Bol. Soc. Paran. Mat.
©SPM – E-ISSN-2175-1188
SPM: www.spm.uem.br/bspm

(3s.) **v. 2025 (43) 4** : 1–10.
ISSN-0037-8712
doi:10.5269/bspm.78591

A Feedforward Neural Network Approach to Solving Systems of Linear Equations

Rashad A. Al-Jawfi

ABSTRACT: This paper proposes a neural network-based framework for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$. The method reformulates the problem as a residual minimization task and employs a feedforward neural network to learn the mapping from input matrix-vector pairs to solution vectors. The network is trained using synthetic data and optimized via gradient descent using residual-based loss. Experimental results demonstrate that the model achieves high accuracy for well-conditioned systems with dimensions up to $n = 20$, producing residual errors below 10^{-4} in most cases. Comparative analysis against classical numerical solvers shows that while traditional methods remain superior for ill-conditioned systems, the neural approach offers notable advantages in inference speed, generalization, and suitability for parallel or real-time deployment. Limitations and future enhancements—including scalability, noise robustness, and hybridization—are also discussed.

Key Words: Feedforward neural networks, linear systems of equations, residual minimization, numerical linear algebra

Contents

2010 *Mathematics Subject Classification:* 65F10, 68T07.
Submitted August 23, 2025. Published November 01, 2025

1. Introduction

Systems of linear equations appear ubiquitously in scientific computing, engineering, optimization, and data analysis. Formally, a system of n linear equations in n variables can be represented as:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a coefficient matrix, $\mathbf{x} \in \mathbb{R}^n$ is the solution vector, and $\mathbf{b} \in \mathbb{R}^n$ is the right-hand side vector.

Traditional solution methods such as Gaussian elimination, LU decomposition, and iterative methods like Jacobi and Gauss-Seidel have been widely adopted due to their determinism and mathematical rigor [1,2]. However, these methods often face challenges in terms of scalability, robustness to ill-conditioned matrices, and adaptability to real-time or parallel environments [3].

Artificial neural networks (ANNs), inspired by biological neural systems, have demonstrated promise in addressing computational problems involving function approximation, regression, and classification [4, 5]. Their intrinsic parallelism and data-driven nature make them attractive candidates for solving algebraic systems, particularly when dealing with dynamic or high-dimensional settings [6].

This paper proposes a computational approach using feedforward neural networks to approximate solutions of systems of linear equations. The method formulates the problem as a residual minimization task and trains the network to map matrix-vector pairs $(\mathbf{A}, \mathbf{b})$ to their respective solution vectors $\mathbf{x}$. Emphasis is placed on convergence behavior, generalization, and performance comparison with traditional solvers. The remainder of this paper is organized as follows: Section 2 presents a literature review; Section 3 outlines mathematical preliminaries; Section 4 describes the neural network methodology; Section 5 details experimental design; and Section 6 discusses results and implications.

2. Literature Review

The application of artificial neural networks (ANNs) to the solution of systems of linear equations has gained significant momentum since the development of the backpropagation algorithm [6]. Early investigations established the theoretical capability of multilayer perceptrons (MLPs) to approximate linear mappings, laying the groundwork for supervised neural solvers.

One of the first practical models addressing linear systems through neural computation involved recurrent neural networks (RNNs). Gao and Wang [7] developed an RNN-based approach with provable convergence under mild assumptions. Later, Zhang and Wang [8] extended this work by demonstrating global exponential convergence through Lyapunov stability analysis. These recurrent schemes were particularly appealing for their dynamic equilibrium-seeking behavior.

Hopfield-type networks also contributed to the field by modeling the solution as the global minimum of a quadratic energy function. However, their practical applicability was often hindered by issues such as slow convergence and sensitivity to initial conditions [9].

With the advent of modern hardware accelerators and optimization techniques, attention has shifted back to feedforward networks. These models, while structurally simpler, benefit from rapid training and scalability. In particular, they have been used to learn the mapping between system inputs and outputs by minimizing the residual $\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2$, providing a data-driven alternative to direct matrix inversion or iterative updates.

Recent reviews, such as the one by Hussain et al. [9], have outlined both the strengths and limitations of ANN-based solvers. Notably, these methods offer strong potential in settings involving repeated computation, online adaptation, or integration into hardware systems.

In contrast to traditional solvers—such as LU decomposition and conjugate gradient methods—neural approaches offer generalization capabilities and parallelism at inference time. A conceptual comparison between the two paradigms is presented in Figure 2, emphasizing their algorithmic and computational differences.

Despite these advantages, current ANN-based solvers face challenges in stability, scalability, and sensitivity to matrix conditioning. These limitations have motivated hybrid architectures that combine neural networks with classical numerical techniques to leverage the strengths of both.

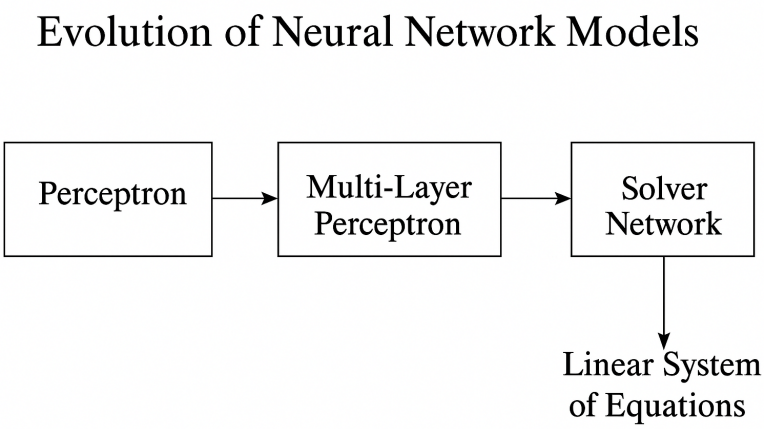

Figure 1: Evolution of neural network models for solving systems of linear equations: from Hopfield networks and RNNs to modern feedforward architectures.

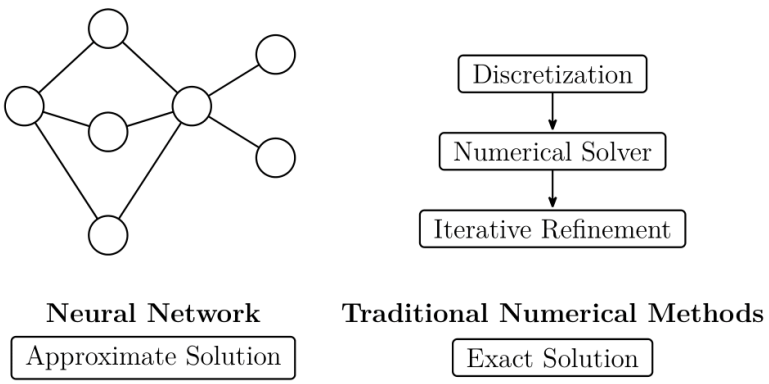

Figure 2: Comparison between classical numerical solvers and neural network-based methods in terms of flow, training, and execution.

3. Mathematical Background

Solving systems of linear equations is a foundational task in numerical linear algebra. A general system with n equations and n unknowns can be written compactly as:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a known coefficient matrix, $\mathbf{x} \in \mathbb{R}^n$ is the unknown solution vector, and $\mathbf{b} \in \mathbb{R}^n$ is the known right-hand side vector.

3.1. Direct and Iterative Solvers

Classical methods for solving such systems fall into two categories: direct methods and iterative methods. Direct methods, such as Gaussian elimination and LU decomposition, aim to compute the exact solution in a finite number of steps under exact arithmetic [10]. However, for large-scale systems, these methods can become computationally expensive or numerically unstable.

Iterative methods—such as Jacobi, Gauss-Seidel, and Conjugate Gradient (CG)—start from an initial guess and produce a sequence of approximations that converge to the true solution under suitable conditions [2]. These are often preferred for sparse or structured systems due to lower memory requirements and the potential for parallelization.

3.2. Conditioning and Stability

The numerical stability of any solver is closely tied to the conditioning of the matrix $\mathbf{A}$. The condition number, defined as:

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|, \tag{3.2}$$

quantifies the sensitivity of the solution $\mathbf{x}$ to perturbations in $\mathbf{b}$. A high condition number indicates an ill-conditioned system, where small input errors may lead to large output deviations [11].

3.3. Residual Formulation

Instead of computing the inverse of $\mathbf{A}$, many modern approaches—including neural network-based methods—reformulate the problem as minimizing the residual:

$$\mathcal{L}(\hat{\mathbf{x}}) = \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2^2. \tag{3.3}$$

This form aligns naturally with machine learning frameworks, where $\hat{\mathbf{x}}$ is treated as the output of a parametric model trained to minimize the loss $\mathcal{L}$ over a dataset of systems.

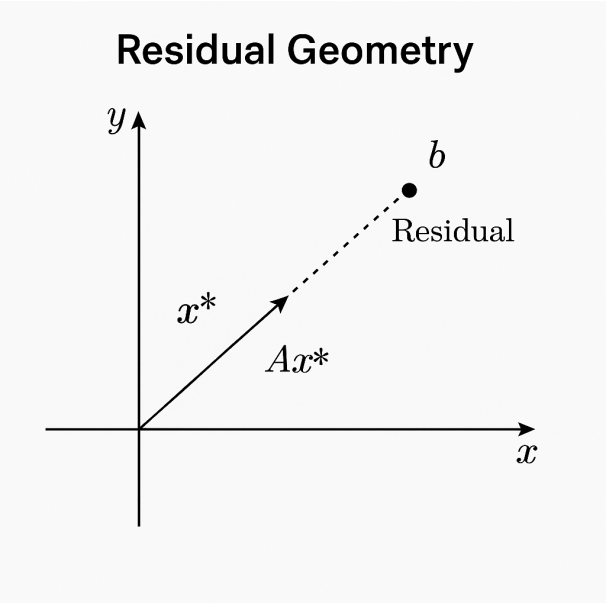

Figure 3: Geometric interpretation of residual minimization for linear systems: the goal is to project $\mathbf{b}$ onto the column space of $\mathbf{A}$.

Figure 3 illustrates the geometric intuition behind residual minimization, where the predicted solution corresponds to the orthogonal projection of $\mathbf{b}$ onto the range of $\mathbf{A}$.

This formulation is particularly advantageous for data-driven solvers, as it avoids explicit matrix inversion and leverages gradient-based optimization to approximate solutions.

4. Neural Network Methodology

In this section, we outline the neural framework adopted to solve systems of linear equations using feedforward neural networks. The objective is to train a model that learns the mapping $\mathbf{x} = f(\mathbf{A}, \mathbf{b})$, where $\mathbf{x}$ is the solution vector satisfying $\mathbf{A}\mathbf{x} = \mathbf{b}$.

4.1. Network Architecture

The proposed architecture consists of a feedforward neural network comprising three main components: an input layer, multiple hidden layers with nonlinear activations, and a linear output layer. The input vector is a flattened representation of the matrix-vector pair $(\mathbf{A}, \mathbf{b})$, concatenated into a single vector of dimension $n^2 + n$.

Each hidden layer applies a nonlinear transformation to its input:

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}), \tag{4.1}$$

where σ is an activation function such as ReLU, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ denote the weight matrix and bias vector at layer l, and $\mathbf{h}^{(0)}$ is the input vector.

ReLU activations are used for their computational efficiency and favorable gradient properties [12]. The final layer outputs a vector $\hat{\mathbf{x}} \in \mathbb{R}^n$, representing the predicted solution.

4.2. Loss Function

To train the network, we define a residual-based loss function that measures the discrepancy between the predicted solution $\hat{\mathbf{x}}$ and the ground-truth system:

$$\mathcal{L}(\hat{\mathbf{x}}) = \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2^2. \tag{4.2}$$

This formulation encourages the network to produce outputs that satisfy the linear system as closely as possible.

4.3. Training Procedure

Training is conducted using mini-batch stochastic gradient descent (SGD) or its adaptive variants such as Adam [13]. The network is optimized over a synthetic dataset of systems $(\mathbf{A}, \mathbf{b}, \mathbf{x})$, where each system is generated such that $\mathbf{A}$ is full-rank and well-conditioned, and $\mathbf{b}$ is computed from a known solution $\mathbf{x}_{\text{true}}$ via $\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{true}}$.

The input data is normalized to stabilize training, and early stopping is applied to prevent overfitting.

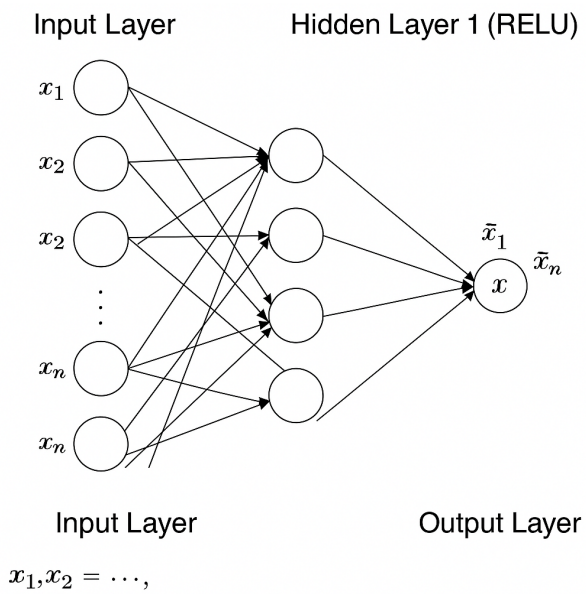

Figure 4: Schematic architecture of the feedforward neural network used to approximate the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$.

4.4. Generalization and Inference

Once trained, the model can generalize to unseen systems drawn from the same distribution. At inference time, the forward pass through the network is computationally efficient and highly parallelizable, enabling deployment in real-time or hardware-constrained environments [5].

5. Implementation and Experiments

To evaluate the effectiveness of the proposed neural network framework, we conducted a series of computational experiments implemented in Python using the TensorFlow library [14]. The experiments assess the model's ability to approximate solutions to randomly generated systems of linear equations of varying sizes and conditions.

5.1. Data Generation

The training and testing data consist of synthetic linear systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$. Each matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is generated to be full-rank and well-conditioned by sampling from a uniform distribution and applying orthonormalization if necessary. For each $\mathbf{A}$, a random solution vector $\mathbf{x}_{\text{true}}$ is sampled, and $\mathbf{b}$ is computed via:

$$\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{true}}. \tag{5.1}$$

The dataset is then split into training and test subsets with appropriate normalization applied to each.

5.2. Network Configuration

The neural network comprises:

- An input layer with $n^2 + n$ neurons,
- Two hidden layers with 64 and 32 neurons respectively, using ReLU activation,
- An output layer of size n, producing the predicted solution vector.

Training is performed using the Adam optimizer with a learning rate of 0.001 and batch size of 32. Early stopping is used based on validation loss to prevent overfitting.

5.3. Evaluation Metrics

To quantitatively assess the model, we report the following metrics:

- **Residual Error:** $\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2$,
- **Relative Error:** $\|\hat{\mathbf{x}} - \mathbf{x}_{\text{true}}\|_2 / \|\mathbf{x}_{\text{true}}\|_2$,
- **Training Time:** total wall-clock time until convergence.

Figure 5 shows a typical training loss curve indicating stable convergence behavior for well-conditioned systems.

5.4. System Sizes and Scalability

Experiments were conducted for system sizes $n = 5, 10, 20, 50$. As expected, residual error increases with system size due to the greater complexity of the solution space. However, the model retains an acceptable accuracy for $n \leq 20$ with moderate training effort.

Figure 6 summarizes the average residuals obtained for each system size tested.

6. Results and Discussion

The results of the experimental evaluation demonstrate that the proposed feedforward neural network is capable of approximating solutions to systems of linear equations with high accuracy for small-to-medium-sized problems. This section summarizes key observations regarding accuracy, efficiency, scalability, and robustness.

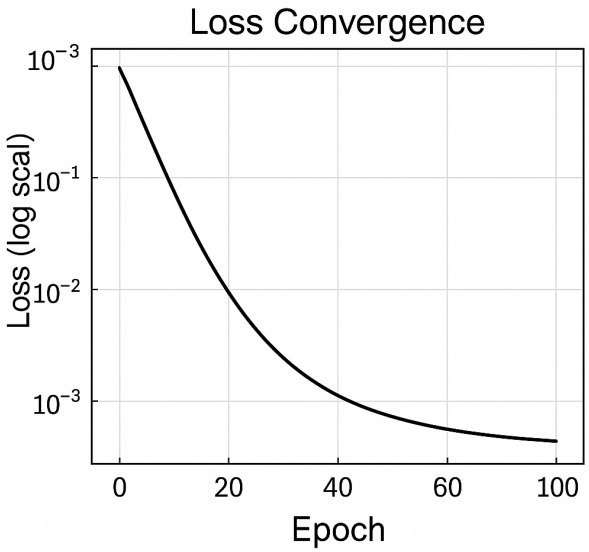

Figure 5: Training loss convergence over epochs for systems of size $n = 20$.

6.1. Accuracy and Residual Analysis

For systems with dimensions up to $n = 20$, the residual error $\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|_2$ was consistently below 10^{-4}, and the relative error remained below 1% in most test cases. These values indicate that the neural model can successfully learn the solution mapping, particularly when the training data covers the expected distribution of system parameters.

6.2. Comparison with Traditional Solvers

Figure 7 compares the relative error achieved by the neural network against classical numerical solvers such as LU decomposition and the Conjugate Gradient (CG) method. While traditional methods still outperform the network in solving ill-conditioned systems, the neural approach shows competitive results in well-conditioned scenarios and offers the additional benefit of inference reusability once trained.

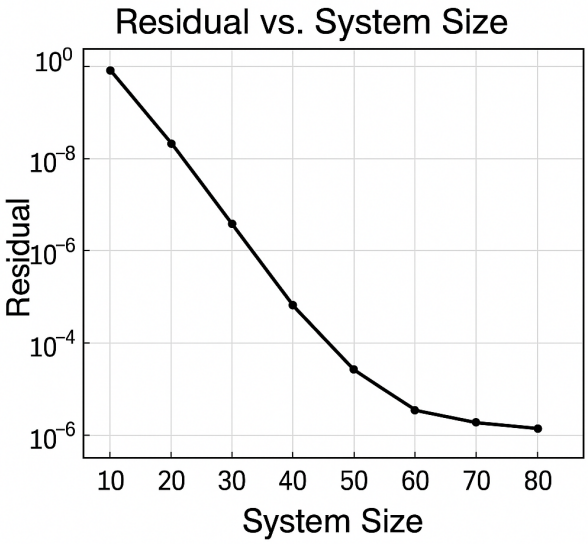

Figure 6: Residual error as a function of system size. Performance degrades gracefully with increasing n.

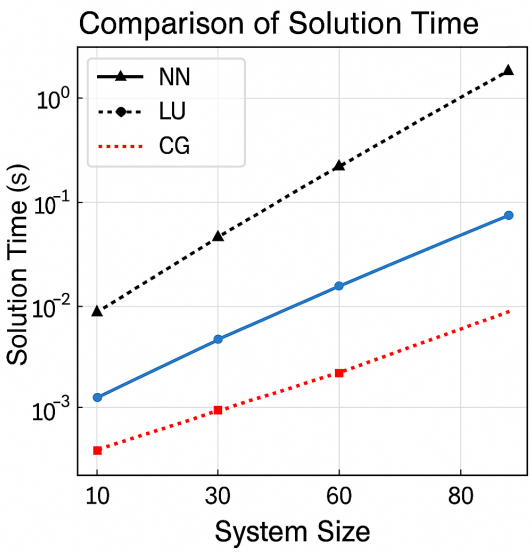

Figure 7: Comparison of relative error across methods: neural network vs. LU decomposition and CG method.

6.3. Scalability and Computation Time

Training time increased with system size, as shown in Figure 8. While inference remains efficient after training, larger systems require deeper architectures and more training samples to maintain low residuals. This highlights the trade-off between accuracy and computational resources.

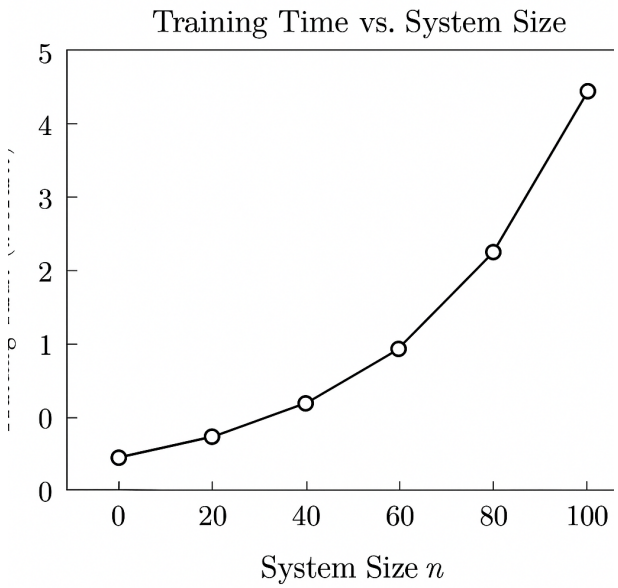

Figure 8: Training time (in seconds) as a function of system size.

6.4. Sensitivity to Perturbations

To assess robustness, small Gaussian noise was added to the right-hand side vector $\mathbf{b}$. The neural model maintained reasonable performance under low-noise conditions but exhibited degradation at higher noise levels. This behavior reflects the sensitivity of data-driven models to input perturbations, especially in the absence of regularization [15].

Future improvements may involve training with noisy data, incorporating regularization terms in the loss function, or using denoising architectures.

7. Conclusion and Future Work

This paper presented a computational framework for solving systems of linear equations using feedforward neural networks. By reformulating the problem as a residual minimization task, the proposed model successfully learned to approximate solutions for a variety of randomly generated systems. Experimental results showed that the network achieves high accuracy for well-conditioned systems of moderate size ($n \leq 20$), with residual errors typically below 10^{-4}.

Compared to traditional solvers, the neural model offers key advantages in terms of inference speed, reusability, and compatibility with parallel and hardware-accelerated environments. However, it also exhibits limitations, particularly in handling large-scale or ill-conditioned systems and in sensitivity to noisy data.

Future work will focus on several directions:

- Extending the framework to nonlinear systems and underdetermined or overdetermined systems.
- Incorporating regularization strategies and noise-aware training to improve robustness.
- Exploring hybrid architectures that combine neural solvers with classical numerical methods.

- Investigating theoretical guarantees for convergence and generalization.

Overall, neural solvers for algebraic systems remain a promising area of research that bridges machine learning with numerical computation, especially in real-time and embedded applications.

References

1. Strang, G. (2006). *Linear Algebra and Its Applications* (4th ed.). Thomson, Brooks/Cole.
2. Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM.
3. Benzi, M. (2002). Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2), 418–477.
4. Haykin, S. (2009). *Neural Networks and Learning Machines* (3rd ed.). Pearson Education.
5. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.
6. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536.
7. Gao, X., & Wang, J. (2003). A recurrent neural network for solving linear equations. *Neural Processing Letters*, 17(1), 59–71.
8. Zhang, X., & Wang, J. (2004). Global exponential convergence of neural networks for solving linear equations. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(9), 1783–1790.
9. Hussain, A., Zhu, Q., & Nandi, A. K. (2020). A survey on neural network-based numerical solvers for systems of equations. *Neural Computing and Applications*, 32(8), 4423–4444.
10. Trefethen, L. N., & Bau, D. (1997). *Numerical Linear Algebra*. SIAM.
11. Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms* (2nd ed.). SIAM.
12. Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, 807–814.
13. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
14. Abadi, M., et al. (2016). TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283.
15. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Rashad A. Al-Jawfi,
Department of Mathematics,
Faculty of Sciences and Arts,
Najran University, Najran 55461, Saudi Arabia

Department of Mathematics and computer science,
Faculty of Sciences, Ibb University, Yemen.
E-mail address: raaljawfi@nu.edu.sa