2024 IEEE 15th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON) | 979-8-3315-4090-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/UEMCON62879.2024.10754702

An Adaptive Dropout and Parallel Computing Approaches for Accelerating RNN Controller

Kushal Kalyan Devalampeta Surendranath
Department of Computer Science
North Carolina A&T State University
Greensboro, NC, USA
ksurendranath@aggies.ncat.edu

Maxwell Sam
Department of Computer Science
North Carolina A&T State University
Greensboro, NC, USA
msam@aggies.ncat.edu

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

***Abstract*—The efficiency and performance of neural network (NN) controllers present a significant challenge in the rapidly evolving landscape of real-time closed-loop control systems, such as those used in solar inverters. This paper introduces a novel approach that enhances training efficiency by combining adaptive dropout with parallel computing techniques, utilizing the Levenberg-Marquardt (LM) algorithm and Forward Accumulation Through Time (FATT). Unlike traditional dropout methods that apply a fixed dropout rate uniformly across all neurons, Adaptive Dropout dynamically adjusts the dropout rate based on each neuron's calculated importance and its stage in the training process. This allows for the protection of more critical neurons while regularizing less significant ones, thereby improving convergence speed and enhancing generalization in the neural network controller. To further accelerate the training process, the Adaptive Dropout method is seamlessly integrated into a parallel computing architecture. This architecture employs multiple cores to compute Dynamic Programming (DP) costs and Jacobian matrices for various trajectories simultaneously. This approach not only harnesses the computational power of modern multi-core systems but also ensures efficient processing across all trajectories. The experimental results demonstrate that adaptive dropout with parallel computing provides improvements in training efficiency and overall performance than both no dropout and weight dropout control schemes.**

***Index Terms*—Neural Network Controller, Solar Inverter, Adaptive Dropout, Parallel Computing, Levenberg-Marquardt (LM) Algorithm, Forward Accumulation Through Time (FATT)**

I. Introduction

The growing demand for efficient energy systems, particularly in the context of renewable energy sources like solar power, has driven the advancement of technologies such as solar inverters. These inverters play a critical role by converting the direct current (DC) generated by solar panels into alternating current (AC) suitable for grid integration. Traditionally, control methods like the d-q vector control mechanism have been employed to manage these inverters. However, these approaches involve complex algorithms and computations, as highlighted in [1], which limit their effectiveness in dynamic and resource-constrained environments [2].

[1]This work was supported by the U.S. National Science Foundation under Award 2131175. Corresponding author: Letu Qingge.

Recent studies have explored various control strategies for DC-DC and DC-AC converters. The Archimedes Optimization Algorithm (AOA) has been used to fine-tune Proportional-Integral-Derivative (PID) controllers for DC-DC buck converters, achieving superior voltage regulation and response times [3]. A robust nonlinear control strategy for DC-AC converters in grid-connected fuel cell systems was presented in [4], using partial feedback linearization to improve dynamic grid performance. For AC-DC-AC converters, [5] proposed a virtual-impedance approach to reduce current stress and enhanced efficiency. Transient modeling for faulty DC microgrids, considering converter control effects, was discussed in [6], emphasizing the importance of accurate transient analysis for reliability. Data-driven PID control for DC-DC buck-boost converters, as shown in [7], further improved controller performance and trajectory tracking accuracy.

Neural network controllers have emerged as a promising alternative, offering greater adaptability and efficiency in managing the nonlinearities inherent in power electronic systems [8]. Recurrent Neural Networks (RNNs), in particular, are well-suited for this task due to their ability to process sequential data and retain memory of past inputs, making them ideal for real-time closed-loop control systems. However, deploying RNN controllers in embedded systems, such as digital signal processors (DSPs) and field-programmable gate arrays (FPGAs), presents significant challenges. The training process for these networks is often computationally intensive, necessitating innovative approaches to ensure their effective implementation in real-world systems.

Dropout is a widely used technique in neural networks to mitigate overfitting, effectively applied across various architectures, including Convolutional Neural Networks (CNNs) [9] and Recurrent Neural Networks (RNNs) [10]. By randomly deactivating neurons during training, dropout helps prevent overfitting and enhances the generalization of the model [11][12]. However, traditional dropout methods use a fixed dropout rate uniformly across all neurons, regardless of their importance within the network. This approach can limit the efficiency and effectiveness of the training process, especially in resource-constrained environments.

Previous works on dropout techniques, such as those by Gal and Ghahramani [13], introduced variational dropout for

RNNs, which used a Bayesian framework to maintain dropout masks over time, providing a more structured way to handle the temporal nature of RNNs. Although this method improved generalization, it still applied dropout uniformly throughout the network, which lead to suboptimal training when neurons of varying importance are treated equally. This uniformity in dropout rate application has been a recurring limitation in several studies. For example, Srivastava et al. [14] applied a static dropout rate throughout training, and Pham et al. [10] also did not account for the dynamic importance of neurons during the training process.

To accelerate the neural network controller, this paper introduces an Adaptive Dropout technique that dynamically adjusts the dropout rate for each neuron based on its importance and the stage of training. By protecting more critical neurons and regularizing less significant ones, this method not only enhances the network's convergence speed but also improves its generalization capabilities [14]. This adaptive approach is further integrated into a parallel computing framework, leveraging the computational power of modern multi-core systems. By distributing the Dynamic Programming (DP) costs and Jacobian matrix computations across multiple cores, the training process is further accelerated [12].

This paper focuses on the integration of adaptive dropout with the Levenberg-Marquardt (LM) algorithm and Forward Accumulation Through Time (FATT) techniques within a parallel computing architecture [12]. This combination not only enhances training efficiency, but also ensures that neural network (NN) controller can be effectively deployed in real-time systems, such as those required for managing solar inverters in grid-connected environments.

The contributions of this research are threefold: First, an adaptive dropout mechanism is developed to improve the training efficiency of RNN controllers by selectively regularizing neurons during training. Second, this adaptive dropout mechanism is incorporated into a parallel computing framework, demonstrating its effectiveness in handling large-scale trajectory datasets on platforms such as GPU clusters. Third, a comprehensive evaluation of the method shows significant improvements in both training speed and model performance.

The remainder of this paper is structured as follows: Section 2 introduces the NN controller used in closed-loop control systems for solar inverters. Section 3 details the adaptive dropout approach and its integration with parallel computing. Section 4 discusses the training of the NN controller using the adaptive dropout technique. Section 5 presents the implementation of these methods in C++ on a GPU platform together with experimental results. Finally, Section 6 concludes the paper with a summary of the key findings and contributions.

II. NN CONTROLLER IN CLOSED-LOOP CONTROL SYSTEMS FOR SOLAR INVERTERS

A. Solar Microinverter

A neural network (NN) controller is implemented within the Piccolo real-time digital controller, as illustrated in Fig. 1 [15]. The photovoltaic (PV) panel generates direct current (DC) electricity from sunlight, which is then optimized using a converter that incorporates Maximum Power Point Tracking (MPPT) to maximize energy extraction. The high-voltage DC Bus transfers this optimized DC power to the next stage of the system. The inverter then converts the DC power into alternating current (AC), making it suitable for use in the electrical grid. The final stage outputs AC electricity, which can be either fed into the grid or used locally. The Neural Network Controller, integrated with the Piccolo Digital Controller, manages the operation of the DC-AC inverter by receiving control signals and adjusting the system in real-time. It utilizes feedback from the AC output to make continuous adjustments, thereby improving the system's efficiency and stability.

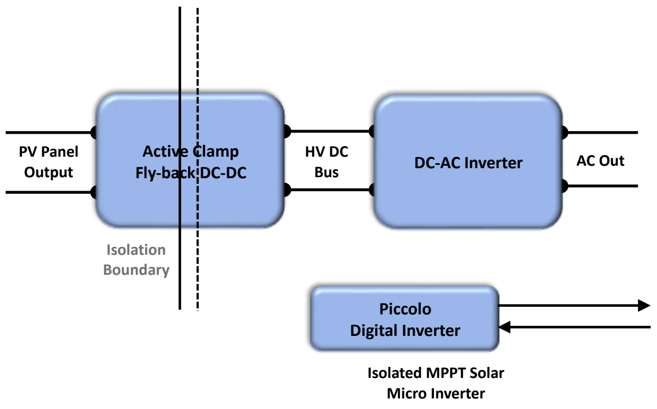


Fig. 1: Microinverter Block Diagram [16]

B. NN Controller with Special Tracking Error Integrals

The structure of the NN Controller, as shown in Fig. 2, consists of an input layer with four neurons, two hidden layers with six neurons each, and an output layer with two neurons to control the system outputs. The optimal number of neurons in each hidden layer was determined through extensive trial and error. After numerous iterations, it was found that using six neurons in each hidden layer consistently provided satisfactory real-time control performance. This configuration effectively captures the complexity of the control tasks while maintaining a balance between model complexity and performance in real-time applications. Additionally, the dropout technique has proven valuable in reducing the number of active neurons or weights, enhancing the NN Controller's compatibility with embedded real-time computing systems [11].

The NN Controller receives tracking error input signals $\overrightarrow{e_{dq}}$ and their corresponding integral values $\overrightarrow{s_{dq}}$ in the input block. To prevent input saturation, $\overrightarrow{e_{dq}}$ and $\overrightarrow{s_{dq}}$ are normalized using the hyperbolic tangent function, which restricts the values to the interval $[-1, 1]$. These normalized values are then further scaled by constant gains, $Gain$ and $Gain2$, respectively.

For step references [17], the special error integral terms $\overrightarrow{s_{dq}}$ ensure that there is no steady-state error. The NN controller can be formally expressed as shown in equation (1) [11], where the weights from the input layer to the first hidden layer, second hidden layer, and output layer are represented by $\overrightarrow{w_1}$, $\overrightarrow{w_2}$, and $\overrightarrow{w_3}$, respectively. Biases for each layer are incorporated within these weights.

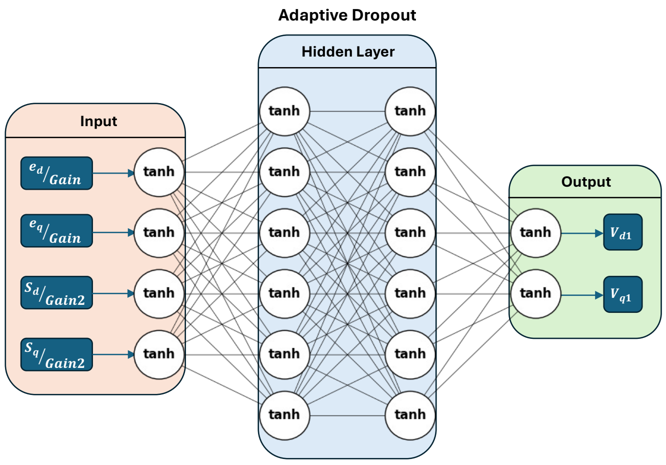


Fig. 2: The NN Controller with special tracking error integrals [11]

In addition, to further enhance the performance and efficiency of the NN controller, particularly in resource-constrained environments, an adaptive dropout technique is employed. This method dynamically adjusts the dropout rate based on the importance of each neuron and the progress of training, ensuring that the NN controller remains both robust and computationally efficient.

$$R\left(\overrightarrow{e_{dq}}, \overrightarrow{s_{dq}}, \overrightarrow{w_1}, \overrightarrow{w_2}, \overrightarrow{w_3}\right) = \tanh\left\{\overrightarrow{w_3}\begin{bmatrix} \tanh\left\{\overrightarrow{w_2}\begin{bmatrix} \tanh\left\{\overrightarrow{w_1}\begin{bmatrix} \tanh\begin{bmatrix} \frac{e_d}{Gain} \\ \frac{e_q}{Gain} \\ \frac{s_d}{Gain2} \\ \frac{s_q}{Gain2} \\ -1 \end{bmatrix} \end{bmatrix}\right\} \\ -1 \end{bmatrix}\right\} \\ -1 \end{bmatrix}\right\} \tag{1}$$

III. Training Controller with Adaptive Dropout Technique and Integration of Parallel Computing

The training of neural network controllers is optimized by incorporating adaptive dropout and parallel computing within the combined Forward Accumulation Through Time (FATT) and Levenberg-Marquardt (LM) algorithm.

The cost function for training the RNN using Dynamic Programming (DP) is defined as in equations (2)(3):

$$C_{dp} = \sum_{k=j}^{\infty} \gamma^{k-j} U(\vec{e}_{dq}(k)) \tag{2}$$

$$= \sum_{k=j}^{\infty} \gamma^{k-j} \sqrt{(id(k) - id_{\text{ref}}(k))^2 + (iq(k) - iq_{\text{ref}}(k))^2} \tag{3}$$

where $j > 0$ denotes the starting point, $0 < \gamma \leq 1$ represents the discount factor, and U is the local cost or utility function. The function C_{dp}, given the initial time j and the initial state $\vec{idq}(j)$, represents the cost-to-go for the state $\vec{idq}(j)$ in the Dynamic Programming (DP) problem.

A. LM Algorithm

The Levenberg-Marquardt (LM) algorithm is a widely used optimization technique for minimizing nonlinear least-squares problems [12]. It integrates the gradient descent method with the Gauss-Newton algorithm, dynamically adjusting the step size to balance local and global search strategies and achieve faster convergence.

In the adaptive dropout approach, the LM algorithm is modified to incorporate a dynamic dropout rate that adjusts based on each neuron's importance during training. This adaptation ensures that the most significant neurons contribute more consistently to gradient calculations, thereby enhancing the learning process.

The cost function C_{dp} is rewritten in a Sum-Of-Squares form as follows (4):

$$C_{dp} = \sum_{k=1}^{N} U(\overrightarrow{e_{dq}}(k)) = \sum_{k=1}^{N} V^2(k) \tag{4}$$

The gradient of the cost function is given by (5):

$$\frac{\partial C_{dp}}{\partial \overrightarrow{w}} = 2 J_v(\overrightarrow{w})^T V \tag{5}$$

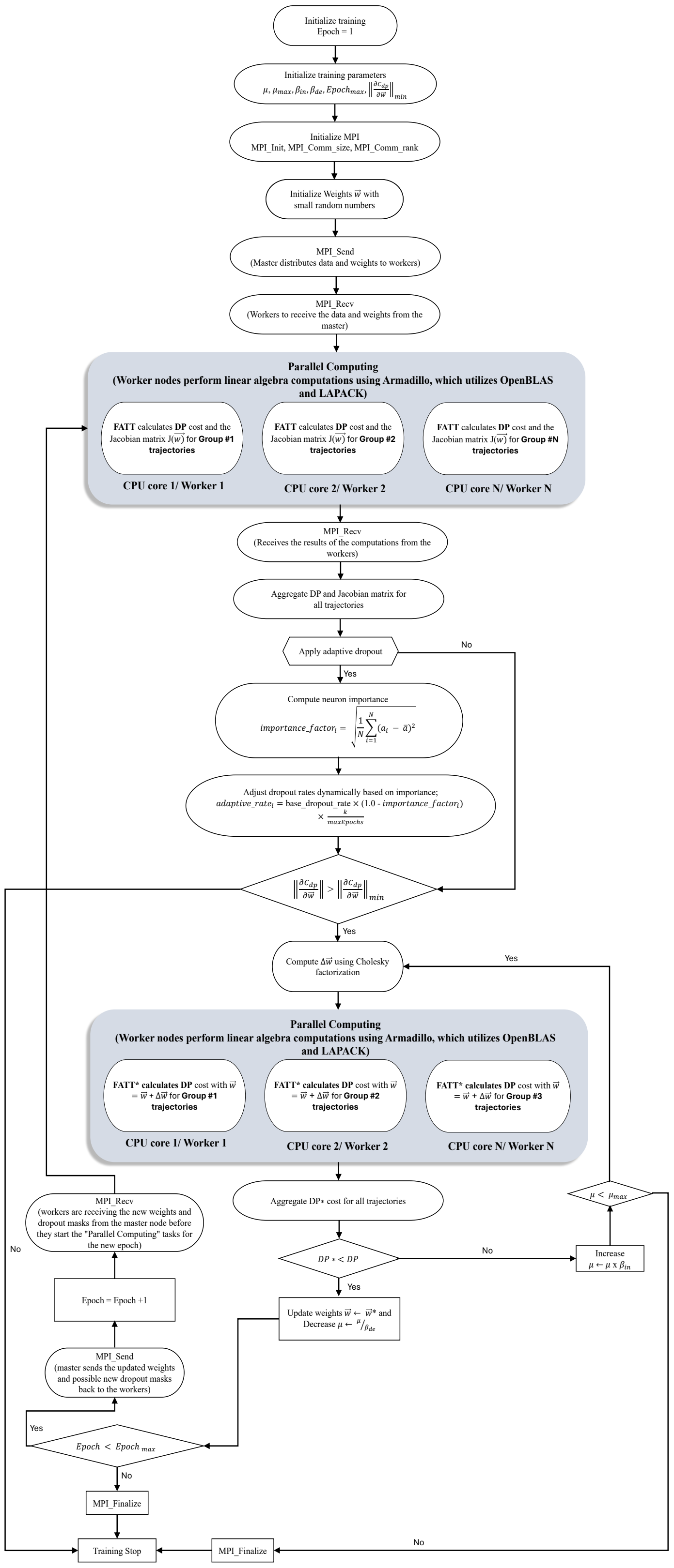
where $J_v(\overrightarrow{w})$ is computed with adaptive dropout applied, ensuring that only the neurons not dropped contribute to the calculation. The LM weight update equation (6) with adaptive dropout is:

$$\triangle \overrightarrow{w} = -[J_v(\overrightarrow{w})^T J_v(\overrightarrow{w}) + \mu I]^{-1} J_v(\overrightarrow{w})^T V \tag{6}$$

B. FATT (Forward Accumulation Through Time)

The FATT algorithm is a method used in recurrent neural networks (RNNs) to compute gradients efficiently during backpropagation through time. It involves accumulating the contributions of each time step forward, which helps in managing memory and computational resources while handling long sequences in RNNs[12].

The Forward Accumulation Through Time (FATT) algorithm incorporates adaptive dropout by dynamically adjusting the dropout rates during the unrolling of the system in the forward path. This change ensures a more robust and efficient training process. Parallel computing is integrated to speed up



Initialize training
Epoch = 1
Initialize training parameters
$\mu, \mu_{max}, \beta_{in}, \beta_{de}, Epoch_{max}, \left\|\frac{\partial C_{dp}}{\partial \vec{w}}\right\|_{min}$
Initialize MPI
MPI_Init, MPI_Comm_size, MPI_Comm_rank
Initialize Weights $\vec{w}$ with small random numbers
MPI_Send
(Master distributes data and weights to workers)
MPI_Recv
(Workers to receive the data and weights from the master)
Parallel Computing
(Worker nodes perform linear algebra computations using Armadillo, which utilizes OpenBLAS and LAPACK)
FATT calculates DP cost and the Jacobian matrix $J(\vec{w})$ for Group #1 trajectories
FATT calculates DP cost and the Jacobian matrix $J(\vec{w})$ for Group #2 trajectories
FATT calculates DP cost and the Jacobian matrix $J(\vec{w})$ for Group #N trajectories
CPU core 1/ Worker 1
CPU core 2/ Worker 2
CPU core N/ Worker N
MPI_Recv
(Receives the results of the computations from the workers)
Aggregate DP and Jacobian matrix for all trajectories
Apply adaptive dropout
No
Yes
Compute neuron importance
$importance\_factor_i = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})^2}$
Adjust dropout rates dynamically based on importance;
$adaptive\_rate_i$ = base_dropout_rate × (1.0 - $importance\_factor_i$) × $\frac{k}{maxEpochs}$
$\left\|\frac{\partial C_{dp}}{\partial \vec{w}}\right\| > \left\|\frac{\partial C_{dp}}{\partial \vec{w}}\right\|_{min}$
Yes
Compute $\Delta\vec{w}$ using Cholesky factorization
Yes
Parallel Computing
(Worker nodes perform linear algebra computations using Armadillo, which utilizes OpenBLAS and LAPACK)
FATT* calculates DP cost with $\vec{w} = \vec{w} + \Delta\vec{w}$ for Group #1 trajectories
FATT* calculates DP cost with $\vec{w} = \vec{w} + \Delta\vec{w}$ for Group #2 trajectories
FATT* calculates DP cost with $\vec{w} = \vec{w} + \Delta\vec{w}$ for Group #3 trajectories
CPU core 1/ Worker 1
CPU core 2/ Worker 2
CPU core N/ Worker N
MPI_Recv
(workers are receiving the new weights and dropout masks from the master node before they start the "Parallel Computing" tasks for the new epoch)
No
Aggregate DP∗ cost for all trajectories
$\mu < \mu_{max}$
$DP* < DP$
No
Increase
$\mu \leftarrow \mu \times \beta_{in}$
Yes
Epoch = Epoch +1
Update weights $\vec{w} \leftarrow \vec{w}^*$ and Decrease $\mu \leftarrow {}^{\mu}/_{\beta_{de}}$
MPI_Send
(master sends the updated weights and possible new dropout masks back to the workers)
Yes
$Epoch < Epoch_{max}$
No
MPI_Finalize
Training Stop
MPI_Finalize
No

Fig. 3: Parallel LM+FATT trajectory training algorithm for an RNN controller with adaptive dropout

Algorithm 1 FATT Algorithm to Calculate the Jacobian Matrix and to Accumulate DP Cost for One Trajectory and adaptive dropout [18]

```
Initialize weights: $w$
maxEpochs $\leftarrow$ Maximum number of epochs
base_dropout_rate $\leftarrow$ Initial dropout probability
Initialize dropout parameters: base_dropout_rate, training_progress
**for** $k = 1$ to maxEpochs **do**
    **for** each trajectory **in parallel do**
        **for** $n = 0$ to $N-1$ **do**
            $\overrightarrow{vdq1}(n) \leftarrow k_{PWM} R(\overrightarrow{edq}(n), \overrightarrow{sdq}(n), w)$
            $\frac{\partial \overrightarrow{s_{dq}}(n)}{\partial w} \leftarrow T_s \left[\frac{\partial \overrightarrow{\phi_{dq}}(n)}{\partial w} - \frac{1}{2}\frac{\partial \overrightarrow{idq}(n)}{\partial w}\right]$
            $\frac{\partial \overrightarrow{vdq1}(n)}{\partial w} \leftarrow k_{PWM} \left[\frac{\partial R(n)}{\partial w} + \frac{\partial R(n)}{\partial \overrightarrow{edq}(n)}\frac{\partial \overrightarrow{idq}(n)}{\partial w} + \frac{\partial R(n)}{\partial \overrightarrow{sdq}(n)}\frac{\partial \overrightarrow{sdq}(n)}{\partial w}\right]$
            $\frac{\partial \overrightarrow{idqs}(n+1)}{\partial w} \leftarrow A\frac{\partial \overrightarrow{idqs}(n)}{\partial w}$
            $\frac{\partial \overrightarrow{idqs}(n+1)}{\partial w} \leftarrow \frac{\partial \overrightarrow{idqs}(n+1)}{\partial w} + B\frac{\partial \overrightarrow{udq}(n+1)}{\partial w}$
            $\frac{\partial \overrightarrow{idq}(n+1)}{\partial w} \leftarrow$ the first two terms of $\frac{\partial \overrightarrow{idqs}(n+1)}{\partial w}$
            $\frac{\partial \overrightarrow{\phi_{dq}}(n+1)}{\partial w} \leftarrow \frac{\partial \overrightarrow{\phi_{dq}}(n)}{\partial w} + \frac{\partial \overrightarrow{idq}(n+1)}{\partial w}$
            $\overrightarrow{idqs}(n+1) \leftarrow A\overrightarrow{idqs}(n) + B\overrightarrow{udqs}(n)$
            $\overrightarrow{edq}(n+1) \leftarrow \overrightarrow{idq}(n+1) - \overrightarrow{idq}_{\text{ref}}(n+1)$
            $\overrightarrow{sdq}(n+1) \leftarrow \overrightarrow{sdq}(n) + \frac{T_s}{2}\left[\overrightarrow{edq}(n) + \overrightarrow{edq}(n+1)\right]$
            $C \leftarrow C + U(\overrightarrow{edq}(n+1))$ accumulate DP cost
            $\frac{\partial \vec{V}(n+1)}{\partial w} \leftarrow \frac{\partial \vec{V}(n+1)}{\partial \overrightarrow{edq}(n+1)}\frac{\partial \overrightarrow{idq}(n+1)}{\partial w}$
            the $(n+1)$th row of $J_v(w) \leftarrow \frac{\partial \vec{V}(n+1)}{\partial w}$
        **end for**
    **end for**
    **while** $\lambda < \lambda_{\text{max}}$ **do**
        $\Delta w \leftarrow \text{solve}(J^T J + \lambda I, P)$
        $w_{\text{temp}} \leftarrow w + \Delta w$
        **Apply Adaptive Dropout:**
            importance_factor$_i \leftarrow \sqrt{\frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})^2}$ (for each neuron $i$)
            adaptive_rate$_i \leftarrow$ base_dropout_rate $\times$ (1.0 $-$ importance_factor$_i$) $\times \frac{k}{\text{maxEpochs}}$
            Apply dropout to neurons with adaptive_rate$_i$
        $C_2 \leftarrow \text{FATT}(X, w_{\text{temp}})$
        **if** $C_2 < C_1$ **then**
            $w \leftarrow w_{\text{temp}}$
            $\lambda \leftarrow \min(\lambda \times \lambda_{\text{dec}}, \lambda_{\text{min}})$
            **break**
        **else**
            $\lambda \leftarrow \lambda \times \lambda_{\text{inc}}$
        **end if**
    **end while**
    **if** $\lambda = \lambda_{\text{max}}$ **then**
        **break**
    **end if**
**end for**
{On exit, the Jacobian matrix $J_v(w)$ is finished for each trajectory in parallel.}
```

the calculation of the Dynamic Programming (DP) cost and the Jacobian matrix for multiple trajectories.

The process begins by initializing the weights and dropout parameters, followed by calculating the Jacobian matrix $J_v(\overrightarrow{w})$ and accumulating the DP cost as in equation (7):

$$C = \sum_{n=1}^{N} U(\overrightarrow{e_{dq}}(n+1)) \tag{7}$$

Adaptive dropout is applied at each iteration, where the dropout rate is dynamically adjusted based on the importance of each neuron, calculated as in equation (8):

$$\text{importance\_factor} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(a_i - \bar{a})^2} \tag{8}$$

where a_i represents the activation of the neuron, $\bar{a}$ is the mean activation, and N is the number of observations. Neurons with a higher standard deviation (importance factor) are considered more crucial for learning and are thus less likely to be dropped.

C. Adaptive dropout approach

The adaptive dropout methodology is an enhanced technique of the standard dropout method whereby constantly modifying the dropout rate according to the training progress and the relative relevance of each neuron [19]. Conventional dropout prevents overfitting by randomly deactivating a subset of neurons during training; however, it has a constant dropout rate for all neurons, which may not be suitable for complex models like Recurrent Neural Networks (RNNs). To address this limitation, adaptive dropout dynamically adjusts the dropout rate, allowing the network to prioritize critical neurons during training. This approach enhances generalization and accelerates convergence.

1) Neuron Importance Calculation

The importance of each neuron is determined by calculating the standard deviation of its activations. The neurons with higher activation variance are likely contributing more to the learning process. The importance factor for each neuron is calculated using equation (8).

2) Adaptive Dropout Rate

The dropout rate for each neuron is not static; it adapts based on the calculated importance factor and the current stage of training. The formula for the adaptive dropout rate is as in equation (9):

$$\begin{aligned}\text{adaptive\_rate} = {} & \text{base\_dropout\_rate} \times \\ & (1.0 - \text{importance\_factor}) \times \text{training\_progress}\end{aligned} \tag{9}$$

where, base_dropout_rate $= 0.4$ and the training_progress is $\frac{k}{\text{maxEpochs}}$. k is the current epoch.

In this equation, the base dropout rate serves as the initial dropout probability, while the importance factor adjusts this probability according to the neuron's significance. The training progress, representing the fraction of the training process completed, further refines the dropout rate, allowing it to decrease as training nears completion, ensuring that critical neurons are preserved more often as the network approaches convergence.

3) Application of Dropout

During each iteration of training, the adaptive dropout rate is used to determine whether each neuron should be dropped. This decision is made using a Bernoulli distribution, where the probability of dropping a neuron is guided by the adaptive dropout rate. This ensures that each iteration might drop a different set of neurons, preventing the network from becoming overly dependent on specific neurons and further enhancing generalization.

4) Temporary and Random Dropout

The deactivation of neurons is temporary, occurring only during the current iteration of training. This random and iterative dropout process ensures that different neurons are dropped at different times, based on their adaptive dropout rates, which vary with the neuron's importance and training progress.

5) Tracking Dropped Neurons

To monitor the dropout dynamics, the number of neurons dropped in each layer is tracked and logged during every iteration. This tracking allows for a detailed assessment of how the dropout strategy is affecting the network and provides insights into the effectiveness of the adaptive dropout technique.

IV. Training controller with adaptive dropout technique

The flowchart in Fig. 3 illustrates the training process of a neural network (NN) controller using the Forward Accumulation Through Time (FATT) algorithm combined with the Levenberg-Marquardt (LM) optimization technique and adaptive dropout within a parallel computing framework. Unlike traditional dropout, where a fixed dropout rate is applied uniformly across all neurons, adaptive dropout adjusts the dropout rate dynamically based on each neuron's importance. The neuron importance is calculated using the variance of neuron activations, which reflects how crucial each neuron is to the learning process at different stages of training. During each forward pass in the training loop, the adaptive dropout is applied to the neurons in the hidden layers. This means that less important neurons are more likely to be deactivated, while more important neurons are preserved, allowing the model to focus on the most significant features during learning.

The training process starts with initializing the training parameters which are crucial for regulating the learning rate, defining stopping criteria, and ensuring convergence. The MPI framework is initialized with the necessary parameters, which prepare the parallel computing environment by determining the number of available CPU cores and assigning ranks to different workers. Weights $\overrightarrow{w}$ are initialized with small random values to begin the optimization.

The adaptive dropout is applied in a parallel computing environment where multiple trajectories are processed simultaneously. Each worker node represented as CPU cores, computes the DP cost and Jacobian matrices for specific groups of trajectories. FATT is applied in each worker node, leveraging the Armadillo library, which performs the linear algebra operations using OpenBLAS and LAPACK. The master node aggregates the results of the DP cost and Jacobian matrices from each worker node after each computation round.

This parallelization helps speed up the computation of the DP cost and the Jacobian matrix while maintaining the benefits of the adaptive dropout mechanism as in algorithm 1. After aggregating the results, the adaptive dropout algorithm computes the importance factor for each neuron, using eq (8). This factor helps adjust the dropout rate dynamically using the equation (9). Neurons with higher importance factors are preserved, while those with lower values have higher dropout rates.

Once the importance factor is computed, the algorithm proceeds with parallel computing for each CPU core (Worker 1 to N). The DP cost for each group of trajectories is calculated, and the aggregated DP cost is updated. The training loop continues until a termination condition is met, based on the comparison of the cost values.

When the training loop is complete, the weights are updated, and the process moves to the next epoch, repeating the adaptive dropout and parallelization steps. Finally, the training process stops when the maximum number of epochs is reached or when the DP cost converges to a predefined threshold.

By dynamically adjusting the dropout rate, the adaptive dropout mechanism helps in better convergence of the training process, preventing overfitting, and ensuring that the model generalizes well to unseen data.

V. Results and Discussion

The training program was developed using C++17, utilizing the Armadillo C++ library for efficient linear algebra computations, supported by OpenBLAS and LAPACK. The OpenMPI implementation of the Message Passing Interface (MPI) was employed to distribute workloads across multiple nodes, ensuring scalability and efficiency in parallel processing. The experiments were conducted on an AMD Ryzen Threadripper Pro 5975WX system with a Linux operating system. For parallelization, the number of processes was set to 48 to effectively utilize the computational resources. The dataset used for testing consisted of synthetic control data based on established dynamic models of solar inverter systems, designed to emulate real-world power electronic systems. By dividing the training trajectories into groups and assigning each group to individual CPU cores or workers, the program achieves efficient parallelization. For 10-trajectory, 20-trajectory, or up to 100-trajectory, each core independently computes its subset of trajectories in parallel, allowing for faster processing and aggregation of results. This parallelization is crucial for reducing computational time when processing larger numbers of trajectories, as the workload is distributed across available processing units.

By incorporating adaptive dropout into the neural network architecture, the model's complexity is effectively reduced and prevented overfitting, leading to faster convergence and improved generalization. This approach significantly reduced runtime per trajectory, resulting in quicker training times and enhanced overall efficiency. The adaptive dropout rate is dynamically adjusted during training based on the importance of each neuron. Neurons with higher activation significance are retained, while less important neurons are regularly dropped, minimizing the risk of overfitting while optimizing computational resources. Figures 5, 6, and 7 present the average run time plots with adaptive dropout, weight dropout, and without dropout with parallelization is applied. Similarly, Figures 8, 9, and 10 present the speedup comparisons across different numbers of workers and trajectories with adaptive dropout, weight dropout and without dropout on GPU with parallelization applied. The results indicate better average runtimes and speedup performance when utilizing the adaptive dropout with parallel computing compared to both weight dropout with parallel computing and running without dropout with parallel computing. The results, summarized in Table I, provide a numerical comparison of the average running times and speedup achieved with adaptive dropout, weight dropout, and no dropout. As in the figures 14 and 15, adaptive dropout consistently provides lower average running times and higher speedups across all trajectory sizes when compared to both weight dropout and no dropout. This trend is consistent across different trajectory sizes, as shown in Figure 15, where adaptive dropout with parallelization consistently delivers the highest speedup compared to parallelization with no dropout, achieving a 6x speedup compared to 5x for weight dropout and 1x for no dropout at 100 trajectories.

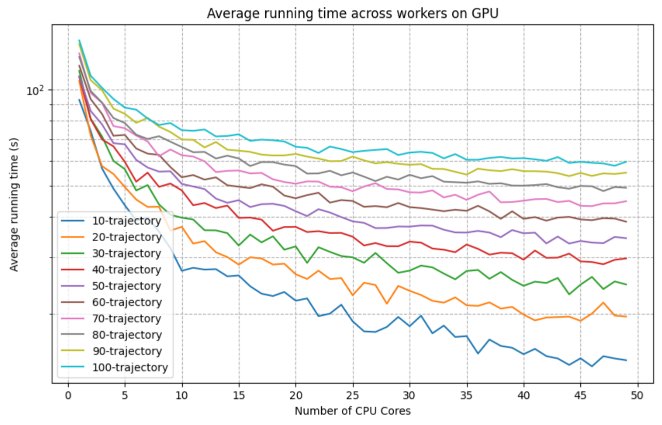


Fig. 5: Average running time across workers on GPU with adaptive dropout

The comparative analysis of weight dropout with adaptive dropout shows that adaptive dropout provides a precise but important advantage in performance, particularly noticeable in scenarios involving a higher number of trajectories and CPU cores. When comparing convergence plots without adaptive dropout from Fig. 11 with adaptive dropout Fig. 12, and with adaptive dropout combined with parallel computing Fig. 13, the combination of adaptive dropout and parallel computing results in significantly reduced oscillations.

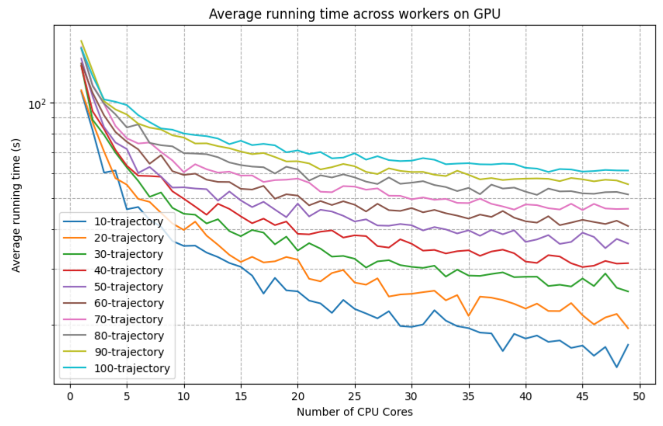


Fig. 6: Average running time across workers on GPU with weight dropout [18]

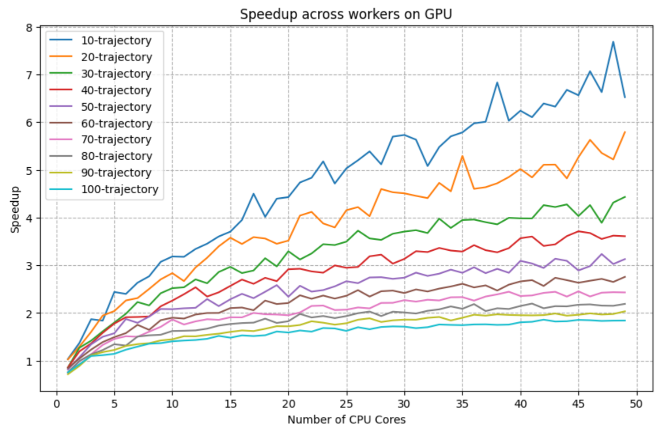


Fig. 9: Speedup across workers with weight dropout [18]

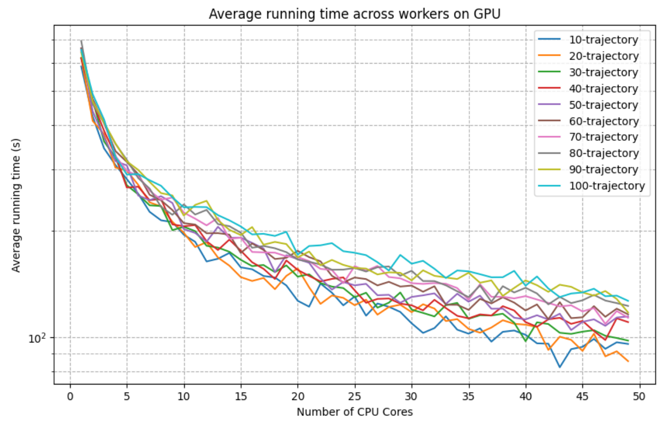


Fig. 7: Average running time across workers on GPU without adaptive dropout [18]

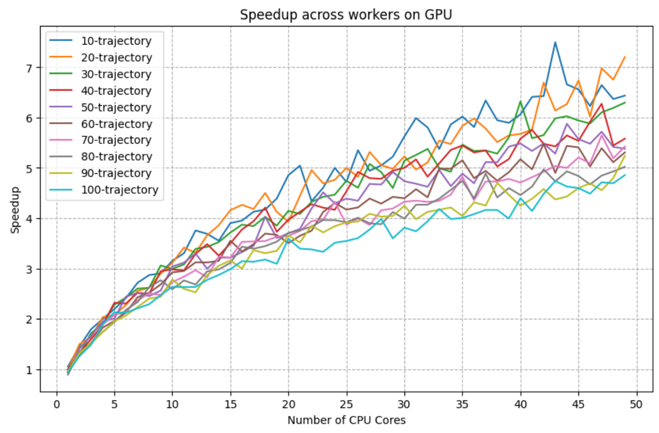


Fig. 10: Speedup across workers without adaptive dropout [18]

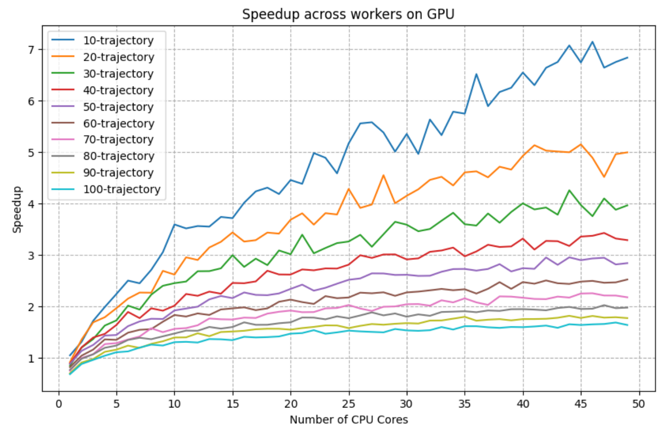


Fig. 8: Speedup across workers with adaptive dropout

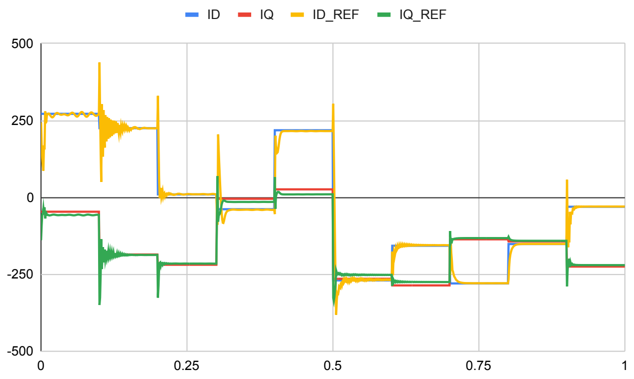


Fig. 11: Convergence without adaptive dropout

Performance comparisons demonstrate that the implementation of adaptive dropout not only reduces computational complexity but also provides a noticeable speedup in performance. The parallelized C++ version with adaptive dropout integration outperformed the version with weight dropout and no dropout, showcasing the efficiency and scalability of the proposed method in handling large numbers of trajectories with high sampling frequencies. The code we developed is available on https://github.com/Kushal-1234/Parallelization-with-adaptive-dropout.

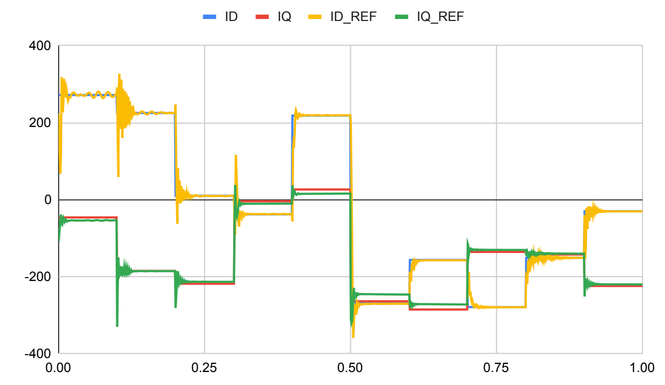


Fig. 12: Convergence with adaptive dropout

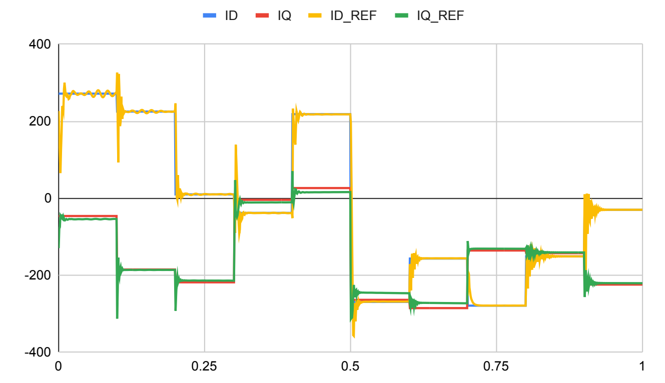


Fig. 13: convergence with adaptive dropout with parallel computing

TABLE I: Numerical Performance Index Results for Adaptive Dropout, Weight Dropout, and No Dropout on GPU with Parallelization

Metric	**Adaptive Dropout**	**Weight Dropout**	**No Dropout**
Avg. Running Time (10 Trajectories)	99.8s	112.7s	615.8s
Avg. Running Time (50 Trajectories)	100.0s	112.6s	624.2s
Avg. Running Time (100 Trajectories)	103.4s	113.0s	625.5s
Speedup (10 Trajectories)	6.16x	5.46x	1.00x
Speedup (50 Trajectories)	6.14x	5.48x	1.00x
Speedup (100 Trajectories)	6.02x	5.48x	1.00x
Overall Improvement (%)	83.0%	81.9%	-

VI. Conclusion

Neural network controllers for real-time closed-loop control systems, such as solar inverters, have demonstrated significant improvements in training efficiency and performance when adaptive dropout is integrated with parallel computing and supported by GPU acceleration. This research shows that adaptive dropout not only accelerates convergence when combined with parallel computing strategies but also enhances the neural network's generalization capacity by dynamically adjusting dropout rates based on the significance of individual neurons. These benefits are further amplified by GPU acceleration, which significantly reduces the time required to compute multiple trajectories simultaneously. Large-scale neural network models particularly benefit from GPU-based training due to its superior scalability and efficiency, as evidenced by the examination of speedup performance. Overall, this study provides a robust foundation for optimizing neural network training, paving the way for more efficient and scalable real-time control applications.

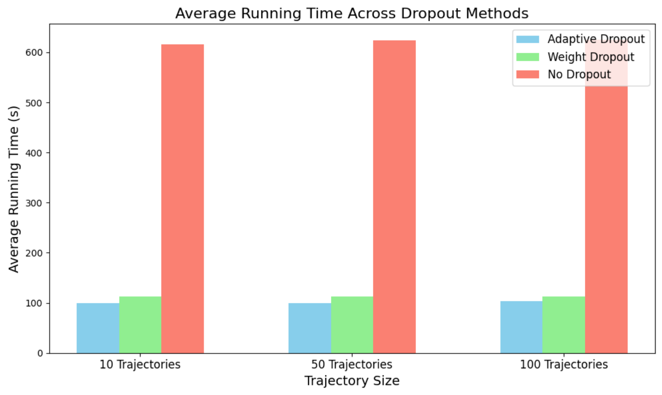


Fig. 14: Average Running Time Across Dropout Methods

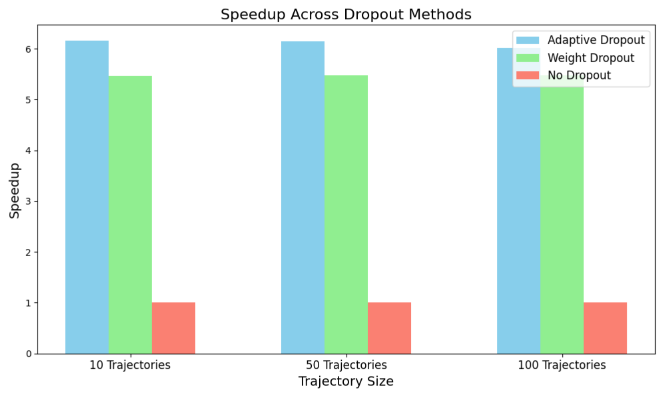


Fig. 15: Speedup Across Dropout Methods

Acknowledgment

This work is supported by the U.S. National Science Foundation under Award 2131175. We thank anonymous reviewers for their insightful comments and inputs.

References

[1] S. Li, T. Haskew, Y.-K. Hong, and L. Xu, “Direct-current vector control of three-phase grid-connected rectifier–inverter,” *Electric Power Systems Research*, vol. 81, pp. 357–366, 02 2011.

[2] L. Xu and Y. Wang, “Dynamic modeling and control of dfig-based wind turbines under unbalanced network conditions,”

IEEE Transactions on power systems, vol. 22, no. 1, pp. 314–323, 2007.

[3] N. F. Nanyan, M. A. Ahmad, and B. Hekimoğlu, "Optimal pid controller for the dc-dc buck converter using the improved sine cosine algorithm," *Results in Control and Optimization*, vol. 14, p. 100352, 2024.

[4] M. R. Mahmud and H. R. Pota, "Robust nonlinear controller design for dc–ac converter in grid-connected fuel cell system," *IEEE Journal of Emerging and Selected Topics in Industrial Electronics*, vol. 3, no. 2, pp. 342–351, 2022.

[5] F. Liu, X. Ruan, X. Huang, Y. Qiu, and Y. Jiang, "Control scheme for reducing second harmonic current in ac–dc–ac converter system," *IEEE Transactions on Power Electronics*, vol. 37, no. 3, pp. 2593–2605, 2022.

[6] L. Kong and H. Nian, "Transient modeling method for faulty dc microgrid considering control effect of dc/ac and dc/dc converters," *IEEE Access*, vol. 8, pp. 150 759–150 772, 2020.

[7] M. R. Ghazali, M. A. Ahmad, and R. M. T. R. Ismail, "Data-driven pid control for dc/dc buck-boost converter-inverter-dc motor based on safe experimentation dynamics," in *2018 IEEE Conference on Systems, Process and Control (ICSPC)*, 2018, pp. 89–93.

[8] C. Bailer-jones, D. Mackay, and P. Withers, "A recurrent neural network for modelling dynamical systems," *Network: Computation in Neural Systems*, vol. 9, 08 2002.

[9] S. Nukala, X. Yuan, K. Roy, and O. T. Odeyomi, "Face recognition for blurry images using deep learning," in *2024 4th International Conference on Computer Communication and Artificial Intelligence (CCAI)*, 2024, pp. 46–52.

[10] V. Pham, T. Bluche, C. Kermorvant, and J. Louradour, "Dropout improves recurrent neural networks for handwriting recognition," in *2014 14th international conference on frontiers in handwriting recognition*. IEEE, 2014, pp. 285–290.

[11] 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 *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2023, pp. 157–163.

[12] X. Fu, J. Sturtz, E. Alonso, R. Challoo, and L. Qingge, "Parallel trajectory training of recurrent neural network controllers with levenberg–marquardt and forward accumulation through time in closed-loop control systems," *IEEE Transactions on Sustainable Computing*, vol. 9, no. 2, pp. 222–229, 2024.

[13] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[15] M. Bhardwaj and S. Choudhury, "Digitally controlled solar micro inverter design using c2000 piccolo microcontroller user's guide," Technical report, Tech. Rep., 2017.

[16] J. Sturtz, K. K. D. Surendranath, M. Sam, X. Fu, C. D. Hingu, R. Challoo, and L. Qingge, "Accelerating the neural network controller embedded implementation on fpga with novel dropout techniques for a solar inverter," *Pervasive and Mobile Computing*, vol. 104, p. 101975, 2024.

[17] X. Fu, S. Li, D. C. Wunsch, and E. Alonso, "Local stability and convergence analysis of neural network controllers with error integral inputs," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[18] M. Sam, K. K. D. Surendranath, X. Fu, and L. Qingge, "Accelerating rnn controllers with parallel computing and weight dropout techniques," in *Advances and Trends in Artificial Intelligence. Theory and Applications*, H. Fujita, R. Cimler, A. Hernandez-Matamoros, and M. Ali, Eds. Singapore: Springer Nature Singapore, 2024, pp. 401–412.

[19] L. J. Ba and B. Frey, "Adaptive dropout for training deep neural networks," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. Curran Associates Inc., 2013, p. 3084–3092.