Efficient Embedded Machine Learning applications using Echo State Networks

L. Cerina[1], M. D. Santambrogio[1], G. Franco[2], C. Gallicchio[2], A. Micheli[2]
[1]Dipartimento Elettronica Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy
[2]Department of Computer Science, University of Pisa, Italy

Abstract—**The increasing role of Artificial Intelligence (AI) and Machine Learning (ML) in our lives brought a paradigm shift on how and where the computation is performed. Stringent latency requirements and congested bandwidth moved AI inference from Cloud space towards end-devices. This change required a major simplification of Deep Neural Networks (DNN), with memory-wise libraries or co-processors that perform fast inference with minimal power. Unfortunately, many applications such as natural language processing, time-series analysis and audio interpretation are built on a different type of Artifical Neural Networks (ANN), the so-called Recurrent Neural Networks (RNN), which, due to their intrinsic architecture, remains too complex and heavy to run efficiently on embedded devices. To solve this issue, the Reservoir Computing paradigm proposes sparse untrained non-linear networks, the Reservoir, that can embed temporal relations without some of the hindrances of Recurrent Neural Networks training, and with a lower memory usage. Echo State Networks (ESN) and Liquid State Machines are the most notable examples. In this scenario, we propose a performance comparison of a ESN, designed and trained using Bayesian Optimization techniques, against current RNN solutions. We aim to demonstrate that ESN have comparable performance in terms of accuracy, require minimal training time, and they are more optimized in terms of memory usage and computational efficiency. Preliminary results show that ESN are competitive with RNN on a simple benchmark, and both training and inference time are faster, with a maximum speed-up of 2.35x and 6.60x, respectively.**

I. Introduction

We can observe in our everyday life how Artificial Intelligence (AI) and embedded technologies are shaping every aspect of our life. Digital assistants listen to us on our smartphone, while a plethora of devices monitor our cities, offices and houses with cameras, microphones and other connected sensors. This huge amount of data is then processed somewhere in the Cloud to extract meaningful informations so that the devices themselves can become more and more proactive. In this scenario, high-quality/low-latency services are becoming a selling point of tech enterprises, while privacy protection is required by regulators. These concerns in particular, along with the congestion of wireless bandwidth and novel computing devices, drove a major redistribution of computing resources from the Cloud to endpoints to perform processing near to the source of data (see Fog Computing [1]).

However, this approach required to transfer algorithms and frameworks already optimized for large scale systems, into devices with an exiguous amount of resources. One of the most notable examples are Deep Neural Networks (DNN): although the training of networks is still performed at high precision levels with energy-hungry GPUs, the inference of Convolutional Neural Networks (CNN) models moved to specialized end devices. The network is reduced using pruning, distillation, weight compression or precision loss among others [2], and executed on mobile CPUs or AI-centred ASICs.

On the other hand, every application that requires the learning of temporal relations, e.g. pattern recognition in time-series, or Natural Language Processing, is usually performed using Recurrent Neural Networks (RNN). RNN architectures are built upon more complex neurons, such as Gated Recurrent Units (GRU) [3], which embeds temporal knowledge through weights and specific activation rules. This intrinsic complexity, unoptimized for CNN co-processors, contributed to a lower utilization of RNN in Edge contexts.

In this case, the Reservoir Computing (RC) field [4] offers valuable alternatives in the RNN field to these complex architectures, with Echo State Networks (ESN) [5] and Liquid State Machines (LSM) [6] as the two most famous examples. ESNs in particular propose peculiar solution: instead of learning a large set of weights, a sparse untrained matrix of random weights, called Reservoir, transforms the input into a N-dimensional non-linear state. After *encoding* the inputs in the reservoir states, the training happens only on the fully connected linear readout layer, that transforms the reservoir states into the output. Under this assumption, ESN provides a notable advantage over classical RNN in memory and computational time, even for complex tasks such as polyphonic music predictions [7]. This is noteworthy on embedded devices, where it will possible to deploy complex models keeping computational and energy requirements low.

We propose here an efficient and comprehensive C++ ESN library which is optimized for embedded devices, but at the same time is guaranteed to be scalable on more powerful systems. The choice of network hyper-parameters is solved using Bayesian Optimization (BO) strategies, that can help overcome the limitation associated to grid-search or random search approaches.

The major contribution of this work are as follows:

- An efficient, portable and modular implementation of ESN.
- A comparison of ESN and RNN on a state-of-the-art classification benchmark.

The rest of the paper is organized as follows. We briefly introduce the current state of the art in Section II. Section

III, presents general ESNs characteristics and our proposed methodology. Experimental results are presented in Section IV, while Section V summarizes our conclusions.

II. Related work

This Section describes the literature concerning applications of Recurrent Neural Networks on embedded or constrained devices.

Although it is not uncommon to find research publications and projects that exploit CNN for the analysis of time-series, they require a certain manipulation of the data, such as domain transformations (e.g. calculating a spectrogram which is fed to the network). In our case, we will focus only on works that uses proper RNN.

The authors from [8] focused on simplifying the structure of GRU and enforcing low-rank, sparse model representation to fit inside constrained devices. The project considers the impact on model compression and prediction costs with good results. Similarly, the authors in [9] compressed the model weights using Kronecker products, resulting in a speed-up up to 4x, but with a slight loss in accuracy. The work in [10] exploits binarized Long-Short Term Memory (LSTM) to reach the maximum compression possible, with good accuracy in activity recognition.

Speech recognition models are analysed in [11], here the authors employ joint factorization methods to reduce the model size by one third, with negligible loss in accuracy.

A comparison of different RNN architectures is available in [12], the model selection was performed starting from the constraints of the final device and limiting the search space of hyper-parameters accordingly. One of the few works that explicitly targets RNN on mobile devices is [13]: both multithreading and GPU offloading are explored to increase inference speed.

III. Proposed methodology

This section outlines the model behind Echo State Networks and the difference with current RNN methods. We present also a brief description of advantages and drawbacks of both methods.

Model Background

The most simple model of ESN with leaky feedback integration is defined as:

$$\begin{cases} x_t &= (1-\alpha)\, x_{t-1} + \alpha\, \sigma(\mathbf{W_{in}}\, u_t + \mathbf{W_r}\, x_{t-1}) \\ y_t &= \mathbf{W_{out}}\, x_t \end{cases} \tag{1}$$

where $\mathbf{x}_t \in \mathbb{R}^{N_r}$ represents the reservoir states, $\mathbf{u}_t \in \mathbb{R}^{N_u}$ are the input signals, and $\mathbf{y}_t \in \mathbb{R}^{N_o}$ are the predicted outputs. The initial state $\mathbf{x}_0$ is usually set to zero. Here $\mathbf{W_{in}}$ is $N_r \times N_u$ random matrix that expand the dimensionality of input $\mathbf{u}_t$, and $\mathbf{W_r}$ is $N_r \times N_r$ highly sparse random Reservoir, which sparsity is determined by the $\mathbf{s}$ *density* parameter (here expressed in the range $\in (0, 1]$). At the output $\mathbf{W_{out}}$ is a $N_o \times N_r$ feed-forward readout, trained (in a least-square sense) to approximate a set of states $\mathbf{x}_{1:t}$ to the true value of $\mathbf{y}_{1:t}$. Depending on the architecture, also the input and the states of each layer (e.g. DeepESN [14]) are concatenated to calculate $\mathbf{W_{out}}$. Additionally, the term $\alpha \in (0, 1]$ represents the *leaky factor* that controls internal dynamics with respect to the variations of $\mathbf{u}_t$. Lastly, $\sigma(\cdot)$, is a non-linear sigmoid activation function, generally $tanh(\cdot)$. At the beginning of the computation the initial state x_0 is initialized to 0, and a small amount of transient states are then discarded as washout. This procedure is performed each time the internal state is reset.

Conversely, GRUs are defined by:

$$\begin{cases} z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\ \tilde{h_t} &= tanh(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h_t} \end{cases} \tag{2}$$

Where the reset gate $\mathbf{r}$ defines how new inputs are combined with the current state, while the update gate $\mathbf{z}$ how much of the past state must be remembered. Also here $\sigma(\cdot)$ is a sigmoid or hyperbolic activation function. The hidden states are then sent to a linear layer to determine the output.

Observing the two models together, we can see how ESNs are fundamentally a simpler version of GRU: what here is done by $\mathbf{z}$ and $\mathbf{r}$, in the ESN is built with linear weights, namely the leaky factor α and the Reservoir itself $\mathbf{W_r}$. The great difference between the two relies on the fact that where GRU require few units, but a long training phase necessary to precisely tune all the weights, the ESN embed memory and input representation in a larger and sparse set of active units (each non-zero element of $\mathbf{W_r}$). The role of trained $\mathbf{W_{out}}$ is transform the chaos of Reservoir states into order, but since training is performed using linear regression, it is quite fast and efficient and does not require multiple runs on the same network.

Nevertheless, this reduced complexity comes at a cost. The quasi-randomness of the ESN must be controlled to avoid divergent behaviours of non-linear states, and this requires a non-trivial tuning of hyper-parameters. More details on this in Section IV. To mitigate this intrinsic randomness, ESN researchers proposed more ordered, low-rank topologies such as Cyclic Reservoir Jump (CRJ) [15].

ESN library

Our Reservoir library is built entirely in C++ and it employs Eigen (eigen.tuxfamily.org) as a mathematical backend. The Reservoir class is self-contained and it provides support for various topologies, as well as DeepESN models. Each layer is highly parametrizable to give the maximum control over the Reservoir.

In the field of RNN training, the network's sparsity remains an open dilemma. Although the effects on memory footprint and inference speed are evident, the efficacy of compression methods is inconsistent and accuracy results are slightly lower [16]. In order to avoid re-training or complex pruning processes, researchers are modifying optimization rules to maintain network's sparsity during the training [17].

On the other hand, our library directly embraces sparsity: ESN layers are initialized with 70% to 99% sparse connections, depending on their topology. This important characteristic allows us to exploit SpMDV (Sparse Matrix multiplies Dense Vector) operations. Thanks to sparse multiplications, it is possible to run large ESN models efficiently on architectures that are not equipped with a GPU or SIMD co-processors. At the same time, our library is built to be scalable also on large computing systems, so it natively supports OpenMP to share operations across multiple threads seamlessly. Currently, we tested the library on x86 machines, ARM Cortex processors (e.g. Raspberry Pi) and Tensilica XTensa LX6 (namely the ESP32 microcontroller).

In addition to the Reservoir module, our library provides a module to update and memorize internal states, and a training module that supports both regression and classification tasks.

Since ESN performance is dependent heavily both on its hyper-parameters and on its weight optimizations, we plugged our library into a Bayesian Optimizer [18] to find the best set for each task. BO starts from a random sampling in the parameter space, it builds an a-priori model of the fitness function, and then sequentially searches the next sample that is expected to improve the score. For each set of parameters, the BO performs three guesses to account instabilities caused by the quasi-randomness of Reservoirs. In this case, the optimization time is influenced only by the number of hyper-parameters (that increase the initial random samples), and the dimension of the ESN in terms of active units.

IV. Experimental results

This Section describes the experimental setup use to test our methodology. The results compare a classification task that involves the analysis of digital trajectories and we will evaluate accuracy, training time, and model efficiency.

We wanted to focus on systems with hard constraints in terms of memory and computational resources, and we narrowed down the comparison of different Machine Learning (ML) frameworks. Numerical environments such as Matlab and Julia were excluded due to their *heavy* runtime. We then focused on PyTorch (pytorch.org), a quite common ML framework, as it provides a *bare-metal* C++ frontend, and it is built upon an efficient and lightweight mathematical library. We selected architectural solutions from the State-of-the-Art, then replicated them and deployed on the embedded system.

Experimental setup

The training phase of both RNN and ESN was performed on a x86 desktop machine, equipped with an Intel i7-6700 CPU (3.4GHz, 8 threads) with 32GB DDR4 RAM memory and a NVidia Geforce GTX 960. Conversely, we tested the inference speed of the models on a Raspberry Pi 3B+ (CortexA53@1.4GHz) with 1GB LPDDR2 memory. Unfortunately PyTorch or any other ML framework are not compatible or optimized for ESP32 microcontroller and it would have provided an unfair comparison.

The dataset employed is the Characters Trajectories Dataset [19], which describes the trajectory and pen-tip force of handwritten letters. The dataset provides 2858 samples divided in 20 classes. We used a 55%-25%-20% separation for Training, Validation and Test and the samples were randomly selected to maximize heterogeneity. Each trajectory was rescaled to $[-1, 1]$ range.

Training and memory occupation

The comparison included three different ESN topologies against two different types of gated RNN. In particular, considering ESN, we tested shallow Reservoir with Random or Cyclic topology (CRJ), and a deep Reservoir with Random topology. The upper boundaries for ESN dimension and sparsity are: $[400, 20\%]$ for random shallow and $[200, 20\%]$ for the deep reservoir. The CRJ network instead has a maximum dimension of 800 and a maximum jump of 40. The other parameters that are optimized by BO are the spectral radius, the input scaling and the leaky factor. The regularization factor of $\mathbf{W_{out}}$ is fixed at 0.001.

In a ESN model, the number of free parameters is entirely dependent by $\mathbf{W_{out}}$ and it is evaluated as the dimension of the last layer, multiplied by the number of classes in output. However, we added also the number of active units in the Reservoir and the input layer $\mathbf{W_{in}}$ to the calculation, even if their *training* has a computational cost near to zero.

We compared the ESN against the GRU and FastGRNN [8], we used the training system proposed by the authors for both networks. This system has more strict requirements in terms of data formatting, so for these cases the time series were zero-padded, in order to grant that all sequences have the same length. The training was performed using 150 epochs, but leaving all the other settings to their default values. In this case the number of Neural Units was chosen in such a way that the amount of free parameters is comparable to those of the ESN architectures.

TABLE I: Performance results, training time and model dimension for different RNN and ESN models. ESN's training time includes both hyper-parameter search and proper training.

Model	**Accuracy**	**Training time**	**Free parameters** (trained weights)	**Neural Units**
GRU	98.870	7m 18s	12920 (all)	60
FastGRNN	98.320	7m 30s	12956 (all)	102
ESN-CRJ	98.31	5m 54s	12118 (9340)	926
ESN-Random	98.739	3m 11s	11680 (8080)	1996
DeepESN	99.159	7m 7s	27121 (5860)	1961

TABLE I summarizes the performance in accuracy of different models: both methods reached a comparable performance in the test set. Although the speed-up in training is limited to 2.35x, there are some differences between the two methods that must be taken in consideration. First of all, our process combines both training and hyper-parameter selection, which was not done for the RNN training. Furthermore, PyTorch exploits the GPU capabilities and extremely parallel batching, while our library runs well even on CPU.

Looking at the DeepESN example, we can also state that the strength of ESN lies in the simplicity of the $\mathbf{W_{out}}$ training: although the number of free parameters is larger than the other two, the final number of trained weights is actually smaller, thanks to smaller concatenated states (293 vs 467, 404). This characteristic implies that complex deep models could be trained in the same amount of time of shallow, but larger reservoirs. The same logic does not apply for RNN.

Inference speed

In order to evaluate the performance of all systems, we deployed them on the Raspberry Pi, and measured how much time was necessary to recognize a single letter. To minimize the variance of the performance, we evaluated all instances of the dataset 10 times and then averaged the results. Furthermore, to minimize the discrepancy between Python and C++, the trained RNN models were loaded and run using the C++ frontend.

TABLE II: Average inference time for RNN and ESN models

Model	**RNN**		**ESN**		
	GRU	FastGRNN	CRJ	Random	DeepESN
Inference time [ms]	30.53	29.72	11.90	4.62	27.60

From TABLE II we can notice that ESN outperform RNN systems, with a speed-up ranging from 1.07x up to 6.60x. The result however is two-fold, although ESN are intrinsically faster due to their lower complexity, the PyTorch framework was designed with desktop CPUs and GPUs in mind, and it is not optimized for ARM devices. With respect to the ESN, we can see that DeepESN is slightly more accurate, thanks to the broader representation of internal states, but the inter-layer matrices are large and dense (100x189 in our example) and slow down the computation. To minimize the problem, some works in literature [20] used dimensionality reduction techniques between subsequent layers.

V. Conclusions

In this paper, we presented an alternative method to Recurrent Neural Networks based on the Reservoir Computing paradigm, optimized for embedded systems. The proposed system obtains similar performance in accuracy, faster inference time due to its simplified architecture and a lower training time. The reduced complexity could allow also to train ESN directly on the target device, without a Cloud-scale system. It should be stated that ESN does not represent a plain substitute for RNN, since there is little or no literature to demonstrate the performance of ESN on complex tasks such as speech recognition. Instead, they could be an alternative in some tasks that are designed to run on smaller, constrained devices where RNN would be way too limited in size or performance.

While our library is competitive in the training time, future developments will include a number of optimization to reduce it, namely mini-batches, parallel guesses and the possibility to take advantage of GPU-based devices.

References

[1] J. Lin, W. Yu *et al.*, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, 2017.

[2] J. Gu, Z. Wang *et al.*, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, 2018.

[3] J. Chung, C. Gulcehre *et al.*, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[4] D. Verstraeten, B. Schrauwen *et al.*, "An experimental unification of reservoir computing methods," *Neural networks*, vol. 20, no. 3, 2007.

[5] H. Jaeger and H. Haas, "Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication," *science*, vol. 304, no. 5667, 2004.

[6] W. Maass and H. Markram, "On the computational power of circuits of spiking neurons," *Journal of computer and system sciences*, vol. 69, no. 4, 2004.

[7] C. Gallicchio, A. Micheli, and L. Pedrelli, "Comparison between deepesns and gated rnns on multivariate time-series prediction," *CoRR*, vol. abs/1812.11527, 2018. [Online]. Available: http://arxiv.org/abs/1812.11527

[8] A. Kusupati, M. Singh *et al.*, "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Advances in Neural Information Processing Systems*, 2018.

[9] U. Thakker, J. G. Beu *et al.*, "Compressing rnns for iot devices by 15-38x using kronecker products," *CoRR*, vol. abs/1906.02876, 2019. [Online]. Available: http://arxiv.org/abs/1906.02876

[10] M. Edel and E. Köppe, "Binarized-blstm-rnn based human activity recognition," in *2016 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*. IEEE, 2016.

[11] R. Prabhavalkar, O. Alsharif *et al.*, "On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016.

[12] Y. Zhang, N. Suda *et al.*, "Hello edge: Keyword spotting on microcontrollers," *CoRR*, vol. abs/1711.07128, 2017. [Online]. Available: http://arxiv.org/abs/1711.07128

[13] Q. Cao, N. Balasubramanian, and A. Balasubramanian, "Mobirnn: Efficient recurrent neural network execution on mobile gpu," in *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, ser. EMDL '17. New York, NY, USA: ACM, 2017.

[14] C. Gallicchio, A. Micheli, and L. Pedrelli, "Deep reservoir computing: a critical experimental analysis," *Neurocomputing*, vol. 268, 2017.

[15] A. Rodan and P. Tino, "Minimum complexity echo state network," *IEEE Transactions on Neural Networks*, vol. 22, no. 1, Jan 2011.

[16] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *CoRR*, vol. abs/1902.09574, 2019. [Online]. Available: http://arxiv.org/abs/1902.09574

[17] T. Dettmers and L. Zettlemoyer, "Sparse networks from scratch: Faster training without losing performance," *CoRR*, vol. abs/1907.04840, 2019. [Online]. Available: http://arxiv.org/abs/1907.04840

[18] A. Cully, K. Chatzilygeroudis *et al.*, "Limbo: A Flexible High-performance Library for Gaussian Processes modeling and Data-Efficient Optimization," *The Journal of Open Source Software*, vol. 3, no. 26, 2018.

[19] B. Williams, M. Toussaint, and A. J. Storkey, "Modelling motion primitives and their timing in biologically executed movements," in *Advances in neural information processing systems*, 2008.

[20] H. M. Nguyen, G. Kalra *et al.*, "Esnemble: an echo state network-based ensemble for workload prediction and resource allocation of web applications in the cloud," *The Journal of Supercomputing*, 2019.