Short Papers

Parallel Trajectory Training of Recurrent Neural Network Controllers With Levenberg–Marquardt and Forward Accumulation Through Time in Closed-Loop Control Systems

Xingang Fu, *Senior Member, IEEE*, Jordan Sturtz, Eduardo Alonso, Rajab Challoo, and Letu Qingge

***Abstract*—This paper introduces a novel parallel trajectory mechanism that combines Levenberg-Marquardt and Forward Accumulation Through Time algorithms to train a recurrent neural network controller in a closed-loop control system by distributing the calculation of trajectories across Central Processing Unit (CPU) cores/workers depending on the computing platforms, computing program languages, and software packages available. Without loss of generality, the recurrent neural network controller of a grid-connected converter for solar integration to a power system was selected as the benchmark test closed-loop control system. Two software packages were developed in Matlab and C++ to verify and demonstrate the efficiency of the proposed parallel training method. The training of the deep neural network controller was migrated from a single workstation to both cloud computing platforms and High-Performance Computing clusters. The training results show excellent speed-up performance, which significantly reduces the training time for a large number of trajectories with high sampling frequency, and further demonstrates the effectiveness and scalability of the proposed parallel mechanism.**

***Index Terms*—Cloud computing, forward accumulation through time, high - performance computing (HPC) cluster, Levenberg–Marquardt, parallel trajectory training, recurrent neural network controller.**

I. Introduction

The Levenberg-Marquardt (LM) algorithm provides a nice compromise between the speed of the second-order Newton's method and the guaranteed convergence of first-order steepest descent method to solve nonlinear least squares problems [1], [2]. Thus it is particularly suitable for training small and medium-sized feed-forward Neural Networks (NNs) [3].

However, the computation loads of the LM algorithm are expensive due to the calculation needs of the Jacobian matrix, and researchers have explored ways to speed it up. For instance, the block-diagonal matrix has been proposed to approximate the Hessian matrix [4], [5], and, in [6], the forward difference method was used to approximate the Jacobian matrix by perturbing one parameter to produce a column, instead of calculating the Jacobian matrix directly.

In addition, several mechanisms have been proposed to parallelize the LM algorithm for NN training, appropriately distributing computational and space requirements. For example, the Single Program and Multiple Data (SPMD) strategy divides training data into groups and each group is distributed on one node in a cluster [7]. [8] also utilized the parallelization of data sets by calculating the objective functions simultaneously. Relatedly, [9] and [10] distributed the computing tasks/data points across parallel GPU multiprocessors to train the LM algorithm in parallel. In [11], the parallel selection of the damping parameter and multicore versions of the Basic-Linear-Algebra-Subprograms (BLAS) were used in the LM algorithm to increase computational efficiency. In any case, with or without parallelization, the application of feed-forward NNs is inherently limited due to their inability to identify and process sequential partners in large data sets.

Recurrent Neural Networks (RNNs) are potentially more powerful than feed-forward NNs thanks to their feedback connections and memory gates [3]. Many algorithms have been used for training RNNs such as Backpropagation Through Time (BPTT) [12], Real-Time Recurrent Learning (RTRL) [13], Extended Kalman Filters (EKF) [14], genetic algorithms [15], and Expectation Maximization (EM) [16], [17]. Notwithstanding their merits, they all suffer from serious drawbacks: the BPTT algorithm may cause gradient exploding and vanishing problems. The high computational cost of RTRL makes it only appropriate for online training of small RNNs. EKF is also computationally expensive since it requires many matrix calculations at each estimation. Evolutionary methods such as genetic algorithms have proved to be successful in training RNNs by formulating the RNN cost function as a nonlinear global optimization problem. However, they may get stuck in local minima and show a low speed of convergence. Finally, the application of EM to training neural networks is limited by the complicated calculations in the expectation step when the number of hidden neurons is large. Such deficiencies have been an impediment to the application of RNNs to real-life problems such as closed-loop control systems, that we take as our benchmark.

Although some research has shown the potential of training RNNs using LM [18], [19], LM has not been used broadly for this purpose. The Forward Accumulation Through Time (FATT) algorithm was proposed to calculate the Jacobian matrix efficiently and combined with the LM algorithm to train an RNN controller applied to a power converter control system, which produced excellent performance [20]. However, training was based on a rather small number of trajectories (e.g., 10 trajectories) and a relatively low sampling frequency (e.g., 1000 Hz) due to constraints in the computational power and the memory size of the single workstation used in the experiments.

Manuscript received 7 May 2023; revised 24 September 2023; accepted 31 October 2023. Date of publication 6 November 2023; date of current version 3 April 2024. This work was supported by the National Science Foundation of the United States under Grants 213214 and 2131175. Recommended for acceptance by P. D. Yoo. *(Corresponding author: Letu Qingge.)*

Xingang Fu is with the Department of Electrical and Biomedical Engineering, The University of Nevada, Reno, NV 89557 USA (e-mail: xfu@unr.edu).

Jordan Sturtz and Letu Qingge are with the Department of Computer Science, North Carolina A&T State University, Greensboro, NC 27411 USA (e-mail: jasturtz@aggies.ncat.edu; lqingge@ncat.edu).

Eduardo Alonso is with the Artificial Intelligence Research Centre (CitAI), University of London, EC1V 0HB London, U.K. (e-mail: e.alonso@city.ac.uk).

Rajab Challoo is with the Department of Electrical Engineering and Computer Science, Texas A&M Kingsville, Kingsville, TX 78363 USA (e-mail: rajab.challoo@tamuk.edu).

Digital Object Identifier 10.1109/TSUSC.2023.3330573

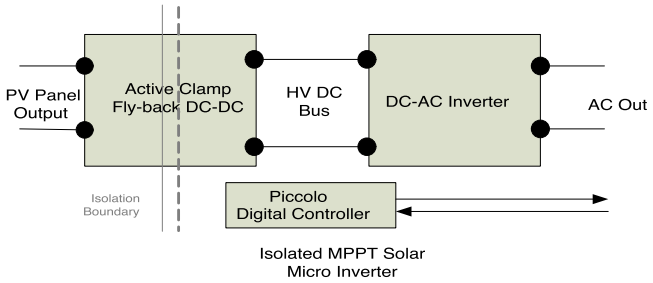

Fig. 1. TI microinverter block diagram [21].

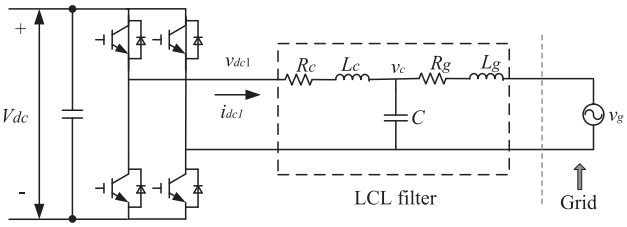

Fig. 2. Schematic of a single-phase DC-AC inverter block with LCL filters.

To extend the LM training of RNNs to a large number of trajectories with high sampling frequency and accelerate the training process, this paper proposes a novel parallel trajectory training mechanism. The key contributions include 1) the introduction of a training mechanism tailored for RNNs with error integral terms in closed-loop control systems, a solar microinverter system in particular; 2) the development of a parallel LM and FATT algorithm designed for trajectory training of RNNs; 3) implementation of a parallel approach that efficiently distributes the FATT-based calculation of training trajectories across Central Processing Unit (CPU) cores/workers, 4) validation conducted through implementations in two programming languages: Matlab and C++; 5) comprehensive validation of the implementations and performance comparison on both cloud computing platforms and High-Performance Computing (HPC) clusters.

The rest of the paper is organized as follows. Section II introduces the RNN controllers in the benchmark test closed-loop control system. The parallel trajectory training algorithm of the RNN controllers is detailed in Section III. Training results using cloud platforms are presented in Section IV. Section V provides detailed implementation and training results on HPC clusters. Finally, the paper concludes with a summary of the main points in Section VI.

II. RNN Controllers in a Closed-Loop Control System for a Solar Inverter

A. A Closed-Loop Control System: A Solar Microinverter

Typically, solar inverters consist of two components: the DC-DC converter and the DC-AC inverter, as illustrated in the case of the Texas Instruments (TI) Microinverter in Fig. 1 [21]. The PhotoVoltaic (PV) solar panels attach to the DC-DC converter, while the DC-AC inverter maintains the voltage of the DC Bus at its rated value while feeding controlled AC current to the main power grid.

Fig. 2 further shows the schematic of a single-phase DC-AC inverter block with the LCL filters, in which a DC-link capacitor/DC bus is on the left, an LCL filter is placed in the middle, and a single-phase voltage source, representing the voltage at the Point of Common Coupling (PCC) of the AC power grid system is on the right [22].

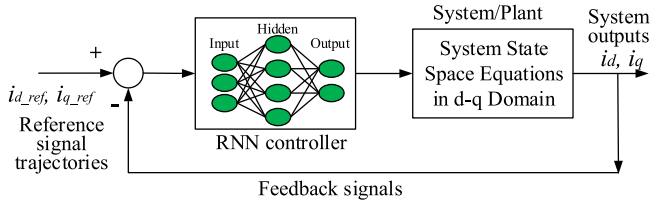

Fig. 3. NN controller in a closed-loop control system. The system equations serve as the feedback connections for the NN controller.

When using the d-q frame, the system state-space equation of the DC-AC inverter block can be described by (1) [23], which will be used for RNN training.

$$\underbrace{\frac{d}{dt}\begin{bmatrix} i_d \\ i_q \\ i_{dc\_d1} \\ i_{dc\_q1} \\ v_{cd} \\ v_{cq} \end{bmatrix}}_{\overrightarrow{i_{dqs}}} = \underbrace{\begin{bmatrix} -\frac{R_g}{L_g} & \omega_s & 0 & 0 & -\frac{1}{L_g} & 0 \\ -\omega_s & -\frac{R_g}{L_g} & 0 & 0 & 0 & -\frac{1}{L_g} \\ 0 & 0 & -\frac{R_c}{L_c} & \omega_s & \frac{1}{L_c} & 0 \\ 0 & 0 & -\omega_s & -\frac{R_c}{L_c} & 0 & \frac{1}{L_c} \\ \frac{1}{C} & 0 & -\frac{1}{C} & 0 & 0 & \omega_s \\ 0 & \frac{1}{C} & 0 & -\frac{1}{C} & -\omega_s & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} i_d \\ i_q \\ i_{dc\_d1} \\ i_{dc\_q1} \\ v_{cd} \\ v_{cq} \end{bmatrix}}_{\overrightarrow{i_{dqs}}} + \underbrace{\begin{bmatrix} \frac{1}{L_g} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{L_g} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{L_c} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{L_c} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} v_d \\ v_q \\ v_{d1} \\ v_{q1} \\ 0 \\ 0 \end{bmatrix}}_{\overrightarrow{u_{dqs}}} \tag{1}$$

where ω_s is the angular frequency of the grid voltage, and all other symbols are consistent with those shown in Fig. 2. i_d and i_q represent system states that need to be controlled. The controller outputs will be v_{d1} and v_{q1}. System parameters in (1) can be obtained from the user guide or datasheet of TI microinverter [21].

To train the RNN digital controller, the continuous state space model in (1) must first be converted into an equivalent discrete model using (2), either through a zero-order or a first-order hold discrete equivalent mechanism with a sampling time of T_s. For example, if the sampling frequency equals 10000 Hz, then $T_s = 1/10000 = 0.1$ ms.

$$\overrightarrow{i_{dqs}}(k+1) = A\overrightarrow{i_{dqs}}(k) + B\overrightarrow{u_{dqs}}(k) \tag{2}$$

in which, A stands for system matrix and B is the input matrix.

B. The NN Controller in a Closed-Loop Control System

A NN will be implemented in the Piccolo real-time digital controller in Fig. 1 to regulate the currents (i_d and i_q) to follow the reference trajectories (i_{d_ref} and i_{q_ref}) in a closed-loop control system, instead of conventional Proportional–Integral (PI) controllers, as shown in Fig. 3.

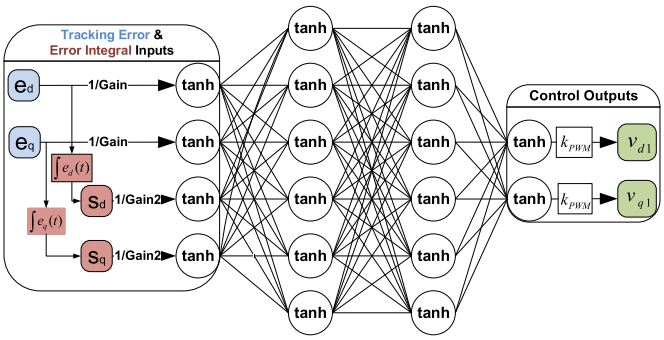

Fig. 4. NN controller with special tracking error integrals [25].

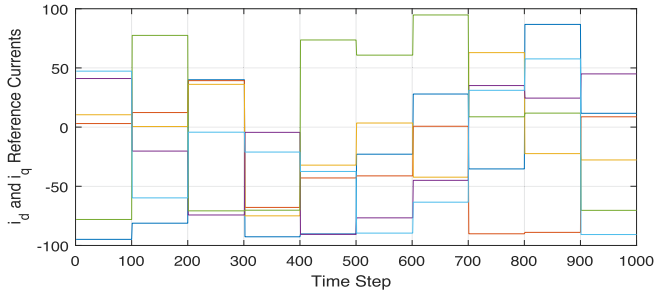

Fig. 5. Reference trajectories for RNN training.

C. The RNN With Error Integral Inputs

The structure of the proposed NN controller is shown in Fig. 4. The NN has two hidden layers, each with six neurons, and one two-neuron layer that controls the outputs. The selection of the number of neurons in each hidden layer was conducted through trial and error tests. After implementing many trial and error tests, 6 nodes in each hidden layer were found to be able to generate good enough results in real-time control. Further, the number of weights or neurons can be further reduced through the dropout approach to better fit the embedded real-time computing [24].

The input block of the NN takes the tracking error input signals $\overrightarrow{e_{dq}}$ and their special error integral values $\overrightarrow{s_{dq}}$. To avoid input saturation, $\overrightarrow{e_{dq}}$ and $\overrightarrow{s_{dq}}$ are divided by constant gain values, $Gain$ and $Gain2$, respectively, and normalized by the hyperbolic tangent function, whose values are limited in the range [-1, 1]. Specifically, $\overrightarrow{e_{dq}}$ is defined as $\overrightarrow{e_{dq}}(k) = \overrightarrow{i_{dq}}(k)$ - $\overrightarrow{i_{dq\_ref}}(k)$ and $\overrightarrow{s_{dq}}(k)$ is calculated by

$$\overrightarrow{s_{dq}}(k) = \int_0^{kT_s} \overrightarrow{e_{dq}}(t)dt \approx T_s \sum_{j=1}^{k} \frac{\overrightarrow{e_{dq}}(j-1) + \overrightarrow{e_{dq}}(j)}{2} \tag{3}$$

in which the trapezoid formula was used to compute the integral term $\overrightarrow{s_{dq}}(k)$ and $\overrightarrow{e_{dq}}(0) \equiv 0$. The special error integral terms $\overrightarrow{s_{dq}}$ will guarantee that there is no steady-state error for step references [26].

The system equations (1) and (2) serve as the feedback connections for the NN controller as seen in Fig. 3. Moreover, the calculation of the error integration terms $\overrightarrow{s_{dq}}(k)$ (3) has to accumulate all past error terms $\overrightarrow{e_{dq}}(j)$ from $j = 0$ to $j = k$ and each past error term $\overrightarrow{e_{dq}}(j)$ computation will involve the outputs of the NN controller in the corresponding past step j. Thus, the proposed NN is a recurrent NN and will be denoted as RNN thereafter.

Further, the RNN controller can be represented explicitly by equation (4), where $W1$, $W2$, and $W3$ stand for the weights of the input layer to the first hidden layer, second hidden layer, and output layer, respectively. The bias for each layer is incorporated into weights $W1$, $W2$, and $W3$.

$$R(\overrightarrow{e_{dq}}, \overrightarrow{s_{dq}}, W_1, W_2, W_3) =$$
$$\tanh\left\{W_3\left[\tanh\left\{W_2\left[\tanh\left\{W_1\left[\tanh\left[\begin{matrix}\frac{e_d}{Gain}\\ \frac{e_q}{Gain}\\ \frac{s_d}{Gain2}\\ \frac{s_q}{Gain2}\end{matrix}\right]\right]\right\}\right]\right\}\right]\right\} \tag{4}$$
$$-1 \quad -1 \quad -1$$

The outputs from the RNN multiplied by the gain value of the Pulse-Width-Modulation (PWM) (k_{PWM}) will constitute the control action $\overrightarrow{v_{dq1}}$, which is expressed by

$$\overrightarrow{v_{dq1}} = k_{PWM} R(\overrightarrow{e_{dq}}, \overrightarrow{s_{dq}}, W_1, W_2, W_3) \tag{5}$$

III. Parallel Trajectory Training of RNN Controllers

A. Training Objective: Approximate Optimal Control

Adaptive Dynamic Programming (ADP) [27] methods that combine incremental optimization techniques with parametric structures that approximate optimal cost are typically used to control a system. Specifically, a discrete-time ADP approach based on the principle of Bellman's optimality [28] uses a discrete-time system model along with a performance index or cost [29].

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

$$C_{dp} = \sum_{k=j}^{\infty} \gamma^{k-j} U(\overrightarrow{e_{dq}}(k))$$
$$= \sum_{k=j}^{\infty} \gamma^{k-j} \sqrt{[i_d(k) - i_{d\_ref}(k)]^2 + [i_q(k) - i_{q\_ref}(k)]^2} \tag{6}$$

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 $\overrightarrow{i_{dq}}(j)$, the function C_{dp} is referred to as the cost-to-go of state $\overrightarrow{i_{dq}}(j)$ of the DP problem. The training objective is to find an optimal RNN controller that minimizes the DP cost C_{dp} by regulating $\overrightarrow{i_{dq}}$.

B. Trajectory Tracking

Fig. 5 demonstrates the reference trajectories for RNN training, which contains 6 trajectories: 3 for i_d and 3 for i_q. The reference trajectories were generated randomly within the system's controllable range. This range can be determined by the physical current/voltage ratings of solar inverters. For demonstration purposes, the range was set as [-100, 100] in Fig. 5. The reference i_d and i_q values were set to change after certain time steps, e.g. 100, which is a tunable parameter. Normally, the control system needs time to reach its steady state, the 100 time steps turned out to be a well-balanced number in training the RNN controller for a solar inverter. The total time steps/trajectory length was set to a certain number, e.g. 1000 in Fig. 5, which is determined by training duration and sampling time. For example, if the sampling time $T_s = 1$ ms and the training duration 1 s are used, the trajectory length will be $1\ s/T_s = 1000$. The total number of training trajectories can vary from 10 to several hundred. Utilizing a large number of trajectories

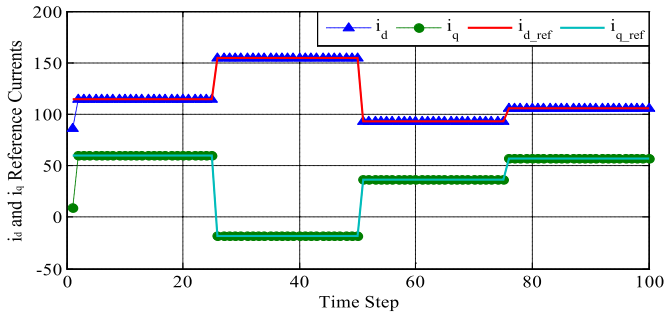

Fig. 6. Idea tracking performance for RNN training.

will significantly slow down training, which is the reason for proposing the parallel trajectory training algorithm for overcoming this challenge.

Fig. 6 shows the ideal tracking performance after the RNN was well-trained. After training, an RNN controller will able to regulate the system states i_d and i_q to follow the reference currents i_{d_ref} and i_{q_ref}.

C. LM Algorithm

If the performance error function is not a sum of squares, then the LM weight update equation is not directly applicable. To implement LM training, the cost function defined in (6) needs to be rewritten in a Sum-Of-Squares form. Consider the cost function C_{dp} with $\gamma = 1$, $j = 1$ and $k = 1, \ldots, N$, then it can be written in the form

$$C_{dp}=\sum_{k=1}^{N} U(\overrightarrow{e_{dq}}(k)) \xleftrightarrow{\text{def } V(k)=\sqrt{U(\overrightarrow{e_{dq}}(k))}} C_{dp}=\sum_{k=1}^{N} V^2(k) \tag{7}$$

and the gradient $\frac{\partial C_{dp}}{\partial w}$ can be written in a matrix product form

$$\frac{\partial C_{dp}}{\partial \overrightarrow{w}} = \sum_{k=1}^{N} V(k)\frac{\partial V(k)}{\partial \overrightarrow{w}} = 2J_v(\overrightarrow{w})^T V \tag{8}$$

where the Jacobian matrix $J_v(\overrightarrow{w})$ is

$$J_v(w) = \begin{bmatrix} \frac{\partial V(1)}{\partial w_1} & \cdots & \frac{\partial V(1)}{\partial w_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial V(N)}{\partial w_1} & \cdots & \frac{\partial V(N)}{\partial w_M} \end{bmatrix}, V = \begin{bmatrix} V(1) \\ \vdots \\ V(N) \end{bmatrix} \tag{9}$$

Therefore, the weight update using LM for an RNN controller can be expressed as

$$\triangle \overrightarrow{w} = -[J_v(\overrightarrow{w})^T J_v(\overrightarrow{w}) + \mu I]^{-1} J_v(\overrightarrow{w})^T V \tag{10}$$

D. FATT Algorithm

In order to calculate the Jacobian matrix $J_v(\overrightarrow{w})$ efficiently, FATT was used, which incorporates the procedures of unrolling the system, calculating the derivatives of the Jacobian matrix, and calculating the DP cost into one single process for each training epoch [20]. Fig. 7 illustrates the process of unrolling the trajectory in the forward path, and Algorithm 1 specifies FATT [18], where $\overrightarrow{\phi}(k)=\sum_{j=1}^{k} \overrightarrow{i_{dqs}}(j)$ and $\frac{\partial \overrightarrow{\phi_{dq}}(k)}{\partial \overrightarrow{w}}=\sum_{j=1}^{k} \frac{\partial \overrightarrow{i_{dqs}}(j)}{\partial \overrightarrow{w}}$.

E. Parallel Training Combination of LM and FATT Algorithms

Fig. 8 presents the proposed parallel training combination of LM and FATT algorithms for training an RNN controller. FATT* in Fig. 8 refers

Algorithm 1: FATT Algorithm to Calculate the Jacobian Matrix and to Accumulate DP Cost for One Trajectory.

1: $C \leftarrow 0, \overrightarrow{e_{dq}}(0) \leftarrow 0, \overrightarrow{s_{dq}}(0) \leftarrow 0, \frac{\partial \overrightarrow{i_{dqs}}(0)}{\partial \overrightarrow{w}} \leftarrow 0, \frac{\partial \overrightarrow{\phi_{dq}}(0)}{\partial \overrightarrow{w}} \leftarrow 0$
2: {Calculate the Jacobian matrix $J_v(\overrightarrow{w})$}
3: **for** $k = 0$ to $N - 1$ **do**
4: $\overrightarrow{v_{dq1}}(k) \leftarrow k_{PWM} R(\overrightarrow{e_{dq}}(k), \overrightarrow{s_{dq}}(k), \overrightarrow{w})$
5: $\frac{\partial \overrightarrow{s_{dq}}(k)}{\partial \overrightarrow{w}} \leftarrow T_s \left[\frac{\partial \overrightarrow{\phi_{dq}}(k)}{\partial \overrightarrow{w}} - \frac{1}{2}\frac{\partial \overrightarrow{i_{dq}}(k)}{\partial \overrightarrow{w}}\right]$
6: $\frac{\partial \overrightarrow{v_{dq1}}(k)}{\partial \overrightarrow{w}} \leftarrow k_{PWM}$ $\left[\frac{\partial R(k)}{\partial \overrightarrow{w}} + \frac{\partial R(k)}{\partial \overrightarrow{e_{dq}}(k)}\frac{\partial \overrightarrow{i_{dq}}(k)}{\partial \overrightarrow{w}} + \frac{\partial R(k)}{\partial \overrightarrow{s_{dq}}(k)}\frac{\partial \overrightarrow{s_{dq}}(k)}{\partial \overrightarrow{w}}\right]$
7: $\frac{\partial \overrightarrow{i_{dqs}}(k+1)}{\partial \overrightarrow{w}} \leftarrow A\frac{\partial \overrightarrow{i_{dqs}}(k)}{\partial \overrightarrow{w}} + B\frac{\partial \overrightarrow{u_{dq}}(k+1)}{\partial \overrightarrow{w}}$
8: $\frac{\partial \overrightarrow{i_{dq}}(k+1)}{\partial \overrightarrow{w}} \leftarrow$ the first two terms of $\frac{\partial \overrightarrow{i_{dqs}}(k+1)}{\partial \overrightarrow{w}}$
9: $\frac{\partial \overrightarrow{\phi_{dq}}(k+1)}{\partial \overrightarrow{w}} \leftarrow \frac{\partial \overrightarrow{\phi_{dq}}(k)}{\partial \overrightarrow{w}} + \frac{\partial \overrightarrow{i_{dq}}(k+1)}{\partial \overrightarrow{w}}$
10: $\overrightarrow{i_{dqs}}(k+1) \leftarrow A\overrightarrow{i_{dqs}}(k) + B\overrightarrow{u_{dqs}}(k)$
11: $\overrightarrow{e_{dq}}(k+1) \leftarrow \overrightarrow{i_{dq}}(k+1) - \overrightarrow{i_{dq\_ref}}(k+1)$
12: $\overrightarrow{s_{dq}}(k+1) \leftarrow \overrightarrow{s_{dq}}(k) + \frac{T_s}{2}[\overrightarrow{e_{dq}}(k) + \overrightarrow{e_{dq}}(k+1)]$
13: $C \leftarrow C + U(\overrightarrow{e_{dq}}(k+1))$ accumulate DP cost
14: $\frac{\partial \overrightarrow{V}(k+1)}{\partial \overrightarrow{w}} \leftarrow \frac{\partial \overrightarrow{V}(k+1)}{\partial \overrightarrow{e_{dq}}(k+1)}\frac{\partial \overrightarrow{i_{dq}}(k+1)}{\partial \overrightarrow{w}}$
15: the $(k+1)th$ row of $J(\overrightarrow{w}) \leftarrow \frac{\partial \overrightarrow{V}(k+1)}{\partial \overrightarrow{w}}$
16: **end for**
17: {On exit, the Jacobian matrix $J_v(\overrightarrow{w})$ is finished for one trajectory.}

to a modified version of Algorithm 1 that only calculates the DP cost by eliminating lines 5–9 and 14–15 to save computation time. The most time-consuming parts include the DP cost calculation and the Jacobian matrix for each trajectory, which is conducted by the FATT Algorithms. To solve this challenge, the basic idea is to parallelize them as follows: First, all training trajectories are divided into N groups, each with a size from 1 to a number smaller than the number of total trajectories. Then, the calculation of each group of trajectories is allocated to one Worker or Central Processing Unit (CPU) core. The detailed implementation will depend on the specific programming language, platforms, etc. For example, for the MATLAB implementation, the computing unit will be one MATLAB worker, which corresponds to one CPU core. For the C++ implementation, this single worker could correspond to one CPU core or thread. For a fair comparison, one single CPU core was also used in the C++ implementation. The implementation and comparison of both cases will be detailed in Sections IV and V.

Fig. 8 also illustrates how the algorithm dynamically adjusts μ. When μ increases, training is closer to a gradient descent algorithm with a small learning rate, whereas when μ decreases, training approaches the Gauss-Newton method, which provides faster convergence than gradient descent. There are three stopping conditions used for training: 1) when the training epoch reaches a maximum acceptable value $\text{Epoch}_{\max}$; 2) when μ is larger than $\mu_{\max}$; and 3) when the gradient is smaller than the predefined minimum acceptable value $\|\partial C_{dp}/\partial \overrightarrow{w}\|_{min}$.

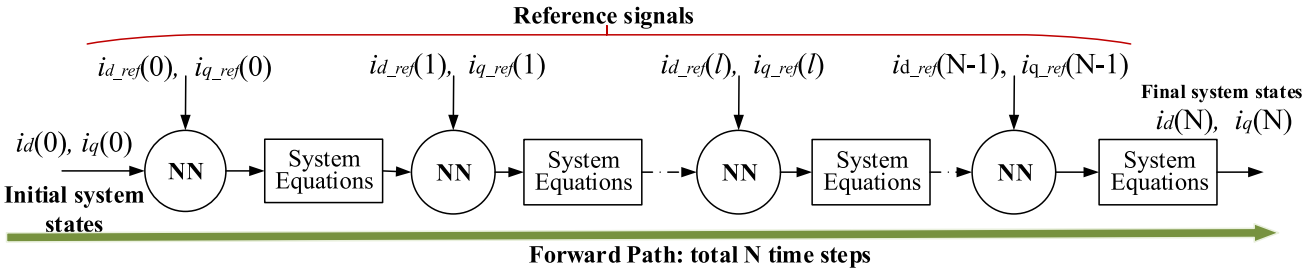

Fig. 7. Unrolling the forward path in the FATT algorithm for training an NN controller in a closed-control loop system.

IV. Amazon EC2 Cloud Training Performance

A. Amazon EC2 Cloud

Amazon EC2 cloud was utilized as the test cloud platform. The cloud cluster was configured and connected to Amazon EC2. For the Amazon EC2 cluster, the region is selected as US East (N. Virginia). The worker machine type is General Purpose (m5.24 × large, 48 core), which supports 48 workers per machine. The headnode machine type is Standard (c5d. × large, 2core,1 × 100NVMe). It is noted that the m5.24 × large type instance uses up to 3.1 GHz Intel Xeon Platinum Processor, however, the specific CPU processor number is not provided by Amazon EC2 [30].

B. MATLAB Implementation and Speedup Performance

Matlab [31] was first used to develop the training program and validate the proposed parallel training algorithm. Matlab Online running version R2022b was used. For the two parallel computing parts described in Fig. 8, two parfor-loop structures were used to execute for-loop iterations in parallel on workers in a parallel pool. To execute the parfor-loop, Matlab starts a parallel pool with one worker per physical CPU core, not CPU thread. A parallel pool with 48 workers was used.

To test whether training is successful or not, the current tracking performance in the closed-loop control was performed. Fig. 9 shows the current tracking performance after the RNN controller was well-trained. The current i_d and i_q can track their reference currents i_{d_ref} and i_{q_ref} very well only with some slight oscillations at the beginning of the reference change.

Fig. 10 shows the running time comparison on different numbers of Matlab workers and trajectories. The horizontal axis represents the number of CPU cores/Matlab workers, while the vertical axis stands for the average running time. To test the performance of the proposed approach, all inputs across all runs and any randomness inherent in the code were fixed. The test program was run with trajectories in $\{10, 20, 30, \ldots 100\}$ and with Matlab workers from $\{1, 2, 3, \ldots 48\}$. When the number of Matlab workers equals 1, no parallel computing was used. For each running case, the program was repeated 10 times, and the average running time was calculated to remove possible variations of the running time in each run for a fair comparison. Regardless of the number of trajectories used in the experiment, a consistent trend emerges: as the number of CPU cores/Matlab workers increases to a certain threshold, the average running time stabilizes and cannot be further reduced.

The speedup is defined as the ratio of serial execution time over parallel execution time, which is $speedup = Serial\ Execution\ Time/Parallel\ Execution\ Time$. Fig. 11 shows the speedup comparison across different numbers of Matlab workers. The horizontal axis represents the number of CPU cores/Matlab workers and the vertical axis stands for the speedup. Figs. 10 and 11 clearly show an excellent, although nonlinear speedup performance. When the number of Matlab workers surpasses the number of trajectories, no speedup benefits can be further achieved and the running time is slightly longer due to communication loads between Matlab workers. The trend was observed in all number of trajectories. Compared to nonparallel training, the parallel approach achieves a least 4 times speedup. When the number of trajectories increases, the speedup becomes even more significant.

V. HPC Cluster Training Performance

A. HPC Cluster Platform

The selected HPC platform is a cluster consisting of four Non-Uniform Memory Access (NUMA) system [32] compute nodes running the AlmaLinux distribution of the Linux operating system [33]. The CPU of each compute node is Intel(R) Xeon(R) Platinum 8180 M CPU. Each compute node contains 2 sockets, one 28-core CPU per socket, and 2 threads per CPU core, providing a total of $\mathbf{2 * 28 = 56}$ CPU cores and $\mathbf{56 * 2 = 112}$ threads. See Fig. 12 for an illustration of the HPC architecture.

B. MATLAB Implementation and Speedup Performance

The HPC is configured to use Slurm to manage job scheduling [34]. The Matlab cluster configuration supports up to 28*2 = 56 computational CPU cores. For a fair comparison with results running on Amazon EC2 Cloud, the maximum number of Matlab workers was also set to 48.

Fig. 13 shows the running time comparison across different numbers of Matlab workers and trajectories. Fig. 14 shows the speedup comparison across different numbers of Matlab workers. Figs. 13 and 14 show slightly better speedup performance running on the HPC cluster than Figs. 10 and 11 running on Amazon EC2 Cloud. For example, the maximum speedup running on the HPC cluster is 6 times compared to only 5 times speedup on the Amazon EC2 Cloud for 10-trajectory training.

C. C++ Implementation and Speedup Performance

The c++17 standard [35] and the compiler g++ [36] were used to develop the training program. An important third-party package used in the training program is the C++ package Armadillo, which is a fast linear algebra library to perform linear algebra computations [37], [38]. Armadillo relies on BLAS [39] and Linear Algebra PACKage (LAPACK) [40]. BLAS is a specification of low-level routines for performing basic linear algebra operations. The training program used OpenBLAS, which is an open-source implementation in Fortran of this software specification [41]. The configuration of an implementation of BLAS is paramount since it can be tuned to particular architectures to

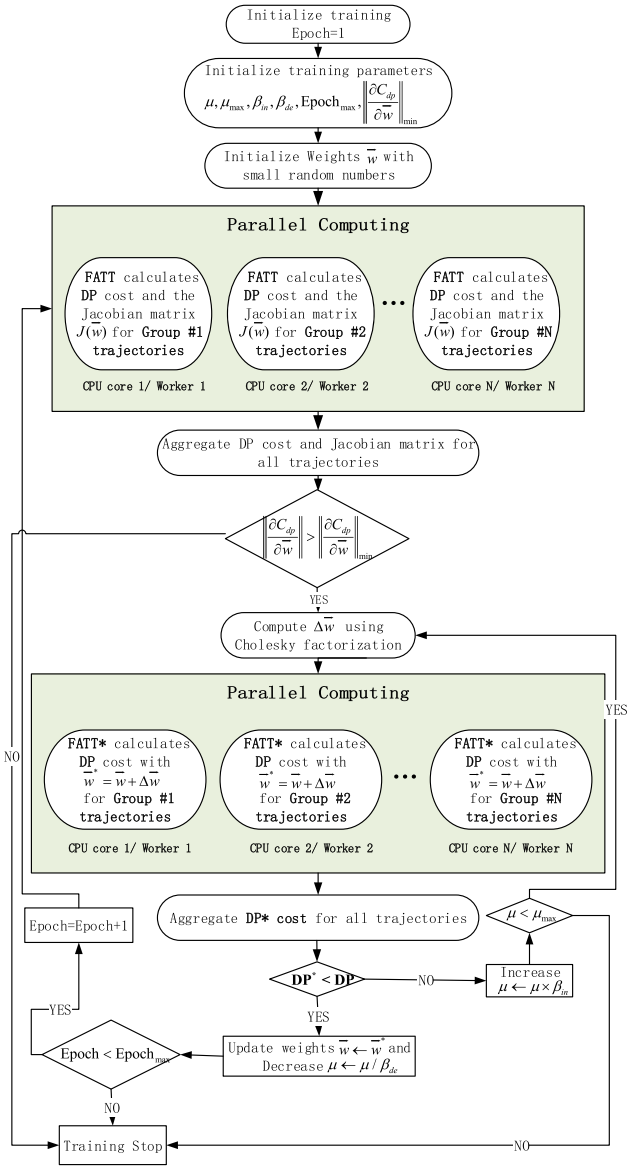

Fig. 8. Proposed parallel LM+FATT trajectory training algorithm for an RNN controller. $\mu_{\max}$ stands for maximum μ, and β_{de} and β_{in} signify the decreasing and increasing factors, respectively.

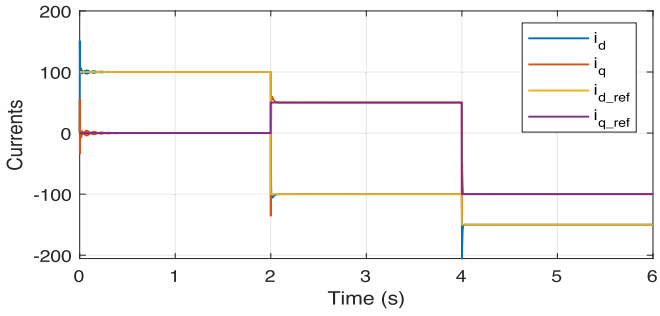

Fig. 9. Tracking performance in a closed-loop after successful RNN training.

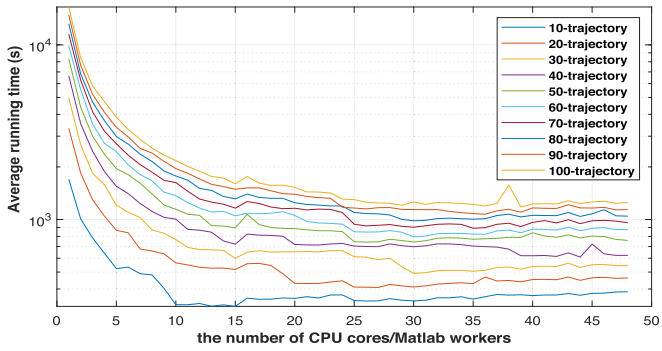

Fig. 10. Average running time across Matlab workers on Amazon EC2 Cloud.

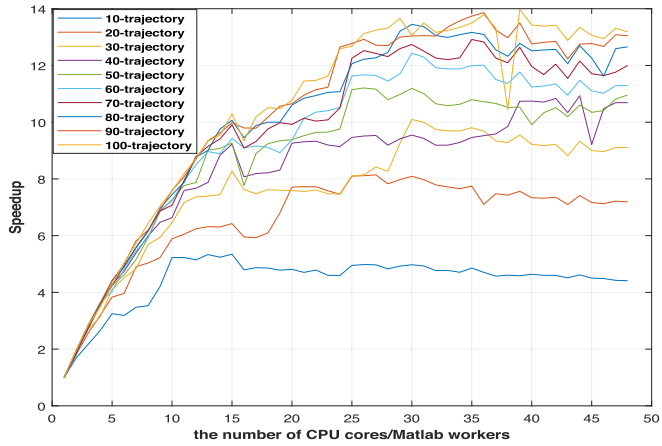

Fig. 11. Speedup across Matlab workers on Amazon EC2 Cloud.

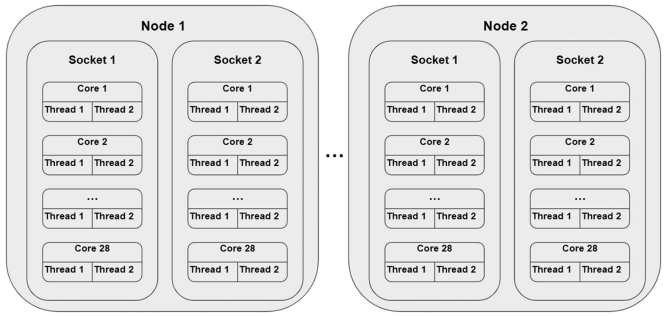

Fig. 12. Schematic of the HPC Cluster architecture.

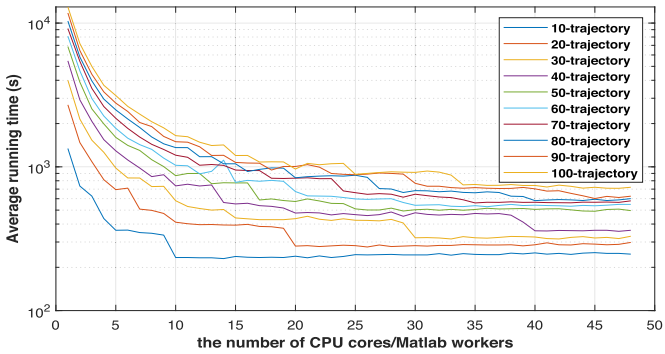

Fig. 13. Average running time across Matlab workers on the HPC cluster.

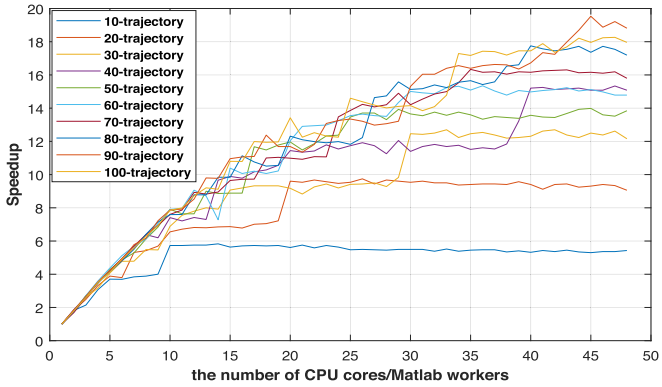

Fig. 14. Speedup across Matlab workers on the HPC cluster.

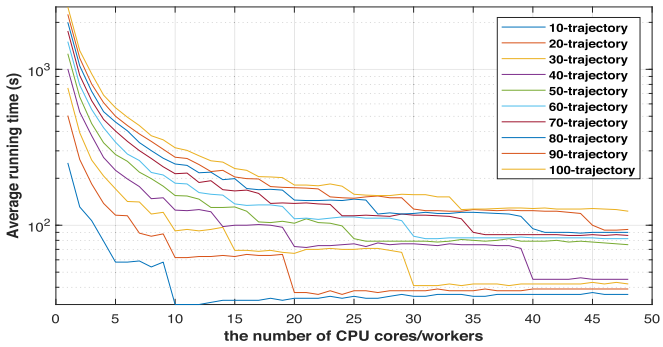

Fig. 15. Average running time across workers on the HPC cluster.

improve performance. LAPACK is a software library of more complex linear algebra operations also written in Fortran.

The OpenMPI implementation of the Message Passing Interface (MPI) standard [42] was utilized to parallelize the workload among the available worker nodes. An advantage of the Open MPI framework is that programs are scalable to larger and more powerful clusters. To parallelize the workload, the MPI program has a sequential part and parallel parts as illustrated in Fig. 8. The master process in the MPI program handles the sequential parts and sends messages to each worker process to manage its parallel part of the workload.

Fig. 15 shows the running time comparison across different numbers of workers. The horizontal axis represents the number of CPU cores/workers, while the vertical axis stands for the average running time. Fig. 16 shows the speedup comparison on different numbers of workers. The horizontal axis represents the number of CPU cores/workers and the vertical axis stands for the speedup. Compared with Figs. 13 and 14, the parallel version of the C++ program (Figs. 15 and 16) runs faster than the Matlab version on the HPC cluster. For example, the maximum speedup of the C++ version is around 8 times compared to the 6 times speedup of the corresponding Matlab version for 10-trajectory training. Overall, among the three implementations: the C++ version on the HPC cluster, the Matlab version on the HPC cluster, and the Matlab version on the cloud, the C++ version on the HPC cluster achieves the best results. The next one is the Matlab version on the HPC cluster, while the Matlab version on the cloud is the slowest one.

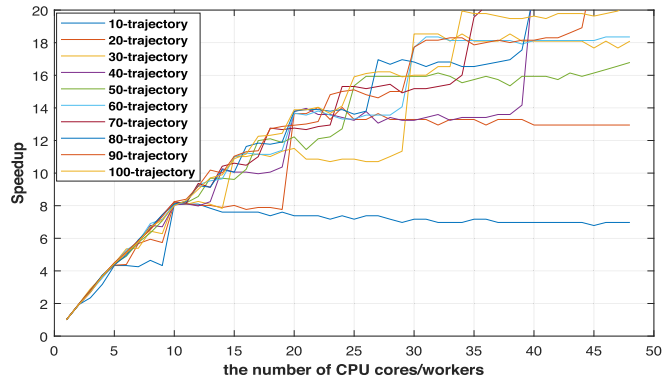

Fig. 16. Speedup across workers on the HPC cluster.

VI. Conclusion

This paper investigated how to use parallel computing to accelerate the LM algorithm for training an RNN controller in a closed-loop control system. The proposed parallel trajectory training algorithm incorporates LM and FATT algorithms. The training programs were implemented using both Matlab and C++. The developed programs were tested on two computing platforms, namely the Amazon EC2 Cloud and an HPC cluster. Performance comparison results show that the parallel training algorithm can provide a significant speedup compared to its non-parallelized counterparts. Specifically, the C++ implementation achieves better speedup than the corresponding MATLAB implementation. The program running on HPC clusters can further yield even better speedup than that on cloud platforms. The significant speedup performance makes it suitable to train RNN controllers with a large number of trajectories and long-duration trajectories.

Acknowledgments

This work was supported by the National Science Foundation of the United States under Grants 2131214 and 2131175. The authors thank anonymous reviewers for their insightful comments and inputs.

References

[1] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quart. Appl. Math.*, vol. 2, pp. 164–168, 1944.
[2] D.W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *J. Soc. Ind. Appl. Math.*, vol. 11, no. 2, pp. 431–441, 1963.
[3] M. T. Hagan, H. B. Demuth, and M. H. Beale, *Neural Network Design*. Boston, MA, USA: PWS, 2002, pp. 19–23.
[4] L. Chan and C. Szeto, "Training recurrent network with block-diagonal approximated Levenberg-Marquardt algorithm," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, 1999, pp. 1521–1526.
[5] Y. Kwak, "An accelerated Levenberg-Marquardt algorithm for feedforward network," *J. Korean Data Inf. Sci. Soc.*, vol. 23, no. 5, pp. 1027–1035, 2012.
[6] T. Tawara, Levenberg-Marquardt with sparse block matrices on the GPU. [Online]. Available: https://on-demand.gputechconf.com/gtc/2012/presentations/S0231-Levenberg-Marquardt-Using-Block-Sparse-Matrices-on-CUDA.pdf
[7] N. N. R. Ranga Suri, D. Deodhare, and P. Nagabhushan, "Parallel Levenberg-Marquardt-based neural network training on Linux clusters - a case study," in *Proc. 3rd Indian Conf. Comput. Vis. Graph. Image Process.*, 2002, pp. 1–6.
[8] J. Cao, K. A. Novstrup, A. Goyal, S. P. Midkiff, and J. M. Caruthers, "A parallel levenberg-marquardt algorithm," in *Proc. 23rd Int. Conf. Supercomput*, 2009, pp. 450–459. [Online]. Available: https://doi.org/10.1145/1542275.1542338

[9] A. Przybylski, B. Thiel, J. Keller-Findeisen, B. Stock, and M. Bates, "Gpufit: An open-source toolkit for GPU-accelerated curve fitting," *Sci. Rep.* vol. 7, 2017, Art. no. 15722. [Online]. Available: https://doi.org/10.1038/s41598--017-15313-9

[10] X. Zhu and D. Zhang, "Efficient parallel Levenberg-Marquardt model fitting towards real-time automated parametric imaging microscopy," *PLoS One*, vol. 8, no. 10, 2013, Art. no. e76665. [Online]. Available: https://doi.org/10.1371/journal.pone.0076665

[11] Y. Lin, D. O'Malley, and V. V. Vesselinov, "A computationally efficient parallel Levenberg-Marquardt algorithm for highly parameterized inverse model analyses," *Water Resour. Res.*, vol. 52, no. 9, pp. 6948–6977, 2016. [online]. Available: https://doi.org/10.1002/2016WR019028

[12] S. Li, M. Fairbank, D. C. Wunsch, and E. Alonso, "Vector control of a grid-connected rectifier/inverter using an artificial neural network," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Brisbane Australia, 2012, pp. 1–7.

[13] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, no. 2, pp. 270–280, 1989.

[14] H. Jaeger, "Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network"," Gesellschaft für Mathematik und Datenverarbeitung Report, 2002, Art. no. 159.

[15] F. Gomez, J. Schmidhuber, R. Miikkulainen, and M. Mitchell, "Accelerated neural evolution through cooperatively coevolved synapses," *J. Mach. Learn. Res.*, vol. 9, no. 5, pp. 937–965, 2008.

[16] S. Ma and J. Chuanyi, "A unified approach on fast training of feedforward and recurrent networks using EM algorithm," *IEEE Trans. Signal Process.*, vol. 46, no. 8, pp. 2270–2274, Aug. 1998.

[17] S.K. Ng and G.J. McLachlan, "Using the EM algorithm to train neural networks: Misconceptions and a new algorithm for multiclass classification," *IEEE Trans. Neural Netw.*, vol. 15, no. 3, pp. 738–749, May 2004.

[18] X. Fu, S. Li, and I. Jaithwa, "Implement optimal vector control for LCL-filter-based grid-connected converters by using recurrent neural networks," *IEEE Trans. Ind. Electron.*, vol. 62, no. 7, pp. 4443–4454, Jul. 2015.

[19] Y. Tanoto, W. Ongsakul, and C. O.P. Marpaung, "Levenberg-Marquardt recurrent networks for long term electricity peak load forecasting," *Telecommun. Comput. Electron. Control*, vol. 9, no. 2, pp. 257–266, 2011.

[20] X. Fu, S. Li, M. Fairbank, D. C. Wunsch, and E. Alonso, "Training recurrent neural networks with the Levenberg–Marquardt algorithm for optimal control of a grid-connected converter," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 9, pp. 1900–1912, Sep. 2015.

[21] Texas Instruments, Digitally controlled solar micro inverter design using C2000 Piccolo microcontroller user's guide, 2014. [Online]. Available: https://www.ti.com/lit/ug/tidu405b/tidu405b.pdf

[22] W. Wu, Y. Sun, Z. Lin, T. Tang, F. Blaabjerg, and H. S. Chung, "A new LCL-filter with in-series parallel resonant circuit for single-phase grid-tied inverter," *IEEE Trans. Ind. Electron.*, vol. 61, no. 9, pp. 4640–4644, Sep. 2014.

[23] R. Teodorescu, M. Liserre, and P. Rodriguez, *Grid Converters for Photovoltaic and Wind Power Systems*. Hoboken, NJ, USA: Wiley, 2011.

[24] J. Sturtz, X. Fu, C. D. Hingu, and L. Qingge, "A novel weight dropout approach to accelerate the neural network controller embedded implementation on FPGA for a solar inverter," in *Proc. IEEE Int. Conf. Smart Comput.*, Nashville, TN, USA, 2023, pp. 157–163.

[25] W. Waithaka, X. Fu, A. Hadi, R. Challoo, and S. Li, "DSP implementation of a novel recurrent neural network controller into a TI solar microinverter," in *Proc. IEEE PES Gen. Meeting*, 2021, pp. 1–5.

[26] X. Fu, S. Li, D. C. Wunsch, and E. Alonso, "Local stability and convergence analysis of neural network controllers with error integral inputs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 7, pp. 3751–3763, Jul. 2023.

[27] F. Wang, H. Zhang, and D. Liu, "Adaptive dynamic programming: An introduction," *IEEE Comput. Intell. Mag.*, vol. 4, no. 2, pp. 39–47, May 2009.

[28] R. E. Bellman, *Dynamic Programming*. Princeton, NJ, USA: Princeton Univ. Press, 1957.

[29] D. V. Prokhorov and D. C. Wunsch, "Adaptive critic designs," *IEEE Trans. Neural Netw.*, vol. 8, no. 5, pp. 997–1007, Sep. 1997.

[30] Amazon EC2 instance Types. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[31] Matlab. Available: https://www.mathworks.com/products/matlab.html

[32] C. Lameter, "NUMA (Non-Uniform Memory Access): An overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors," *Queue*, vol. 11, no. 7, pp. 40–51, Jul. 2013.

[33] AlmaLinux OS Foundation, Almalinux. [Online]. Available: https://almalinux.org/

[34] Slurm: Workload Manager. [Online]. Available: https://slurm.schedmd.com/overview.html

[35] C++17. [Online]. Available: https://en.cppreference.com/w/cpp/17

[36] Free Software Foundation, Inc, G++(1) - Linux man page. [Online]. Available: https://linux.die.net/man/1/g++

[37] C. Sanderson and R. Curtin, "Armadillo: A template-based C++ library for linear algebra," *J. Open Source Softw.*, vol. 1, 2016, Art. no. 26.

[38] C. Sanderson and R. Curtin, "A user-friendly hybrid sparse matrix class in C++," *Lecture Notes Comput. Sci.*, vol. 10931, pp. 422–430, 2018.

[39] BLAS. [Online]. Available: https://netlib.org/blas/

[40] LAPACK. [Online]. Available: https://netlib.org/lapack/

[41] OpenBLAS. [Online]. Available: https://github.com/xianyi/OpenBLAS

[42] Open MPI: Open Source High Performance Computing. [Online]. Available: https://www.open-mpi.org/