G-SpNN: GPU-Accelerated Passivity Enforcement for S-Parameter Modeling with Neural Networks

Lijie Zeng[1], Jiatai Sun[1], Xiao Wu[2], Dan Niu[3], Tianshi Wang,[4], Yibo Lin[5], Zuochang Ye[6], and Zhou Jin[7]
[1]SSSLab, China University of Petroleum-Beijing, China, [2]Huada Empyrean Software Co. Ltd, China,
[3]School of Automation, Southeast University, China, [4]HiSilicon Technologies Co. Ltd
[5]School of Integrated Circuits, Peking University, Beijing, [6]School of Integrated Circuits, Tsinghua University,
[7]College of Integrated Circuits, Zhejiang University, China
Email: z.jin@zju.edu.cn

***Abstract*—The increasing complexity of high-frequency circuits calls for efficient and accurate passive macromodeling techniques. Existing passivity enforcement methods, including those in commercial tools, often encounter convergence issues or compromise accuracy. The Domain-Alternated Optimization (DAO) framework seeks to restore accuracy through an additional optimization step but is hampered by high memory consumption and slow convergence, particularly for large-scale problems. This paper presents G-SpNN, a novel GPU-accelerated framework that recasts the passivity-enforced macromodeling problem as a neural network training task. This approach significantly enhances both the speed and scalability of passivity enforcement. Experimental results show that G-SpNN achieves an average speedup of 7.63× in convergence compared to DAO, while reducing memory usage by two orders of magnitude. This enables G-SpNN to handle complex, high-port-count circuits with greater accuracy and efficiency, paving the way for robust high-frequency circuit simulations.**

***Index Terms*—S-Parameter Macromodeling, Passivity Enforcement, Neural Network, GPU Acceleration**

I. INTRODUCTION

S-parameters, and frequency-domain parameters in general, pose significant challenges in circuit simulation. Running time-domain analyses with them, which typically involves frequency-domain interpolation and extrapolation, causality enforcement, impulse response computation and the convolution analysis, is computationally intensive and often leads to convergence issues. Macromodeling, also known as broadband SPICE (bbspice) in the circuit design community, offers a preferred alternative [1], but its effectiveness hinges on robust passivity enforcement to guarantee accurate and stable high-frequency simulations.

Early attempts to address passivity enforcement through convex optimization faced limitations due to their high computational complexity, scaling with $O(n^6)$ [1], where n represents the number of poles. This restricted their applicability to small-scale problems ($n < 100$) [1]. Mainstream methods typically adopt a two-step approach: first generating a macromodel without considering passivity constraints, often using techniques like Vector Fitting (VF) [2], and then applying specialized algorithms to restore passivity, such as the Eigenvalue Perturbation (EPM) [3], Residue Perturbation (RPM) [4], and Local Compensation (LC) [5]. However, these methods frequently compromise model accuracy, significantly increase computational costs, and are not robust in their performances, making them challenging to implement in practice.

Despite these challenges, almost all commercial solutions rely on these two-step methods. In practice, designers routinely observe commercial broadband SPICE fitting tools struggling to fit S-parameter data from Touchstone files, particularly for the simulations of radio-frequency integrated circuits (RFICs) and chip packaging. The iterative nature of these methods often introduces cumulative fitting errors, leading to reduced accuracy. To mitigate this, commercial tools often relax passivity constraints, either by internally setting a tolerance for violations or enforcing passivity only within limited bandwidths. While this approach may improve fitting success rates, it can hinder simulation convergence later in the design process.

The Domain-Alternated Optimization (DAO) framework [6] seeks to enhance accuracy by incorporating a third optimization step after the conventional two-step approach. By transforming the original modeling problem constrained by passivity criteria into an unconstrained optimization problem, DAO enables the use of unconstrained optimizers to improve macromodel accuracy while maintaining passivity throughout the process. However, DAO still suffers from high memory consumption and slow convergence, particularly when dealing with large-scale problems, limiting its practical adoption. For example, using the DAO method on a 64-port system resulted in an average memory usage of 22GB per iteration, while a 138-port system exceeded 31.8GB.

Therefore, passivity enforcement is arguably still an unsolved problem, especially for large-scale systems with many ports. Today, the increasing prevalence of complex integrated circuits, with a rising number of ports driven by trends like chiplets and multi-core architectures, further exacerbates the challenges of passivity enforcement [7]. This surge in complexity renders existing methods inadequate, necessitating new solutions capable of efficiently and accurately enforcing passivity in large-scale systems.

In the meanwhile, the development of neural networks, combined with the emergence of large-scale optimizers and optimization frameworks for training these networks, presents new possibilities for addressing the challenge of passivity enforcement [8]. These frameworks, which can be accelerated using GPUs, provide a missing piece in the three-step DAO

framework. By drawing an analogy between passivity enforcement and neural network training, specifically the process of finding optimal weights [9] [10], it becomes possible to handle large-scale problems more effectively.

This work proposes G-SpNN, a novel GPU-accelerated framework that recasts passivity-enforced macromodeling as a neural network training problem. This framework, built upon the deep learning toolkit PyTorch, maps the constrained optimization problem of passivity enforcement onto an unconstrained optimization objective within the neural network training process. By leveraging PyTorch's automatic differentiation mechanism, G-SpNN eliminates the need for manual gradient derivation, simplifying the implementation and improving efficiency. Moreover, G-SpNN employs vectorized representations and QR decomposition to streamline error function calculations and utilizes the LBFGS method to approximate the Hessian inverse matrix, thereby accelerating convergence.

G-SpNN's key contributions include:

1) Casting the passive macromodeling problem to neural network training, thus leveraging GPU acceleration.
2) Using the LBFGS method to efficiently approximate the Hessian inverse matrix, proven to work the best in practice.
3) Keeping the memory usage almost constant with an increasing number of ports. Experimental results show that G-SpNN not only converges more stably and quickly than DAO, with a average speedup of 7.63×, its memory usage can be reduced by two orders of magnitude in test cases.

From above, G-SpNN offers a robust passivity-enforced macromodeling framework, presented in more detail in the following sections.

II. PRELIMINARIES

A. Macromodel Generation of S-Parameters

S-parameters describe signal scattering in the frequency domain. Macromodeling fits a rational function to frequency data s_k (frequency points) and $H(s_k) \in \mathbb{C}^{m\times m}$ (system response):

$$f(s_k) = \sum_{n=1}^{N_q} \frac{c_n}{s_k - a_n} + d + s_k h \tag{1}$$

where c_n, a_n, d, h are fitting coefficients.

To improve stability and efficiency, Eq. (1) is often expressed in a state-space form [11]:

$$G(s_k) = C(s_k \cdot I - A)^{-1}B + D \tag{2}$$

where $A \in \mathbb{C}^{n\times n}, B \in \mathbb{C}^{n\times m}, C \in \mathbb{C}^{m\times n}, D \in \mathbb{C}^{m\times m}$, and I as the identity matrix.

For this system, $G(s_k)$ needs to satisfy two conditions [12]:

1) The error needs to be minimized.

$$Error = \min\left(\sum_k |G(s_k) - H(s_k)|\right) \tag{3}$$

2) It needs to satisfy the passivity requirement [13].

$$G(s_k) + G^H(s_k) \geq 0 \tag{4}$$

where G^H represents the conjugate transpose of G.

B. Three-Step Modeling Framework

The first step in passivity macromodeling is to fit Eq. (1) using methods such as VF [2] or Rational Fitting [14], and then transform it into the state-space representation as shown in Eq. (5), thereby obtaining the initial matrix.

$$G_0(s_k) = C_0(s_k \cdot I - A_0)^{-1}B_0 + D_0 \tag{5}$$

Although this step can easily provide a locally optimal fit with minimal error, the resulting matrix may not necessarily satisfy the required passivity [15].

The second step involves passivity fixing on the initial $G_0(s_k)$ to construct $G_1(s_k)$ by using EPM, RPM and LC, such that $G_1(s_k)$ satisfies the passivity [16].

The third step begins with $G_1(s_k)$ and solves the optimization model shown in Eq. (6) to find a system $G_2(s_k)$ that minimizes the error [17].

$$\begin{cases} \min\limits_{C,D} \sum\limits_k |G_2(s_k) - H(s_k)| \\ \text{subject to: } G_2(s_k) \text{ is passive} \end{cases} \tag{6}$$

C. Domain-Alternated Optimization

Actually, the third step of the Three-Step Modeling Framework often faces challenges in enforcing passivity constraints. Robust implementation of Eq. (6) is impractical [18] [19]. To address this problem, the DAO framework simplifies the optimization process by introducing a system $W(s_k)$ as optimization variables, thereby removing passivity constraints and converting the problem into an unconstrained form [6].

Algorithm 1 DAO Algorithm

1: **Step 1 :** Perform SPF on the initial passive system $G_1(s_k)$ to obtain $W(s_k)$.
2: **Step 2 :** Use the matrices (L, Q) of $W(s_k)$ as parameters and perform unconstrained optimization to minimize error.
3: **Step 3 :** Perform PFE to the optimized $W(s_k)$ to obtain the final system $G(s_k)$.

The Spectral Factorization (SPF) operation in Algorithm 1 is given by Eq. (7)-(10). Its essence lies in computing the matrices L and Q in order to derive $W(s_k)$ as shown in Eq. (11) [20].

$$R = Q^T Q = D + D^T \tag{7}$$

$$\tilde{A} = BR^{-1}C - A, \quad \tilde{B} = BQ^{-1}, \quad \tilde{C} = C^T R^{-1} C \tag{8}$$

$$K\tilde{A} + \tilde{A}^T K - K\tilde{B}\tilde{B}^T K - \tilde{C} = 0 \tag{9}$$

$$L = Q^{-T}C - Q^{-T}B^T K \tag{10}$$

$$W(s_k) = L(s_k \cdot I - A)^{-1}B + Q \tag{11}$$

The Partial Fractional Expansion (PFE) operation, defined by Eq. (12)-(14), focuses on computing the matrices C and D to transform $W(s_k)$ into $G(s_k)$ [21] [22] [23].

$$M = \text{kron}(A^T, I_n) + \text{kron}(I_n, A^T) \tag{12}$$

$$\text{vec}(K) = -M^{-1}\text{vec}(L^T L) \tag{13}$$

$$C = B^T K + Q^T L, \quad D = \frac{1}{2}Q^T Q \tag{14}$$

where kron$(\cdot)$ represents the Kronecker product, and vec(K) represents the vectorized form of matrix K [24].

Based on the principles and theorems related to SPF and PFE [21], the system $G(s_k)$ derived from $W(s_k)$ is guaranteed to maintain passivity. Consequently, the original optimization problem in Eq. (6) can be reformulated as an unconstrained optimization problem in Eq. (15).

$$\begin{cases} \min\limits_{L,Q} \sum\limits_{k} |G(s_k) - H(s_k)| \\ \text{subject to:} \\ W(s_k) = L(s_k \cdot I - A)^{-1} + Q \\ G(s_k) = pfe(W(s_k)) \end{cases} \tag{15}$$

III. G-SpNN

For the optimization problem defined in Eq. (15), it is feasible to directly compute the gradient vector and Hessian matrix [25]. However, compared to Eq. (6), the larger computational scale results in increased time costs and memory consumption, brings significant challenges. To this end, we propose G-SpNN, which achieves an optimal passive macromodel with faster convergence and lower memory consumption. More importantly, it exhibits excellent scalability, capable of handling ultra-large-scale passive macromodeling problems.

The overall workflow of G-SpNN is shown in Figure 1. Starting with a given passive system $G_1(s_k)$, an unconstrained system $W(s_k)$ is first derived by SPF transformation and represented as a layer in the neural network. Next, the PFE transformation is applied to generate a new network layer $G(s_k)$, corresponding to a passive system. This reformulates the problem as a neural network training task. During forward propagation, the system, together with the tabulated data $H(s_k)$, is used to compute the loss value. For efficient training, the LBFGS method is futher incorporated with backpropagation to compute gradients and update the network parameters. Once the iteration stopping criteria are satisfied, the optimized passive system is obtained. Details are in Algorithm 2.

Algorithm 2 Proposed G-SpNN Algorithm

1: **Preprocess:** Calculate R_F, b, δ, M^{-1} from (20), (22) and (12)
2: **SPF Transformation:** Apply SPF to matrices A, B, C, D from (2) to obtain A, B, L, Q
3: **Initialize:** Set optimizer with parameters L, Q, define number of epochs
4: **for** i in {1, 2, ..., epoch} **do**
5: Apply PFE to transform L, Q into C, D
6: Vectorize C, D into y, and L, Q into x
7: Compute loss: $f(x)$ from (21)
8: Backpropagate and using chain rule: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x}$
9: Approximate the Hessian inverse via k LBFGS iterations
10: Update L, Q
11: **end for**
12: **Final Transformation:** Convert L, Q to final C, D using PFE to obtain final $G(s_k)$

A. Analogy to Neural Network Training

Both solving Eq. (15) and training a neural network represent fundamentally nonlinear optimization challenges. In this context, the error function in Eq. (15) can be viewed as analogous to the prediction error encountered in neural network training, highlighting a direct connection between the two optimization frameworks. This analogy not only emphasizes the shared nature of the problems but also suggests that leveraging neural network techniques can facilitate faster optimization. As illustrated in Figure 2, the comparison between the two problem-solving approaches is visually clear. In neural network training, each data instance consists of an input vector x_i and a corresponding label y_i, and the predicted label $\phi(x_i, w)$ is obtained through the neural network. The training task is to minimize an objective function, which primarily consists of prediction errors.

By drawing an analogy between unconstrained optimization, as expressed in Eq. (15), and neural network training, we represent parameters L, Q in simplified form as w, and treat parameters s_k and $H(s_k)$ (derived from the simulation or measurement process) as data instances. The neural network processes these data instances and computes $g(s_k, w)$ ($g(\cdot, w)$ represents $G(s)$ in Eq. (15), where A and B are constants), after which the corresponding objective function is calculated to iteratively update the network, ultimately yielding the optimal solution to Eq. (15). Through this construction, the unconstrained optimization problem can be seamlessly transformed into the neural network training process, making it possible to leverage advanced deep learning techniques to address the issues in the DAO framework.

It is important to note that, as shown in Figure 1, during the complete equivalent neural network training process, the transformations between L, Q and C, D through PFE and SPF do not affect the gradient computation. The gradient still follows the chain rule and is automatically handled by PyTorch, a feature that the DAO framework lacks.

B. Loss Function

During the forward propagation phase of neural network training, the need to compute prediction errors N times, with each error computed through matrix norm calculations, introduces significant complexity throughout the training process. To address this issue, this section will reformulate the objective function from Eq. (15) to make the neural network training process more computationally efficient.

For the objective function in Eq. (15):

$$Error = \min_{C,D} \sum_{k} |G(s_k) - H(s_k)| \tag{16}$$

We can vectorize it as follows:

$$Error\_vec = |Fy - h| \tag{17}$$

where:

$$y = \begin{bmatrix} \text{vec}(C) \\ \text{vec}(D) \end{bmatrix} \qquad \text{F}_i = \begin{bmatrix} \text{kron}([s_i I - A]^{-1}, I_n) & I_m^2 \end{bmatrix} \tag{18}$$

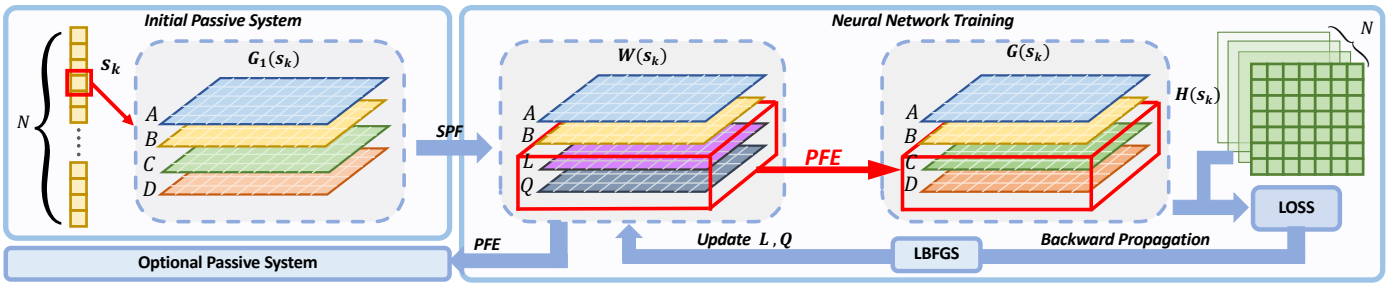

Fig. 1: The overall framework of G-SpNN.

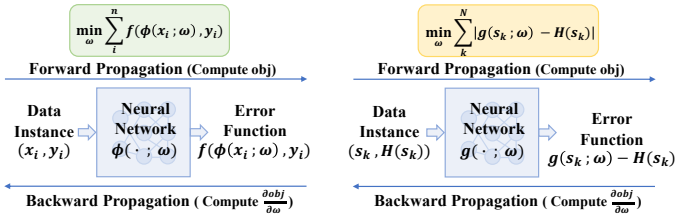

Fig. 2: Analogy between unconstrained optimization and neural network training. (a) Train a network for weights $\mathbf{w}$. (b) Solve a set of parameters for macromodel with $\mathbf{w} = (\mathbf{L}, \mathbf{Q})$.

To further simplify the computation of the objective function, a more detailed decomposition of Eq. (17) can be performed. The specific decomposition process is as follows. Firstly, QR factorization can be applied to F in Eq. (19):

$$h = \begin{bmatrix} \mathrm{vec}(\mathrm{Re}(H(s_1))) \\ \vdots \\ \mathrm{vec}(\mathrm{Re}(H(s_N))) \\ \mathrm{vec}(\mathrm{Im}(H(s_1))) \\ \vdots \\ \mathrm{vec}(\mathrm{Im}(H(s_N))) \end{bmatrix} \quad F = \begin{bmatrix} \mathrm{Re}(F_1) \\ \vdots \\ \mathrm{Re}(F_N) \\ \mathrm{Im}(F_1) \\ \vdots \\ \mathrm{Im}(F_N) \end{bmatrix} \tag{19}$$

$$F = Q_F R_F \tag{20}$$

Subsequently, Eq. (17) can be further transformed into the following form Eq. (21), which also represents the final definition of the loss function:

$$loss = Error\_vec = (R_F y - b)^T (R_F y - b) + \delta^2 \tag{21}$$

where:

$$b = Q_F^T h, \quad \delta^2 = h^T h - b^T b \tag{22}$$

By decomposing F into Q_F and R_F, the quadratic term $(R_F y - b)^T (R_F y - b)$ is simplified to a standard least-squares problem, which is computationally more efficient. This simplification reduces the need for repeated computation of matrix norms, which would otherwise be required for every iteration in the optimization process. Furthermore, the decomposition helps in minimizing the number of operations required to update gradients and compute the loss, thereby enhancing overall computational efficiency.

C. Further Optimization for Memory and Time Consumption

Based on the new definition of the loss function, optimization problem Eq. (15) can be further refined as follows:

$$\begin{cases} \text{variable: } L, Q \\ \text{min: } f(y) = (R_F y - b)^T (R_F y - b) + \delta^2 \\ \text{subject to:} \\ x = \begin{bmatrix} \mathrm{vec}(L) \\ \mathrm{vec}(Q) \end{bmatrix} \\ y = pfe(x) \end{cases} \tag{23}$$

For nonlinear optimization problems like Eq.(23), traditional methods such as DAO typically compute the gradient vector and Hessian matrix to determine the search direction and step size, thereby accelerating convergence. However, the full calculation of the Hessian matrix imposes significant memory and time overhead [26], especially for high-dimensional optimization problems. Therefore, we employ the LBFGS method [27] to approximate the inverse of the Hessian matrix, which reduces memory usage while still accurately selecting the gradient descent direction, thus accelerating neural network convergence.

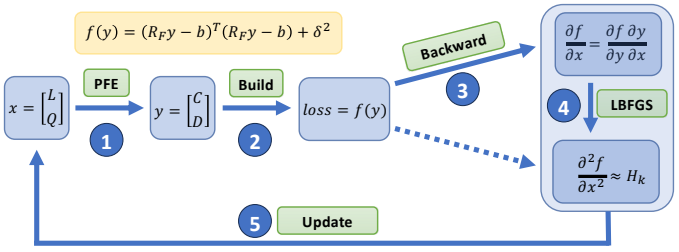

Fig. 3: Computational graph with LBFGS Method.

Figure 3 illustrates the computational graph. The input parameters are derived from Eq.(11) (the unconstrained system $W(s)$ in Figure 1), which include the parameter matrices L and Q. These matrices are vectorized into x. The computational graph includes five steps. ***Step 1***, x undergoes the PFE operation to generate parameters y (the passive system $G(s)$). ***Step 2***, using y along with other matrices from Eq. (20) and Eq. (22), the loss function is constructed. ***Step 3***, automatic differentiation is employed to compute the gradient information, applying the chain rule to perform backpropagation and calculate the first-order derivative of the loss function with respect to the network weights x. ***Step 4***, the LBFGS method is used to approximate the inverse Hessian matrix. ***Step 5***, the previously calculated

first-order derivative information and the approximated inverse Hessian matrix are used to update the parameters L and Q of the initial passive system. Above all, this completes one optimization iteration.

In the approximation process of ***Step 4***, it is important to note that the LBFGS algorithm does not store the full Hessian matrix directly [28]. Instead, it maintains a limited history of recent iterations to perform the calculations. The update formula for the Hessian inverse approximation in LBFGS is given by:

$$H_{k+1} = H_k - \frac{H_k \Delta g_k \Delta g_k^T H_k}{\Delta g_k^T \Delta x_k} + \frac{\Delta x_k \Delta x_k^T}{\Delta x_k^T \Delta g_k} \tag{24}$$

where:

1) H_k is the Hessian inverse approximation at iteration k.
2) $\Delta g_k = (\frac{\partial f}{\partial x})_{k+1} - (\frac{\partial f}{\partial x})_k$ is the change in the gradient.
3) $\Delta x_k = x_{k+1} - x_k$ is the change in the parameter.

The LBFGS is well-suited for this optimization problem due to its efficiency in managing large-scale variables and its ability to converge rapidly to a solution with a limited memory overhead. As experiments have shown, this optimization method, which combines efficient memory management and rapid convergence speed, provides robust support for optimizing complex circuit models.

IV. Experiments

A. Experimental Setup

We implement and test the proposed G-SpNN on an i7-14700KF @5.6GHz CPU with 32GB of memory, and a GeForce RTX 4070 SUPER GPU with 12GB of VRAM. G-SpNN is implemented based on PyTorch and compared against the state-of-the-art (SOTA) framework DAO, which is implemented in MATLAB and is open-sourced on GitHub[1].

For the DAO, we utilize MATLAB's built-in method $user.MemUsedMATLAB$ to record the memory consumption. For the G-SpNN, we monitor the GPU memory usage by calling PyTorch's $torch.cuda.memory\_allocated()$.

The input files used in the experiments are standard Touchstone format files, derived from actual real-world simulations. After reading the files, the parameters s_k and $H^{m\times m}(s_k)$ $(1 \le k \le N)$ can be obtained. Before conducting the comparative experiments, the first two steps of macromodeling (as outlined in the preliminaries section) need to be performed on the raw data. Specifically, this involves applying the VF and LC method to process the data and generate four matrices: $A^{n\times n}$, $B^{n\times m}$, $C^{m\times n}$, and $D^{m\times m}$. The detailed information about these matrices and the results of the VF and LC method can be found in Tables I and II.

In this experiment, steady-state error (SS Error) is used as the basis for comparison. According to the data in Table II, the VF method initially produces a result with a very low steady-state error for the test samples. However, this result does not satisfy the passivity requirement. After applying the LC method for passivity correction, the passivity condition is met, but the steady-state error increases to some extent.

[1]https://www.github.com/yezuochang/pmm

TABLE I: Touchstone files.

Num	Case	Ports	n	m	N
1	Telluride	11	56	11	258
2	test_5	30	199	30	2000
3	sp125_uniform_2	64	342	64	400
4	CKDIST_TUNEDBUF	64	338	64	2000
5	pll_testcase	138	727	138	300

TABLE II: Fitting results of VF and LC.

Num	VF			LC		
	SS Error	Time(s)	Passivity	SS Error	Time(s)	Passivity
1	1.52e-4	0.8906	non-passive	5.91e-1	0.1875	passive
2	1.72e-7	17.78	non-passive	6.56e-5	7.64	passive
3	6.29e-4	25.59	non-passive	6.56e-4	6.906	passive
4	4.27e-3	96.89	non-passive	5.16e-3	17.21	passive
5	5.49e-4	75.23	non-passive	5.70e-4	91.79	passive

B. Passivity and Error

Table III presents the passivity results and steady-state errors for the G-SpNN and DAO methods. It is important to note that the steady-state error of the VF method represents the theoretical lower bound of the errors for both the G-SpNN and DAO methods. From the results, it is clear that both methods maintain passivity and further reduce errors compared to the LC method, with the G-SpNN error being closer to the VF error. Figure 4 shows the fitting results and errors for two points (H_{11} and H_{14}) in the H matrix of Case 1. It can be seen that G-SpNN provides a better overall fit to the original data, with errors lower than both the VF and LC method and the DAO method.

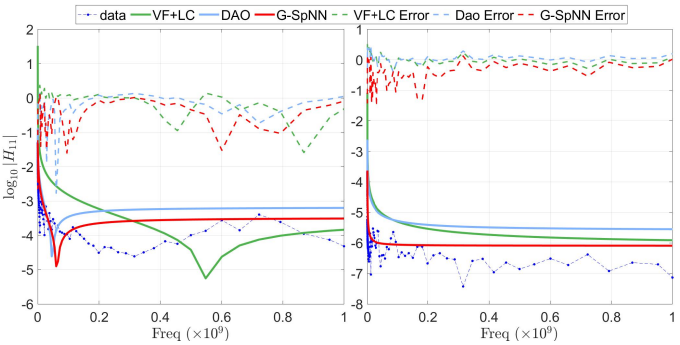

Fig. 4: Fitting accuracy of LC, DAO and G-SpNN.

C. Convergence Speed and Memory Usage

In Table III, we present the runtime, iteration count, initial loss, and final loss for both the G-SpNN and DAO methods. The convergence criterion is defined as a change in the loss function $\Delta\text{Loss} < 10^{-3}$. It is important to note that for Case 1, although the runtime of DAO appears shorter, the comparison of the final loss and steady-state error shows that DAO actually experiences pseudo-convergence and does not reach the optimal solution. In contrast, G-SpNN demonstrates better convergence performance. For Case 3 and Case 4, it should be noted that the DAO method is forcibly terminated during the iteration process due to memory overflow and does not reach the predefined convergence criterion. For Case 5, DAO experiences a memory overflow during the first iteration and could not complete the iteration.

TABLE III: Comparison of G-SpNN and DAO. The "–" indicates memory overrun during execution.

Num	Initial Loss	DAO [6]					G-SpNN				
		Time (s)	#Iteration	Final Loss	SS Error	Passivity	Time (s)	#Iteration	Final Loss	SS Error	Passivity
1	6.17e12	17.47	93	4.8656	2.26e-3	passive	94.19	232	4.19e-2	2.46e-4	passive
2	6.50e8	104.65	7	3.86e-2	2.65e-5	passive	97.93	328	3.89e-2	2.65e-5	passive
3	17.36	2116.48	4	17.21	6.51e-4	passive	300.46	803	17.15	6.47e-4	passive
4	2.31e3	3923.14	9	2.24e3	4.97e-3	passive	176.56	469	2.14e3	4.69e-3	passive
5	78.32	–	–	–	–	–	403.35	456	77.62	5.58e-4	passive
Average		1540.44	28	565.53	1.98e-3		214.49	458	446.97	1.23e-3	

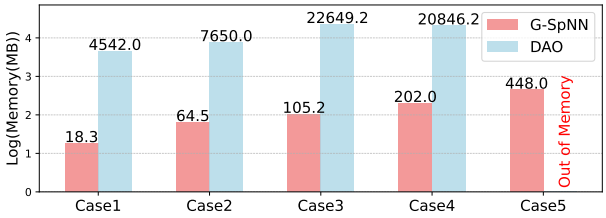

Fig. 5: G-SpNN vs. DAO in memory consumption.

For the tested cases, G-SpNN achieves an average speedup of 7.63× compared to DAO, and this speedup will increase even further with larger test cases. Therefore, G-SpNN is highly suitable for stable iterative convergence on large-scale cases.

We record the memory usage at each iteration to calculate the average memory consumption, as shown in Figure 5. Due to the high time and space complexity of the DAO method, we limit the number of poles in the VF method for cases 3-5 to ensure computational feasibility (which leads to higher SS Error and limits the reduction in loss). Nevertheless, it is still evident that as the test scale increases, DAO's memory usage rises significantly, while G-SpNN's memory usage remains relatively stable. The test results show that DAO's average memory consumption is 171.3 times that of G-SpNN.

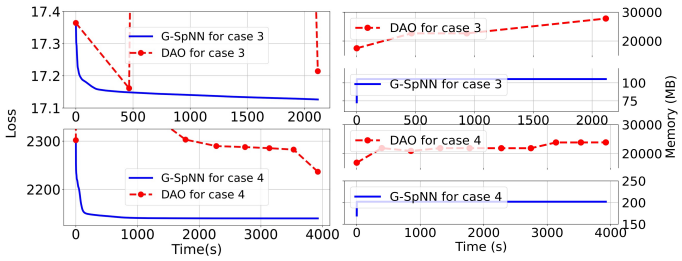

Fig. 6: G-SpNN vs. DAO in loss and memory over time (Case 3 and Case 4).

D. Detailed Analysis

To further compare the convergence efficiency of DAO and G-SpNN, we use Case 3 and Case 4 as examples, taking DAO's runtime for these cases as the benchmark. Figure 6 shows the loss and memory usage variations during iterations. It can be observed that G-SpNN has a smoother convergence process with better performance, achieving lower loss values per unit time compared to DAO. In contrast, DAO experiences gradient explosion, causing poor convergence and high memory usage, with significant fluctuations leading to memory explosions and premature termination. G-SpNN maintains significantly stable and lower memory usage throughout.

E. Comparison with Adam

For neural network training, optimization algorithms can leverage either first-order or higher-order gradient information. A commonly used first-order method is Adam, which is less sensitive to learning rate selection and performs well in many scenarios. We conduct tests comparing G-SpNN + Adam and G-SpNN + LBFGS on case 3 (using the same learning rate and configurations, only substituting the gradient computation method), with G-SpNN running for 80 seconds.

As shown in Figure 7, the Adam optimizer not only fails to reach lower loss values but also demonstrates instability, making its convergence behavior unpredictable. In contrast, the LBFGS method enables G-SpNN to progress steadily toward convergence, owing to second-order information guiding more effective update directions.

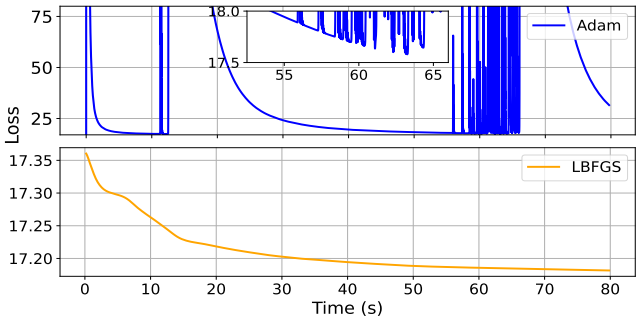

Fig. 7: Loss over time for G-SpNN with Adam and with LBFGS (Case 3).

V. Conclusion

In this paper, we introduce a novel perspective by mapping the passivity-enforced macromodeling problem into a neural network training task. The proposed optimization framework, G-SpNN, leverages advanced deep learning techniques as well as the LBFGS method. Compared to the current SOTA framework DAO, G-SpNN achieves significant improvements in both convergence speed and memory efficiency, which showcases exceptional capabilities to handle larger systems with a greater number of ports effectively.

Acknowledgment

The corresponding author is Zhou Jin. This work was supported by NSFC (Grant No.62234010, 92473107, 62204265, U23A20301, 62374031), Beijing National Research Center for Information Science and Technology (BNRist), Semiconductor Technology Innovation Center (Beijing) (Grant No.QYJS-2021-0900-B), and NSF of Jiangsu Province (BK20240173).

REFERENCES

[1] C. Coelho, J. Phillips, and L. Silveira, "A convex programming approach for generating guaranteed passive approximations to tabulated frequency-data," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 2, pp. 293–301, 2004.

[2] B. Gustavsen and A. Semlyen, "Rational approximation of frequency domain responses by vector fitting," IEEE Transactions on Power Delivery, vol. 14, no. 3, pp. 1052–1061, 1999.

[3] S. Grivet-Talocia, "An adaptive sampling technique for passivity characterization and enforcement of large interconnect macromodels," IEEE Transactions on Advanced Packaging, vol. 30, no. 2, pp. 226–237, 2007.

[4] B. Gustavsen, "Fast passivity enforcement for pole-residue models by perturbation of residue matrix eigenvalues," IEEE Transactions on Power Delivery, vol. 23, no. 4, pp. 2278–2285, 2008.

[5] T. Wang and Z. Ye, "Robust passive macro-model generation with local compensation," IEEE Transactions on Microwave Theory and Techniques, vol. 60, no. 8, pp. 2313–2328, 2012.

[6] Z. Ye, T. Wang, and Y. Li, "Domain-alternated optimization for passive macromodeling," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 10, pp. 2244–2255, 2015.

[7] T. Bradde, A. Zanco, and S. Grivet-Talocia, "Bivariate macromodeling with passivity constraints," Electrical Performance of Electronic Packaging and Systems. IEEE, pp. 1–3, 2021.

[8] Z. Jin, W. Li, Y. Bai, T. Wang, Y. Lu, and W. Liu, "Machine learning and gpu accelerated sparse linear solvers for transistor-level circuit simulation: A perspective survey," in 2024 29th Asia and South Pacific Design Automation Conference. IEEE, pp. 96–101, 2024.

[9] Y. Jiang, J. Song, X. Yin, X. Dong, S. Sun, Y. Lin, Z. Jin, X. Yang, and C. Zhuo, "A parallel simulation framework incorporating machine learning-based hotspot detection for accelerated power grid analysis," in Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD, pp. 1–7, 2024.

[10] J. Sun, X. Zha, C. Wang, X. Wu, D. Niu, W. Xing, and Z. Jin, "Pseudo adjoint optimization: Harnessing the solution curve for spice acceleration," in International Conference on Computer Aided Design, 2024.

[11] D. Deschrijver, M. Mrozowski, T. Dhaene, and D. De Zutter, "Macromodeling of multiport systems using a fast implementation of the vector fitting method," IEEE Microwave and Wireless Components Letters, vol. 18, no. 6, pp. 383–385, 2008.

[12] B. Gustavsen, "Passivity enforcement by residue perturbation via constrained non-negative least squares," IEEE Transactions on Power Delivery, vol. 36, no. 5, pp. 2758–2767, 2020.

[13] S. Grivet-Talocia, "Passivity enforcement via perturbation of hamiltonian matrices," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 51, no. 9, pp. 1755–1769, 2004.

[14] C. Coelho, J. Phillips, and L. Silveira, "Robust rational function approximation algorithm for model generation," in Proceedings 1999 Design Automation Conference, pp. 207–212, 1999.

[15] J. Rodriguez, "A loewner/mpm—vf combined rational fitting approach," in 2020 IEEE Power & Energy Society General Meeting. IEEE, pp. 1–1, 2020.

[16] A. Carlucci, T. Bradde, and S. Grivet-Talocia, "Improving robustness to termination conditions in passivity enforcement of rational macromodels," IEEE Transactions on Components, Packaging and Manufacturing Technology, pp. 1–1, 2024.

[17] L. F. Rodrigues, L. P. Ihlenfeld, and G. H. da Costa Oliveira, "A novel subspace identification approach with passivity enforcement," Automatica, vol. 132, pp. 109798, 2021.

[18] L. P. Ihlenfeld and G. H. Oliveira, "A faster passivity enforcement method via chordal sparsity," Electric Power Systems Research, vol. 204, pp. 107706, 2022.

[19] S. Grivet-Talocia, L. M. Silveira et al., "5 post-processing methods for passivity enforcement," Also of Interest, pp. 139, 2021.

[20] D. Youla, "On the factorization of rational matrices," IRE Transactions on Information Theory, vol. 7, no. 3, pp. 172–189, 1961.

[21] B. Anderson, K. Hitz, and N. Diem, "Recursive algorithm for spectral factorization," IEEE Transactions on Circuits and Systems, vol. 21, no. 6, pp. 742–750, 1974.

[22] B. Anderson, "A system theory criterion for positive real matrices," SIAM Journal on Control, vol. 5, no. 2, pp. 171–182, 1967.

[23] P. Lancaster, Algebraic Riccati Equations. Oxford Science Publications/The Clarendon Press, Oxford University Press, 1995.

[24] K. Filipiak, D. Klein, A. Markiewicz, and M. Mokrzycka, "Approximation with a kronecker product structure with one component as compound symmetry or autoregression via entropy loss function," Linear Algebra and its Applications, vol. 610, pp. 625–646, 2021.

[25] Z. Ye, Y. Li, M. Gao, and Z. Yu, "A novel framework for passive macromodeling," in Proceedings of the 48th Design Automation Conference, pp. 546–551, 2011.

[26] W.-K. Chen, Y. Zhang, B. Jiang, W.-H. Fang, and G. Cui, "Efficient construction of excited-state hessian matrices with machine learning accelerated multilayer energy-based fragment method," The Journal of Physical Chemistry A, vol. 124, no. 27, pp. 5684–5695, 2020.

[27] R. Bollapragada, J. Nocedal, D. Mudigere, H.-J. Shi, and P. T. P. Tang, "A progressive batching l-bfgs method for machine learning," in International Conference on Machine Learning, pp. 620–629, 2018.

[28] M. Borhani, "Multi-label log-loss function using l-bfgs for document categorization," Engineering Applications of Artificial Intelligence, vol. 91, pp. 103623, 2020.