activations [8]. They have massive computation and memory budgets and are difficult to train [9]. There are four significant challenges in RNN training that must be addressed to meet performance needs [4]. They are (1) vanishing gradients problem, (2) exploding gradients problem, (3) handling long-range dependencies, and (4) model fitting and generalization issues. Though there are a wide range of solutions [10,11] available to combat their training challenges in the RNN literature, so they are still a problematic choice [4,12].

1.1. Motivation and Challenges

Deep learning is penetrating the Internet of Things (IoT) paradigm, and interest in mapping trained models to edge devices has recently surged. Microcontrollers (MCUs) have even found applications in keyword spotting or even video applications [13]. Of the RNNs, Long Short-Term Memory (LSTM) networks are one of the most popular variants of RNNs. LSTM networks are known for their long-term memory capability, and they also overcome the most common RNN training challenges, the vanishing and exploding gradients problems. However, they are not readily deployable models for the edge. LSTM networks are memory and compute-intensive. As an illustration, LSTM for a speech recognition system may consist of an input vector of length 153, an output vector of length 512, 1024 hidden units, and 2 layers. From the example shown in Tables 1 and 2, numbers marked in red show that the computation and memory needs are huge for the LSTM chosen. To cater to such hardware needs [14] and be capable of accurately performing the trained task, RNNS have to be simplified or compressed.

Table 1. Memory requirements of Long Short-Term Memory-based speech model.

Computation	Recurrent Weights	Recurrent Nodes	Nonrecurrent Weights	Nonrecurrent Nodes	Peep-Hole Diagonal Weights	Peep-Hole Nodes	Bias
$f_t = [W_{fh}.r_{t-1} + W_{fx}.x_t + W_{fc}.c_{t-1} + b_f]$	W_{fr}	r_{t-1}	W_{fx}	x_t	W_{fc}	c_{t-1}	b_f
Weight Matrix Dimensions	[1024,512]	[512,1]	[1024,153]	[153,1]	[1024,1]	[1024,1]	[1024,1]
$i_t = [W_{ih}.r_{t-1} + W_{ix}.x_t + W_{ic}.c_{t-1} + b_f]$	[1024,512]	[512,1]	[1024,153]	[153,1]	[1024,1]	[1024,1]	[1024,1]
$g_t = [W_{ch}.r_{t-1} + W_{cx}.x_t + b_c]$	[1024,512]	[512,1]	[1024,153]	[153,1]	-	-	[1024,1]
$c_t = f_t \odot c_{t-1} + g_t \odot i_t - 1$	No weights, only element-wise multiplications, element-wise additions						
$ot = [W_{oh}.r_{t-1} + W_{ox}.x_t + W_{oc}.c_{t-1} + b_o]$	[1024,512]	[512,1]	[1024,153]	[153,1]	[1024,1]	[1024,1]	[1024,1]
$h_t = o_t \odot h(c_t)$	No weights, only element-wise multiplications						
$r_t = W_{rh}.h_t$	[512,1024]	-					
Number of parameters stored in memory	3248128						

In practice, there has been a fundamental understanding of how the Recurrent Neural Networks like the LSTM networks [15,16], GRUs [17–19], Unitary RNNs [20] and so on show outstanding performance, but their portability on edge devices remains insufficient. The exact training procedures, compression, and evaluation methods to host RNN models on edge devices need to be created and benchmarked.

From an implementation standpoint, we are in the transition period of ML. The transition is clear in four important frontiers: *data*, *models*, *frameworks*, and *devices*. We have mant DL tools available for training (TensorFlow, Theano, PyTorch, Caffe, and so on) and RNN-based source code repositories. However, the core component or "labeled data" is still proprietary. Furthermore, we have interpreter-based frameworks (Google TensorFlow Lite Micro [21]) and compiler-based frameworks (Microsoft ELL library [22]) as *inference frameworks* on which to run DL models on embedded System-on-Chip (SoC) devices and MCUs. Conversely, there has been no unified framework that particularly caters edge

ML models or applications in products. Ref. [21] provides a list of various issues with existing frameworks for edge mapping. For an RNN, SRAM is related to the activation size (read and write), whereas Flash is related to the model size (read only) [23]. If the device's external memory is also chosen, memory accesses have adverse implications on inference speed and power consumption.

Table 2. Computations involved in one Long Short-Term Memory cell for speech recognition.

Computation	MACs	Muls (Elem. Wise)	Adds (Adder Tree)
f_t	680960	1024	3072
i_t	680960	1024	3072
g_t	680960	0	2048
c_t	0	2048	1024
o_t	680960	1024	3072
h_t	0	1024	0
r_t	524288	0	0
1 LSTM cell computation	3248128	6144	12,288

This study aimed to evaluate and optimize RNNs to target edge platforms. We chose HAR since wearable-sensors-based systems are essential in many healthcare monitoring applications like elderly care support, fitness tracking, sleep quality assessment, predictive health, etc. Edge-based and adept HAR systems can replace expensive healthcare monitors if the solutions are accurate, reliable, low-powered, and small. However, the lacks of standard workflows and differences in evaluation protocols, evaluation metrics [24], data generation methods, and their quality make comparison of different approaches a challenging task and do not allow fair comparison of results [25,26].

1.2. Contributions and Key Features

1. We present a comprehensive application of device mapping research workflow that can be commonly adapted for optimizing RNN models onto a resource-constrained edge (see Section 3). This will help with reducing the research time required for methodical workflows in this context.
2. We focused on the HAR-based EI-RNN evaluation and optimization of five different RNN units sandboxed apart from the classic LSTM structures (see Sections 2.1–2.6).
3. We used eight different HAR datasets (see Tables 3 and 4, Section 4.2.9 for important details) and two evaluation methods, namely, the hold-out method and cross-validation method.
4. We conducted an in-depth performance analysis of both training and compression techniques applied to RNNs (see Sections 5.1 and 5.2). To the best of our knowledge, the exact details of both these critical aspects have rarely been presented with clarity from an implementation point of view.
5. The key takeaways based on this empirical evaluation study will be important for practitioners and researchers in this problem domain.

We describe the background and related work with respect to RNNs, their training challenges, and compression schemes for edge mapping studies in Section 2. We then present our research workflow that captured the essential steps to be carried out for mapping any application to a device via RNN modeling and evaluation in Section 3. Furthermoere, we present the experimental details covering the dataset description, evaluation methods and metrics used in Section 4. We then discuss EI-RNN modeling and provide an analysis in Section 5, which includes training, compression and inference. Next, we discuss

the results obtained and present the key takeaways in Section 6. Finally, we present the conclusions drawn in Section 7 including the future scope of the research problem.

2. Background and Related Work

RNNs are neural networks that allow information from both the past and present to be processed while having hidden states. Mathematically, the fundamental RNN equations take different forms based on the structure of the gates, residual connections, and memory components. In this study, we focused on six different RNN units, namely, Vanilla Recurrent Neural Networks (RNNs) [27], Long Short-Term Memory (LSTM) units [15,16], Gated Recurrent Units (GRUs) [17–19], Fast Recurrent Neural Networks (FRNNs) [28], Fast Gated Recurrent Neural Networks (FGRNNs) [28], and (Unitary Gated Recurrent Neural Networks (UGRNNs) [20]. Structurally, these RNN units can be stacked, peep-holed, have residual connections or projections, can be bidirectional, and can be attention-based depending on the complexity of the task they intend to learn. The following are the RNNs we used in our study.

2.1. Vanilla RNN

This is the very basic form of a Recurrent Neural Network. Figure 1 shows the internal structure of the Vanilla RNN. Mathematically, Equations (1) and (2) represent its time sequencing behavior. The input feature vector x_t is associated with the nonrecurrent weight matrix W_{hx}, and the activations from previous time step h_{t-1} are associated with recurrent weight matrix U_{hh}. The output vector y_t is the associated weight matrix W_{yh}. $f1$ is a nonlinear function like a *tanh* or *sigmoid* nonlinearity. $f2$ can be a softmax output nonlinearity.

$$h_t = f1\left[U_{hh}h_{t-1} + W_{hx}x_t + b_h\right] \quad (1)$$

$$y_t = f2\left[W_{yh}h_t + b_y\right] \quad (2)$$

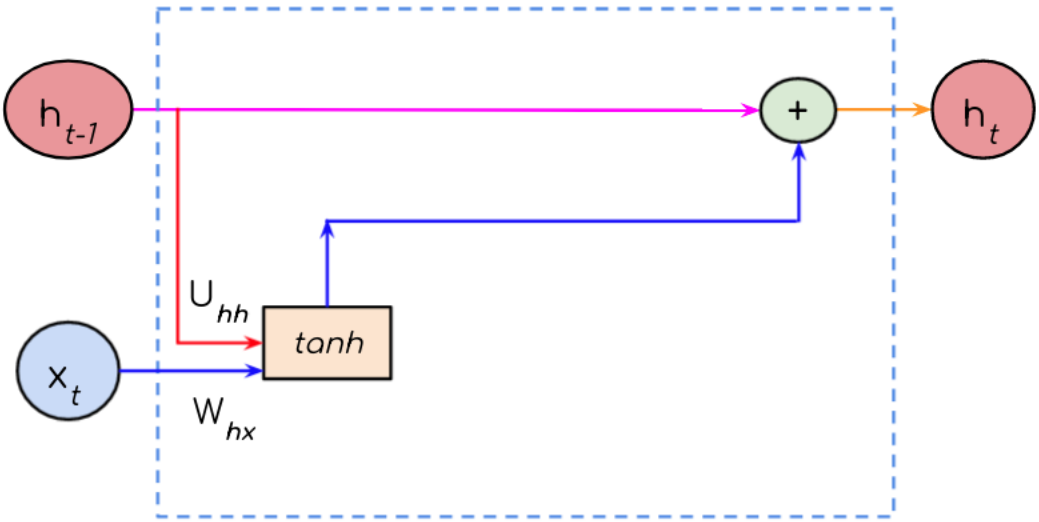


Figure 1. Internal structure of a Vanilla Recurrent Neural Network Unit.

2.2. LSTM RNN

In an LSTM RNN, the recurrent cell is modified into the reset gate for Equation (3), forget gate for Equation (4), candidate memory cell for Equation (6), and output gate Equation (7). The internal diagram of an LSTM cell is shown in Figure 2.

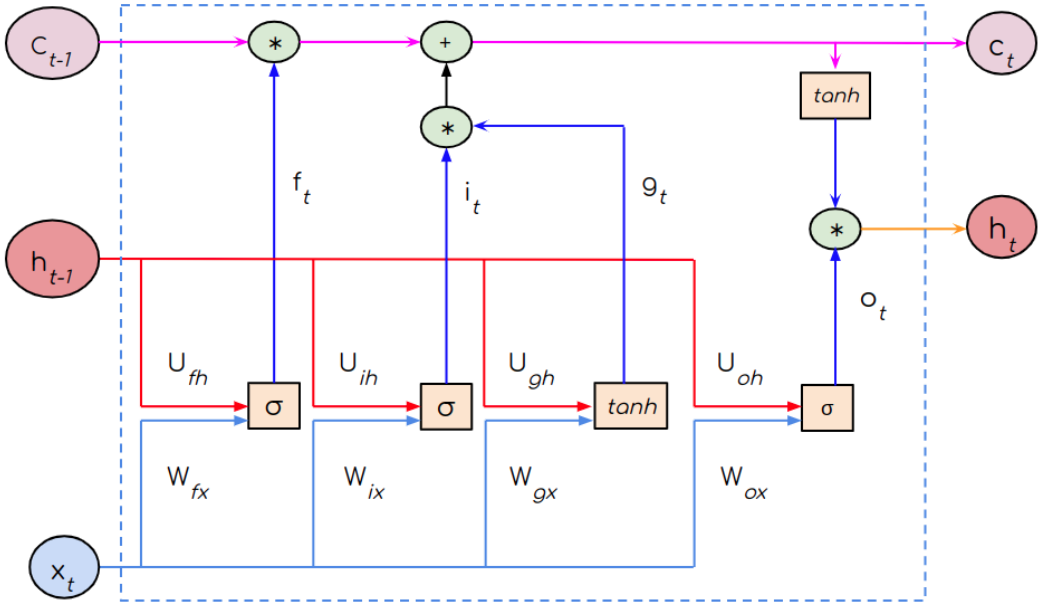


Figure 2. Internal structure of Long Short-Term Memory Recurrent Neural Network Unit.

$$\textit{input gate, } i_t = \sigma\left[U^l_{ih}\, h^l_{t-1} + W^l_{ix}\, x^l_t + diag[p^l_i]\, c_{t-1} + b_i\right] \tag{3}$$

$$\textit{forget gate, } f_t = \sigma\left[U^l_{fh}\, h^l_{t-1} + W^l_{fx}\, x^l_t + diag[p^l_f]\, c_{t-1} + b_f\right] \tag{4}$$

$$g_t = \tanh\left[U^l_c\, h^l_{t-1} + W^l_{cx}\, x^l_t + b^l_c\right] \tag{5}$$

$$\textit{memory cell, } c_t = f^l_t \bigodot c^l_{t-1} + i^l_t \bigodot g^l_t \tag{6}$$

$$\textit{output gate, } o_t = \sigma\left[U^l_{oh}\, h^l_{t-1} + W^l_{ox}\, x^l_t + diag[p^l_o]\, c_{t-1} + b^l_o\right] \tag{7}$$

$$\textit{recurrent hidden state, } h_t = o^l_t \bigodot \tanh c^l_t \tag{8}$$

$$\textit{where, } x^l_t = \begin{cases} h^{l-1}_t\ (\textit{hidden temporal features}), & l > 1 \\ \textit{input features}, & l = 1 \end{cases} \tag{9}$$

$$\textit{and final output at the last layer will be} \quad y_t = \phi[W_{yh} h_t + b_y] \tag{10}$$

ϕ represents the output activation function, for example, a softmax activation. $\bigodot$ represents element-wise multiplication. Peep-hole connections ($diag[p]$) are optional for each of the gates. In all our experiments, we used single-layer structures without any peep-hole connections. The recurrent weight matrix U_{hh} of an LSTM is obtained by vertically stacking the four-gate, recurrent weight matrices as $[U_{ih}, U_{oh}, U_{fh}, U_{ch}]^T$; likewise, the nonrecurrent weight matrix W_{hx} is obtained by vertically stacking the four-gate, nonrecurrent weight matrices as $[W_{ix}, W_{ox}, W_{fx}, W_{cx}]^T$.

2.3. GRU

In a Gated Recurrent Unit, the recurrent cell is modified as input and update gates to control what information to allow from the current input (via update gate) and what information to reset from the previous state (via the reset gate). The reset and update gates are described by Equation (11) and Equation (12), respectively. Unlike LSTM, however, it has no cell memory passed to the next time step separately from the output hidden state. The internal diagram of a GRU cell is shown in Figure 3.

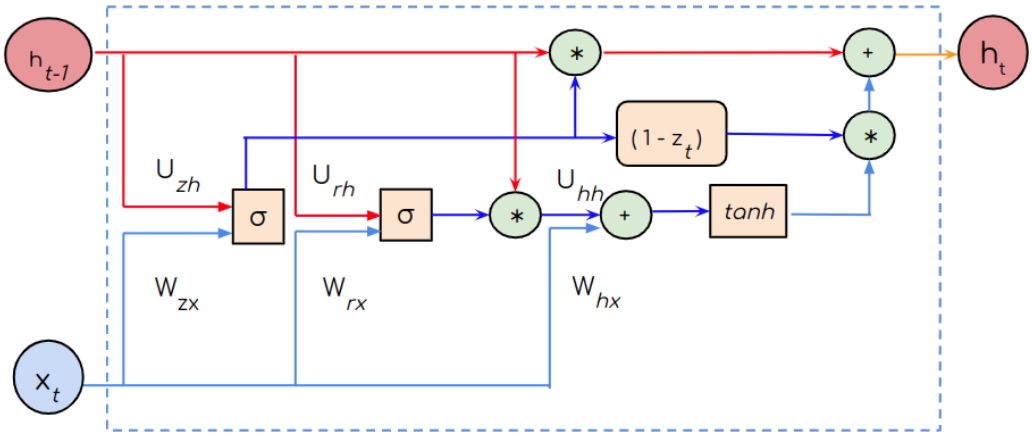


Figure 3. Internal structure of Gated Recurrent Unit.

$$\textit{reset gate, } r_t = \sigma\left[U_{rh}\, h_{t-1} + W_{rx}\, x_t + b_r\right] \tag{11}$$

$$\textit{update gate, } z_t = \sigma\left[U_{zh}\, h_{t-1} + W_{zx}\, x_t + b_z\right] \tag{12}$$

$$\tilde{h}_t = \tanh\left[U_{hh}(r_t \odot h_{t-1}) + W_{hx}\, x_t + b_h\right] \tag{13}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{14}$$

2.4. FGRNN

LSTM and GRU address the vanishing and exploding gradient problems. However, they are intense in terms of computation and memory complexities. Reference [28] describes a tiny yet efficient RNN unit that is simple in both computation and memory spaces and addresses the vanishing and exploding gradient problems. The RNN was named FGRNN since it is a gated version of the Vanilla RNN and is faster to train. The internal diagram of an FGRNN cell is shown in Figure 4. There are just two additional scalar hyperparameters, namely, ζ and ν, in the RNN cell structure, as shown in Equation (17).

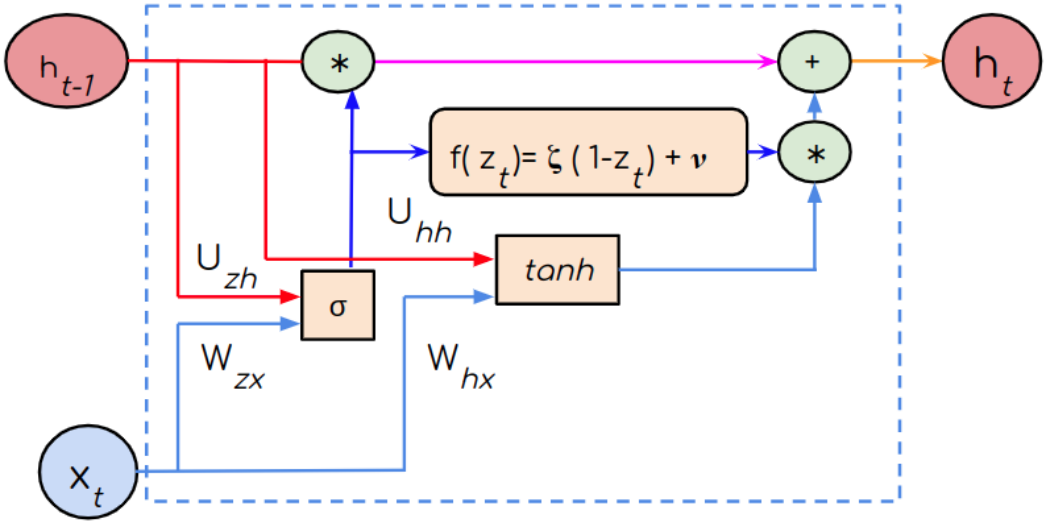


Figure 4. Internal structure of Fast Gated Recurrent Neural Network Unit.

$$z_t = \sigma\left[U_{zh}\, h_{t-1} + W_{zx}\, x_t + b_z\right] \tag{15}$$

$$\tilde{h}_t = \tanh\left[U_{hh}\, h_{t-1} + W_{hx}\, x_t + b_h\right] \tag{16}$$

$$h_t = z_t \odot h_{t-1} + (\zeta(1 - z_t) + \nu) \odot \tilde{h}_t \tag{17}$$

2.5. FRNN

Reference [28] proposes another RNN unit called Fast RNN (FRNN) with two additional scalar hyperparameters called α and β, as shown in Equation (19). Like the FGRNN, this RNN also provides stable training and is simple in its gating structure. The details of the architecture are shown in Figure 5.

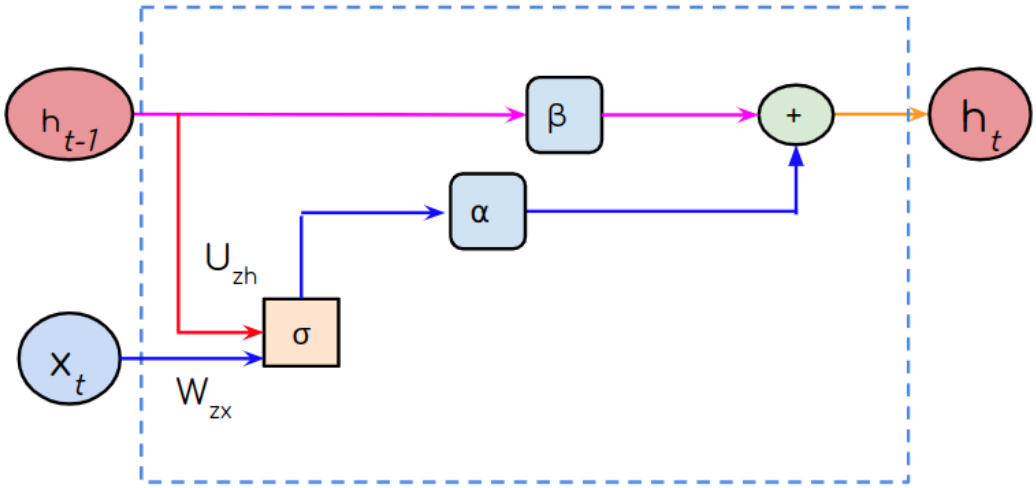


Figure 5. Internal structure of Fast Recurrent Neural Network Unit.

$$\tilde{h}_t = \tanh\left[U_{hh}\, h_{t-1} + W_{hx}\, x_t + b_h\right] \tag{18}$$

$$h_t = \alpha \odot \tilde{h}_t + \beta \odot h_{t-1} \tag{19}$$

2.6. UGRNN

Unitary Gated RNNs or UGRNNs provide stable training and address the vanishing and exploding gradient problems by limiting the range of the singular values of the hidden state transition matrix. This structure leads to higher RNN prediction accuracy but has impacted the model size compared to FGRNNs and FRNNs. However, they are simpler than LSTM and GRU, as shown in Figure 6. The internal gating structure follows Equations (20)–(22).

$$z_t = \sigma\left[U_{zh}\, h_{t-1} + W_{zx}\, x_t + b_z\right] \tag{20}$$

$$\tilde{h}_t = \tanh\left[U_{hh}\, h_{t-1} + W_{hx}\, x_t + b_h\right] \tag{21}$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \tag{22}$$

Porting dense and architecturally complex RNNs on resource-constrained edge devices is not directly feasible. Memory and computation spaces on edge devices are far smaller than those of CPU, GPU, or cloud-based machines. Compressing neural networks is one of the solutions to this problem.

The existing studies have employed Neural Architecture Search (NAS), pruning [29,30], and quantization [14] for edge-based DL deployment. TinyNAS and TinyEngine [13] as CNN models' MCUs have been recent and popular advancements. Pruning is a common and efficient strategy used for model compression. Pruning refers to removing redundant connections in a network [31] while retaining the accuracy level of the task defined. This results in sparse [32] and irregular data flow computations from a hardware perspective [33]. Techniques that shrink the network without such side effects have been beneficial [34,35]. Therefore, the weight matrices become structured (blocked) [36,37] or unstructured [14] based on the pruning technique.

Low-Rank Matrix Factorization (LMF) expresses a base matrix A of dimension m × n as a product of two smaller matrices U and V of dimensions m × d and d × n, respectively.

Parameter d controls the compression factor [38]. The low-rank parameterization of weight matrices has resulted in smaller models while producing mixed results in maintaining model accuracy. A novel optimization algorithm via low-rank constraint and sparsity projection is discussed in Ref. [39]. The low-rank and diagonalization of weight matrices were adopted in Ref [40]. Ref. [41] studied mechanisms for learning compact RNNs and LSTMs via low-rank factorizations and parameter-sharing schemes. Ref. [42] employed the Singular Value Decomposition (SVD) technique on a recurrent projection matrix of a speech model to map onto an embedded platform. Ref. [41] applied low-rank factorization and projection compression technique on large-scale vocabulary speech signal modeling at a small cost of 0.3% increase in Word Error Rate (WER) but a significant 75% decrease in model parameters.

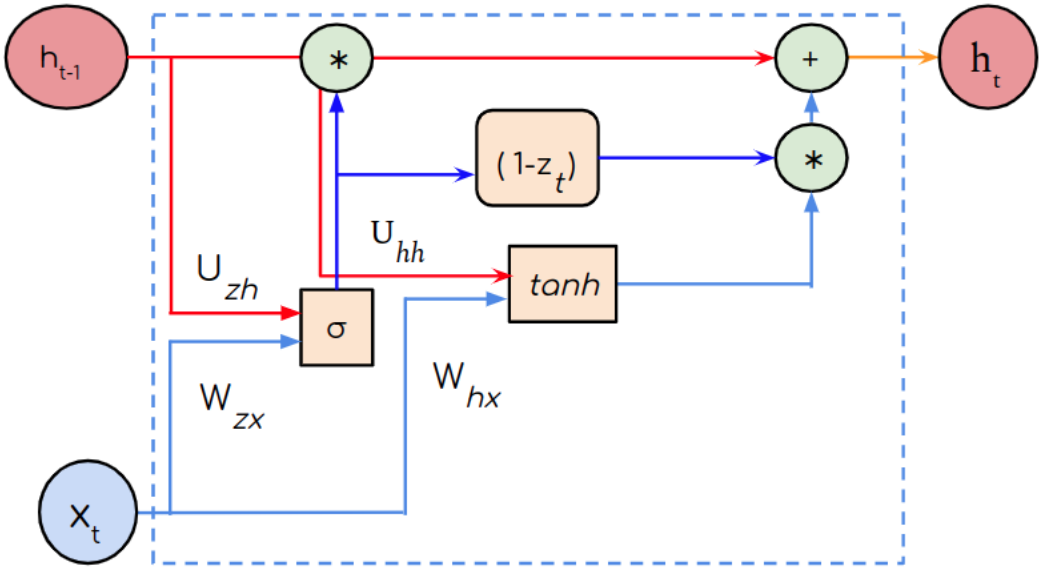


Figure 6. Internal structure of Unitary Gated Recurrent Neural Network Unit.

Quantization refers to tuning the precision of the weights and activations of a network to lower levels while retaining accuracy. It is a very common step in memory optimization for RNNs. Ref. [29] employed quantization of weights using the k-means clustering mechanism. Ref. [43] compared floating to fixed point behavior of neural network parameters and their impact on speedup for a speech recognition task. The details of various RNN algorithmic optimizations for edge-based inference are discussed in Section V of Ref. [4]. Quantization, pruning, and tensor decomposition methods are some of the common methods used for compressing RNNS and CNNs [44].

3. Research Workflow

The lack of consensus over methodical data handling procedures, training mechanisms, and rapidly evolving tools are the main reasons that have hindered the adaptation of RNN-based HAR models and have sometimes led to incorrect results. To be able to run the newest, best models on the most commonly used frameworks that have constantly changing landscapes requires paramount effort and skill [21,45]. Figure 7 gives an overview of the steps involved in mapping a HAR-based RNN model onto a resource-constrained edge. This workflow is agnostic of the application, architecture, and DL framework. This is an iterative procedure used to fetch light model solutions for an edge device. Reference [26,46] also used workflow designs similar to ours, but we used RNNs for training, and we did not evaluate power as a performance metric. In the context of HAR, or any other application, data collection and preprocessing are the most crucial components in the workflow and the most time consuming. Even though we represent it as a single block in the workflow, many details must be carefully addressed. We provide as many details as possible regarding the

finer details of the datasets (see Section 4.2.9). We applied the hold-out and k-fold cross-validation protocols to the datasets as per Reference [24,25]. We also found that architecture selection and hyperparameter choices have become the key steps in Automated Machine Learning (AutoML) frameworks. We incorporated these key features into our research workflow, as shown in Figure 7.

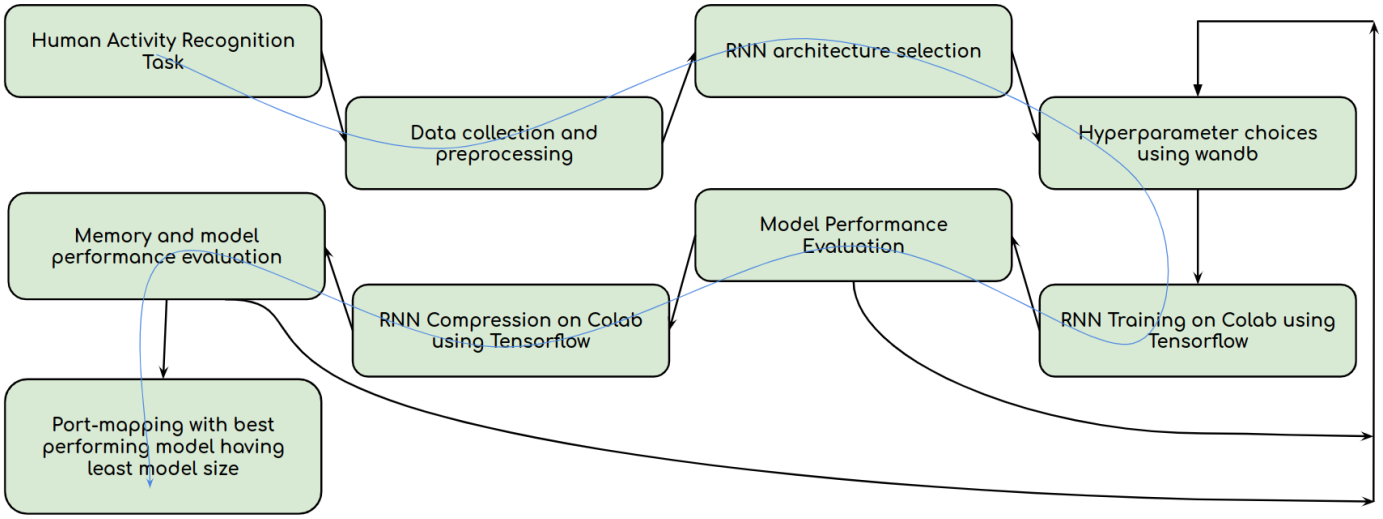


Figure 7. Recurrent Neural Network-based application-to-device edge-mapping workflow.

4. Experiments

In this section, we discuss the details of the experimental setup, training hardware, training framework and experiment tracking tool, application and datasets used, evaluation methods, and metrics used in our study.

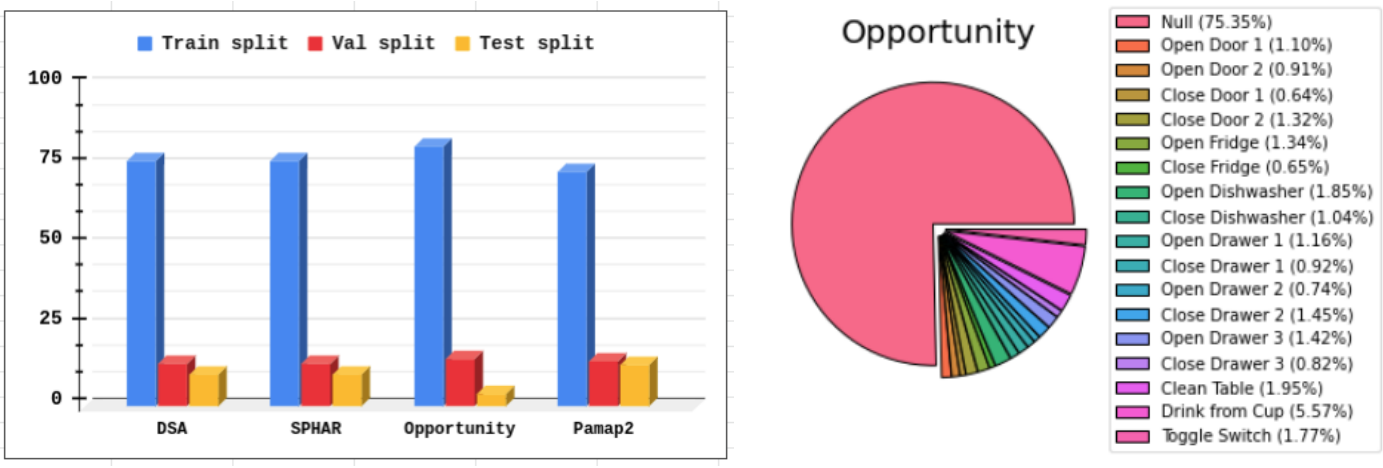
4.1. Experimental Settings

All experiments were conducted on Google Colab leveraging an NVidia Tesla T4 Tensor Core GPU (with a clock of 1.59 GHz, 14.27 GB memory, and 7.5 compute capability) on the Compute Unified Device Architecture, Version 11.2, platform. We first mounted the program files and folders of our experiments from Google Drive. Next, we performed a `pip install` of the `requirements.txt` file for all the libraries and package installations on Colab. These steps were much easier and quicker apriori to Colab, where we struggled to set up the coding environment and manage them. Training was carried out using TensorFlow Deep Learning Library (v1.15) with a Python programming environment. Hyperparameter tuning was carried out using a deep learning experiment tracking tool called Weights And Biases (wandb) [47]. Its important feature, hyperparameter sweeps, helps find the best model for various hyperparameter choices. The tool is also useful in terms of visualization and debugging DL errors. We set specific seed values in TensorFlow during the initialization of weight matrices.

4.2. Application and Datasets

For the Human Activity Recognition (HAR) tasks using EI-RNNs, we used eight static time-series datasets for our experiments. The datasets are captured from accelerometers, gyroscopes, temperature sensors, smartphones with sensors, and magnetometers. The details of the dataset are given in Tables 3 and 4. In the tables, input features represent the total input dimension, which is equal to the total time steps times the number of features from the sensors. We applied the hold-out protocol to the datasets described in Table 3, where they underwent train–validation–test splits for RNN training (see Figure 8a). We applied the k-fold cross-validation protocol to the datasets described in Table 4.

There are a few challenges in the datasets retrieved. Few datasets are balanced, and categories are definite, while a few are unbalanced (see Figure 8b) and have the null class problem (Opportunity dataset). Imbalances occur when few activities take place for a longer duration (and so are recorded several times), and few activities take place rarely (and so are recorded fewer times). Typically, only a few parts of the data are relevant, and irrelevant information constitute the null class [48].



(**a**) (**b**)

Figure 8. (**a**) Train–val–test split ratios are uneven across datasets; (**b**) an example for imbalanced classes in Opportunity dataset.

Next, there can be large number of sensor modalities, and the number of participants and trial records may differ in each dataset. Both these factors influence the performance of the model under study. (Ref. [49], Figure 1 shows how the accelerometer dataset is annotated for the HAR task).

Table 3. Details of the datasets where train, validation, and test samples were generated. The semi non-overlapping windowing technique was used for generation of Opportunity and Pamap2. A = accelerometer, G = gyroscope, M = magnetometer, B = balanced dataset, N = null categories present.

SI No.	Dataset	Input Sensor	Train Samples	Val Samples	Test Samples	Time Steps	Input Features	Output Labels	Freq. (Hz)
1	DSA [49] (B)	A, M	6976	1232	912	125	5625	19	25
2	SPHAR [50] (B)	A, G	7878	1391	1030	128	1152	6	50
3	Opportunity [51] (N)	A, G, M	54,246	9894	2684	24	1896	18	30
4	Pamap2 [52] (B)	A, G, M	39,452	7566	6946	24	1248	12	100

Table 4. Details of the datasets under k-fold cross validation protocol. A = accelerometer, G = gyroscope, M = magnetometer.

SI No.	Dataset	Input Sensor	No. of Samples (SNOW)	No. of Samples (FNOW)	No. of Classes	Sampling Frequency (Hz)	No. of Features	Balanced
1	MHEALTH [53]	A, G, M	2555	1335	12	50	5750	True
2	USCHAD [54]	A, G	9824	5038	12	100	3000	False
3	WHARF [55]	A	3880	2146	12	32	480	False
4	WISDM [56]	A	20,846	10,516	6	20	300	False

4.2.1. Daily and Sports Activities Dataset

DSA comprises data from nine sensors placed at five different body parts of eight users for 5 min per activity. The 5 min signals are divided into 5 s segments so that 480 (=60 × 8) signal segments were obtained for each activity. A total of 45 signals with 25 features extracted every 5 s yield a 5625-dimensional feature vector. Some of the activities include sitting, standing, lying on the back and right side, ascending and descending stairs, cycling on an exercise bike in horizontal and vertical positions, rowing, and so on. It is a 19-category and well-balanced dataset.

4.2.2. Smart Phone Human Activity Recognition Dataset

The UCI HAR from the smartphone dataset consists of data from 30 users performing daily activities from a waist-mounted smartphone with embedded inertial sensors. The sensor signals (accelerometer and gyroscope) were preprocessed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 s and 50% overlap (128 readings/window). Some of the activities include sitting, standing, walking, lying, and so on. It is a six-category and well-balanced dataset.

4.2.3. Opportunity Dataset

This dataset consists of data from a rich sensor environment where subjects performed daily activities in the kitchen and living area. It comprises recordings of 12 subjects using 15 networked sensor systems, with 72 sensors of 10 modalities integrated into the environment, in objects, and on the body. Some of the activities included open door, close door, open fridge, close fridge, open drawer, close drawer, clean table, etc. It is an 18-category dataset with null cases also included. The dataset is not well-balanced with the null class problem (see Section 4.2 for null class description).

4.2.4. Physical Activity Monitoring for Aging People Dataset

This PAMAP2 dataset consists of recordings from nine users using three different types of sensors (accelerometer, gyroscope, and magnetometer) and a heart rate monitor. The sensors were placed at three different body positions, and data from these were sampled at 100 Hz. The heart rate monitor was set to a 9 Hz sampling frequency. Some of the activities included lying, sitting, standing, walking, running, cycling, and so on. It is a 12-class and balanced dataset.

4.2.5. Mobile Health Dataset

This MHealth dataset was collected from 10 users with four different types of sensors, i.e., three-axis accelerometer sensors, three-axis gyroscope sensors, three-axis magnetometer sensors, and two-lead electrocardiogram sensors, to record 12 different activities at a 50 Hz sampling frequency. Some of the activities included standing still, sitting and relaxing, lying down, walking, climbing stairs, waist bend forward, the frontal elevation of arms, etc. It is a 12-category, well-balanced dataset.

4.2.6. University of Southern California Human Activity Dataset

This USCHAD dataset was collected from 14 subjects from accelerometers and gyroscopes recording 12 activities. Some of the activities included walking forward, walking left, walking right, walking upstairs, walking downstairs, running forward, and so on. It is a 12-category and unbalanced dataset.

4.2.7. Wearable Human Activity Recognition Folder Dataset

This WHARF dataset was collected from 17 subjects through a single wrist-worn triaxial accelerometer recording 12 activities. Some activities included getting up from bed, sitting down on a chair, drinking from a glass, eating with a fork and knife, eating with a spoon, pouring water, etc. The dataset is a 12-category and unbalanced dataset.

4.2.8. Wireless Sensor Data Mining Dataset

The WISDM dataset was collected from the WIreless Sensor Data Mining laboratory using a pocket-placed mobile device with an accelerometer. Six activities were recorded at a 20 Hz sampling frequency. Some of the activities included walking, jogging, sitting, standing, and so on. It is a six-category dataset, unbalanced dataset.

4.2.9. Finer Details of the Datasets

The datasets were normalized and preprocessed before pipelining them to the RNNs for training. NaNs were replaced with zeros. The output labels were one-hot-encoded. An important point to note here is that the original dataset reference article (citations in the second columns of Tables 3 and 4) gives details of the dataset as per their data acquisition and processing steps. These datasets can be downloaded, processed, and formatted to a certain file type for better usability by other researchers in the community (see Appendix A). There may be a few variations between the two, for example, combining two labels (standing up and standing) into one (standing up). To the best of our knowledge, we cross-checked the details of the base dataset paper and the source of download. The reader may also find that the training, validation, and testing split across these datasets were not perfectly uniform (see Figure 8a). This is because a few datasets were already split into training, validation, and testing groups (Opportunity and Pamap2). We retained them as they were. DSA and SPHAR had only train and test splits. We further split the training into training and validation groups. More data handling procedures and preprocessing details can be obtained from Reference [26].

As a data generation procedure, Opportunity and Pamap2 datasets fall into the semi nonoverlapping windowing (SNOW) methods, where the actual dataset is transformed using the sliding window technique with a sliding window length of 24 and window stride of 12. Figure 9 shows the sliding window technique used for dataset transformation. The datasets in Table 4 were downloaded after applying windowing methods like SNOW and Fully Nonoverlapping Window (FNOW). Since they were not normalized, we normalized them before usage. More details regarding these methods can be obtained from Reference [25].

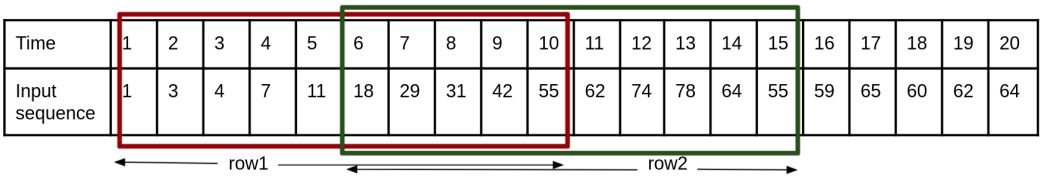


Figure 9. Semi non-overlapping windowing data generation technique. The red box indicates an input sequence with ten features. With a window stride of fived, the green box indicates the next ten input features.

4.3. Evaluation Protocols and Metrics

In this study, we adopted two evaluation protocols (items 1 and 2 below) and two evaluation metrics to assess model performance (items 3 and 4 below).

1. Holdout method: In this method, datasets are split into three sets, i.e, training set, validation/holdout set, and testing set. Here, the training set is the subset of data used to learn the temporal pattern in the time-series data. The error associated with this is called training error. Validation or hold-out set is the subset of data used to guide the selection of hyperparameters. The error associated with this is called validation error. The test set is the subset of data used to measure the model's performance on a new unseen sample. The error associated with this is called testi error. The split ratio affects the model's performance.
2. Cross-validation method: In this method, datasets are split into k nonoverlapping subsets. In each trial, one of them is chosen as a test set and the rest is used as the training set. Test error is estimated by taking the average test error across k trials.

3. Accuracy: This metric denotes the total number of correct predictions of classes against their actual labels.
4. F1 score: This metric is the weighted average of precision and recall. Precision is the ratio of correctly classified positive observations to the total classified positive observations. Recall is the ratio of correctly classified positive observations to all observations in actual class.

5. EI-RNN Optimization and Analysis

Optimizing recurrent architectures for the edge comprises three steps:

1. EI-RNN training on host/cloud GPU;
2. EI-RNN compression on host/cloud GPU;
3. EI-RNN inference on the edge (RPi).

5.1. EI-RNN Training

Any learning process is a combination of representation, evaluation, and optimization [57]. Representations of datasets and neural architectures are covered in Section 4.2 and Section 2, respectively. In this section, we describe the optimization techniques for EI-RNN modeling. We also describe the initial training setup and how we examined the fit issues and choice of hyperparameters affecting model performance. The base source code is Reference [58], which underwent several significant changes, including hyperparameter tuning, experiment tracking, and visualization using wandb, and experiments covering hold-out and k-fold cross-validation methods of RNN training.

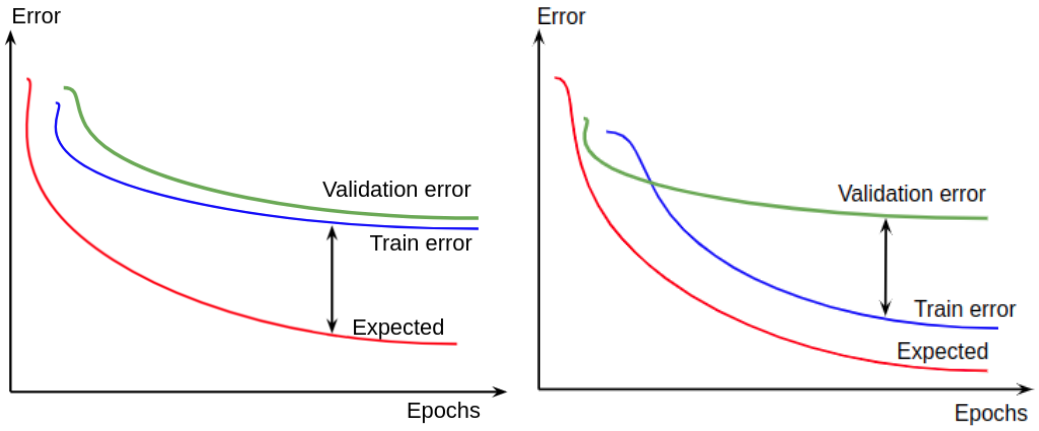
5.1.1. Initial Settings

From a set of cleaned and structured datasets and architectures, we initiated a training procedure with a simple set of hyperparameters: a number of hidden units of the RNN cell of 8, a number of layers of 1, and the number of input features to be processed at every time step t. The time steps of the RNN cell (125 for the DSA dataset) was given by the dimension of the input features (5625)/number of features to be processed (45). Furthermore, we set the number of epochs (300), batch size (128), optimizer (Adam/RMS Prop), and learning rate (0.01) based on references. The activation functions of each RNN are given in Section 2. We used the random normal initialization of the RNN weight matrices. We adopted the cross-entropy loss function since the problem was a classification problem and the output function was a softmax nonlinearity function.

5.1.2. Train–Debug Cycle

We chose a total of 8 datasets, which were grouped into 2 to experiment with two methods, namely, the train–val–test split method and the cross-validation method. Usually cross-validation is applied. To the four datasets in Table 3, we applied the train–val–test split hold out method, whereas to the four datasets in Table 4, we applied the cross-validation method (see Section 5.1). In the train–debug cycle phase of modeling, we observed the loss and accuracy plots across the training and validation datasets. We used six different recurrent units, namely, Vanilla RNN, LSTM, GRU, FGRNN, FRNN, and UGRNN. We addressed the under-fitting case (see Figure 10a) to improve the model performance on the training set by increasing the number of hidden units of the RNN. We addressed the over-fitting case (see Figure 10b) to improve model performance on the validation set using regularization methods. The implementation included the adaptive learning rate during training with the help of TensorFlow placeholders (tf.compat.v1.placeholder ('float', name = 'lr')). Dropout regularization (by setting *input_keep_prob* and *output_keep_prob* hyperparameters) and regularization through early stopping criteria were incorporated. The RNN underwent unstacking (with the help of tf.unstack (x, timeSteps, 1) Application Programming Interface (API)), and the Dropout wrapper (available in the TensorFlow library) was applied whenever we encountered over-fitting. A compute graph was built

in TensorFlow v1.15 to find the final hidden states, logits (output of the classifier), and softmax predictions.



(**a**) (**b**)

Figure 10. (**a**) High bias problem where training error is far from expected, an under-fitting case, and (**b**) high variance problem where training error is close to expected, but there is a large gap between training and validation error, an over-fitting case.

5.1.3. Hyperparameter Tuning and Generalization

After model creation with initial settings and train–debug cycles, we performed a hyperparameter search via the wandb tool. We chose the grid search method and observed the impact of the hyperparameter hidden units, batch size(for example, see Figure 11), and optimizer in various sweeps (for example, see. Figure 12a). We found the best batch size and best number of hidden units for each RNN and each dataset. Table 5 shows the variation in the hidden units for the UGRNN architecture on the SPHAR dataset. Increases in the number of hidden units improved performance but adversely affected the model size. An optimal choice had to be made. We chose 16 as the number of hidden units considering both performance and model size. Next, we found the best optimizer without any regularization. This was conducted for each of the RNN architectures for each dataset individually. Figure 11 corresponds to variations in batch size with the FastRNN architecture for the DSA19 dataset. We chose a batch size of 32 for this RNN and dataset. The optimizer hyperparameter check was conducted for each dataset. The choice of optimization technique significantly influenced the efficiency of the learning algorithm, as shown in Figure 12a. Figure 12b corresponds to the performance of the EI-RNNs on the DSA19 dataset with RMS Prop as the optimizer, with each of them having 16 hidden units. The approximate sensitivity of the hyperparameters under study is given in Table 6. Table 7 gives the list of optimizers corresponding to each dataset. Once we found the best set of hyperparameters, we observed the test error or generalization error. The ability of a model to perform well on previously unobserved data is called generalization [59,60]. Regularization methods reduce this error. In our study, we found that dropout regularization reduced the generalization error significantly. Dropout is a regularization technique applied to only nonrecurrent matrices of an RNN [61].

Table 5. Impact of variation in hidden units on performance and model size of UGRNN on SPHAR dataset.

Hidden Units	Test Accuracy	F1	Model_SIZE (KB)
128	0.94	0.94	14.16
64	0.95	0.95	7.16
32	0.94	0.95	3.66
16	0.94	0.94	1.91
8	0.91	0.91	1.04

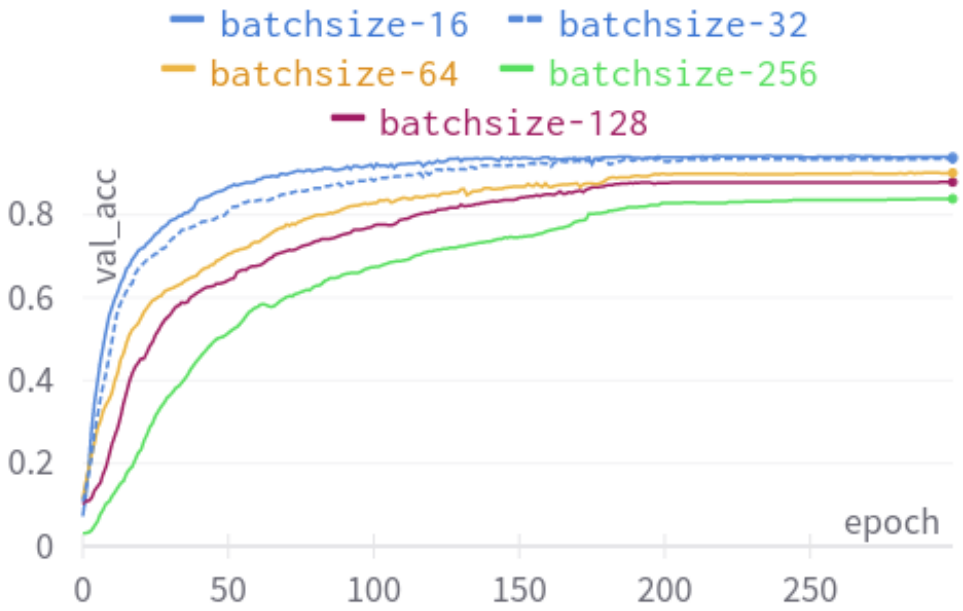
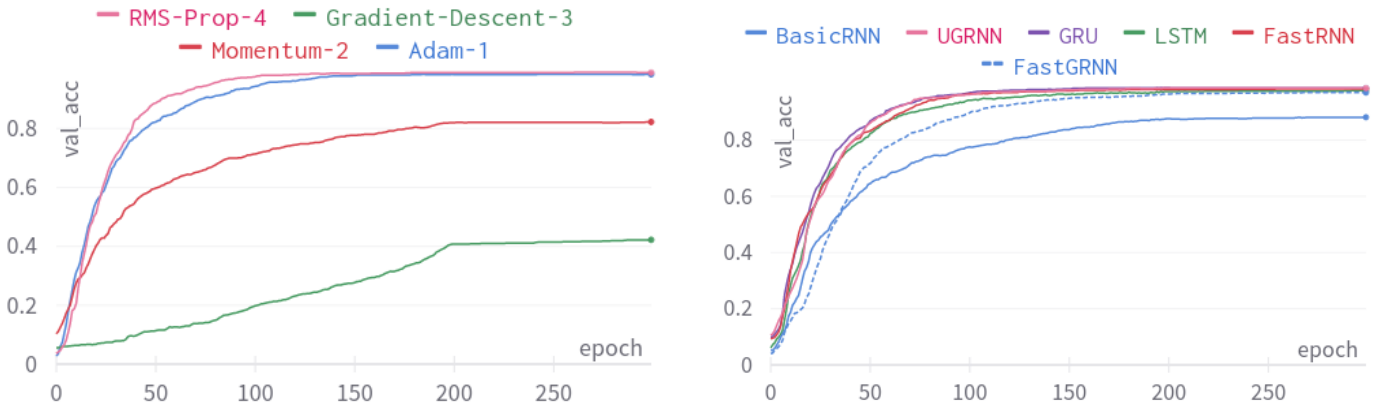


Figure 11. Impact of batch size variation on Fast RNN architecture using DSA19 dataset.



(**a**) (**b**)

Figure 12. (**a**) Validation accuracy variations for different optimizers on DSA19 dataset on GRU, and (**b**) performance of 6 different RNN architectures on the DSA19 dataset using the RMS prop optimizer.

Table 6. Hyperparameters used during training (T), compression (C). Parameters with double arrow represents they are more sensitive than the ones with single arrow.

SI No.	Hyperparameter	App. Sensitivity
1	Activation Function (T)	↑
2	Hidden Size (T, C)	↑ ↑
3	Epochs (T)	↑ ↑
4	Batch Size (T)	↑ ↑
5	Learning Rate (T, C)	↑ ↑
6	Ranks of weight matrices (C)	↑ ↑
7	Sparsity index (C)	↑ ↑
8	Dropout probability (T, C)	↑ ↑
9	Optimizer (T)	↑ ↑
10	Weight matrix initialization (T)	↑
11	Decay rate (T)	↑

Table 7. Choice of optimizers for datasets under study.

SI No.	Dataset	Optimizer
1	DSA	Adam
2	SPHAR	RMS Prop
3	Opportunity	Adam
4	Pamap2	Momentum Nesterov
5	MHEALTH	RMS Prop
6	USCHAD	RMS Prop
7	WHARF	Adam
8	WISDM	Adam

5.2. EI-RNN Compression

The details of the various RNN algorithmic optimizations for edge-based inference are discussed in Section V of Reference [4]. In this section, we briefly present the compression scheme we adopted for edge-mapping studies. As per Reference [28], we carried out train–compress cycles in 3 segments. Training was carried out in the first segment. In the second segment, compression procedures, i.e., hard thresholding, was carried out with low-rank matrix parameterization. Sparse retraining is carried out in the third segment. This process can be formulated in 4 quarters with higher compress and retrain cycles, and the epochs can be increased for finer tuning. The combined list of hyperparameters that affect both training and compression is given in Table 6 with their approximate sensitivity (last column) while performing training (T)–compression (C) cycles. The choice of regularization methods impacts compression. We chose drop out regularization for our experiments, indicated by sparsity indices of matrices W and U. For the 1st trian–test split method, with four datasets and six architectures of RNN, 24 tables of hyperparameter sets were generated. For the 2nd method of k-fold cross validation, with four datasets and six RNNs, we had another set of 24 tables of hyperparameters. We present 2 of them from method one in Tables 8 and 9.

5.3. EI-RNN Inference

The inference was carried on an edge device, the Raspberry Pi. The inference model on the Raspberry Pi was expressed in terms of the respective model weights and biases, as shown in Table 10, using the numpy library. The model and test data were saved on the Pi along with the inference code for each of the 6 RNN architectures. The inference code accounted for the no-rank and low-rank paramterized schemes. Inference was executed using the model, test data, and the inference code; finally, the inference times and memory sizes were reported on the Raspberry Pi.

Table 8. Compression techniques and hyperparameter checks for Fast Gated Recurrent Neural Network or FastGRNN (with 16 hidden units, 32 batch size, 300 epochs, Adam optimizer) on Daily and Sports Activity Recognition (DSA19) dataset for performance and memory optimization. uRank and wRank are ranks of the associated weight matrice; sU and sW are their sparsity indices. Here, 0.9 indicates 10% sparse, 90% dense. Text in blue represents the best hyperparameter choices.

Compression Hyperparameters				Evaluation Metrics					Description
uRank	wRank	sU	sW	Model_SIZE (KB)	train_acc	val_acc	Test_acc	F1 Score	
-	-	1	1	5.20	0.99	0.97	0.98	0.98	Baseline
8	8	1	1	4.30	0.99	0.97	0.96	0.96	Low-Rank Parameterization (LRP)
-	-	0.9	0.9	5.00	0.99	0.98	0.98	0.98	Hard thresholding and sparse retraining (HTSR)
16	16	0.5	0.5	7.20	0.99	0.98	0.97	0.97	Combination of both LRP and HTSR
8	8	0.8	0.8	4.3	0.99	0.98	0.97	0.9732	Memory efficient
12	12	0.6	0.6	5.75	0.99	0.98	0.97	0.97	Performance efficient
5	5	0.9	0.9	3.2	0.98	0.97	0.96	0.96	Our choice for performance and memory savings
10	10	0.5	0.5	5.02	0.97	0.97	0.96	0.962	Other trials
7	7	0.9	0.9	3.93	0.98	0.97	0.95	0.95	Other trails

Table 9. Compression techniques and hyperparameter checks for Fast Recurrent Neural Network or FastRNN (with 16 hidden units, 32 batch size, 300 epochs, root mean square (or RMS Prop optimizer) on SPHAR dataset for performance and memory optimization.

Compression Hyperparameters				Evaluation Metrics					Description
uRank	wRank	sU	sW	Model_Size (KB)	train_acc	val_acc	test_acc	F1 Score	
-	-	1	1	2.03	0.95	0.95	0.94	0.94	Baseline
8	8	1	1	2.25	0.93	0.93	0.935	0.93	Low rank parameterization (LRP)
-	-	0.9	0.9	2.03	0.94	0.94	0.93	0.94	Hard thresholding and sparse retraining (HTSR)
12	12	0.8	0.8	3.20	0.94	0.95	0.932	0.9346	Combination of both LRP and HTSR
8	8	0.9	0.9	1.88	0.93	0.93	0.92	0.92	Our choice for performance and memory savings
16	16	0.5	0.5	4.03	0.94	0.94	0.92	0.94	Performance-efficient
4	4	0.8	0.8	1.3	0.91	0.91	0.91	0.91	Memory-efficient
10	10	0.5	0.5	2.76	0.92	0.91	0.91	0.919	Other trials
12	12	0.4	0.4	2.61	0.93	0.94	0.92	0.93	Other trails

Table 10. Inference model parameters of *Edge-Intelligent* Recurrent Neural Network (EI-RNN) on a Raspberry Pi edge device.

SI No.	RNN Unit	Inference Model Weights and Biases with No Rank	Inference Model Weights and Biases with Low-Rank Parameterization
1	Basic RNN	W, U, Bh	W1, W2, U1, U2, Bh
2	FastGRNN	W, U, Bg, Bh, zeta, nu	W1, W2, U1, U2, Bg, Bh, zeta, nu
3	FastRNN	W, U, B, alpha, beta	W1, W2, U1, U2, B, alpha, beta

Table 10. *Cont.*

SI No.	RNN Unit	Inference Model Weights and Biases with No Rank	Inference Model Weights and Biases with Low-Rank Parameterization
4	UGRNN	W1, W2, U1, U2, Bg, Bh	W, W1, W2, U, U1, U2, Bg, Bh
5	LSTM	W1, W2, W3, W4, U1, U2, U3, U4, Bf, Bi, Bo, Bc	W, W1, W2, W3, W4, U, U1, U2, U3, U4, Bf, Bi, Bo, Bc
6	GRU	W1, W2, W3, U1, U2, Br, Bg	W, W1, W2, W3, U, U1, U2, Br, Bg

6. Results and Discussion

In this section, we discuss the best-performing architectures and smallest models for each dataset as observed from the results in Tables 11–16. We observed that in most of the cases, the fast gate architectures and the basic RNN architecture required the least memory. So, we found memory savings via these fast gates over the best performers. We noticed that fast gates deviated from the best performers. We report the performance deviation (F1 score deviation) when the fast gates were chosen for inference on the edge. All the metrics are reported after training and compression was carried out on Tensorflow 1.15.

The hyperparameters that suit one RNN cell type mostly suit all other types as well. The same set of choices is not applicable to a different dataset. For example, Figure 11 shows the impact of validation accuracy for different batch sizes on the FastRNN architecture on the DSA19 dataset. Table 7 shows the choice of optimizers for the datasets used. The number of hidden units has a direct impact on model size. This is shown inTable 14. Other hyperparameters must be tuned for each dataset separately. Table 8 shows this analysis for the FastGRNN architecture with 16 hidden units on the DSA19 dataset. Also, the regularization and compression techniques are closely related. For example, with respect to the DSA19 dataset, with 16 hidden units per RNN cell, regularization and compression techniques adversely affected both performance and model size. Only hard thresholding and spare retraining did not prove efficient. Only Low-Rank Parameterization showed slightly improved performance and memory savings. A combination of hard thresholding, low-rank parameterization, and sparse retraining showed memory-optimized models and performance improvement. Furthermore, we observed that whenever there were imbalances in the class distribution (Opportunity dataset), increasing the rank of the weight matrices improved performance but adversely affected model size.

6.1. Performance Evaluation Using Hold-Out Method of Training and Subsequent Compression

Here, we present the performance evaluation of the six RNN architectures on four datasets, namely, DSA, SPHAR, Opportunity, and Pamap2, using the hold-out method of training and the subsequent methods of compression discussed in Section 5.2. The hyperparameters for training and compression were tuned as discussed in Sections 5.1 and 5.2. Tables 11 and 12 show the analysis of the six RNN units with different numbers of hidden states: 8, 16, and 32. We report the test accuracy and F1 score (normalized %), training time (minutes), and model size (kilobytes).

From Table 11, we can observe that for the DSA19 dataset, GRU and FastRNN with 32 hidden units were the best performers. FastRNN was the smallest model, followed by UGRNN with 16 hidden units. GRU32 (GRU with 32 hidden units) was 6.27× heavier than the fast gate FastRNN8 and GRU16 (GRU with 16 hidden units) is 3.55× heavier than the fast gate FastRNN8. The performance deviation of FastRNN8 was about 4%, which had a small model capacity of 1.79 KB. On the SPHAR dataset, UGRNN32 proved to perform better than the other architectures. FastRNN and LSTM also performed well but at the cost model size when LSTM was concerned. Regarding model size, LSTM32 was 6.35× bulkier than FastGRNN8, and UGRNN32 was 3.93× bulkier than FastGRNN8. But, the performance deviation of FastGRNN8 was around 6–7% with respect to that of LSTM32 and UGRNN32.

The Opportunity dataset is unbalanced and has more null cases, which resulted in lower F1 scores compared to those of the others. For the Opportunity dataset, GRU16, GRU32, and LSTM16 performed well. Although FastGRNN8 was the smallest, at 3.92 KBytes, its F1 score was very low (0.29) (see figures in red in Table 12). Therefore, we took the next one, i.e., FastGRNN32, as the best model size of 8.05 KBytes. LSTM16 was 1.26× bulkier and GRU32 was 2.09× bulkier than FastGRNN8. Performance deviation was about 5% when we considered fast gates. GRU's F1 score was the highest for this dataset. On the Pamap2 dataset, UGRNN16 provided the highest performance of 81% but an F1 score of 0.77 and was heavier than FastGRNN by 2.03×. The F1 score of GRU32 was the highest on this dataset. UGRNN16 was 2.03× bulkier than FastGRNN8, and GRU32 was 3.86× bulkier than FastGRNN8. FastGRNN deviated from the best performers by around 13 to 16%.

Table 11. Performance of Recurrent Neural Networks across Daily and Sports Activity (DSA) dataset and Smart Phone Human Activity Recognition (SPHAR) dataset with 8, 16, and 32 hidden units. Numbers in bold font represent the best results under each column, and numbers in blue font represent figures that deviated from the best results in each column.

RNN Cell	Hidden Units	*DSA*				*SPHAR*			
		Test Acc. (%)	**F1 Score**	**Train Time (min)**	**Model Size (KB)**	**Test Acc. (%)**	**F1 Score**	**Train Time (min)**	**Model Size (KB)**
BasicRNN	8	0.75	0.73	**4.48**	2.86	0.74	0.72	2.61	1.02
	16	0.84	0.83	4.93	4.23	0.84	0.83	2.51	1.80
	32	0.90	0.90	4.75	6.98	0.86	0.85	**1.71**	3.36
FastGRNN	8	0.92	0.91	10.98	2.09	0.88	0.88	7.15	**0.93**
	16	0.96	0.96	11.65	3.21	0.91	0.91	7.04	1.64
	32	0.97	0.97	8.98	5.46	0.94	0.93	5.62	3.08
FastRNN	8	0.95	0.94	6.76	**1.79**	0.91	0.91	7.73	1.28
	16	0.97	0.97	7.95	2.79	0.94	0.94	7.97	2.25
	32	0.98	0.98	6.54	4.79	0.94	0.94	7.95	4.19
UGRNN	8	0.95	0.94	7.02	2.72	0.92	0.92	11.35	1.23
	16	0.97	0.97	7.18	4.32	0.92	0.92	12.24	1.91
	32	0.98	0.98	10.49	7.50	**0.94**	**0.95**	12.66	3.66
GRU	8	0.95	0.94	14.63	3.92	0.93	0.93	18.07	1.57
	16	0.98	0.98	6.19	6.36	0.93	0.94	11.19	2.95
	32	**0.99**	**0.98**	10.30	11.23	0.93	0.93	9.18	5.70
LSTM	8	0.90	0.90	12.33	6.28	0.91	0.91	13.13	1.60
	16	0.98	0.98	9.52	10.37	0.94	0.94	13.60	3.04
	32	0.98	0.97	13.08	18.56	0.94	0.94	21.41	5.91
Best Performance		GRU with 32 hidden units				UGRNN with 32 hidden units			
Least Model Size		FRNN, FGRNN				FRNN, FGRNN, BasicRNN			
Memory savings via Fast Gates		3.55× with respect to GRU16, 6.27× with respect to GRU32				6.35× with respect to LSTM32, 3.93× with respect to UGRNN32			
Performance deviation via Fast Gates		up to 4%				up to 7%			

Table 12. Performance of Recurrent Neural Networks across Opportunity and PAMAP2 datasets with 8, 16, and 32 hidden units. Numbers in bold font represent the best results under each column, numbers in blue font represent figures that deviate from the best results under each column. F1 scores are low for the Opportunity dataset as it is highly unbalanced. Since Fast Gated Recurrent Neural Network8 (FastGRNN8) had a very low F1 score(in red), we took FastGRNN32 as the best model size under opportunity dataset. Likewise, FastGNN8 was considered best for model size since BasicRNN showed a poor F1 score(in red) on the PAMAP2 dataset.

RNN Cell	Hidden Units	*Opportunity*				*Pamap2*			
		Test Acc. (%)	F1 Score	Train Time (min)	Model Size (KB)	Test Acc. (%)	F1 Score	Train Time (min)	Model Size (KB)
Basic RNN	8	0.84	0.25	12.14	6.70	0.45	0.39	6.46	2.83
	16	0.86	0.34	**4.19**	8.70	0.60	0.52	**8.91**	3.98
	32	0.87	0.38	9.27	12.70	0.59	0.54	6.31	6.30
FastGRNN	8	0.85	0.29	13.25	3.92	0.71	0.64	8.69	**2.99**
	16	0.86	0.34	21.71	5.30	0.73	0.65	7.48	4.59
	32	0.86	0.42	14.89	**8.05**	0.73	0.67	8.05	6.93
FastRNN	8	0.86	0.33	15.99	4.70	0.57	0.63	7.72	4.96
	16	0.85	0.39	15.77	6.23	0.62	0.65	9.21	5.55
	32.00	0.87	0.44	18.66	9.29	0.69	0.67	8.49	8.62
UGRNN	8	0.85	0.34	23.32	4.41	0.75	0.69	12.65	4.08
	16	0.86	0.38	14.36	6.29	**0.81**	0.77	9.48	6.08
	32	0.86	0.44	26.49	8.77	0.69	0.66	8.49	8.62
GRU	8	0.86	0.37	19.45	7.05	0.69	0.63	10.24	4.73
	16	**0.87**	0.42	31.89	10.34	0.78	0.72	12.02	7.39
	32	**0.87**	**0.47**	30.15	16.90	0.80	**0.80**	22.82	11.55
LSTM	8	0.86	0.39	20.72	5.48	0.67	0.59	19.78	4.42
	16	**0.87**	0.46	16.37	10.16	0.57	0.50	16.51	7.17
	32	0.86	0.46	26.24	17.16	0.59	0.53	19.28	12.67
Best Performance		GRU, LSTM with 1632 hidden units				GRU with 32 hidden units			
Smallest Model		FGRNN				FRNN, FGRNN, BasicRNN			
Memory Savings via Fast Gates		2.09× with respect to GRU32, 1.26× with respect to LSTM16				2.03× with respect to UGRNN16, 3.86× with respect to GRU32			
Performance Deviation via Fast Gates		up to 5%				around 13–16%			

6.2. Performance Evaluation Using K-Fold Cross-Validation Training and Subsequent Compression

Next, we present the performance analysis of the six architectures on four datasets, namely, MHEALTH (Table 13), USCHAD (Table 14), WHARF (Table 15), and WISDM (Table 16), using cross-validation training with the subsequent methods of compression discussed in Section 5.2. The hyperparameters for training and compression were tuned as discussed in Sections 5.1 and 5.2. We used the scikit learn library, which has kfold.split method, to perform the five-fold validations to obtain the training and validation sets for each fold in an iterative manner. We report the mean train accuracy, mean validation accuracy across five folds, test accuracy, F1 score (normalized %), and model size (kilobytes) for this set of experiments. We chose 16 hidden units for this set of training experiments since 8 was small and 32 was large for running a five-fold training. For this method, two kinds of data generation methods were employed, namely FNOW and SNOW. Model sizes were the same for both data generation schemes and are thus reflected as a common column (Col. 6) in the next four tables (Tables 13–16).

Table 13. Performance analysis of RNNs for Mobile HEALTH (MHEALTH) Fully Nonoverlapping Window (FNOW) and Semi Nonoverlapping Window (SNOW) based Human Activity Recognition (HAR) task. Mean training accuracy, mean validation accuracy, test accuracy, and F1 score are expressed as normalized %, and model size is expressed in kilobytes. Text in red is the highest in each category, except those under model size, where text in red represents the smallest model.

RNN Type	MHEALTH-FNOW					MHEALTH-SNOW			
	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score	Model Size	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score
Basic RNN	0.77	0.76	0.83	0.78	3.30	0.86	0.85	0.85	0.81
FastGRNN	0.86	0.85	0.93	0.92	3.37	0.97	0.97	0.98	0.98
FastRNN	0.99	0.99	0.99	0.99	3.30	0.99	0.99	0.99	0.99
UGRNN	0.85	0.82	0.90	0.87	5.42	0.96	0.96	0.96	0.96
GRU	1.00	0.98	0.99	0.99	8.30	0.99	0.98	0.99	0.99
LSTM	0.96	0.94	0.96	0.95	10.80	0.99	0.99	0.99	0.99
Best Performance	FRNN and GRU					FRNN, GRU and LSTM			
Smallest Model	FRNN and BasicRNN					FRNN, BasicRNN			
Memory Savings via Fast Gates	2.52× with respect to GRU , 3.27× with respect to LSTM								
Performance Deviation via Fast Gates	Nil								

Table 14. Performance analysis of RNNs on University of Southern California Human Activity dataset or USCHAD Fully Nonoverlapping Window (FNOW)- and Semi Nonoverlapping Window (SNOW)-based Human Activity Recognition (HAR) task. Mean train accuracy, mean validation accuracy, test accurac,y and F1 score are expressed in normalized %, and model size is expressed in KB. Text in red is the highest in each category, except those for model size, where text in red represents the smallest model.

RNN Type	USCHAD-FNOW					USCHAD-SNOW			
	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score	Model Size	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score
Basic RNN	0.51	0.51	0.53	0.44	2.23	0.51	0.51	0.49	0.29
FastGRNN	0.70	0.70	0.73	0.64	2.30	0.71	0.71	0.73	0.73
FastRNN	0.77	0.76	0.79	0.73	2.24	0.87	0.87	0.90	0.86
UGRNN	0.85	0.85	0.87	0.83	3.67	0.85	0.86	0.90	0.87
GRU	0.84	0.83	0.84	0.80	5.11	0.86	0.84	0.88	0.84
LSTM	0.75	0.75	0.70	0.64	6.55	0.79	0.76	0.74	0.67
Best Performance	UGRNN					FastRNN			
Smallest Model	FRNN, FGRNN, BasicRNN					FRNN, FGRNN, BasicRNN			
Memory Savings via Fast Gates	1.63× with respect to UGRNN								
Performance Deviation via Fast Gates	up to 10%					up to 1%			

Table 15. Performance analysis of RNNs for WHARF Fully Nonoverlapping Window (FNOW)- and Semi Nonpverlapping Window (SNOW)-based HAR task. Mean train accuracy, mean validation accuracy, test accuracy, and F1 score are expressed as normalized %, and model size is expressed in KB. Text in red is the highest in each category, except those under model size, where text in red represents the smallest model.

RNN Type	WHARF-FNOW					WHARF-SNOW			
	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score	Model Size	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score
Basic RNN	0.49	0.49	0.44	0.27	2.05	0.51	0.51	0.50	0.30
FastGRNN	0.49	0.50	0.45	0.27	2.12	0.59	0.59	0.55	0.36
FastRNN	0.57	0.59	0.55	0.43	2.05	0.63	0.63	0.61	0.43
UGRNN	0.58	0.57	0.53	0.34	3.30	0.66	0.67	0.67	0.49
GRU	0.57	0.56	0.50	0.34	4.55	0.58	0.56	0.53	0.36
LSTM	0.53	0.52	0.48	0.31	5.80	0.59	0.59	0.57	0.37
Best Performance	FRNN					UGRNN			
Smallest Model	FRNN and BasicRNN					FRNN, BasicRNN			
Memory Savings via Fast Gates	1.6× with respect to UGRNN								
Performance Deviation via Fast Gates	Nil					up to 6%			

Table 16. Performance analysis of RNNs for WISDM Fully Nonoverlapping Window (FNOW)- and Semi Nonoverlapping Window (SNOW)-based HAR task. Mean training accuracy, mean validation accuracy, test accuracy, and F1 score are expressed as normalized %, and model size is expressed in KBy. Text in red is the highest in each category, except those under model size, where text in red represents the smallest model.

RNN Type	WISDM-FNOW					WISDM-SNOW			
	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score	Model Size	Mean Train Acc.	Mean Val Acc.	Test Acc.	F1 Score
Basic RNN	0.82	0.80	0.81	0.69	1.27	0.80	0.79	0.82	0.67
FastGRNN	0.81	0.81	0.81	0.69	1.72	0.83	0.83	0.84	0.72
FastRNN	0.82	0.82	0.82	0.72	1.66	0.84	0.84	0.85	0.76
UGRNN	0.93	0.92	0.93	0.89	2.90	0.87	0.88	0.89	0.82
GRU	0.88	0.87	0.88	0.82	4.15	0.97	0.96	0.97	0.95
LSTM	0.85	0.84	0.75	0.84	5.40	0.79	0.79	0.79	0.62
Best Performance	UGRNN					GRU			
Smallest Model	FRNN and BasicRNN					FRNN, and BasicRNN			
Memory Savings via Fast Gates	1.81× with respect to UGRNN					2.5× with respect to GRU			
Performance Deviation via Fast Gates	up to 17%					up to 19%			

For MHEALTH-FNOW, GRU and FastRNN were the best performers, and FastGate and Vanilla RNN were the smallest. For MHEALTH-SNOW, FastRNN, GRU, and LSTM were the best performers. For USCHAD-FNOW, UGRNN obtained a normalized F1 score of 0.883, and FastGate and Vanilla RNN were the smallest at around 2.3KB. For USCHAD-SNOW, FastRNN performed best. For WHARF-FNOW, UGRNN had the best training accuracy, and FastRNN had the best test accuracy and F1 score. For USCHAD-SNOW, UGRNN had the best performance and was the smallest. For WISDM-FNOW, UGRNN performed best with a 0.89 normalized F1 score, and, for WISDM-SNOW, GRU performed best with a 0.95 normalized F1 score.

6.3. Inference Evaluation on Raspberry Pi

Finally, we present the RNNs that were port-mapped onto the Raspberry Pi. We recorded the inference time and model size for each RNN and for each dataset. Figure 13 shows the model sizes of the RNN architectures for the eight datasets and Figure 14 show the inference time of the models on the eight datasets when port-mapped to the edge device. FastRNN and FastGRNN occupied the least space, whereas LSTM and GRU were bulky. A similar observation was made when we recorded the inference time of the models on the Pi. FastRNN, FastGRNN, BasicRNN, and UGRNN were faster than the GRU and LSTM models.

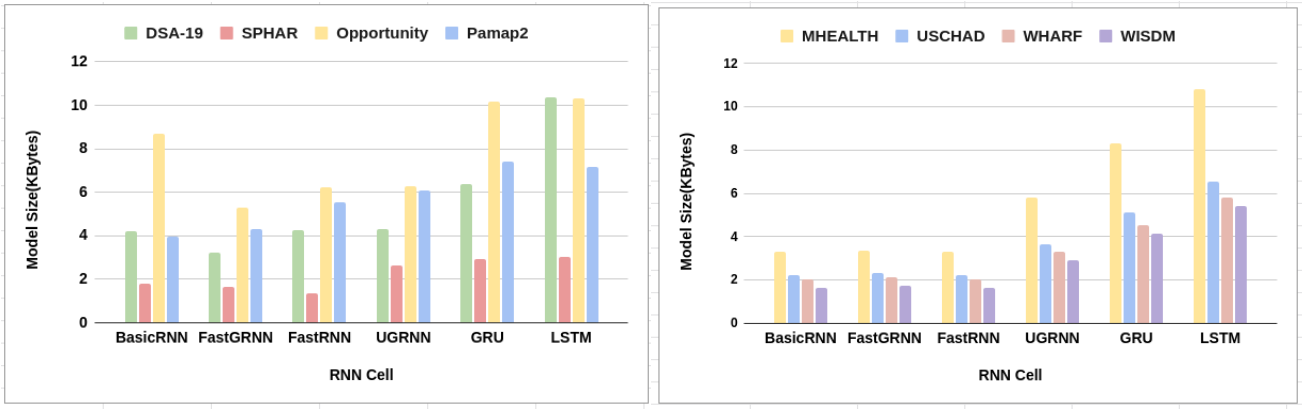
The following are the key takeaways after the training and compression of the six Recurrent Neural Network architectures:

1. Apart from LSTM, the other RNNS like fast gates like FastGRNN, FastRNN, UGRNN, and GRU are potential candidates for application in device-mapping edge-based RNN modeling studies. Similar studies were conducted in Ref. [28]. The Table 17 from Ref. [28] shows the performance of the RNNs but with different hyperparameters used for the RNN architecture.
2. LSTM and GRU require significantly longer training and inference time compared to the other RNN units that we studied.
3. LSTM and GRU are bulkier on the edge device than FastRNN, FastGatedRNN, and UGRNN.
4. Fast gates like FastGRNN and FastRNN are memory-efficient for edge devices, but they show performance deviation compared to the other RNNs of around 0 to 19% across all considered datasets.
5. UGRNN are also potential candidates for edge-based RNN model mapping, showing good performance and smaller memory sizes compared to LSTM and GRU.

6. Data collection and preprocessing play important roles in training the RNNs and consume significant time.
7. If the class distribution of the dataset is unbalanced, we saw a drop in the F1 score of the RNN model.
8. A hidden state size of 16 and a single layer are optimal for edge mapping for HAR applications. Increasing the hidden size or layer size improves performance but adversely affects the model size.
9. Regularization methods like dropout improve model performance after any degradation due to the compression applied on the dense RNN models.
10. A combination of compression techniques is better than singleton methods.
11. For Low-Rank Parameterization, the decomposition rank affects the model size and performance of RNNs. Higher ranks improve performance but increase the model size proportionally.
12. The inference time on a Raspberry Pi was directly dependent on the time steps, the input feature size the RNN was processing, and the complexity of the RNN cell and architecture.
13. The complex workflows for application to device mapping have forced developers to be inventive. Dealing with frameworks and third-party packages and libraries is complex.

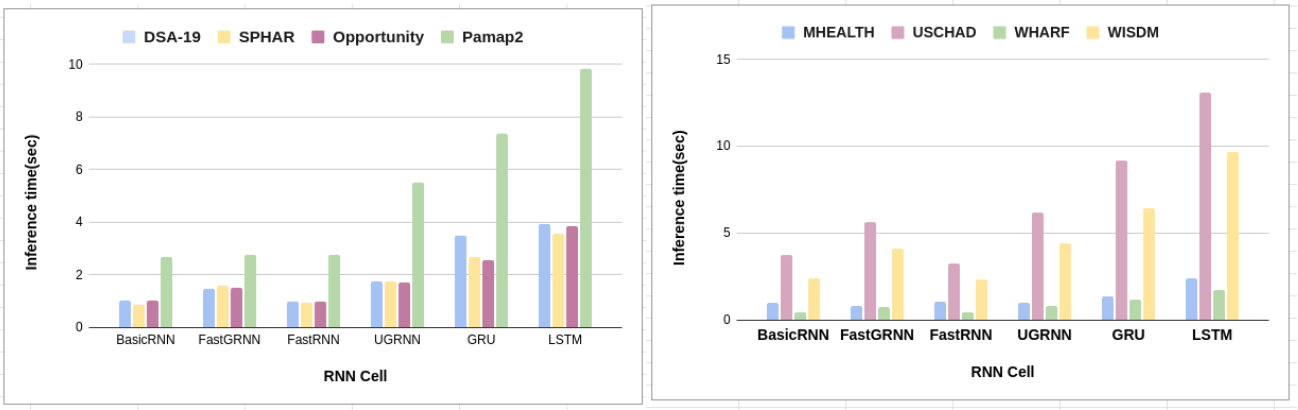
Table 17. Conclusions for SPHAR and DSA from [28].

		SPHAR		DSA	
SI No.	**RNN Unit**	**Accuracy (%)**	**Model Size (KB)**	**Accuracy (%)**	**Model Size (KB)**
1	Basic RNN	91.31	29	71	20
2	FastRNN	94.50	29	84.14	97
3	FastGRNN	95.59	3	83.73	3.25
4	UGRNN	94.53	37	84.74	399
5	LSTM	93.62	71	84.84	270
6	GRU	93.65	74	84.84	526



(**a**) (**b**)

Figure 13. Recurrent Neural Network (RNN) model sizes for different 8 datasets on Raspberry Pi (RPi). (**a**) RNN model size on RPi for Daily and Sports Activity Recognition (DSA-19), Smart Phone Human Activity Recognition (SPHAR), Opportunity, and Physical Activity Monitoring for Aging People (PAMAP2) datasets. (**b**) RNN model size on RPi for Mobile Health (MHEALTH), University of Southern California Human Activity Dataset (USCHAD), Wearable Human Activity Recognition Folder (WHARF), and Wireless Sensor Data Mining (WISDM) dataset.



(**a**) (**b**)

Figure 14. Recurrent Neural Network or RNN inference time for 8 different datasets on Raspberry P (RPi). (**a**) Inference time of RNN models on RPi for Daily and Sports Activity Recognition (DSA-19), Smart Phone Human Activity Recognition (SPHAR), Opportunity, and Physical Activity Monitoring for Aging People (PAMAP2) datasets. (**b**) Inference time of RNN models on RPi for Mobile Health (MHEALTH), University of Southern California Human Activity Dataset (USCHAD), Wearable Human Activity Recognition Folder (WHARF), and Wireless Sensor Data Mining (WISDM) datasets.

7. Conclusions and Future Scope

This study involved the comprehensive empirical evaluation and optimization of an RNN-based HAR application in device mapping. We covered six RNN units, namely, the Vanilla RNN, Fast Gated RNN, Fast RNN, GRU, UGRNN, and LSTM. We used hold-out and cross-validation methods on eight datasets, namely, DSA19, SPHAR, Opportunity, PAMAP2, MHealth, USCHAD, WHARF, and WISDM. We trained and compressed the RNNs. We addressed over-fitting issues while training using dropout regularization. For compression, we used low-rank parameterization, iterative hard thresholding, and spare retraining methods. We found that efficient training and suitable compression methods are critical in optimizing RNNs for performance and memory efficiency. We performed inference on a Raspberry Pi.

For finding the right set of hyperparameters for a model, NAS can be explored. Other compression techniques like the Kronecker Products can be explored, which, in the literature, have proved efficient in edge modeling studies.

Author Contributions: Conceptualization, V.S.L. and V.R.B.; methodology, V.S.L.; validation, V.S.L., V.R.B. and J.A.; formal analysis, V.S.L. and H.S.S.; investigation, V.S.L., V.R.B., J.A. and H.S.S.; writing—original draft preparation, V.S.L.; writing—review and editing, V.S.L. and J.A.; visualization, V.R.B. and H.S.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The instance data and solutions will be made available upon request.

Acknowledgments: We would like to thank the anonymous reviewers for providing valuable feedback for improving the technical quality and segmentation of this journal.

Conflicts of Interest: Author Hariram Selvamurugan Satheesh was employed by the company ABB GISPL. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

RNN	Recurrent Neural Network
HAR	Human Activity Recognition

GPU	Graphics Processing Unit
EI-RNN	*Edge-Intelligent* RNN
IoT	Internet of Things
LSTM	Long Short-Term Memory
GRU	Gated Recurrent Unit
FRNN	Fast Recurrent Neural Network
FGRNN	Fast Gated Recurrent Neural Network
UGRNN	Unitary Gated Recurrent Neural Network
RPi	Raspberry Pi
DL	Deep Learning
AI	Artificial Intelligence
CNN	Convolutional Neural Network
MCU	Microcontroller Unit
ELL	Embedded Learning Library
SoC	System on Chip
LMF	Low-Rank Matrix Factorization
SVD	Singular Value Decomposition
DSA	Daily and Sports Activities
SPHAR	Smart Phone Human Activity Recognition
Oppo	Opportunity
PAMAP	Physical Activity Monitoring for Aging People
MHEALTH	Mobile HEALTH
USC-HAD	University of Southern California Human Activity Dataset
WHARF	Wearable Human Activity Recognition Folder
WISDM	Wireless Sensor Data Mining
NaN	Not a Number
API	Application Programming Interface
wandb	Weights and Biases Tool
RMS Prop	Root Mean Squared Propagation

Appendix A

Table A1. Datasets and their URL. All of them were accessed on 30 June 2022.

Dataset	URL
DSA	https://archive.ics.uci.edu/ml/datasets/daily+and+sports+activities
SPHAR	https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones
Oppo	https://universityofadelaide.app.box.com/s/ag10ugotoqmbw3sw6q74s0pd7b7gkznj
PAMAP2	https://universityofadelaide.app.box.com/s/ag10ugotoqmbw3sw6q74s0pd7b7gkznj
MHealth	https://github.com/colebryant/mhealth-classification
USCHAD	http://sipi.usc.edu/HAD/
WHARF	https://github.com/centaurresearchgroup/WHARF
WISDM	https://www.cis.fordham.edu/wisdm/dataset.php

References

1. Kolen, J.; Kremer, S. Gradient flow in recurrent nets: The difficulty of learning longterm dependencies. In *A Field Guide to Dynamical Recurrent Network*; IEEE: Piscataway, NJ, USA, 2010.
2. Martens, J.; Sutskever, I. Learning recurrent neural networks with hessian-free optimization. In Proceedings of the 28th International Conference on Machine Learning, Washington, DC, USA, 28 June–2 July 2011.
3. Collins, J.; Sohl-Dickstein, J.; Sussillo, D. Capacity and trainability in recurrent neural networks. *arXiv* **2016**, arXiv:1611.09913.
4. Lalapura, V.S.; Amudha, J.; Satheesh, H.S. Recurrent neural networks for edge intelligence: A survey. *ACM Comput. Surv.* **2021**, *54*, 1–38. [CrossRef]
5. Amudha, J.; Thakur, M.S.; Shrivastava, A.; Gupta, S.; Gupta, D.; Sharma, K. Wild OCR: Deep Learning Architecture for Text Recognition in Images. In Proceedings of the International Conference on Computing and Communication Networks, Manchester, UK, 19–20 November 2022; Springer: Berlin/Heidelberg, Germany, 2022; pp. 499–506.
6. Vanishree, K.; George, A.; Gunisetty, S.; Subramanian, S.; Kashyap, S.; Purnaprajna, M. CoIn: Accelerated CNN Co-Inference through data partitioning on heterogeneous devices. In Proceedings of the 2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS), Coimbatore, India, 6–7 March 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 90–95.

7. Sujadevi, V.G.; Soman, K.P. Towards identifying most important leads for ECG classification. A Data driven approach employing Deep Learning. *Procedia Comput. Sci.* **2020**, *171*, 602–608. [CrossRef]
8. Madsen, A. Visualizing memorization in RNNs. *Distill* **2019**, *4*, e16. [CrossRef]
9. Bengio, Y.; Simard, P.; Frasconi, P. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* **1994**, *5*, 157–166. [CrossRef] [PubMed]
10. Asmitha, U.; Roshan Tushar, S.; Sowmya, V.; Soman, K.P. Ensemble Deep Learning Models for Vehicle Classification in Motorized Traffic Analysis. In Proceedings of the International Conference on Innovative Computing and Communications, Delhi, India, 17–18 February 2023; Gupta, D., Khanna, A., Bhattacharyya, S., Hassanien, A.E., Anand, S., Jaiswal, A., Eds.; Spinger: Singapore, 2023; pp. 185–192.
11. Ramakrishnan, R.; Vadakedath, A.; Bhaskar, A.; Sachin Kumar, S.; Soman, K.P. Data-Driven Volatile Cryptocurrency Price Forecasting via Variational Mode Decomposition and BiLSTM. In Proceedings of the International Conference on Innovative Computing and Communications, Delhi, India, 17–18 February 2023; Gupta, D., Khanna, A., Bhattacharyya, S., Hassanien, A.E., Anand, S., Jaiswal, A., Eds.; Spinger: Singapore, 2023; pp. 651–663.
12. Pascanu, R.; Mikolov, T.; Bengio, Y. On the difficulty of training recurrent neural networks. In Proceedings of the International Conference on Machine Learning, PMLR, Atlanta, GA, USA, 17–19 June 2013; pp. 1310–1318.
13. Lin, J. Efficient Algorithms and Systems for Tiny Deep Learning. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2021.
14. Han, S.; Kang, J.; Mao, H.; Hu, Y.; Li, X.; Li, Y.; Xie, D.; Luo, H.; Yao, S.; Wang, Y.; et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 75–84.
15. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
16. Graves, A. Generating sequences with recurrent neural networks. *arXiv* **2013**, arXiv:1308.0850.
17. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv* **2014**, arXiv:1412.3555.
18. Cho, K.; Van Merriënboer, B.; Bahdanau, D.; Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv* **2014**, arXiv:1409.1259.
19. Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv* **2014**, arXiv:1406.1078.
20. Arjovsky, M.; Shah, A.; Bengio, Y. Unitary evolution recurrent neural networks. In Proceedings of the International Conference on Machine Learning, PMLR, New York, NY, USA, 20–22 June 2016; pp. 1120–1128.
21. David, R.; Duke, J.; Jain, A.; Janapa Reddi, V.; Jeffries, N.; Li, J.; Kreeger, N.; Nappier, I.; Natraj, M.; Wang, T.; et al. TensorFlow lite micro: Embedded machine learning for tinyml systems. *Proc. Mach. Learn. Syst.* **2021**, *3*, 800–811.
22. *Microsoft-v2019* ; ELL: Embedded Learning Library; Microsoft Corporation: Redmond, WA, USA, 2018.
23. Banbury, C.; Zhou, C.; Fedorov, I.; Matas, R.; Thakker, U.; Gope, D.; Janapa Reddi, V.; Mattina, M.; Whatmough, P. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proc. Mach. Learn. Syst.* **2021**, *3*, 517–532.
24. Gu, F.; Chung, M.H.; Chignell, M.; Valaee, S.; Zhou, B.; Liu, X. A survey on deep learning for human activity recognition. *ACM Comput. Surv.* **2021**, *54*, 1–34. [CrossRef]
25. Jordao, A.; Nazare, A.C., Jr.; Sena, J.; Schwartz, W.R. Human activity recognition based on wearable sensor data: A standardization of the state-of-the-art. *arXiv* **2018**, arXiv:1806.05226.
26. Demrozi, F.; Turetta, C.; Pravadelli, G. B-HAR: An open-source baseline framework for in depth study of human activity recognition datasets and workflows. *arXiv* **2021**, arXiv:2101.10870.
27. Olah, C. Understanding LSTM Networks. 2015. Available online: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (accessed on 22 May 2022).
28. Kusupati, A.; Singh, M.; Bhatia, K.; Kumar, A.; Jain, P.; Varma, M. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In *Advances in Neural Information Processing Systems*; MIT Press: Cambridge, MA, USA, 2018; Volume 31.
29. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
30. Castellano, G.; Fanelli, A.M.; Pelillo, M. An iterative pruning algorithm for feedforward neural networks. *IEEE Trans. Neural Netw.* **1997**, *8*, 519–531. [CrossRef] [PubMed]
31. Reed, R. Pruning algorithms-a survey. *IEEE Trans. Neural Netw.* **1993**, *4*, 740–747. [CrossRef] [PubMed]
32. Guo, Y.; Yao, A.; Chen, Y. Dynamic network surgery for efficient dnns. In *Advances in Neural Information Processing Systems*; MIT Press: Cambridge, MA, USA, 2016; Volume 29.
33. Gao, C.; Neil, D.; Ceolini, E.; Liu, S.C.; Delbruck, T. DeltaRNN: A power-efficient recurrent neural network accelerator. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 21–30.

34. Yao, S.; Zhao, Y.; Zhang, A.; Su, L.; Abdelzaher, T. Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework. In Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, Delft, The Netherlands, 6–8 November 2017; pp. 1–14.
35. Wang, S.; Li, Z.; Ding, C.; Yuan, B.; Qiu, Q.; Wang, Y.; Liang, Y. C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Delft, The Netherlands, 6–8 November 2018; pp. 11–20.
36. Anwar, S.; Hwang, K.; Sung, W. Structured pruning of deep convolutional neural networks. *ACM J. Emerg. Technol. Comput. Syst. (JETC)* **2017**, *13*, 1–18. [CrossRef]
37. Wen, L.; Zhang, X.; Bai, H.; Xu, Z. Structured pruning of recurrent neural networks through neuron selection. *Neural Netw.* **2020**, *123*, 134–141. [CrossRef]
38. Thakker, U.; Beu, J.; Gope, D.; Dasika, G.; Mattina, M. Run-time efficient RNN compression for inference on edge devices. In Proceedings of the 2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), Washington, DC, USA, 17 February 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 26–30.
39. Shan, D.; Luo, Y.; Zhang, X.; Zhang, C. DRRNets: Dynamic Recurrent Routing via Low-Rank Regularization in Recurrent Neural Networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *34*, 2057–2067. [CrossRef]
40. Zhao, Y.; Li, J.; Kumar, K.; Gong, Y. *Extended Low-Rank Plus Diagonal Adaptation for Deep and Recurrent Neural Networks*; IEEE Press: Piscataway, NJ, USA, 2017. [CrossRef]
41. Lu, Z.; Sindhwani, V.; Sainath, T.N. Learning compact recurrent neural networks. In Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China, 20–25 March 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 5960–5964.
42. Prabhavalkar, R.; Alsharif, O.; Bruguier, A.; McGraw, L. On the compression of recurrent neural networks with an application to LVCSR acoustic modeling for embedded speech recognition. In Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China, 20–25 March 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 5970–5974.
43. Vanhoucke, V.; Senior, A.; Mao, M.Z. Improving the Speed of Neural Networks on CPUs. 2011. Available online: https://research.google/pubs/improving-the-speed-of-neural-networks-on-cpus/ (accessed on 10 February 2024).
44. Ramakrishnan, R.; Dev, A.K.; Darshik, A.; Chinchwadkar, R.; Purnaprajna, M. Demystifying Compression Techniques in CNNs: CPU, GPU and FPGA cross-platform analysis. In Proceedings of the 2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID), Virtual, 20–24 February 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 240–245.
45. Warden, P.; Situnayake, D. TinyML. 2019. Available online: https://www.oreilly.com/library/view/tinyml/9781492052036/ (accessed on 10 February 2024).
46. Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things. *IEEE Internet Things J.* **2020**, *7*, 4403–4417. [CrossRef]
47. Biewald, L. Experiment Tracking with Weights and Biases. 2020. Available online: https://www.wandb.com (accessed on 30 June 2022).
48. Bulling, A.; Blanke, U.; Schiele, B. A tutorial on human activity recognition using body-worn inertial sensors. *ACM Comput. Surv. (CSUR)* **2014**, *46*, 1–33. [CrossRef]
49. Altun, K.; Barshan, B. Human activity recognition using inertial/magnetic sensor units. In Proceedings of the International Workshop on Human Behavior Understanding, Istanbul, Turkey, 22 August 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 38–51.
50. Anguita, D.; Ghio, A.; Oneto, L.; Parra Perez, X.; Reyes Ortiz, J.L. A public domain dataset for human activity recognition using smartphones. In Proceedings of the 21th International European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges, Belgium, 24–26 April 2013; pp. 437–442.
51. Chavarriaga, R.; Sagha, H.; Calatroni, A.; Digumarti, S.T.; Tröster, G.; Millán, J.d.R.; Roggen, D. The Opportunity challenge: A benchmark database for on-body sensor-based activity recognition. *Pattern Recognit. Lett.* **2013**, *34*, 2033–2042. [CrossRef]
52. Reiss, A.; Stricker, D. Creating and benchmarking a new dataset for physical activity monitoring. In Proceedings of the 5th International Conference on PErvasive Technologies Related to Assistive Environments, Crete, Greece, 6–9 June 2012; pp. 1–8.
53. Banos, O.; Garcia, R.; Holgado-Terriza, J.A.; Damas, M.; Pomares, H.; Rojas, I.; Saez, A.; Villalonga, C. mHealthDroid: A novel framework for agile development of mobile health applications. In Proceedings of the International Workshop on Ambient Assisted Living, Belfast, UK, 2–5 December 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 91–98.
54. Zhang, M.; Sawchuk, A.A. USC-HAD: A daily activity dataset for ubiquitous activity recognition using wearable sensors. In Proceedings of the 2012 ACM Conference on Ubiquitous Computing, Pittsburgh, PA, USA, 5–8 September 2012; pp. 1036–1043.
55. Bruno, B.; Mastrogiovanni, F.; Sgorbissa, A. Wearable inertial sensors: Applications, challenges, and public test benches. *IEEE Robot. Autom. Mag.* **2015**, *22*, 116–124. [CrossRef]
56. Lockhart, J.W.; Weiss, G.M.; Xue, J.C.; Gallagher, S.T.; Grosner, A.B.; Pulickal, T.T. Design considerations for the WISDM smart phone-based sensor mining architecture. In Proceedings of the Fifth International Workshop on Knowledge Discovery from Sensor Data, San Diego, CA, USA, 21 August 2011; pp. 25–33.
57. Domingos, P. A few useful things to know about machine learning. *Commun. ACM* **2012**, *55*, 78–87. [CrossRef]

58. Dennis, D.K.; Gaurkar, Y.; Gopinath, S.; Goyal, S.; Gupta, C.; Jain, M.; Jaiswal, S.; Kumar, A.; Kusupati, A.; Lovett, C.; et al. EdgeML: Machine Learning for Resource-Constrained Edge Devices. 2019. Available online: https://github.com/Microsoft/EdgeML (accessed on 30 June 2022).
59. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016. Available online: http://www.deeplearningbook.org (accessed on 20 May 2022).
60. LeCun, Y.A.; Bottou, L.; Orr, G.B.; Müller, K.R. Efficient backprop. In *Neural Networks: Tricks of the Trade*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 9–48.
61. Zaremba, W.; Sutskever, I.; Vinyals, O. Recurrent neural network regularization. *arXiv* **2014**, arXiv:1409.2329.