Accelerating RNN Controllers with Parallel Computing and Weight Dropout Techniques

Maxwell Sam[1], Kushal Kalyan Devalmapeta Surendranath[1], Xingang Fu[2], and Letu Qingge[1(✉)]

[1] Department of Computer Science, North Carolina A&T State University, Greensboro, NC, USA
lqingge@ncat.edu

[2] Department of Electrical and Biomedical Engineering, University of Nevada, Reno, NV, USA

Abstract. Training recurrent neural network controllers in closed-loop control systems with combined Levenberg-Marquardt and Forward Accumulation Through Time algorithm advances the research in a grid-connected converter for solar integration to a power system. However, an effective training algorithm is needed for a large number of trajectories with a high sampling frequency. Thus, we propose a new effective training mechanism based on parallel computing and weight dropout techniques for recurrent neural network controllers in this paper. Experimental results on both the Amazon Web Services (AWS) cloud and the Graphical Processing Unit (GPU) show that our proposed training mechanism runs at a more promising acceleration rate than the existing algorithms.

Keywords: Recurrent Neural Network Controllers · Levenberg-Marquardt · Forward Accumulation Through Time · Cloud Computing · Parallel Trajectory Training · Weight Dropout

1 Introduction

Recurrent Neural Networks (RNNs) have become a capstone in the field of sequential data processing and control systems due to their ability to maintain a memory of previous inputs through internal state dynamics. However, the training of RNNs for control tasks can be challenging due to the complexity of their error surfaces and the need for efficient computation of the training algorithm. The Levenberg-Marquardt (LM) algorithm, a powerful optimization technique that combines the rapid convergence of Newton's method with the stability of gradient descent, has shown promise in training feed-forward networks but is less explored in the context of RNNs. Recent advancements propose the integration

This work is supported by the National Science Foundation of the United States under Award 2131175 and 2131214.

H. Fujita et al. (Eds.): IEA/AIE 2024, LNAI 14748, pp. 401–412, 2024.
https://doi.org/10.1007/978-981-97-4677-4_33

of the LM algorithm with a novel Forward Accumulation Through Time (FATT) method to efficiently compute the Jacobian matrix required for RNN training [1]. This approach aims to accelerate the training process by leveraging the strengths of LM in handling non-linear optimization problems while addressing the computational demands of backpropagation through time (BPTT) in RNNs.

Novel computational approaches have contributed to recent improvements in the training efficiency of recurrent neural network (RNN) controllers in closed-loop systems. To improve the training process, this research combines weight dropout techniques with parallel trajectory training, utilizing the Levenberg-Marquardt (LM) and Forward Accumulation Through Time (FATT) algorithms. Previous research, most notably the study by [7], which presented a weight dropout technique to optimize neural network controllers for solar inverters and greatly reduced the computational burden on FPGA implementations, served as an inspiration for our strategy.

Further foundational to this research is the study by Ranga Suri, Deodhare, and Nagabhushan [1], which explores "Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters," offering early insights into the effectiveness of parallel computing for neural network training in a case study context [1]. Similarly, Cao et al. present "A Parallel Levenberg-Marquardt Algorithm," detailing an algorithmic approach that underscores the potential of parallel processing to significantly accelerate the computational aspects of neural network training [2]. This body of work and the suggested improvements together highlight the value of novel algorithmic techniques and parallel computing in enhancing the effectiveness of RNN controller training procedures essential for their use in demanding real-world scenarios requiring strong control capabilities and high computational efficiency.

Taking this work further, this research extends the use of weight dropout in a parallel trajectory training framework to greatly accelerate the training of RNN controllers. This methodology shows a significant improvement in training time over conventional methods, as demonstrated by tests conducted on platforms like GPU clusters and Amazon EC2.

The structure of the paper unfolds as follows: In Sect. 2, we delve into the experiment, offering insights into the RNN controllers' parallel trajectory training algorithm, with and without the implementation of dropout techniques. Section 3 elaborates on the detailed training results and trajectory performance. Finally, the paper concludes with a summary of key points in Sect. 4, including the notable findings and contributions.

2 RNN Controllers in a Closed Loop Control System

2.1 A Solar Microinverter

Solar inverters have emerged as pivotal components in photovoltaic systems, revolutionizing solar energy technology by enabling individual panel optimization, and enhancing energy production and system performance as compared to traditional string inverters [4]. Essential components of a solar inverter are the

DC-DC converter and the DC-AC inverter as illustrated in Fig. 1 [5]. The PhotoVoltaic (PV) solar panels are connected to the DC-DC converters for voltage optimization. Pulse Width Modulation (PWM) controls a signal generator that adjusts the duty cycle of the output waveform to regulate the power delivered to the load. DC-AC inverters maintain DC Bus voltage and supply controlled AC current to the main power grid. This process ensures efficient conversion of solar energy for grid integration.

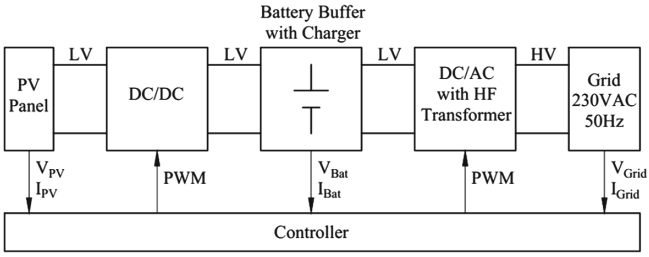


Fig. 1. Microinverter Block Diagram [5]

2.2 NN Controller in a Closed-Loop

A Neural Network Controller will be implemented in the Piccolo real-time digital controller in Fig. 1. This is intended to regulate the currents following the reference trajectories in a closed-loop control system. The structure of the proposed NN Controller is shown in Fig. 2. Two hidden layers, each with six neurons, and an additional layer with two neurons controlling the outputs make up the neural network (NN).

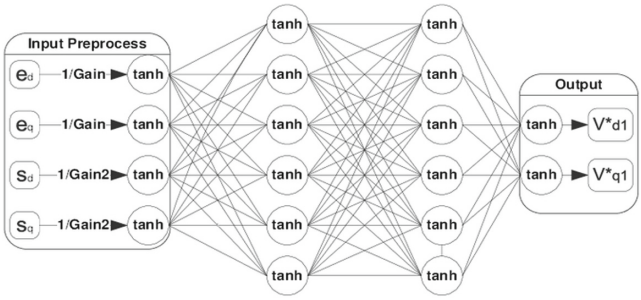


Fig. 2. The NN Controller with special tracking error integrals [6]

The tracking error input signals $\overrightarrow{e_{dq}}$ and their particular error integral values $\overrightarrow{s_{dq}}$ are sent into the NN's input block. In order to prevent input saturation, $\overrightarrow{e_{dq}}$ and $\overrightarrow{s_{dq}}$ are normalized using the hyperbolic tangent function, whose values are restricted to the interval [-1, 1], and divided by constant gain values, $Gain$ and $Gain2$, respectively. For step references [9], the special error integral terms $\overrightarrow{s_{dq}}$ will ensure that there is no steady-state error.

Through several trial and error trials, the number of neurons in each hidden layer was determined. It was found that using six nodes in each hidden layer produced good real-time control results. The process involved several iterations of experimentation to determine the optimal number of neurons in each hidden layer of the neural network. Different configurations of hidden layer neuron counts were tested, and their performance in achieving real-time control results was evaluated. After conducting multiple trials, it was observed that utilizing six nodes in each hidden layer consistently yielded satisfactory real-time control outcomes. Additionally, the dropout technique makes it possible to reduce the number of neurons or weights, which improves compatibility with embedded real-time computing systems [7].

3 Parallel Trajectory Training with Weight Dropout Technique

Recurrent neural networks (RNNs) for closed-loop control systems can be better trained by parallelizing trajectory training using dropout regularization and parallel computing resources. Adaptive dynamic programming (ADP) methodologies, according to [10], integrate incremental optimization techniques with parametric structures to estimate the optimal cost for system control. Specifically, a discrete-time ADP method is based on Bellman's optimality principle [11] and employs a discrete-time system model in conjunction with a cost or performance index [12]. The cost function associated with training recurrent neural networks (RNNs) is called Dynamic Programming (DP).

The Dynamic programming (DP) cost function associated with the RNN training is defined as:

$$C_{dp} = \sum_{k=j}^{\infty} \gamma^{k-j} U(\boldsymbol{e}_{dq}(k)) = \sum_{k=j}^{\infty} \gamma^{k-j} \sqrt{(id(k) - id_{\text{ref}}(k))^2 + (iq(k) - iq_{\text{ref}}(k))^2} \tag{1}$$

where $j > 0$ is the starting point, $0 < \gamma \leq 1$ is a discount factor, and U is the local cost or utility function. Depending on the initial time j and the initial state $\boldsymbol{idq}(j)$, the function C_{dp} is referred to as the cost-to-go of state $\boldsymbol{idq}(j)$ of the DP problem.

3.1 LM + FATT Algorithm

The Levenberg-Marquardt (LM) algorithm is a versatile optimization technique that minimizes least squares objective functions in nonlinear regression and curve-fitting problems by combining the advantages of the Gauss-Newton algorithm and the steepest descent approach as discussed in [14]. In this context, if $\overrightarrow{\mathrm{w}}$ represents a parameter vector of a model and $f(x_i, \overrightarrow{\mathrm{w}})$ denotes the loss function for the ith sample, the sum of squared errors, denoted as $J(\overrightarrow{\mathrm{w}})$, is given by the following expression:

$$J(\overrightarrow{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f(x_i, \overrightarrow{w}))^2 \tag{2}$$

where n is the number of data points, y_i is the observed value, $f(x_i, \theta)$ is the predicted value by the model, and θ represents the parameters being optimized. The LM algorithm adjusts the parameters using the following update equation:

$$\overrightarrow{w}_{k+1} = \overrightarrow{w}_k + (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e} \tag{3}$$

where $\overrightarrow{w}_k$ and $\overrightarrow{w}_{k+1}$ represent the parameter vectors at iteration k and $k+1$, respectively, $\mathbf{J}$ is the Jacobian matrix, $\mathbf{e}$ is the error vector, λ is the damping parameter, and $\mathbf{I}$ is the identity matrix. The Jacobian matrix $\mathbf{J}$ and the error vector $\mathbf{e}$ are defined as:

$$\mathbf{J} = \frac{\partial f(x_i, \theta)}{\partial \theta} \tag{4}$$

Hence, the change in weight of the LM for an RNN controller can be expressed as:

$$\Delta w = -\left[J_v(w)\left(J_v(w) + \mu I\right)\right] J_v(w) V \tag{5}$$

The integration of the Forward Accumulation Through Time (FATT) algorithm, as discussed in [15], significantly enhances the training process of the NN Controller by efficiently computing crucial elements such as the cost function ($C_1 = S(\overrightarrow{w})$), the Jacobian matrix ($J$), and the training pattern (P), which encapsulates the deviation between target trajectories (y) and the NN Controller's output ($f(\overrightarrow{w})$) across all time steps. Leveraging the Jacobian (J) and training pattern (P), we calculate $\triangle\overrightarrow{w}$ to gauge the cost of the updated estimate ($C_2 = S(\overrightarrow{w} + \triangle\overrightarrow{w})$). The comparison between these costs (C_1 and C_2) enables the adjustment of the damping factor λ according to the Levenberg-Marquardt (LM) algorithm, ensuring optimal training.

3.2 Adding Weight Dropout

The implementation of a strategic weight dropout method [7] is implemented in this research, which significantly boosts neural network efficiency and generalization. The weight dropout approach used in this research sets a chosen weight

to zero, making it inactive and making sure that it does not affect the neural network's output. This method is especially important since the Levenberg-Marquardt (LM) algorithm updates the weight vector iteratively by solving a system of linear equations. The algorithm closely monitors the lowered weight indices to preserve the integrity of the weight deactivation during the training cycles. This makes sure that once a weight is deactivated, it stays that way across all iterations, avoiding any unintentional changes that may cause it to become active again.

The least significant weights are those with the lowest absolute values, which are dropped first. Every non-diagonal element in the weight matrices is eligible to be dropped, but diagonal elements cannot be dropped because they have a big effect on the system's eigenvalues and, by extension, its stability and convergence. To ensure that the dropout process does not negatively impact the system's overall behavior, this distinction is essential to maintaining the structural and functional integrity of the network.

After identifying the weight, it is determined to be dropped. The algorithm updates the set of candidate weights and records the dropped weights. Through this dynamic modification, the architecture of the network can be adaptively refined during the training process, possibly leading to the discovery of new local minima that could provide better performance.

The training procedure involves applying the Forward Accumulation Through Time (FATT) and LM algorithms until a predefined condition is satisfied, like achieving the maximum lambda value (λ_{max}). The dropout algorithm steps in at this point and removes the smallest weight from the list of possible weights. To enable retraining with the modified weight configuration, the algorithm resets the lambda value (λ) to its initial value (λ_{start}) after removing a weight. Until the algorithm achieves the predetermined maximum number of dropped weights (20 weights in this case) or the maximum number of epochs, this cycle is repeated. The training algorithm converges until 18 weights are dropped and after that, the cost is increased.

The adoption of weight dropout has notably enhanced the parallel trajectory training process, making it more efficient by reducing computational complexity and facilitating faster convergence. This is particularly effective in environments utilizing LM and Forward Accumulation Through Time (FATT) algorithms, where the network iterates until reaching a predetermined threshold, continually optimizing by dropping insignificant weights. This method has not only increased the speed of the training process but also significantly improved the network's ability to generalize, which is crucial for RNN controllers deployed in diverse, real-world scenarios.

3.3 Parallel Algorithm

The purpose of a parallel algorithm is to use many processing units, such as Central Processing Unit (CPU) cores or Graphics Processing Units (GPUs), to

accomplish multiple computational tasks at once. This is mainly done through the division of jobs into smaller sub-tasks that may be performed concurrently across multiple processing units such as distributing computing nodes. This goes a long way to increase computing efficiency and decrease computing time. To achieve parallelization, all training trajectories are divided into N groups, each with a size from 1 to a number smaller than the total number of trajectories. The calculation of each group of trajectories is allocated to one worker or Central Processing Unit core. Each core independently computes the trajectory for a subset of training data and the results are aggregated to update the network parameters.

In accelerating Recurrent Neural Network (RNN) Controllers, parallelization of the Levenberg-Marquardt (LM) algorithm and Forward Accumulation Through Time (FATT) algorithm is crucial. LM algorithm parallelization involves distributing the computation of parameter updates across multiple processing units while ensuring consistency and convergence properties [1]. This is achieved by partitioning the training data and implementing synchronization mechanisms to coordinate parameter updates. Similarly, FATT algorithm parallelization entails distributing the computation of gradients through time across multiple processing units, with each unit computing gradient contributions for a subset of time steps or data samples independently [8]. Synchronization mechanisms are essential to ensure accurate gradient computation and prevent inconsistencies between parallel tasks.

It's crucial to remember that while parallel algorithms can result in shorter execution times, they often come with additional complexity related to synchronization and communication across parallel threads or processes. Furthermore, not all neural network calculations can be parallelized efficiently, and the amount of parallelism that can be achieved varies depending on the hardware infrastructure, data dependencies, and network architecture. Therefore, in order to optimize performance improvements while avoiding overhead, these aspects must be carefully considered while building efficient parallel algorithms for neural networks (Fig. 3).

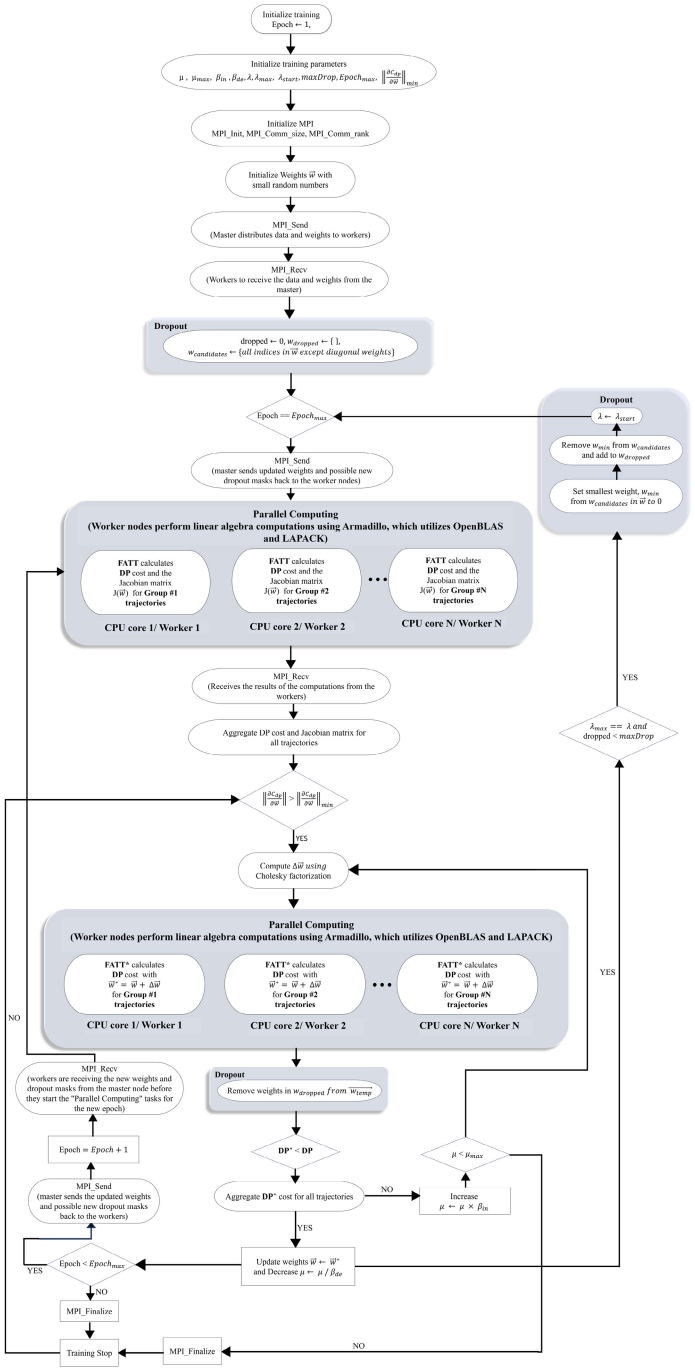


Fig. 3. The algorithm for training a parallel LM+FATT trajectory with weight dropout for an RNN controller. Maximum μ is denoted by μmax, while the decreasing and rising components are represented by βde and βin, respectively.

4 Performance and Training Results

4.1 Cloud Platform and $C++$ Implementation

In the deployment, the Amazon EC2 instance was specifically set up in the US East (N. Virginia) region. For the worker nodes, we opted for general-purpose instances of the m5.24×large type, each boasting a robust 48-core architecture, capable of accommodating up to 48 workers per machine. Conversely, the head node was provisioned with a Standard instance of the c5d.×large type, featuring 2 cores and 1×100NVMe storage capacity. Notably, the m5.24×large instances are powered by an Intel Xeon Platinum Processor, delivering clock speeds of up to 3.1 GHz. While Amazon does not disclose the specific CPU processor number, the performance is optimized for our computational needs.

The training program was developed using the C++17 standard and the g++ compiler. A key component of the program is the Armadillo C++ library, renowned for its swift execution of linear algebra computations. Armadillo relies on two fundamental packages: BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package). For our implementation, OpenBLAS, an open-source Fortran-based version of BLAS, was employed. The optimization of BLAS implementation is critical for enhancing performance across different architectures. LAPACK, also written in Fortran, offers a rich collection of complex linear algebra operations.

To distribute the workload across multiple worker nodes, we leveraged the OpenMPI implementation of the Message Passing Interface (MPI) standard [16]. The scalability of the Open MPI framework allows seamless scaling to larger and more potent clusters. The MPI program comprises both sequential and parallel sections, with the master process orchestrating the sequential tasks and dispatching messages to individual worker processes to handle parallel computations.

4.2 Speedup Performance

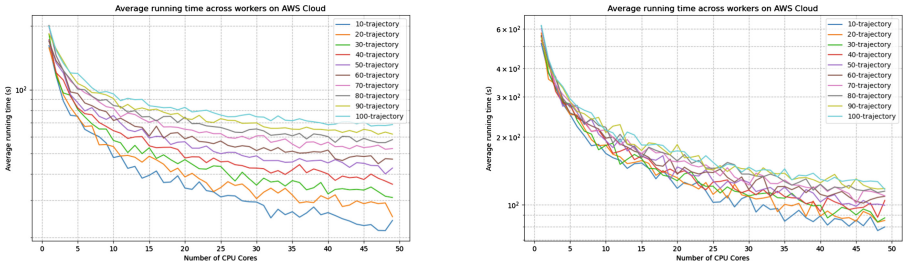
By carefully incorporating dropout into the neural network architecture, we were able to reduce the model's complexity and keep it from learning to recognize noise or outliers in the training set. This regularization accelerated the training process's convergence while also improving the network's capacity to generalize to previously unknown material.

Consequently, there was a significant reduction in the runtime per trajectory, which resulted in quicker training times and enhanced overall efficiency. The dropout strategy was essential in maximizing the effectiveness of our training program and ultimately resulting in better outcomes seen in the comparison study by reducing overfitting and speeding up convergence.

In the performance comparison across various worker configurations, as shown in Fig. 4(a) where dropout is implemented, the horizontal axis denotes the number of CPU cores or workers, while the vertical axis represents the average running time. Figure 4 (b) further elucidates the comparison without dropout, where the number of CPU cores/workers is depicted on the horizontal axis, and

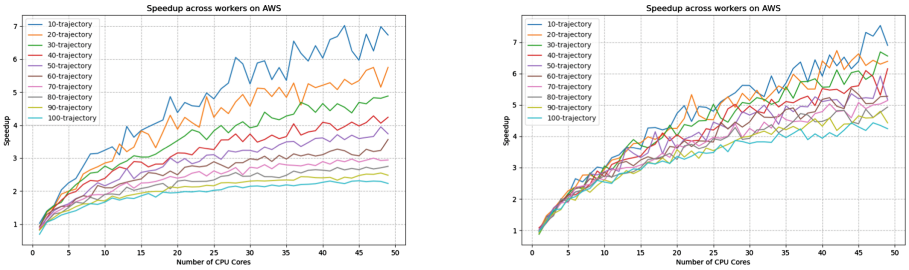
the average running time is also depicted on the vertical axis. It is evident the implementation of dropout technique reduces the time computational complexity which essentially demonstrates a speedup in performance as compared to without dropout. Figure 5 shows the speedup time comparison across different numbers of workers with and without the implementation of dropout. The horizontal axis represents the number of CPU cores/workers, while the vertical axis stands for the speedup time.

Notably, the integration of dropout into the parallelized C++ version significantly enhances its performance compared to the version without dropout integration. From Fig. 7, the results further indicate the efficiency and scalability of the proposed parallel mechanism with outstanding speed-up performance, which greatly decreases the training time for a large number of trajectories with high sampling frequency as compared to with and without the implementation of dropout (Fig. 6).



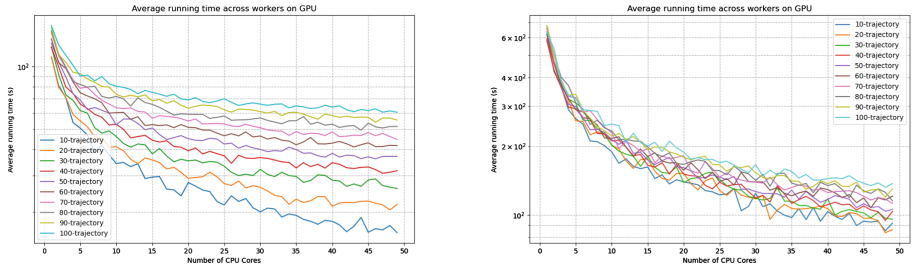
(a) Dropout (b) Without Dropout

Fig. 4. Average running time across workers with (a) and without (b) on AWS



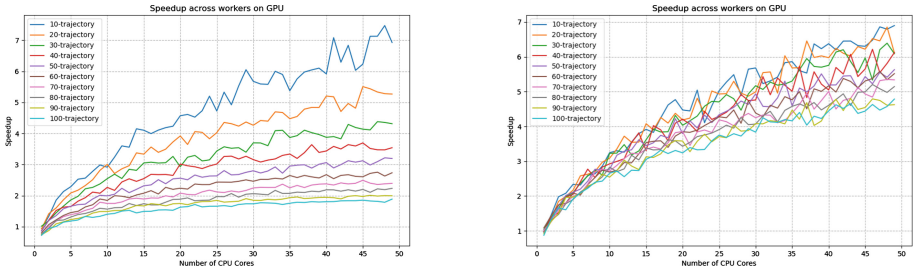
(a) Dropout (b) Without Dropout

Fig. 5. Speedup across workers with (a) and without (b) on AWS



(a) Dropout (b) Without Dropout

Fig. 6. Average running time across workers with (a) and without (b) on GPU



(a) Dropout (b) Without Dropout

Fig. 7. Speedup across workers with (a) and without (b) on GPU

5 Conclusion

This paper demonstrates how to speed up training an RNN controller in a closed-loop control system by combining weight dropout techniques with parallel trajectory training. The LM and FATT algorithms are integrated into the suggested parallel trajectory training technique. The C++ programming language was used to implement the training courses. Two computer platforms were used to evaluate the generated program: an Amazon EC2 instance and a GPU cluster. When the weight dropout technique is used, performance comparison findings demonstrate that the parallel training algorithm can significantly outperform its non-parallelized equivalents in terms of speed. The proposed training mechanism is appropriate for training RNN controllers with a large number of trajectories and long-duration trajectories due to the notable speedup performance.

Acknowledgements. This work is supported by the National Science Foundation of the United States under Award 2131175 and 2131214. We thank anonymous reviewers for their insightful comments and inputs.

References

1. Suri, N.N., Deodhare, D., Nagabhushan, P.N.: Parallel levenberg-marquardt-based neural network training on linux clusters - a case study. In: Indian Conference on Computer Vision, Graphics & Image Processing (2002)
2. Cao, J., Novstrup, K.A., Goyal, A., Midkiff, S.P., Caruthers, J.M.: A parallel levenberg-marquardt algorithm. In: ICS 2009: Proceeding of the 23rd International Conference on Supercomputing, pp. 450–459 (2009). https://doi.org/10.1145/1542275.1542338
3. Przybylski, A., Thiel, B., Keller-Findeisen, J., Stock, B., Bates, M.: 'Gpufit: an open-source toolkit for GPU-accelerated curve fitting. Sci. Rep. **7**, 15722 (2017). https://doi.org/10.1038/s41598-017-15313-9
4. Wang, J., Yang, B., Chen, Z.: Solar inverters: technologies and applications in photovoltaic power systems. Renew. Sustain. Energy Rev. **71**, 789–802 (2017)
5. Texas instruments, digitally controlled solar micro inverter design using C2000 piccolo microcontroller user's guide. https://www.ti.com/lit/ug/tidu405b/tidu405b.pdf
6. Waithaka, W., Fu, X., Hadi, A., Challoo, R., Li, S.: DSP implementation of a novel recurrent neural network controller into a TI solar microinverter. In: Proceeding of 2021 IEEE PES General Meeting, July 26–July 29, 2021
7. Sturtz, J., Fu, X., Hingu, C.D., Qingge, L.: A novel weight dropout approach to accelerate the neural network controller embedded Im- plementation on FPGA for a solar inverter. In: 2023 IEEE International Conference on Smart Computing (SMARTCOMP), Nashville, TN, USA, pp. 157–163 (2023)
8. Fu, X., Sturtz, J., Alonso, E., Challoo, R., Qingge, L.: Parallel trajectory training of recurrent neural network controllers with levenberg-marquardt and forward accumulation through time in closed-loop control systems. IEEE Trans. Sustain. Comput. **09**, 222–229 (2023)
9. Fu, X., Li., D. Wunsch, C., Alonso, E.: Local stability and convergence analysis of neural network controllers with error integral inputs. IEEE Trans. Neural Netw. Learn. Syst. **34**(7), 3751–3763 (2021)
10. Wang, F., Zhang, H., Liu, D.: Adaptive dynamic programming: an introduction. IEEE Comput. Intell. Mag. **4**(2), 39–47 (2009)
11. Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton, New Jersey, USA (1957)
12. Prokhorov, D.V., Wunsch, D.C.: Adaptive critic designs. IEEE Trans. Neural Netw. **8**(5), 997–1007 (1997)
13. Zhu, X., Zhang, D.: Efficient parallel levenberg-marquardt model fitting towards real-time automated parametric imaging microscopy. PLOS ONE **8**(10), e76665 (2013). https://doi.org/10.1371/journal.pone.0076665
14. Pujol, J.: The solution of nonlinear inverse problems and the Levenberg-Marquardt method. Geophysics **72**(4), W1–W16 (2007)
15. Werbos, P.J.: Backpropagation through time: what it does and how to do it. Proc. IEEE **78**(10), 1550–1560 (1990)
16. Sanderson, C., Curtin, R.: A user-friendly hybrid sparse matrix class in C++. Lecture Notes Comput. Sci. (LNCS) **10931**, 422–430 (2018)
17. Sanderson, C., Curtin, R.: A user-friendly hybrid sparse matrix class in C++. In: Davenport, J.H., Kauers, M., Labahn, G., Urban, J. (eds.) ICMS 2018. LNCS, vol. 10931, pp. 422–430. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96418-8_50