NLAFORMER: TRANSFORMERS LEARN NUMERICAL LINEAR ALGEBRA OPERATIONS *

ZHANTAO MA[†], YIHANG GAO[‡], AND MICHAEL K. NG[§]

Abstract. Transformers are effective and efficient at modeling complex relationships and learning patterns from structured data in many applications. The main aim of this paper is to propose and design NLAFormer, which is a transformer-based architecture for learning numerical linear algebra operations: pointwise computation, shifting, transposition, inner product, matrix multiplication, and matrix-vector multiplication. Using a linear algebra argument, we demonstrate that transformers can express such operations. Moreover, the proposed approach discards the simulation of computer control flow adopted by the loop-transformer, significantly reducing both the input matrix size and the number of required layers. By assembling linear algebra operations, NLAFormer can learn the conjugate gradient method to solve symmetric positive definite linear systems. Experiments are conducted to illustrate the numerical performance of NLAFormer.

Key words. Transformers, numerical linear algebra, machine learning

 $\mathbf{MSC\ codes.}\ 65\mathrm{F}10,\,68\mathrm{Q}32$

1. Introduction. The transformer model, introduced by Vaswani et al. in 2017, was originally developed for natural language processing (NLP) tasks [12]. By combining multi-head self-attention with feedforward neural networks (FFNs), transformers effectively capture global dependencies in input sequences. Transformers have proven to be effective in modeling complex relationships and learning patterns from structured data [3]. Their success has revolutionized fields such as natural language processing, computer vision, and scientific reasoning, see [2, 6].

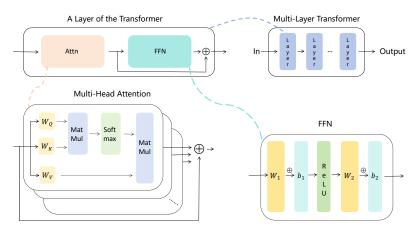


Fig. 1. The components of a multi-layer transformer

In Figure 1, we illustrate the components of a multi-layer transformer. Each

^{*}Submitted to the editors August 2025. This work was partly supported by the National Key Research and Development Program of China under Grant 2024 YFE 0202900 and Joint NSFC and RGC N-HKU769/21.

[†]Department of Mathematics, The University of Hong Kong, Pokfulam, Hong Kong (mazhantao@connect.hku.hk).

[‡]Department of Mathematics, National University of Singapore, Singapore (gaoyh@nus.edu.sg).

[§]Department of Mathematics, Hong Kong Baptist University, Kowloon Tong, Hong Kong (michael-ng@hkbu.edu.hk).

single-layer transformer consists of a multi-head attention module and a feedforward neural network. The attention module contains three elements: queries (Q), keys (K), and values (V). These three elements are represented by three matrices \mathbf{W}_Q , \mathbf{W}_K and \mathbf{W}_V respectively. For an input token \mathbf{P} , the attention mechanism computes similarity scores between queries and keys to determine how much attention each position should pay to others. These scores are normalized using softmax and used to calculate a weighted sum of the values, allowing each position to incorporate context from the entire sequence and efficiently model global dependencies. The output $\mathrm{TF}(\mathbf{P})$ is given by the following computational procedure:

$$\operatorname{Attn}(\mathbf{P}) = \mathbf{P} + \sum_{i=1}^{h} \mathbf{W}_{V}^{(i)} \mathbf{P} \cdot \operatorname{softmax} \left(\mathbf{P}^{\top} \mathbf{W}_{K}^{(i) \top} \mathbf{W}_{Q}^{(i)} \mathbf{P} \right),$$

$$\operatorname{FFN}(\mathbf{P}) = \mathbf{W}_{2} \operatorname{ReLU}(\mathbf{W}_{1} \mathbf{P} + \mathbf{b}_{1} \mathbf{1}^{\top}) + \mathbf{b}_{2} \mathbf{1}^{\top},$$

$$\operatorname{TF}(\mathbf{P}) = \operatorname{Attn}(\mathbf{P}) + \operatorname{FFN}(\operatorname{Attn}(\mathbf{P})),$$

$$\operatorname{where softmax}(\mathbf{Z})_{i,j} = \frac{e^{Z_{i,j}}}{\sum_{k=1}^{d} e^{Z_{k,j}}}, \quad \operatorname{ReLU}(\mathbf{Z})_{i,j} = \max(0, Z_{i,j}),$$

1 is a vector of all ones.

In this process, multi-head attention is used to capture different types of relationships. The output is then passed through a feedforward network, which enhances the ability of the model to learn complex mappings using two linear transformations $(\mathbf{W}_1, \mathbf{W}_2)$ with biased vectors $(\mathbf{b}_1, \mathbf{b}_2)$, and a nonlinear rectified linear unit (ReLU). A multi-layer transformer stacks several single-layer transformers sequentially, with each layer refining the output of the previous one to capture increasingly complex patterns in the input.

1.1. Numerical Linear Algebra Operations. Numerical linear algebra supports a wide range of applications, including scientific computing, engineering simulations, and economic modeling. It serves as a foundational tool for precise and efficient problem solving in these fields [7]. At its core, numerical linear algebra comprises fundamental mathematical operators such as shifting, matrix transposition, multiplication, inner products, etc. These operators are not merely computational tools; they form the theoretical basis of classical iterative numerical algorithms [11].

A previous study [1] experimentally verified that the transformer can perform basic numerical linear algebra operations by adopting a very specialized encoding. In [5], Giannou et al. developed a loop-transformer model as a programmable computer. More precisely, they designed and mapped transformer models to fundamental computer modules such as scratchpad, memory, and instruction. The loop-transformer model can be employed to simulate the execution of basic numerical linear algebra operations. In [13], Yang et al. designed the looped transformers to learn algorithms such as sparse linear regression. In [4], Gao et al. incorporated pre- and post-transformer modules, inspired by the common pre-processing and post-processing steps in traditional algorithms, to enhance performance in complex learning tasks.

Despite these promising developments, existing approaches that simulate computer control flow with transformers remain inefficient and ill-suited for numerical linear algebra operations. The fundamental issue lies in the attempt to emulate each primitive operation, such as memory access, data movement, and basic arithmetic, using a whole transformer block. For example, transposing an $n \times n$ matrix using the loop-transformer [5] requires $O(n^3)$ input and a 4-layer transformer architecture

to ensure successful learning. Moreover, for matrix multiplication between $n \times n$ matrices: $\mathbf{A}^{\top}\mathbf{B}$, the loop-transformer requires $O(n^3)$ input and a 6-layer transformer architecture. These observations motivate a deeper exploration of transformers in the domain of numerical linear algebra by distilling the transformer architecture, identifying the components that are genuinely useful for numerical linear algebra while eliminating those that introduce unnecessary computations and storage.

1.2. Our Contributions. In this paper, we propose and design Numerical Linear Algebra transformer (NLAFormer), which is a transformer-based architecture for learning numerical linear algebra operations. Using a linear algebra argument, we show that the proposed NLAFormer has the capacity to express numerical linear algebra operations such as pointwise computation, shifting, transposition, inner product, matrix multiplication, and matrix-vector multiplication. Moreover, our approach discards the simulation of computer control flow adopted by loop-transformer, significantly reducing both the size of the input matrix and the required number of layers. For example, in the case of n-by-n matrix transposition, our approach only requires an input of size $O(n^2)$ and a 2-layer transformer; see the results in Section 2. Similar reductions apply to other basic numerical linear algebra operations. Our NLAFormer can focus on learning meaningful representations, making it highly effective for general-purpose numerical linear algebra solvers. By assembling linear algebra operations, we demonstrate that NLAFormer can learn the conjugate gradient method for solving symmetric positive definite linear systems. Experimental results are conducted to illustrate the numerical performance of the proposed NLAFormer, showing that it can internalize the iterative logic of classical solvers and, through data-driven training, identify update strategies for faster convergence.

The outline of this paper is given as follows. In Section 2, we develop and design transformers for numerical linear algebra operations and show their expressiveness for such operations. In Section 3, we demonstrate how NLAFormer can learn a conjugate gradient method to solve symmetric positive definite linear systems. In Section 4, we present numerical results to illustrate the numerical performance of the proposed NLAFormer. In Section 5, we provide concluding remarks and present future work.

- 1.3. Notations. We use boldface capital and lowercase letters to denote matrices and vectors, respectively. Non-bold letters represent the elements of matrices, vectors, or scalars. The subscripts represent the serial numbers or the elements of the specified rows and columns of the matrix. For example, \mathbf{P}_r means the rth row of \mathbf{P} , $P_{i,j}$ means the i-th row, j-th column of \mathbf{P} . Calligraphic letters are used to represent sets.
- 2. Learning Numerical Linear Algebra Operations. A central step toward understanding the expressive capacity of transformers in numerical linear algebra is to investigate whether they can represent the fundamental operators that form the basis of classical numerical methods. Regardless of the complexity of a numerical algorithm, its computational structure ultimately relies on a small set of core operations, including pointwise computation, shifting, transposition, inner product, matrix multiplication, matrix-vector multiplication, etc. We compare the different requirements of NLAFormer and loop-transformer in handling basic operators in Table 1. As both methods require a few heads, we only present their required input matrix sizes and number of layers in the table.

Here we present Theorem 2.1 to show the expressiveness of transformers for handling ten basic numerical linear algebra operations in terms of the number of layers

Operation	Requirement	NLAFormer	Loop-Transformer	
Pointwise $+, -, \cdot, \div$	Input size	O(n)	O(n)	
between vectors	Number of layers	1	12n	
Column shift	Input size	O(n)	O(n)	
	Number of layers	1	4	
Row shift	Input size	O(n)	$O(n\log(n))$	
	Number of layers	1	4n	
Vector transpose	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	1	4	
$\mathbf{a}^{\top}\mathbf{b}$	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	1	6	
$\mathbf{a}\mathbf{b}^{\top}$	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	1	6	
Matrix transpose	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	1	4	
$\mathbf{A}^{\top}\mathbf{B}$	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	1	6	
AB	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	2	10	
Ab	Input size	$O(n^2)$	$O(n^3)$	
	Number of layers	2	6	
Table 1				

 $Different\ requirements\ between\ NLAFormer\ and\ loop-transformer.$

and heads in the structure, where each layer consists of a self-attention mechanism followed by an FFN, and each head represents an independent attention mechanism that captures different structural aspects of the input, as illustrated in Figure 1. Throughout the following proof, it is assumed that all vectors $\mathbf{a} \in \mathbb{R}^n$ are in a compact subset $\mathcal{K}_n \subseteq \mathbb{R}^n$, and that all matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ are assumed to lie in a compact subset $\mathcal{K}_{n \times n} \subseteq \mathbb{R}^{n \times n}$. Unless specified, norm $\|\cdot\|$ is taken to be the Frobenius norm for matrices.

THEOREM 2.1. (i) For any $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, there exists a one-layer, one-head transformer to execute the point-wise addition, subtraction, multiplication, and division (with $b_i \neq 0$) between vectors:

(2.1)
$$\mathbf{P} = \begin{bmatrix} a_1 & \dots & a_n \\ b_1 & \dots & b_n \\ 0 & \dots & 0 \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} a_1 & \dots & a_n \\ b_1 & \dots & b_n \\ d_1 & \dots & d_n \end{bmatrix},$$

where $\mathbf{d} = \mathbf{a} \square \mathbf{b}$, with $\square \in \{+, -, \cdot, \div\}$, and a_i, b_i, d_i , for $1 \le i \le n$ denote the ith elements of the respective vectors;

(ii) For any $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, there exists a one-layer, two-head transformer to execute the column shifting:

(2.2)
$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{a} & \mathbf{b} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{b} & \mathbf{a} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix},$$

where \mathbf{e}_1 , \mathbf{e}_2 and \mathbf{e}_3 are the first, second and third unit vectors of the 3-by-3 identity matrix; there exists a one-layer, one-head transformer to execute the

row shifting:

(2.3)
$$\mathbf{P} = \begin{bmatrix} \mathbf{a}^{\top} \\ \mathbf{b}^{\top} \end{bmatrix} \quad and \quad TF(\mathbf{P}) \approx \begin{bmatrix} \mathbf{b}^{\top} \\ \mathbf{a}^{\top} \end{bmatrix};$$

there exists a one-layer, two-head transformer to execute the transpose of ${\bf a}$ such that

$$(2.4) \ \mathbf{P} = \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad \mathit{TF}(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \dots & \mathbf{0} & \mathbf{a} \\ 0 & \dots & 0 & 0 \\ \mathbf{e}_1 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1} \end{bmatrix},$$

where $\mathbf{e}_1, \dots, \mathbf{e}_{n+1}$ are the first to (n+1)th unit vectors of the $(n+1) \times (n+1)$ identity matrix;

(iii) For any $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, there exists a one-layer, two-head transformer to execute the inner product $\mathbf{a}^{\top}\mathbf{b}$:

$$(2.5) \mathbf{P} = \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ 0 & \mathbf{b}^{\top} \\ 0 & 0 & \dots & 0 \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ 0 & \mathbf{b}^{\top} \\ 0 & 0 & \dots & \mathbf{a}^{\top} \mathbf{b} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix};$$

there exists a one-layer, two-head transformer to execute the product between vectors \mathbf{ab}^{\top} :

(2.6)

$$\mathbf{P} = \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ 0 & \mathbf{b}^{\top} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ 0 & \mathbf{b}^{\top} \\ \mathbf{0} & \mathbf{a}\mathbf{b}^{\top} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix};$$

(iv) For any $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}, \mathbf{b} \in \mathbb{R}^n$, there exists a one-layer, two-head transformer to execute the transpose of a matrix such that

$$(2.7) \mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{A}^\top \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix};$$

there exists a one-layer, two-head transformer to execute the multiplication between matrices: $\mathbf{A}^{\top}\mathbf{B}$ such that (2.8)

$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{B} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad \mathit{TF}(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{B} \\ \mathbf{0} & \mathbf{A}^\top \mathbf{B} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix};$$

there exists a two-layer, two-head transformer to execute the multiplication between matrices: **AB** such that (2.9)

$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{B} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad TF(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ \mathbf{0} & \mathbf{B} \\ \mathbf{0} & \mathbf{AB} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix};$$

there exists a two-layer, two-head transformer to execute the multiplication between matrix and vector: **Ab** such that (2.10)

$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ 0 & \mathbf{b}^{\top} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix} \quad \text{and} \quad \mathit{TF}(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ 0 & \mathbf{b}^{\top} \\ \mathbf{0} & \mathbf{A}\mathbf{b} \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix}.$$

Proof. For (i), we consider the input prompt matrix:

(2.11)
$$\mathbf{P} = \begin{bmatrix} a_1 & \dots & a_n \\ b_1 & \dots & b_n \\ 0 & \dots & 0 \end{bmatrix}.$$

We take all attention parameters to be zero, i.e., $\mathbf{W}_{Q,K,V} = 0$, so that the attention module simply returns the input unchanged:

$$(2.12) Attn(\mathbf{P}) = \mathbf{P}.$$

We now consider how the FFN processes this matrix. Since FFN acts independently and same on each column, it suffices to analyze a single column of \mathbf{P} . For simplicity, we write FFN as acting on a column vector below, with the understanding that the same operation is applied to every column. By using the universal approximation theorem [9, Theorem 2], for continuous function $\mathbf{f} \in C(\mathcal{K}, \mathbb{R}^3)$ on a compact set $\mathcal{K} \subseteq \mathbb{R}^3$ and any $\varepsilon > 0$, there exists an FFN such that:

(2.13)
$$\sup_{\mathbf{u} \in \mathcal{K}} \|\text{FFN}(\mathbf{u}) - \mathbf{f}(\mathbf{u})\| < \frac{\varepsilon}{n}, \quad \mathbf{f}(\mathbf{u})_r = \begin{cases} \mathbf{u}_1 \square \mathbf{u}_2, & \text{if } r = 3\\ 0, & \text{otherwise} \end{cases}$$

where $\Box \in \{+, -, \cdot, \div\}$ (with denominator $\neq 0$ when dividing). Therefore, for each column of \mathbf{P} , the FFN can be constructed to approximate the result of $a_j \Box b_j$ and write it into the third row. Applying this column-wise to \mathbf{P} , we obtain:

(2.14)
$$\|\operatorname{FFN}(\operatorname{Attn}(\mathbf{P})) - \mathbf{T}\| < \varepsilon$$
, where $\mathbf{T} = \begin{bmatrix} 0 & \dots & 0 \\ 0 & \dots & 0 \\ d_1 & \dots & d_n \end{bmatrix}$, $d_j = a_j \Box b_j$.

Thus, we obtain the final result, where the approximation comes from the FFN:

(2.15)
$$TF(\mathbf{P}) = Attn(\mathbf{P}) + FFN(Attn(\mathbf{P})) \approx \mathbf{P} + \mathbf{T} = \begin{bmatrix} a_1 & \dots & a_n \\ b_1 & \dots & b_n \\ d_1 & \dots & d_n \end{bmatrix},$$

Remark that if d_i is replaced by a function $f(a_i, b_i)$, where f is any composition of addition, subtraction, multiplication, or division, the result still holds.

For (ii), the input prompt is depicted as follows:

$$\mathbf{P} = egin{bmatrix} \mathbf{0} & \mathbf{a} & \mathbf{b} \ \mathbf{0} & \mathbf{0} & \mathbf{0} \ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix}.$$

We construct

$$\mathbf{W}_Q = \begin{bmatrix} \mathbf{0} & C & C & C \\ \mathbf{0} & 0 & 0 & c \\ \mathbf{0} & 0 & c & 0 \end{bmatrix}, \ \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} C & C & C \\ 0 & 0 & c \\ 0 & c & 0 \end{bmatrix}, \ \mathbf{W}_K = \begin{bmatrix} \mathbf{0} & \mathbf{E}_3 \end{bmatrix}, \ \mathbf{W}_K \mathbf{P} = \begin{bmatrix} \mathbf{E}_3 \end{bmatrix},$$

$$\mathbf{W}_V = e^C \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{E}_n & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_V \mathbf{P} = e^C \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{a} & \mathbf{b} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix},$$

where C, c are positive constants, and the n-by-n identity matrix \mathbf{E}_n . Then we mark:

$$\mathbf{Z} = \mathbf{P}^T \mathbf{W}_K^T \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} C & C & C \\ 0 & 0 & c \\ 0 & c & 0 \end{bmatrix}.$$

By using similar deduction in [4, 5], we know the following equation holds for $2 \le i, j \le 3$:

(2.16)
$$(e^C \operatorname{softmax}(\mathbf{Z}))_{i,j} \approx 1 + Z_{i,j},$$

where the approximation can be arbitrarily well, as C is sufficiently large and c is sufficiently small. After we introduce another head to cancel the constant (see [4]), we obtain that the output of the attention layer:

$$(2.17) \qquad \sum_{h=1}^{2} \mathbf{W}_{V}^{(h)} \mathbf{P} \cdot \operatorname{softmax}(\mathbf{P}^{T} \mathbf{W}_{K}^{(h)T} \mathbf{W}_{Q}^{(h)} \mathbf{P}) \approx c \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{b} & \mathbf{a} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

Therefore, we have

(2.18)
$$\operatorname{Attn}(\mathbf{P}) \approx \mathbf{U} = \begin{bmatrix} \mathbf{0} & \mathbf{a} & \mathbf{b} \\ \mathbf{0} & c\mathbf{b} & c\mathbf{a} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix}.$$

First, let r_1 denotes the row containing \mathbf{a} and \mathbf{b} , r_2 denotes the row containing $c\mathbf{b}$ and $c\mathbf{a}$ in \mathbf{U} . Although the FFN takes the matrix as input, it processes each column independently and identically. Thus, it suffices to prove the approximation for a single column; the same applies to all others. For simplicity, we treat FFN as acting on a column vector below, with the understanding that the same operation is applied to every column. By using the universal approximation theorem [9, Theorem 2], for all $\varepsilon > 0$, compact $\mathcal{K} \subseteq \mathbb{R}^{2n+3}$, the FFN can approximate the continuous function $\mathbf{f} \in C(\mathcal{K}, \mathbb{R}^{2n+3})$:

(2.19)
$$\sup_{\mathbf{u} \in \mathbf{K}} \|\text{FFN}(\mathbf{u}) - \mathbf{f}(\mathbf{u})\| < \frac{\varepsilon}{3}, \quad \mathbf{f}(\mathbf{u})_r = \begin{cases} -\mathbf{u}_{r_1} + \frac{1}{c}\mathbf{u}_{r_2}, & \text{if } r = r_1 \\ -\mathbf{u}_{r_2}, & \text{if } r = r_2 \end{cases}.$$

$$\mathbf{0}, \quad \text{otherwise}$$

Therefore, by applying f column-wise to U, we obtain:

(2.20)
$$\|\text{FFN}(\mathbf{U}) - \mathbf{T}\| < \varepsilon, \text{ where } \mathbf{T} = \begin{bmatrix} \mathbf{0} & -\mathbf{a} + \mathbf{b} & -\mathbf{b} + \mathbf{a} \\ \mathbf{0} & -c\mathbf{b} & -c\mathbf{a} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

Hence, we obtain the final result:

(2.21)
$$\operatorname{TF}(\mathbf{P}) \approx \mathbf{U} + \mathbf{T} = \begin{bmatrix} \mathbf{0} & \mathbf{b} & \mathbf{a} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \end{bmatrix}.$$

The rest of the statements in (ii) can be proved similarly. For (iii), the input prompt is depicted as follows:

$$\mathbf{P} = \begin{bmatrix} 0 & \mathbf{a}^{\top} \\ 0 & \mathbf{b}^{\top} \\ 0 & 0 & \dots & 0 \\ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix}.$$

We construct:

$$\mathbf{W}_{Q} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & c \\ \mathbf{0} & \mathbf{1} & 1 \end{bmatrix}, \quad \mathbf{W}_{Q} \mathbf{P} = \begin{bmatrix} 0 & \cdots & 0 & c \\ 1 & \cdots & 1 & 1 \end{bmatrix}, \quad \mathbf{W}_{K} = \begin{bmatrix} 1 & \mathbf{0} & 0 & \mathbf{0} \\ 0 & \mathbf{0} & C & \mathbf{0} \end{bmatrix},$$
$$\mathbf{W}_{K} \mathbf{P} = \begin{bmatrix} 0 & \mathbf{a}^{\mathsf{T}} \\ C & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_{V} = \begin{bmatrix} 0 & 0 & \mathbf{0} \\ 0 & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_{V} \mathbf{P} = e^{C} \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{b}^{\mathsf{T}} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

for positive constants C, c. By using similar argument in (ii), we have the desired results:

(2.22)
$$\operatorname{TF}(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{a}^{\top} \\ \mathbf{0} & \mathbf{b}^{\top} \\ 0 & 0 & \dots & \mathbf{a}^{\top} \mathbf{b} \\ \mathbf{e}_{1} & \mathbf{e}_{2} & \dots & \mathbf{e}_{n+1} \end{bmatrix}.$$

The rest of the statements in (iii) can be proved similarly. For (iv), the input prompt is depicted as follows:

$$\mathbf{P} = egin{bmatrix} \mathbf{0} & \mathbf{A} & \ \mathbf{0} & \mathbf{0} & \ \mathbf{e}_1 & \mathbf{e}_2 & \dots & \mathbf{e}_{n+1} \end{bmatrix}.$$

We construct

$$\mathbf{W}_Q = \begin{bmatrix} \mathbf{0} & \mathbf{0} & c\mathbf{E}_n \\ \mathbf{0} & 1 & \mathbf{1} \end{bmatrix}, \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} \mathbf{0} & c\mathbf{E}_n \\ 1 & \mathbf{1} \end{bmatrix}, \quad \mathbf{W}_K = \begin{bmatrix} \mathbf{E}_n & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & C & \mathbf{0} \end{bmatrix},$$
$$\mathbf{W}_K \mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{A} \\ C & 0 & \cdots & 0 \end{bmatrix}, \quad \mathbf{W}_V = e^C \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{E}_n \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_V \mathbf{P} = e^C \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{E}_n \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

for positive constants C, c. By using deduction similar to (ii) and the results in [4, 5], we have the desired result:

(2.23)
$$\operatorname{TF}(\mathbf{P}) \approx \begin{bmatrix} \mathbf{0} & \mathbf{A}^{\top} \\ \mathbf{0} & \mathbf{0} \\ \mathbf{e}_{1} & \mathbf{e}_{2} & \dots & \mathbf{e}_{n+1} \end{bmatrix}.$$

The rest of the statements in (iv) can be proved similarly.

Next, we briefly describe the requirements of the loop-transformer, as summarized in Table 1. First, according to Theorem 4 in the paper [5], performing $+,-,\cdot,\div$ between two elements requires an input size of O(n) and 12 layers. Thus, performing full pointwise operations $(+,-,\cdot,\div)$ between two vectors of length n requires an input size of O(n) and 12n layers. For row shifting, according to Lemmas 2 and 3 in the

paper [5], swapping a single element between two row vectors requires two read and write operations, each with an input size of $O(n \log n)$ and one layer. Therefore, performing a full row swap requires an input size of $O(n \log n)$ and 4n layers of the transformer. The column shift can be deduced similarly. According to Lemma 6 in [5], matrix transposition requires an input size of $O(n^3)$ and four layers of the transformer. For matrix multiplication $\mathbf{A}^{\top}\mathbf{B}$, according to Lemma 20 and Lemma 6 in [5], the matrix \mathbf{B} must be first transposed before matrix multiplication to obtain the result. Therefore, the final requirement is an input size of $O(n^3)$ and 6 layers. The requirement of $\mathbf{A}\mathbf{B}$ can be deduced similarly. For all other vector operations, the vector is first zero-padded into a square matrix, and then matrix theorems are applied; thus, the requirements are similar to those for matrices.

According to Theorem 2.1, we demonstrate that the transformer architecture is sufficiently expressive to represent essential numerical operators. This finding confirms that transformers can effectively embody fundamental numerical linear algebra operations within a single cohesive framework, highlighting their potential as a unified solver architecture. NLAFormer discards the simulation of the computer control flow adopted by loop-transformer [5], significantly reducing both the size of the input matrix and the required number of layers.

3. Learning the Conjugate Gradient Algorithm. This section demonstrates how NLAFormer can assemble numerical linear algebra operations and learn the conjugate gradient algorithm for solving symmetric positive definite systems. Solving systems of linear equations is a fundamental task in numerical linear algebra. The objective is to find the solution vector \mathbf{x} that satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A} = [\mathbf{a}_1, \dots, \mathbf{a}_n]^T \in \mathbb{R}^{n \times n}$ is a positive definite matrix, and $\mathbf{b} = [b_1, \dots, b_n]^T \in \mathbb{R}^n$ is the corresponding vector on the right side. The iterative process of the CG method is as follows:

(3.1)
$$\mathbf{d}_{0} = \mathbf{r}_{0} = \mathbf{b} - \mathbf{A}\mathbf{x}_{0}, \quad k = 0,$$

$$\alpha_{k} = \frac{\|\mathbf{r}_{k}\|^{2}}{\mathbf{d}_{k}^{T}A\mathbf{d}_{k}}, \quad \mathbf{x}_{k+1} = \mathbf{x}_{k} + \alpha_{k}\mathbf{d}_{k},$$

$$\mathbf{r}_{k+1} = \mathbf{r}_{k} - \alpha_{k}\mathbf{A}\mathbf{d}_{k}, \quad \beta_{k+1} = \frac{\|\mathbf{r}_{k+1}\|^{2}}{\|\mathbf{r}_{k}\|^{2}},$$

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_{k+1}\mathbf{d}_{k}, \quad k = k+1,$$

as introduced in [10].

The conjugate gradient method inherently follows an iterative dependency structure, with each step building on the results of previous iterations. To align with the workflow of iterative numerical algorithms, NLAFormer is composed of three sequential transformer blocks: a pre-processing block (TF $_{\rm pre}$), an iterative loop block (TF $_{\rm loop}$), and a post-processing block (TF $_{\rm post}$). The complete architecture is deonated as NLAF and is defined as follows:

(3.2)
$$\operatorname{NLAF}(\mathbf{P}) = \operatorname{TF}_{\operatorname{post}}\left(\underbrace{\operatorname{TF}_{\operatorname{loop}}(\cdots \operatorname{TF}_{\operatorname{loop}}}_{\operatorname{looping}}(\operatorname{TF}_{\operatorname{pre}}(\mathbf{P}))\cdots\right)\right).$$

The truncated variants are denoted by NLAF^t , for t = 0, ..., T. Specifically, NLAF^0 includes only the pre-processing block TF_{pre} . For $1 \leq t \leq T-1$, NLAF^t consists of TF_{pre} followed by t applications of the loop block TF_{loop} . Finally, NLAF^T executes the complete pipeline. An illustration of the structured iterative model is provided in Figure 2.



Fig. 2. Structured iterative model diagram

The preprocessing module corresponds to the initialization and input data preparation steps in the algorithm, serving to prepare the data for subsequent processing. The loop module performs multiple iterations, shares consistent parameters between them, and encapsulates the core iterative computations. Finally, the postprocessing module performs the final iteration and outputs the result in the required format. The effectiveness of this structured iterative module design has been validated in the paper [4]. This design mimics the control flow of traditional iterative solvers but differs from conventional approaches that rely on explicitly hand-coded rules. Instead, each transformer module learns to represent the underlying update rules by observing numerical patterns during training. This ability to learn underlying procedures from data is precisely the capability that we seek to develop for tasks in numerical linear algebra.

The following theorem demonstrates that there exists an NLAFormer capable of expressing the conjugate gradient algorithm.

Theorem 3.1. There exists a specially crafted transformer module, comprised of TF_{pre} (a single-layer, four-head transformer) and TF_{loop} (a single-layer, two-head transformer), that can execute the conjugate gradient method, as outlined in Equation (3.1), for solving linear systems.

Let us first establish the following results that a single-layer, two-head transformer is capable of executing a single step of the conjugate gradient method when employed with a specific configuration of input.

LEMMA 3.2. A single-layer, dual-head transformer has the capability to execute a single iteration of the conjugate gradient method for solving linear systems as described in Equation (3.1).

Proof. We consider the following input prompt:

(3.3)
$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\ 0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1} \end{bmatrix}$$

We construct

$$\begin{aligned} \mathbf{W}_Q &= \begin{bmatrix} \mathbf{0} & c\mathbf{E}_n & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}, \quad \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} \mathbf{0} & c\mathbf{d}_k \\ \mathbf{1} & \mathbf{1} \end{bmatrix}, \quad \mathbf{W}_K = \begin{bmatrix} \mathbf{E}_n & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & C & \mathbf{0} \end{bmatrix}, \\ \mathbf{W}_K \mathbf{P} &= \begin{bmatrix} \mathbf{0} & \mathbf{a}_1 & \dots & \mathbf{a}_n \\ C & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{W}_V = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & e^C \mathbf{E}_n \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_V \mathbf{P} = e^C \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & e^C \mathbf{E}_n \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \end{aligned}$$

for positive constants C, c. Also, we denote

$$\mathbf{Z} = \mathbf{P}^T \mathbf{W}_K^T \mathbf{W}_Q \mathbf{P} = egin{bmatrix} C & C & \dots & C & C \ 0 & 0 & \dots & 0 & c \mathbf{a}_1^T \mathbf{d}_k \ dots & dots & \ddots & dots & dots \ 0 & 0 & \dots & 0 & c \mathbf{a}_n^T \mathbf{d}_k \end{bmatrix}.$$

By using similar deduction in [4, 5], we know the following equation holds for $2 \le l \le n+1$:

$$(3.4) (e^C \operatorname{softmax}(\mathbf{Z}))_{l,n+1} \approx 1 + Z_{l,n+1},$$

as long as C is sufficiently large and c is sufficiently small. After we introduce another head to cancel the constant [4], we know the output of the attention layer can be given as follows:

(3.5)
$$\sum_{h=1}^{2} \mathbf{W}_{V}^{(h)} \mathbf{P} \cdot \operatorname{softmax}(\mathbf{P}^{T} \mathbf{W}_{K}^{(h)T} \mathbf{W}_{Q}^{(h)T} \mathbf{P}) \approx c \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} \mathbf{d}_{k} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

We denote $Attn(\mathbf{P}) \approx \mathbf{U}$, where

(3.6)
$$\mathbf{U} = \begin{bmatrix} \mathbf{0} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\ 0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & c\mathbf{A}\mathbf{d}_k \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1} \end{bmatrix}.$$

It is evident that all of $\mathbf{x}_{k+1} - \mathbf{x}_k$, $\mathbf{r}_{k+1} - \mathbf{r}_k$, and $\mathbf{d}_{k+1} - \mathbf{d}_k$ are results of continuous mappings from \mathbf{x}_k , \mathbf{r}_k , \mathbf{d}_k , and $\mathbf{A}\mathbf{d}_k$, respectively. Therefore, similar to the proof of Theorem 1, for all $\varepsilon > 0$, the following approximation holds according to the universal approximation theorem [9, Theorem 2]:

(3.7)
$$\|\text{FFN}(\mathbf{U}) - \mathbf{T}\| < \varepsilon, \ \mathbf{T} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_{k+1} - \mathbf{d}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_{k+1} - \mathbf{x}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_{k+1} - \mathbf{r}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & -c\mathbf{A}\mathbf{d}_k \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \end{bmatrix} .$$

Therefore, the output of the transformer layer can be represented as follows:

(3.8)
$$\operatorname{TF}_{loop}(\mathbf{P}) \approx \mathbf{U} + \mathbf{T} = \begin{bmatrix} \mathbf{0} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\ 0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_{k+1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_{k+1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_{k+1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1} \end{bmatrix} .$$

This concludes that the specifically crafted transformer is capable of implementing a single iteration of the conjugate gradient method as stated in (3.1).

We now proceed to prove Theorem 3.1.

Proof for Theorem 3.1. The initial input is given as follows:

(3.9)
$$\mathbf{P} = \begin{bmatrix} \mathbf{0} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\ 0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_0 \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1} \end{bmatrix}.$$

We construct

$$\mathbf{W}_Q = \begin{bmatrix} \mathbf{0} & \mathbf{0} & & 1 \\ \mathbf{0} & \mathbf{1} & & 1 \end{bmatrix}, \quad \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{W}_K = \begin{bmatrix} \mathbf{0} & c & \mathbf{0} & 0 & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} & C & \mathbf{0} \end{bmatrix},$$

$$\mathbf{W}_{K}\mathbf{P} = \begin{bmatrix} 0 & cb_{1} & cb_{2} & \dots & cb_{n} \\ C & 0 & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{W}_{V} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & e^{C}\mathbf{E}_{n} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \quad \mathbf{W}_{V}\mathbf{P} = e^{C} \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{E}_{n} \\ \mathbf{0} & \mathbf{0} \end{bmatrix},$$

for positive constants C, c. Also, denote

$$\mathbf{Z} = \mathbf{P}^T \mathbf{W}_K^T \mathbf{W}_Q \mathbf{P} = \begin{bmatrix} C & C & \dots & C & C \\ 0 & 0 & \dots & 0 & cb_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & cb_n \end{bmatrix}.$$

We know the following equation holds for $2 \le l \le n+1$:

$$(3.10) (e^C \operatorname{softmax}(\mathbf{Z}))_{l,n+1} \approx 1 + Z_{l,n+1},$$

as long as C is sufficiently large and c is sufficiently small. After we introduce another head to cancel the constant [4], we know the output of the attention layer can be:

(3.11)
$$\sum_{h=1}^{2} \mathbf{W}_{V}^{(h)} \mathbf{P} \cdot \operatorname{softmax}(\mathbf{P}^{T} \mathbf{W}_{K}^{(h),T} \mathbf{W}_{Q}^{(h)} \mathbf{P}) \approx c \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{b} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

Subsequently, following the procedure outlined in Lemma 3.2, we employ two additional attention heads to obtain the following result:

(3.12)
$$\sum_{h=3}^{4} \mathbf{W}_{V}^{(h)} \mathbf{P} \cdot \operatorname{softmax}(\mathbf{P}^{T} \mathbf{W}_{K}^{(h),T} \mathbf{W}_{Q}^{(h)} \mathbf{P}) \approx c \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} \mathbf{x}_{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

Referring to the identity: $\mathbf{d}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, we confirm that the required quantities have been obtained through attention heads, thus yielding the following results after passing through the FFN:

$$\begin{bmatrix}
\mathbf{0}_{n,1} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\
0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_0 \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1}
\end{bmatrix},$$

By applying Lemma 3.2 iteratively, after n applications of the looped transformer, the resulting matrix adheres to the following pattern:

$$\begin{bmatrix}
\mathbf{0} & \mathbf{a}_1 & \mathbf{a}_2 & \dots & \mathbf{a}_{n-1} & \mathbf{a}_n \\
0 & b_1 & b_2 & \dots & b_{n-1} & b_n \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{d}_n \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{x}_n \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{r}_n \\
\mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\
\mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \dots & \mathbf{e}_n & \mathbf{e}_{n+1}
\end{bmatrix},$$

which completes the proof.

We note that NLAFormer effectively captures iterative procedures, confirming its utility not only for isolated algebraic operations but also for structured iterative algorithms. Note that a sequence of basic operators required by the algorithm is effectively assembled to complete the construction. Similarly, for other iterative methods, the relevant basic operators can be composed to support their representation. Therefore, our detailed demonstration of the conjugate gradient method serves as a representative example that clearly illustrates how the logic of iterative algorithms can be systematically expressed within NLAFormer. This methodological clarity enables related algorithms to be constructed in a step-by-step manner. It facilitates the structured and intuitive integration of future algorithmic extensions and reinforces NLAFormer as a general and expressive computational paradigm.

4. Experimental Results. In this section, we present empirical evaluations of the NLAFormer. In the first part, we evaluate whether NLAFormer can effectively learn iterative solutions of the CG method. In the second part, we investigate whether NLAFormer can autonomously discover iterative procedures that achieve faster convergence than the CG method.

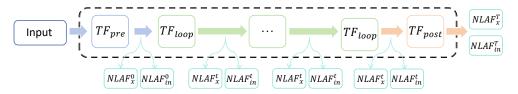


Fig. 3. Diagram of the probe mechanism.

4.1. The First Setting. We introduce two probes to extract variable information from NLAFormer's iterative output at each step, as depicted in Figure 3. The first probe captures the iterative solution (NLAF $_x^t \in \mathbb{R}^n$), corresponding to CG's approximation of the solution at each step, while the second probe extracts intermediate variables (NLAF $_{\rm in}^t \in \mathbb{R}^{2n}$), corresponding to CG's residual (\mathbf{r}_t) and direction (\mathbf{d}_t) vectors. Moreover, we implement two training approaches to investigate the extent to which the model internalizes CG's iterative logic. The first one (**Result supervision**) is that only iterative solutions (NLAF $_x^t$) are supervised with CG results whose loss is given as follows:

$$\min_{\mathbf{\Theta}} \mathbb{E}_{\mathbf{P}} \left[\frac{1}{n(T+1)} \sum_{t=0}^{T} \left\| \text{NLAF}_{\mathbf{x}}^{t}(\mathbf{P}; \mathbf{\Theta}) - \mathbf{x}_{t} \right\|_{2}^{2} \right].$$

The second one (**Joint supervision**) is that both iterative solutions ($NLAF_x^t$) and intermediate variables ($NLAF_{in}^t$) are explicitly supervised using the full loss function defined as follows:

(4.1)

$$\min_{\boldsymbol{\Theta}} \mathbb{E}_{\mathbf{P}} \left[\frac{1}{n(T+1)} \left(\sum_{t=0}^{T} \| \text{NLAF}_{x}^{t}(\mathbf{P}; \boldsymbol{\Theta}) - \mathbf{x}_{t} \|_{2}^{2} + \eta \| \text{NLAF}_{\text{in}}^{t}(\mathbf{P}; \boldsymbol{\Theta}) - (\mathbf{r}_{t}, \mathbf{d}_{t}) \|_{2}^{2} \right) \right],$$

where $(\mathbf{r}_t, \mathbf{d}_t)$ are the intermediate variable of CG at the tth iterative in (3.1), **P** is the input, Θ is the transformer parameter, and η is a regularization parameter.

In our experiment, we utilize a linear system with dimension n=20, which is the same as the setting in the paper [4]. This moderate scale facilitates a precise evaluation of the iterative learning capability of the model. The embedding size is d=256. The matrix **A** is constructed by adding a symmetric positive definite matrix, generated using standard scikit-learn libraries, to a diagonal matrix whose entries are independently drawn from a log normal distribution with zero mean and variance $\sigma=1.2$. This construction ensures positive definiteness while easing the ill-conditioning, making the test instances more representative of realistic numerical settings. The vector **x** is sampled from the standard normal distribution, i.e., $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The model training employs an Adam optimizer [8] with a dynamically adjusted learning rate schedule designed to achieve stable convergence over 700k epochs, as shown in Figure 4(a). The batch size of 64 was employed in the training procedure.

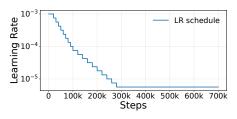
In Table 2, we present a comparative analysis of the intermediate discrepancy between the beginning of training and after 100k training steps. The discrepancy calculates the difference between intermediate variables generated by the model (NLAF_{in}^t) and those of the CG method ($\mathbf{r}_t, \mathbf{d}_t$), defined as follows: $\frac{1}{n} \sum_{t=2}^{n} \frac{1}{n-1} \|\text{NLAF}_{in}^t - (\mathbf{r}_t, \mathbf{d}_t)\|_2^2$. We observe that, over training, intermediate variables spontaneously become increasingly aligned with the CG under result supervision, despite no explicit intermediate guidance. This behavior emerges without any external pressure to match the internal structure of CG, indicating that the NLAFormer is not arbitrarily generating intermediates, but rather finds a computational path that naturally aligns with the CG recursive structure. This convergence suggests that NLAFormer implicitly identifies a CG-like recursive structure from the provided data.

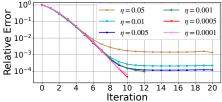
Next we conducted tests with the hyperparameter η set to values in the range $\{0.05,\ 0.01,\ 0.005,\ 0.001,\ 0.0005,\ 0.0001\}$ as shown in Figure 4(b). It shows that effective training requires result supervision to dominate, but intermediate supervision should not be too weak. We ultimately selected 0.0005, which yielded the best performance, for use in the following experimental results.

Method	Initial epoch	100k epoch	Percentage decrease
Joint supervision	46.8370	0.1920	99.7%
Result supervision	45.9350	0.5058	98.0%

Table 2

The discrepancy between the model-generated and the CG intermediate variables before and after training

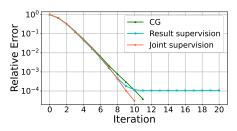


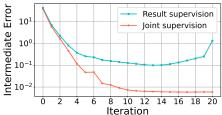


- (a) The trend of learning rate.
- (b) The relative error with different η .

Fig. 4. The experimental setting.

4.1.1. Experimental Results. After NLAFormer is trained, we demonstrate the testing performance of NLAFormer in Figure 5(a). In the figure, we display the average relative errors $(\frac{\|\text{NLAF}_{x}^{t} - x_{true}\|^{2}}{\|x_{true}\|^{2}})$ over 512 tested linear systems against iterations. Here x_{true} refers to the solution of a linear system. According to the results, we see that NLAFormer can effectively mimic the iterative process of CG across both types of supervision. Moreover, a quantitative comparison of intermediate variable errors, defined as $\frac{1}{n}\|\text{NLAF}_{in}^{t} - (\mathbf{r}_{t}, \mathbf{d}_{t})\|_{2}^{2}$, is shown in Figure 5(b). The results demonstrate that joint supervision significantly enhances alignment with the true intermediate values of CG. The marked improvement under explicit intermediate guidance conclusively indicates that NLAFormer is not merely fitting input-output mappings but effectively internalizing the iterative logic of the CG method.





(a) The relative error curves of the iterative (b) The curves of intermediate variable errors results with different training strategies.

Fig. 5. The experimental results.

4.2. The Second Setting. Building upon our theoretical analysis, if a transformer is equipped with the ability to incorporate numerical algebraic operators as learnable components, it should be capable not only of replicating operators but also of learning to combine them in more effective ways to solve the equation. This suggests that, beyond simply imitating existing algorithms such as CG, NLAFormer has the potential to autonomously learn iterative procedures that lead to faster convergence compared to established solvers. To support this exploration, we introduce losses composed of two parts. The first part, Step supervision, which guides the model to follow reference CG iteration steps, helping it internalize the logic of the CG:

$$\min_{\boldsymbol{\Theta}} \mathbb{E}_{\mathbf{P}} \left[\frac{1}{(T - T_0 + 1)n} \left(\sum_{t = T_0}^{T} \| \text{NLAF}_{\mathbf{x}}^t(\mathbf{P}; \boldsymbol{\Theta}) - \mathbf{x}_t \|_2^2 \right) \right],$$

and the second part, **Solution supervision**, which focuses on minimizing the distance to the solution of a linear system, encouraging the model to improve iteration efficiency. Their combination forms the final loss:

(4 2)

$$\min_{\boldsymbol{\Theta}} \mathbb{E}_{\mathbf{P}} \left[\frac{1}{(T - T_0 + 1)n} \left(\sum_{t = T_0}^{T} \|\text{NLAF}_{\mathbf{x}}^t(\mathbf{P}; \boldsymbol{\Theta}) - \mathbf{x}_t\|_2^2 + \lambda \|\text{NLAF}_{\mathbf{x}}^t(\mathbf{P}; \boldsymbol{\Theta}) - \mathbf{x}\|_2^2 \right) \right],$$

where $T_0 = \max\{T - K + 1, 0\}$, with K denoting the length of the learning window. It determines how many recent iterations are used during training and helps balance between learning efficiency and the model's ability to capture long-term dynamics. NLAF_x^t is the t-th iteration solution of the NLAFormer, as defined in (3.2) and depicted in Figure 3, where $t = 0, \ldots, T$. The vector \mathbf{x} is the underlying solution to the given linear system, and λ is a regularization parameter.

We conducted tests with the hyperparameter λ set to values in the range {13, 10, 7, 4, 1, 0.5} as shown in Figure 6. It shows that increasing the weight of solution supervision can accelerate convergence, but an excessively large weight may degrade performance. We ultimately selected 10, which yielded the best performance. This value was used in the following experimental results.

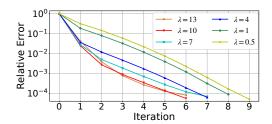


Fig. 6. The relative error curves of the iterative results with different λ

4.2.1. Experimental Results. We first verify whether the model can internalize CG's recursive behavior under the framework. Table 3 presents a comparative analysis of the intermediate discrepancy between the beginning of the training and after 100k training epochs. It can be observed that even without explicit supervision of the intermediate variables of \mathbf{r}_t , \mathbf{d}_t , the intermediates generated by the model still tend to approximate the intermediates of the CG method. This suggests that the model is capable of capturing the structured update logic of CG.

Initial step	100k epoch	Percentage decrease		
46.1199	0.4355	99%		
Table 3				

The discrepancy between the model-generated and the CG intermediate variables before and after training

In Figure 7, we show the average relative error (over 512 testing examples) for each iteration: $\frac{\|\mathrm{NLAF}_{\mathbf{x}}^t - x_{true}\|^2}{\|x_{true}\|^2}$ for both NLAFormer and CG. To ensure positive definiteness, the input matrices were generated using standard numerical libraries and by adding diagonal matrices whose entries were drawn from log-normal distributions with variances $\sigma = 1, 1.2$, and 1.4. This variation in condition numbers facilitates a more comprehensive assessment of the model performance in a variety of numerical

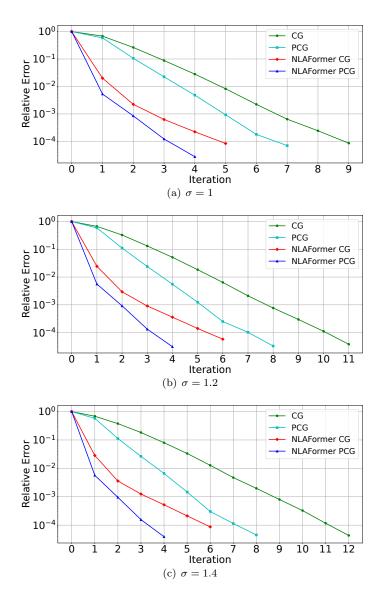


Fig. 7. Comparison results of NLAFormer and traditional algorithms on different datasets.

scenarios. As shown in Figure 7, NLAFormer shows a consistent reduction in the number of iterations required to reach the accuracy threshold compared to CG in multiple cases, even reducing the number of iterations by up to nearly 50%. Crucially, the improvement does not arise from providing explicit additional supervision regarding how to combine operators beyond CG, indicating that the model inherently grasps variations in update dynamics that lead to quicker convergence. This offers empirical backing for the theory that a more effective solver can be crafted through data-driven training.

Further, we examine whether NLAFormer can generalize its optimization capability when exposed to a stronger and more sophisticated iterative method, the Jacobi preconditioned conjugate gradient (PCG) algorithm. In this setting, we replace the

stepwise supervision signal from CG to PCG without modifying the model structure or providing explicit guidance on how to optimize PCG itself. The findings presented in Figure 7 demonstrate that NLAFormer, when trained using PCG, achieves a notably faster convergence. In contrast, the classical PCG algorithm requires more than 50% additional iterations to achieve an equivalent error level. Furthermore, the number of iterations is even reduced by approximately 30% compared to NLAFormer trained with CG. Its ability to generalize beyond specific baselines and adapt to varying algorithmic patterns highlights its potential as a flexible and learnable framework that not only replicates but also refines classical solvers, providing a foundation for exploring new ways in which data-driven training and classical methods can inform and shape one another.

4.2.2. Parameter Analysis. We assess how NLAFormer's convergence behavior is influenced by key training parameters.

- Embedding dimension. As shown in the first row of Figures 8 and 9, larger embedding dimensions generally yield better convergence performance, indicating that increased representational capacity enhances the model's ability to capture iterative dynamics.
- Learning window size. The second row of Figures 8 and 9 shows that while performance slightly degrades when reducing the learning window (e.g., from 20 to 15), the model maintains competitive convergence behavior. This suggests that NLAFormer does not overfit to a specific window size and retains robustness under reduced supervision length.
- Distribution robustness. Across various matrix distributions of the system (see Figure 7), NLAFormer consistently maintains a strong performance. This adaptability contrasts with CG, whose iteration count varies significantly depending on matrix conditioning, highlighting the potential of data-driven models to learn representations that generalize across problem variations.

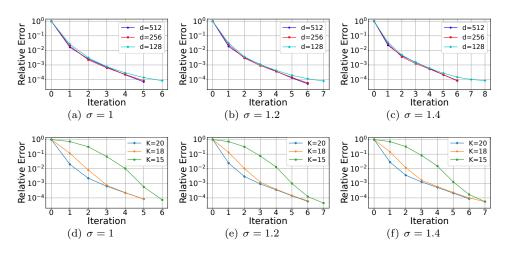


Fig. 8. Comparison results across different dimensions and learning windows on CG tasks.

5. Concluding Remarks. We proposed NLAFormer, a transformer-based framework for numerical linear algebra that efficiently represents fundamental operations with far smaller input size and fewer layers than loop-transformers. Using the conjugate gradient method as a case study, we demonstrated its ability to express complex

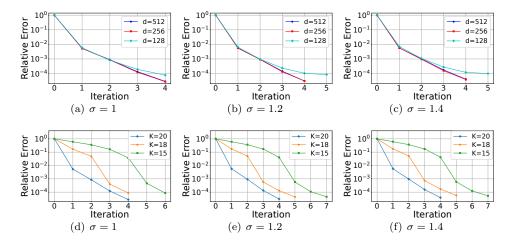


Fig. 9. Comparison results across different dimensions and learning windows on PCG tasks.

iterative numerical algorithms. Experiments further show that NLAFormer can internalize the iterative logic of classical solvers and, through data-driven training, identify update strategies for faster convergence, underscoring its potential as a compact and versatile approach for combining deep learning with numerical computation.

Looking ahead, several promising directions remain to be explored. Many important numerical computation tasks, such as nonlinear systems and partial differential equations, fall outside the current category. Extending our framework to these areas will test its generality and advance a unified neural approach to numerical analysis. On the other hand, integrating NLAFormer with classical solvers in a complementary fashion could combine the data-driven adaptability of deep models with the rigor of established numerical methods. For example, NLAFormer could rapidly generate high-quality initial guesses, which classical algorithms can then refine for precision and convergence. Pursuing these directions will strengthen the theoretical foundation and practical applicability of NLAFormer, potentially redefining how numerical computation is conceived and implemented in both academic research and industrial applications.

References.

- [1] F. Charton, *Linear algebra with transformers*, Transactions on Machine Learning Research, (2022).
- [2] E. Choi, Z. Xu, Y. Li, M. Dusenberry, G. Flores, E. Xue, and A. Dai, Learning the graphical structure of electronic health records with graph convolutional transformer, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, 2020, pp. 606–613.
- [3] C. Diao and R. Loynd, Relational attention: Generalizing transformers for graph-structured tasks, in International Conference on Learning Representations, 2023.
- [4] Y. GAO, C. ZHENG, E. XIE, H. SHI, T. HU, Y. LI, M. NG, Z. LI, AND Z. LIU, Algoformer: An efficient transformer framework with algorithmic structures, Transactions on Machine Learning Research, (2025).
- [5] A. GIANNOU, S. RAJPUT, J.-Y. SOHN, K. LEE, J. D. LEE, AND D. PA-PAILIOPOULOS, Looped transformers as programmable computers, in International

- Conference on Machine Learning, PMLR, 2023, pp. 11398–11442.
- [6] S. Islam, H. Elmekki, A. Elsebai, J. Bentahar, N. Drawel, G. Rjoub, and W. Pedrycz, *A comprehensive survey on applications of transformers for deep learning tasks*, Expert Systems with Applications, 241 (2024), p. 122666.
- [7] K. JBILOU AND M. MITROULI, Numerical Linear Algebra and the Applications, MDPI-Multidisciplinary Digital Publishing Institute, 2021.
- [8] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, in International Conference on Learning Representations, 2015.
- [9] M. LESHNO, V. Y. LIN, A. PINKUS, AND S. SCHOCKEN, Multilayer feedforward networks with a nonpolynomial activation function can approximate any function, Neural Networks, 6 (1993), pp. 861–867.
- [10] J. R. Shewchuk, An introduction to the conjugate gradient method without the agonizing pain, Tech. Report CMU-CS-94-125, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, Aug. 1994.
- [11] L. N. Trefethen and D. Bau, Numerical linear algebra, SIAM, 2022.
- [12] A. VASWANI, N. SHAZEER, N. PARMAR, J. USZKOREIT, L. JONES, A. N. GOMEZ, Ł. KAISER, AND I. POLOSUKHIN, Attention is all you need, Advances in Neural Information Processing Systems, 30 (2017).
- [13] L. Yang, K. Lee, R. D. Nowak, and D. Papailiopoulos, *Looped transformers are better at learning learning algorithms*, in International Conference on Learning Representations, 2024.