Neural Computing and Applications
https://doi.org/10.1007/s00521-024-10392-y

ORIGINAL ARTICLE

Neural acceleration of incomplete factorization preconditioning

Joshua Dennis Booth[1] · Hongyang Sun[2] · Trevor Garnett[1]

Received: 6 June 2024 / Accepted: 21 August 2024

Abstract
The solution of a sparse system of linear equations is ubiquitous in scientific applications. Iterative methods, such as the preconditioned conjugate gradient (PCG) method and the generalized minimal residuals (GMRES) method, are normally chosen over direct methods due to memory and computational complexity constraints. However, the efficiency of these methods depends largely on the preconditioner utilized. The development of a preconditioner normally requires some insight into the sparse linear system and the desired trade-off between generating the preconditioner and the reduction in the number of iterations. Incomplete factorization is a popular black box method to generate these preconditioners. However, it may fail for several reasons, including numerical issues that require searching for adequate scaling, shifting, and fill-in while utilizing a difficult-to-parallelize algorithm. With a move toward heterogeneous computing, many sparse applications find GPUs that are optimized for dense tensor applications like training neural networks being underutilized. In this work, we demonstrate that a simple artificial neural network trained either at compile time or in parallel to the running application on a GPU can provide an incomplete LL^T factorization that can be used as a preconditioner. This generated preconditioner is as good as or better than the ones found using multiple preconditioning techniques such as scaling and shifting in terms of reduction in number of iterations. Moreover, the generated method also works and never fails to produce a preconditioner that does not reduce the iteration count.

Keywords Neural networks · Sparse linear algebra · Iterative methods

1 Introduction

Scientific applications in many domains depend on the solution of sparse linear systems. While traditional Gaussian elimination-based methods (i.e., direct methods for factorizing) offer the best numerical stability, iterative methods dominate implementations in distributed-memory systems and accelerators (i.e., GPUs). The reason is that iterative methods require less memory and utilize vectorized operations in many cases. A common iterative method for symmetric positive definite (SPD) systems is the preconditioned conjugate gradient (PCG) method [1, 2] as it only relies on sparse matrix–vector multiplication (SpMV) and sparse triangular solve (Stri). However, the question of generating "good" preconditioners for a generic SPD system can be more of an art than a science. Incomplete sparse factorization methods, e.g., incomplete Cholesky, are black box methods that are typically used to generate these preconditioners. These methods normally require trying techniques such as scaling, shifting, and identifying fill-in to achieve the desired reduction in iterations. However, the algorithm of incomplete factorization tends to be difficult to parallelize due to the low computational intensity, i.e., the ratio of the number of floating-point operations to memory accesses [3]. In this work, we explore the use of neural acceleration to generate a preconditioner in order to automate this process for scientific application users and better utilize the heterogeneous computing environments common in high-performance computing.

✉ Joshua Dennis Booth
joshua.booth@uah.edu

Hongyang Sun
hongyan.sun@ku.edu

Trevor Garnett
Tjg0020@uah.edu

1 Computer Science, University of Alabama in Huntsville, Huntsville, AL 35899, USA

2 Electrical Engineering & Computer Science, University of Kansas, Lawrence, KS 66046, USA

Neural acceleration is the method of replacing key computationally expensive kernels in code with very simple and cheap artificial neural networks [4, 5]. This is very similar to simple surrogate models used in engineering. The idea is aimed at modern workloads that execute on heterogeneous systems. These systems commonly contain GPU accelerators that are optimized for the dense tensor computations utilized in training neural networks. These neural networks are small enough to easily be trained and executed during compile time or in parallel to the application. In order to utilize neural acceleration, the coder flags functions that are computationally expensive but may not suffer from being approximated by a neural network. Normally, a small amount of information is provided about the function, such as expected inputs, outputs, and computational flow. A cheap neural network is then trained on the GPU. At the time of execution, the neural network can be utilized on the GPU in place of the function call.

The use of neural acceleration for generating a preconditioner is ideal as these preconditioners are normally an approximation of the input sparse matrix and the computation of the preconditioner can be less than ideal due to low computational intensity. Moreover, many techniques are being developed for sparse computations of neural networks on GPUs due to the growing importance of graph neural networks [6]. However, the question exists if a neural network could be utilized in this relatively simple manner. In order to explore this, we first evaluate generating incomplete Cholesky (LL^T) preconditioners for SPD systems for PCG based on the computational flow [7] (i.e., the structure of LL^T). We demonstrate the choices of scaling, ordering, and overhead costs. We then extend this to solving systems that may not be SPD with the generalized minimal residuals method (GMRES). We utilize what we discover in the SPD case to generate incomplete LU and approximate LL^T (i.e., even though it may not be a candidate for Cholesky) to demonstrate its usefulness and a case for the dimensionality of approximate models.

In particular, we explore a neural acceleration method for generating an incomplete Cholesky factorization with zero fill-in that performs as good as or better than a tuned incomplete Cholesky factorization without the overhead of trying different techniques. As such, our method works as a black box for a wide range of sparse matrices and works for our own test suite of sparse matrices while most traditional methods fail in some cases. Moreover, we demonstrate that this can also be useful in some nonsymmetric cases for GMRES. Our contributions are as follows:

- A method to generate a high-quality preconditioner with a given sparsity pattern using neural networks (Sect. 3);
- A comparison of our method to other standard incomplete factorization methods that utilize a given sparsity pattern (Sect. 5);
- An analysis of timing costs to justify the use of neural acceleration (Sect. 6);
- A comparison of generating preconditioners for GMRES (Sect. 7).

2 Background and related work

This section provides a background into sparse incomplete factorization used as a preconditioner and the concept of neural acceleration.

2.1 Sparse preconditioning

2.1.1 Traditional methods

Most traditional methods focus on providing a universal robust method to generate a preconditioner for iteration methods such as PCG [1] and GMRES [8]. The most common of these is incomplete decomposition as it fits a wide array of unstructured systems. We will cover the background related to incomplete Cholesky factorization (i.e., *IChol*) for PCG though many of the same options are available in the nonsymmetric case or incomplete LU factorization (*ILU*) for GMRES. There are two forms of these incomplete methods, i.e., $IChol(k)$ and $IChol(\tau)$ [2, 3, 9]. The former, $IChol(k)$, is based on the level of fill-in, i.e., zero elements becoming nonzeros during factorization, of a sparse matrix. Here, it is common to utilize $IChol(0)$, i.e., allowing no fill-in and thus having the same nonzero pattern as the input matrix, or $IChol(1)$ as these have a small memory footprint. The second method, $IChol(\tau)$, is based on the numerical value of elements related to the off-diagonal. Off-diagonal elements that are smaller than some τ or $\tau|a_{ii}|$ are dropped. Additionally, some combination of these two, i.e., $IChol(k, \tau)$, can be utilized. However, the nonzero values in all these methods are derived from the truncated factorization method. This means that errors (e.g., ϵ that is removed by dropping a nonzero earlier in the incomplete factorization) might have a large impact on some values later on in the computation.

Due to loss of precision from dropping nonzeros, many times the incomplete factorization may fail even though the input matrix is SPD. In these cases, several options exist. The first is to simply allow for more fill-in, but the factorization will suffer from increased computation and memory costs. The other two methods try to deal with the numerical issues directly [10]. The first numerical method is applying scaling to the sparse matrix. Sparse matrices from multiphysics problems can have element values that come from a large distribution and result in diagonal values tending toward zero when updated. Relating back to neural networks, scaling of data is very common because of the numerical values desired by optimization methods utilized in training. The second numerical method is to shift the diagonal values by some small amount to prevent them from tending toward zero when updated. The value of the shift should be large enough to prevent the incomplete factorization method from failing, but small enough that the preconditioner is close to the original sparse matrix. However, both of these introduce two more parameters to consider while constructing a preconditioner.

A last consideration exists in the form of a sparse matrix ordering. The amount of fill-in during factorization is a factor or the nonzero pattern. Certain orderings, e.g., nested dissection (ND) [11] and reverse Cuthill–McKee (RCM) [12], are known to reduce fill-in. In theory, the reduction in fill-in should result in better preconditioners (i.e., a reduction in the number of iterations to converge) when a fixed number of nonzeros are applied (e.g., *ICHOL*(0)) as there should be less error due to truncation. While this idea holds in general, it does not hold for all orderings. An example of this is approximate minimal degree (AMD) [13] ordering which reduces fill-in but may not reduce iteration count to the same degree as RCM [9, 14].

2.1.2 Theoretical methods

The traditional incomplete factorization methods work well in practice, although they do require tuning parameters to match the desired convergence rate while trading off memory usage and time. There have been attempts to build more theoretically constructed preconditioners. The reason for this is both academic and due to concerns with performance. One such method is the use of support graphs to construct preconditioners for M-matrices [15]. In this method, sparse matrices are viewed with their graph representation (i.e., rows/columns as vertices and nonzero values as edges with weights). The idea is to determine the importance of an edge or additional edges and what the weights of these edges should be using the metric of a matrix pencil or eigenvalue problem. While this work is ideal for small input matrices, the concept tends not to scale to larger and more general sparse systems [16]. A more modern approach to this is preconditioners built from graph sparsification [17]. This approach utilizes larger and more global information to remove edges and reweight the graph representation to have a more ideal eigenvalue distribution. However, both of these methods suffer from not working on a wide range of sparse matrices, and the algorithms are difficult to scale (i.e., graph and sparse eigenvalue computations).

2.2 Neural networks and acceleration

The concept of utilizing neural networks to either solve or help solve systems of linear equations is not new. This is not surprising as there exists a direct relationship between solving a system and a general dense layer neural network. Figure 1 shows a visual representation of the matrix–vector multiplication ($Ax = y$). The nonzero values of the matrix (A) are represented in the network as the edge weights while the input (x_i) and outputs (y_j) represent the input and output nodes. No place is this connection seen more than in the foundational work related to online training such as Hopfield networks [18–21]. These types of neural computations look at solving the system by finding the parameters of the connections (i.e., solving for A^{-1}) in an online manner (i.e., during runtime to include training). In particular, they train the network by allowing for a fully connected network (i.e., all input nodes connected to all output nodes) and flip the inputs and outputs (i.e., $x_i \leftrightarrow y_j$). Though there has been a lot of fundamental work in this area for things like embedded systems, these are designed for small dense networks with very specific restrictions to parameter distributions. One of the main reasons for this is the numerical instability of deriving an inverse.

In addition to these traditional approaches, modern work has started to analyze where neural networks can be used in

$$\begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & 0 & a_{24} \\ a_{31} & 0 & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

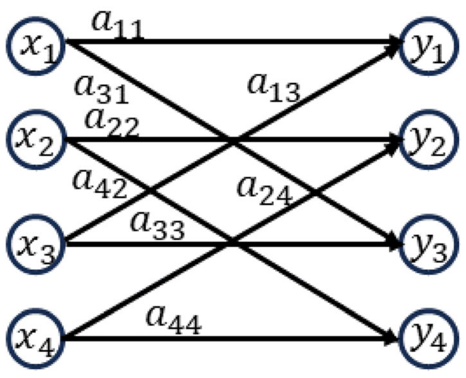


Fig. 1 Neural network representation of $Ax = y$. The input nodes (x_i) represent the elements of vector x, the output nodes (y_j) represent the elements of vector y, and the edge weights are taken from nonzero elements of the sparse matrix A

solving sparse linear systems. The work by Gotz and Anzt [22] utilizes complex convolutional neural networks to identify blocking locations for generating block Jacobi preconditioners. The work demonstrates that a neural network can be used to identify good blocking for this type of preconditioner, but the neural network developed is very large with tens of thousands of parameters even for small problems (i.e., $dimension(A) < 1000$) and it would be difficult to fit into the framework of simple models utilized by neural acceleration. We also note that this type of problem, i.e., identifying blocking patterns, is very similar to cluster and edge detection done often by neural networks within the area of image processing. In particular, the network utilized in this work is very close to that used in LeNet-5 for images.

One modern neural acceleration approach to sparse linear systems considers the problem of identifying the sparse matrix ordering and calculation of fill-in [7]. This work uses a simple graph neural network that represents the computational flow for the calculation of fill-in (i.e., column-by-column calculation). They utilize neural acceleration to outperform traditional methods in CHOLMOD [7, 23] when utilizing a GPU. However, this work does not generate factorization or give insight into the problem of precondition generation for sparse iterative solvers. However, the fundamental takeaway from that work is that the search space for a neural network model used by neural acceleration should match the workflow. This same principle of matching the workflow is utilized here.

3 Neural network construction

3.1 Overview

The current trend in neural networks is to construct a large (and most likely expensive) network that is very generic. This means a large network would be trained (i.e., in a supervised manner) with a large training set of sparse matrices as inputs and ideal preconditioners as outputs. While generic networks like these have many positive attributes, such as being able to be reused, they have a number of downfalls that make them less than ideal for neural acceleration and sparse linear algebra. The reasons are: (a) the training time normally outweighs traditional computational methods even when amortized over the number of uses, and (b) the training set for sparse linear algebra is very tiny. In particular, there are very few different nonzero patterns and numerical values to construct a big enough training set to train a huge neural network model. An example of this is demonstrated in the neural acceleration work related to graph ordering and fill-in [7].

Moving away from these large generic networks, we outline how we construct our neural networks for incomplete Cholesky preconditioners based on the computational flow. Many traditional neural network-inspired approaches try to construct a preconditioner M such that $M \approx A^{-1}$ [19–21]. In particular, the network itself becomes the output. However, constructing an inverse directly is an error-prone task. Numerical methods understand that A^{-1} can be numerically unstable and will likely be dense. This is one of the reasons sparse incomplete factorizations make sense. Sparse factorizations, in general, are used to combat these problems by replacing M with stable sparse triangular solves. As this is the standard workflow, the neural network model should have the same workflow, i.e., the neural network should have the pattern of the incomplete factorization. The problem can be further shrunk by fixing the nonzero pattern of the factorization, e.g., $ICHOL(k)$. We discuss this more about the limitations of this fix pattern assumption in the next subsection.

Figure 2 visualizes a very simple sparse two-layer model. The edges in the first layer represent the nonzeros in L^T, i.e., the transpose of a lower triangular matrix, and the edges of the second layer represent the nonzero in L, i.e., they are ordered in the manner they would be applied to LL^Tx. Our decision to utilize a fixed ordering and nonzero pattern lends itself to us because we are able to know what edges we desire to use for our model. In modern theoretical approaches to constructing incomplete sparse Cholesky (e.g., support-graph and sparsification), both the edges and weights are flexible. In more traditional level-based methods, the edges are fixed and the weights are calculated based on some truncation of the standard factorization method. In this method, the edges are fixed but the weights

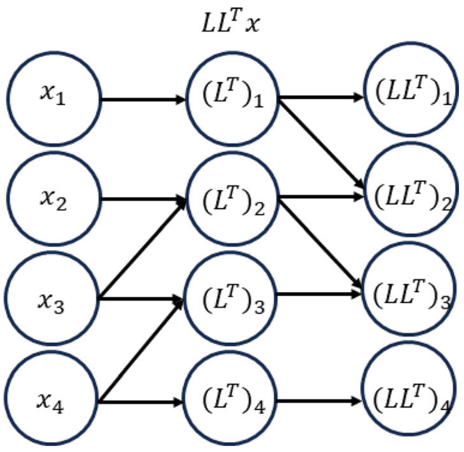


Fig. 2 Neural network model of LL^Tx. Here, the nonzero pattern (i.e., the edges) is based on the same nonzero pattern of Fig. 1. However, one hidden layer is added to the product of L^Tx. While the edges themselves are fixed based on the provided pattern, their numerical value will change based on training via backpropagation

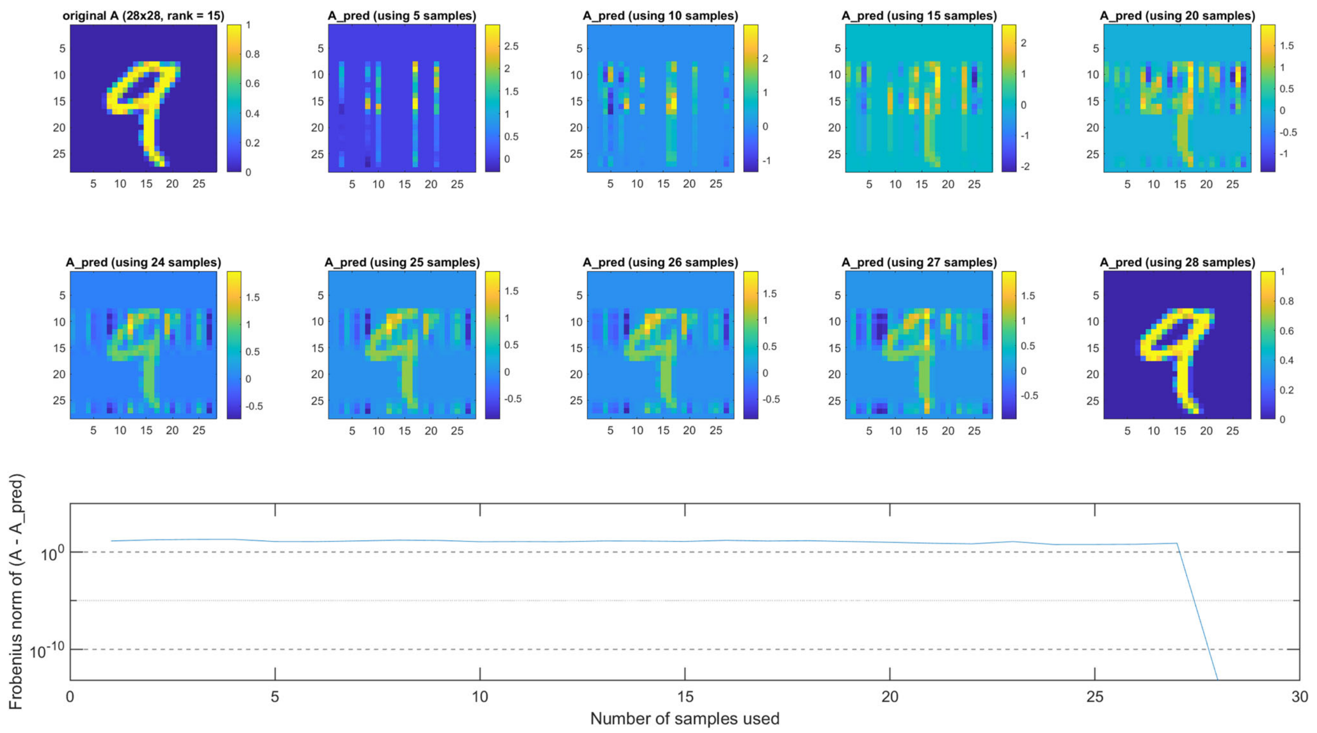


Fig. 3 Reconstruction of an MNIST image (number nine) as a matrix with increasing number of samples. The first two rows of images provide the visual reconstructions and the bottom figure provides the error in terms of the Frobenius norm of the difference between the original and reconstructed images. We note that it is difficult to even make out the number at fewer than 24 samples and that the error norm only decreases at the point of 28 samples (i.e., the number of samples equals the dimension of the image)

for them will be calculated based on the model. Note that this could be applied to any level of $ICHOL(k)$. In our analysis of quality (Sect. 5), we still restrict ourselves to $ICHOL(0)$.

This simple model lends itself well to the current methods of training, namely backpropagation. While methods like those in Hopfield networks [19–21] can be trained online, they require smaller networks with certain distributions. In fact, they generally boil down to the gradient descent method for training. With the current power of GPUs acting as accelerators, backpropagation methods that involve solving an optimization problem make sense. In particular, the objective function of such an optimization method could be written as:

$$\min ||Y - LL^T X||_2^2 \tag{1}$$

Here, we minimize the objective function by finding the numerical values for a fixed set of elements of L using Y and X training values calculated via $Y = AX$, and the 2-norm represents the mean square error.

Therefore, the neural acceleration method could take in the sparse matrix A and generate samples X and Y in order to train L. In our experimental results, we demonstrate that the number of samples needed is relatively small (i.e., $\sqrt{N}$ where $N = dimension(A)$). For the output, the method could either output L to be used by the problem in its iterative solver package or function pointers to apply sparse triangular solve for this on the GPU where it was generated.

3.2 Discussion

Several points of this method and implementation stand out in a manner that requires more discussion. The first point is limiting our method to a fixed nonzero pattern. This has two limiting factors. The first limiting factor is that an ideal nonzero pattern needs to be calculated. In this work, we utilize the nonzero pattern of the input matrix so that no additional calculation needs to be done. Limiting the nonzero pattern of the preconditioner to that of the nonzero pattern of the input matrix is commonly done in many traditional preconditioner methods. Therefore, we do not perceive this as a major limitation as many traditional methods also use it. Moreover, due to the method's speed, additional models with other nonzero patterns could be generated quickly and tested [24]. The second factor is the nonzero pattern is not dynamic to the learning. In particular, this is a limitation if the number of nonzeros needed to form a good preconditioner is less than the given nonzero pattern. The additional nonzeros would account for more storage and more operations during evaluation. However,

Table 1 SPD matrix test suite. ID is the number used to identify the matrix below, N is the matrix dimension, NNZ is the number of nonzeros in the matrix, density is the average number of nonzeros per row, SDD identifying if symmetric diagonally dominate, and area identifying the area of science the matrix is used

ID	Matrix	N	NNZ	Density	SDD	Area
1	minsurfo	40, 806	206k	5.0	Y	Optimization
2	cvxbqp1	50, 000	350k	7.0	N	Optimization
3	gridgena	48, 962	512k	10.5	N	Optimization
4	cfd1	70, 656	1, 826k	25.8	N	Fluid
5	oilpan	73, 752	2, 149k	29.1	N	Structural
6	vanbody	47, 072	2, 329k	49.5	N	Structural
7	ct20stif	52, 329	2, 600k	49.7	N	Structural
8	nasasrb	54, 870	2, 677k	48.8	N	Structural
9	cfd2	123, 440	3, 085k	25.0	N	Fluid
10	s3dkt3m2	90, 449	3, 686k	40.8	N	Structural
11	cant	62, 451	4, 007k	64.1	N	2D/3D
12	shipsec5	179, 860	4, 599k	25.6	N	Structural
13	consph	83, 334	6, 010k	72.1	N	2D/3D
14	G3_circuit	1, 585, 478	7, 661k	4.8	Y	Circuit
15	hood	220, 542	9, 895k	44.9	N	Structural
16	thermal2	1, 228, 045	8, 580k	7.0	N	Thermal
17	af_0_k101	503, 625	17, 551k	34.8	N	Structural
18	af_shell3	504, 855	17, 562k	34.8	N	Structural
19	msdoor	415, 863	19, 173k	46.1	N	Structural
20	StocF-1465	1, 465, 137	21, 005k	14.3	N	Fluid
21	Fault_639	638, 802	27, 246k	43.3	N	Structural
22	inline_1	503, 712	36, 816k	73.1	N	Structural
23	PFlow_742	742, 793	37, 138k	50.0	N	2D/3D
24	ldoor	952, 203	42, 494k	44.6	N	Structural

Table 2 GMRES matrix test suite.ID is the number used to identify the matrix below, N is the matrix dimension, NNZ is the number of nonzeros in the matrix, and density is the average number of nonzeros per row

ID	Matrix	N	NNZ	Density	NSym	Area
1	epb2	25, 228	175k	6.9	N	Thermal
2	wang3	26, 064	177k	6.8	N	Semiconductor
3	minsurfo*	40, 806	206k	5.0	Y	Optimization
4	shyy161	76, 480	330k	4.3	N	Fluid
5	cvxbqp1*	50, 000	350k	7.0	Y	Optimization
6	bcircuit	68, 902	376	5.5	N	Circuit
7	scircuit	170, 998	959k	5.6	N	Circuit
8	torso2	115, 967	1, 033k	8.9	N	2D/3D
9	xenon1	48, 600	1, 1811	24.3	N	Materials
10	ASIC_320ks	321, 671	1, 316k	4.1	N	Circuit
11	vanbody*	47, 072	2, 329k	49.5	Y	Structural
12	poisson3Db	85, 623	2, 3751	27.7	N	Fluid
13	FEM_3D_thermal2	147, 900	3, 489k	23.6	N	Thermal
14	parabolic_fem	525, 825	3, 675k	7.0	Y	Fluid

we have found that this is a trade-off between time to train and evaluate. A non-fixed pattern (e.g., *ICHOL*(τ)) would require additional training parameters (i.e., meta-parameters related to sparsification), and would require a much more expensive training search space. Additionally, we found no cases in our initial investigation while varying the nonzero pattern, which suggests a pattern that was sparser than *ICHOL*(0) is a better preconditioner. This aligns with what we address in the theoretical methods (See Sect. 2.1.2). Therefore, we do not perceive this as a limitation of the neural acceleration technique as many fast parallel incomplete factorization methods make the same assumption [3, 25], and the goal is to achieve a fast approximation.

Table 3 Methods tested for quality

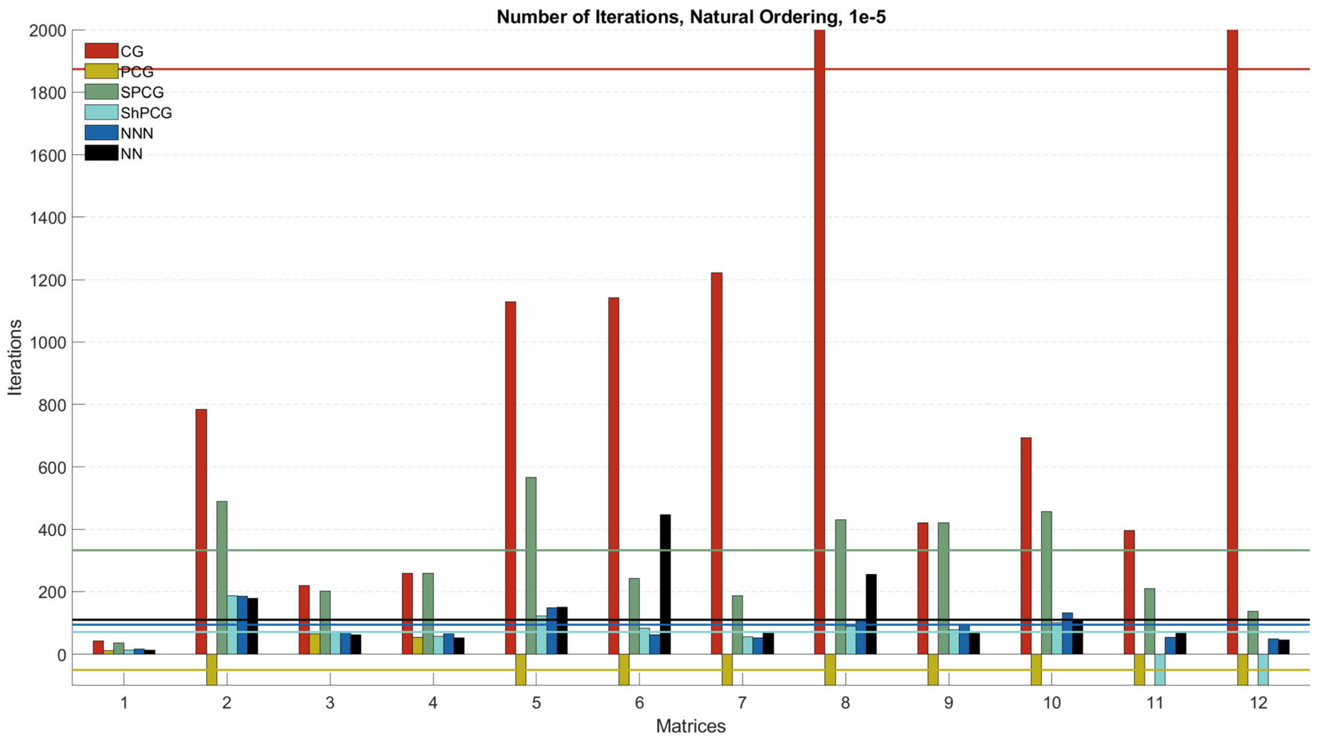
Name	Description	Tol
`CG`	CG	1*e*-5/1*e*-7
`PCG`	PCG with *ICHOL*(0)	1*e*-5/1*e*-7
`SCG`	PCG with scaled matrix	1*e*-5/1*e*-7
`ShCG`	PCG with scaled and shifted (.2)	1*e*-5/1*e*-7
`NNN`	PCG with NN generated LL^T and normalized samples	1*e*-5/1*e*-7
`NN`	PCG with NN generated LL^T	1*e*-5/1*e*-7

All but `CG` have the same number of floating-point operations per iteration

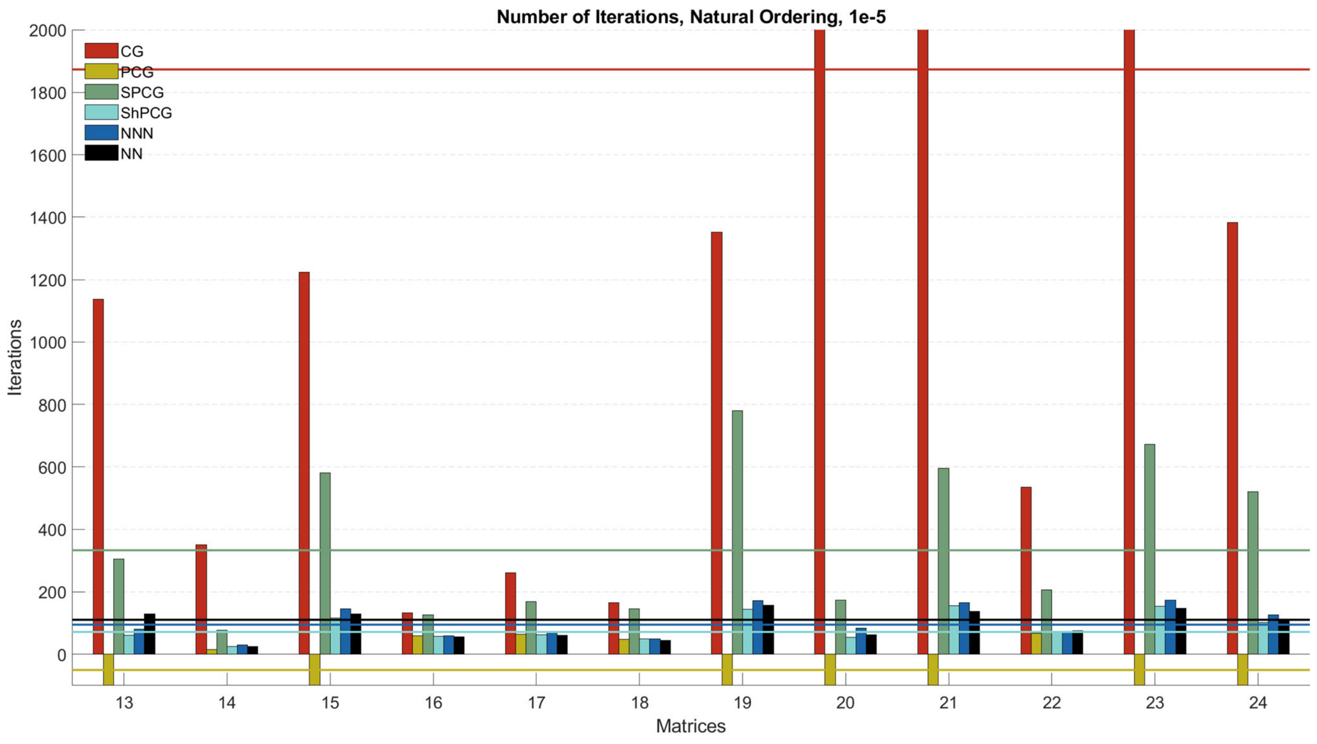
The second point that deserves discussion is training cost. Section 6 provides an empirical analysis of this cost for our less-than-optimal training implementation. The training cost would depend on both the numerical optimization method used and the number of iterations needed to construct a good approximation. The issue with this cost is that better and often more expensive numerical optimization methods require fewer iterations. In our experimentation, we utilize stochastic gradient descent (SGD) and adaptive gradient algorithm (AdaGrad) [26] as our numerical optimizers. SGD is cheaper per iteration. We note that we would not even attempt training with SGD in practice except to demonstrate the versatility of the model. SGD utilizes the same update rate for each parameter. AdaGrad utilizes the second-order information for updating to provide adaptive learning rates for each parameter. AdaGrad is commonly used for training deep learning models with sparse gradients (e.g., recurrent neural networks and transformers). Overall, SGD normally requires about N iterations to achieve the same quality as AdaGrad using $\sqrt{N}$ iterations. With AdaGrad, we find that the maximum time to train any of our test sparse neural networks is very small, and the goal of neural acceleration is to construct an approximation in a timely manner (i.e., having a small search space) that requires the least input from the user. We continue the decision of this along with choices in Sect. 6. Moreover, we only provide the results for AdaGrad in the results as the time to train using SGD for our test suite is too high.

In terms of training time and number of samples, there is an example to consider. We note that in the forward direction of $Ax = y$, where we are trying to reconstruct A from samples X and Y, we would need N samples utilizing a traditional method (e.g., using QR decomposition [27, 28] with back substitution). Figure 3 demonstrates this with an image (of the number nine) taken from the MNIST dataset [29]. This image can be viewed as a sparse matrix without full rank (i.e., the rank does not equal the dimension of A). The samples from X are taken from a random distribution to produce Y values. The reconstruction is done using the standard backs substitution operation (i.e., with QR decomposition when the size is less than N) in MATLAB. While this method does not provide the required incomplete factorization, it does give us a sense of how many samples we really should need to construct the incomplete factorization. Even with a sparse matrix that is not full rank, the image remains difficult to see after 25 samples and the error is very high (i.e., > 1). Therefore, the number of samples should be $\leq N$ to be a successful method.

Lastly, we discuss extending this method from *ICHOL* to *ILU*. The largest issue that normally impacts the generalization of *ICHOL* to *ILU* is the loss of stability that is normally handled with pivoting. Some *ILU* will also generate a permutation matrix P that provides the row permutations as a result of pivoting. Allowing this type of *ILU* may be very expensive and difficult with neural networks, though other neural acceleration methods have considered permutations in general [7]. Because pivot serializes factorization, most parallel packages try to avoid pivoting using some reordering methods (e.g., those that permute large entries to the diagonal) [3] or limit the search for a pivot to a smaller subblock [30]. In this work, we also consider *ILU* after demonstrating the usefulness of *ICHOL*. We avoid the permutation in a similar manner as many other *ILU* codes by trying to permute large values to the diagonal first and excluding matrices that fail due to stability along the way. However, as we demonstrate later in Sect. 7, there is a second issue that must be considered for our method, namely model complexity. The introduction of sometimes more than $2\times$ the number of parameters for having both the upper and lower halves adds considerable more need to train and a slower training rate. As we demonstrate later, LL^T may still be ideal even in the nonsymmetric case simply due to training.



(a) First Half, NNZ $< 5{,}000$k



(b) Second Half, NNZ $> 5{,}000$k

◀ **Fig. 4** Number of iterations to converge to a solution when the sparse matrix is ordered in their natural ordering with a relative tolerance of $1e$-5. The bars represent the raw number of iterations and the lines represent the average iteration for the method across all 24 matrices. In many cases, tradition `PCG` fails because the incomplete factorization fails. In several cases, even scaling with shifting `ShCG` fails. The only methods that works for all cases while constantly reducing iteration count are our two neural network-based methods

4 Experimental setup

4.1 Matrix test suite

We test our method utilizing two test suites taken from the SuiteSparse Matrix Collection [31]. The first suite is a set of SPD matrices for testing *ICHOL* with PCG (Table 1). The second suite is a set of full-rank matrices for testing *ILU* with GMRES (Table 2). For all matrices, a matrix ID, the dimension of the sparse matrix (N), the number of nonzeros (NNZ), the average row density (Density) (i.e., NNZ/N), and application area (Area) where the matrix is found are provided. The ID is used to identify the sparse matrix in the figures in later sections to save space and make them more readable. For the set of SPD matrices (Table 1), a column indicates if the matrix is also symmetric diagonally dominate (SDD), i.e., $|a_{ii}| \geq \sum_{i \neq j} |a_{ij}|$. We provide the column related to SDD to indicate what matrices even fall into the category where a more theoretical method could be utilized. Support graph type theoretical problems require an M-matrix which is a more restrictive group than SDD. Some of the sparsification methods will work with SDD matrices. We do not provide performance results with these theatrical methods as we found they did not improve iteration counts as much as other traditional methods like *ICHOL*(0) for our test suite, but feel that pointing them out to the reader gives them a sense of what matrices may fall into this narrow category. Additionally, a horizontal line is drawn between the matrices with IDs 12 and 13 to represent where the figures are broken into two. The set of matrices used with GMRES includes a column indicating if numerical symmetric *NSym*. Additionally, three SPD matrices are used in this test suite in addition to the first. These matrices are used to examine the impact of using *LU* on these matrices (i.e., providing additional free parameters that may not be needed). Moreover, the overall size (i.e., NNZ) of these matrices is smaller than the first set due to the number of model parameters required.

For the SPD suite, we consider both the natural ordering (i.e., the ordering provided by the input) and the RCM ordering in our quality analysis as this has been shown in the past to be a key factor in iteration count [9]. Therefore, it is interesting to examine if the ordering has an impact when the preconditioner is generated using our neural network model. For the GMRES suite, all sparse matrices are reordering with Dulmage–Mendelsohn decomposition [32] followed by RCM on $A + A^T$ nonzero pattern in order to try to promote large values onto the diagonal.

4.2 Experimental environment

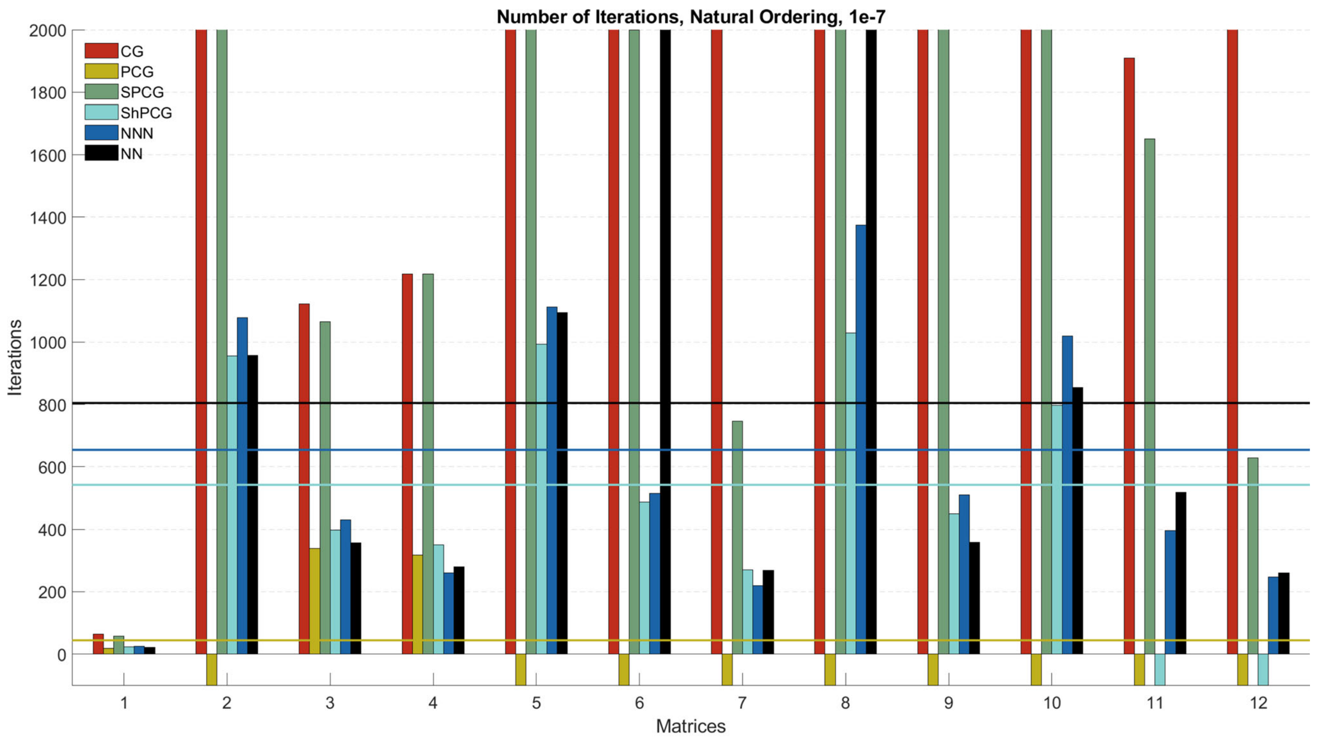
We construct our neural networks within the Tensorflow (2.8.1) framework[1] inside of Python3. MATLAB 2022a is used for PCG and GMRES. The system used is an Intel Xeon Silver 4210R that contains 10 physical cores and supports 20 threads. The CPUs run at 2.4 GHz. The system contains a total of 64 GB of DDR4 (4x16GB 2933MHz). Training is done with the system's Nvidia Quadro RTX4000 GPU with 8GB of GDDR6.

4.3 Network training

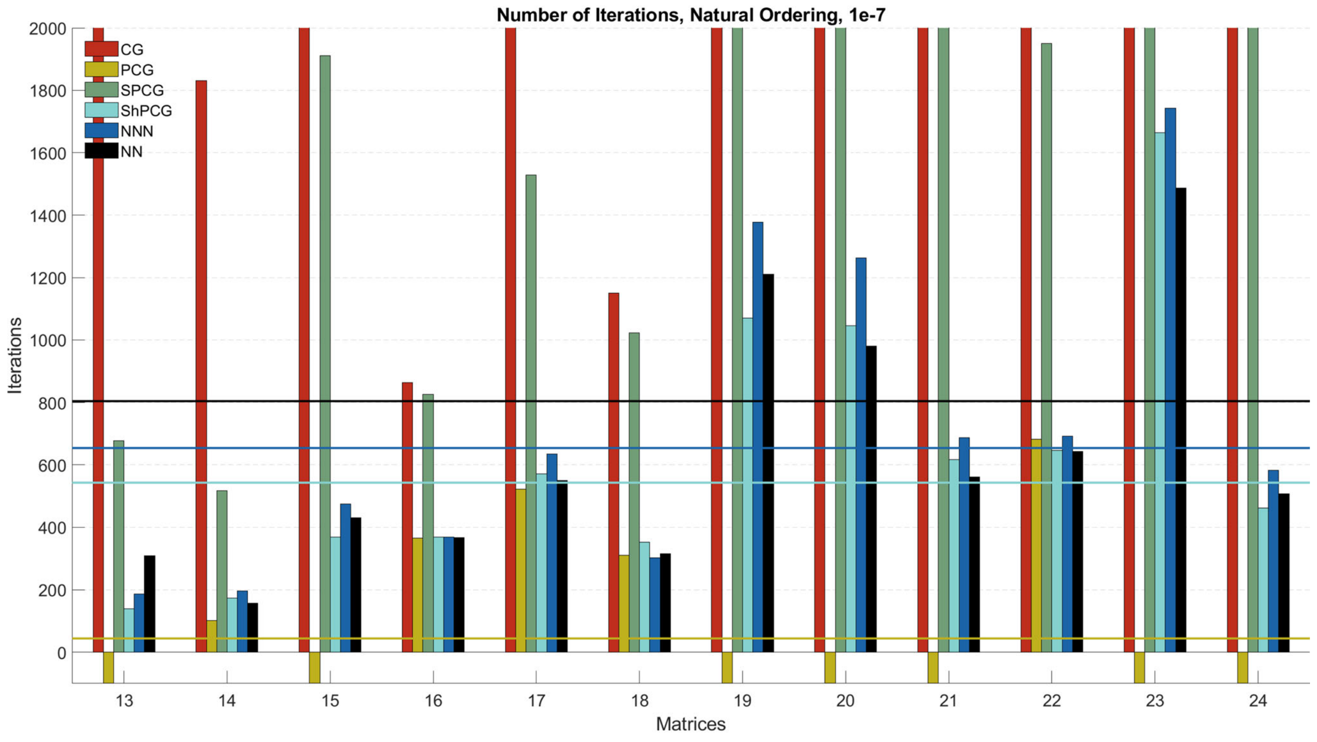
All networks are trained with a custom-made version of AdaGrad [26] built with the Tensorflow framework. The custom-made version allows utilizing only the parameters associated with the nonzero structure of LL^T or LU. Moreover, we could optimize the performance of this over the built-in AdaGrad of Tensorflow/Keras[2] by utilizing SpMV operations. A total of $\sqrt{N}$ training vectors were generated, and the models were trained iteratively utilizing a batch of 1 vector per iteration for all LL^T models. This value is based on the theoretical number of iterations for the convergence of iterative solvers. A total of $100\sqrt{N}$ training vectors were generated, and the model was trained iteratively utilizing a batch of 1 vector per iteration for all LU models. This increase in the number of training vectors is a result of the increase in model parameters and smaller α needed to converge. SGD was considered initially but it required more tuning and iterations to train. With the sparse optimizations and the reduction of iterations, the AdaGrad choice is cheaper in terms of the training time (though theoretically more expensive in terms of the number of floating-point operations and memory). We consider two sets of training for LL^T model. In particular, we train with one set of randomly selected X (denoted as NN) and one set of randomly selected X with normalized samples (denoted as NNN). We set the AdaGrad parameter to be $\alpha = 0.1$ for samples that are not normalized, and set the AdaGrad parameter to be $\alpha = N^{3/2}/20000$ for samples that are normalized. We note that α for the normalized samples is very large compared to what is normally used in

[1] https://www.tensorflow.org.

[2] https://www.tensorflow.org/guide/keras.



(a) First Half, NNZ < 5,000k



(b) Second Half, NNZ > 5,000k

◀ **Fig. 5** Number of iterations to converge to a solution when the sparse matrix is ordered in their natural ordering with a relative tolerance of 1*e*-7. The bars represent the raw number of iterations and the lines represent the average iteration for the method across all 24 matrices. In many cases, tradition `PCG` fails because the incomplete factorization fails. In several cases, even scaling with shifting `ShCG` fails. The only methods that works for all cases while constantly reducing iteration count are our two neural network-based methods

practice, and this indicates a very fast convergence in training. For the *LU* model, we only utilize non-normalized samples and utilize the AdaGrad parameter to be $\alpha = .001$. This value is relatively small and indicates a slow learning rate and influence from the update. The loss function utilized was the mean square error (MSE) as shown in Eq. (1).

5 Experimental evaluation of quality for LL^T

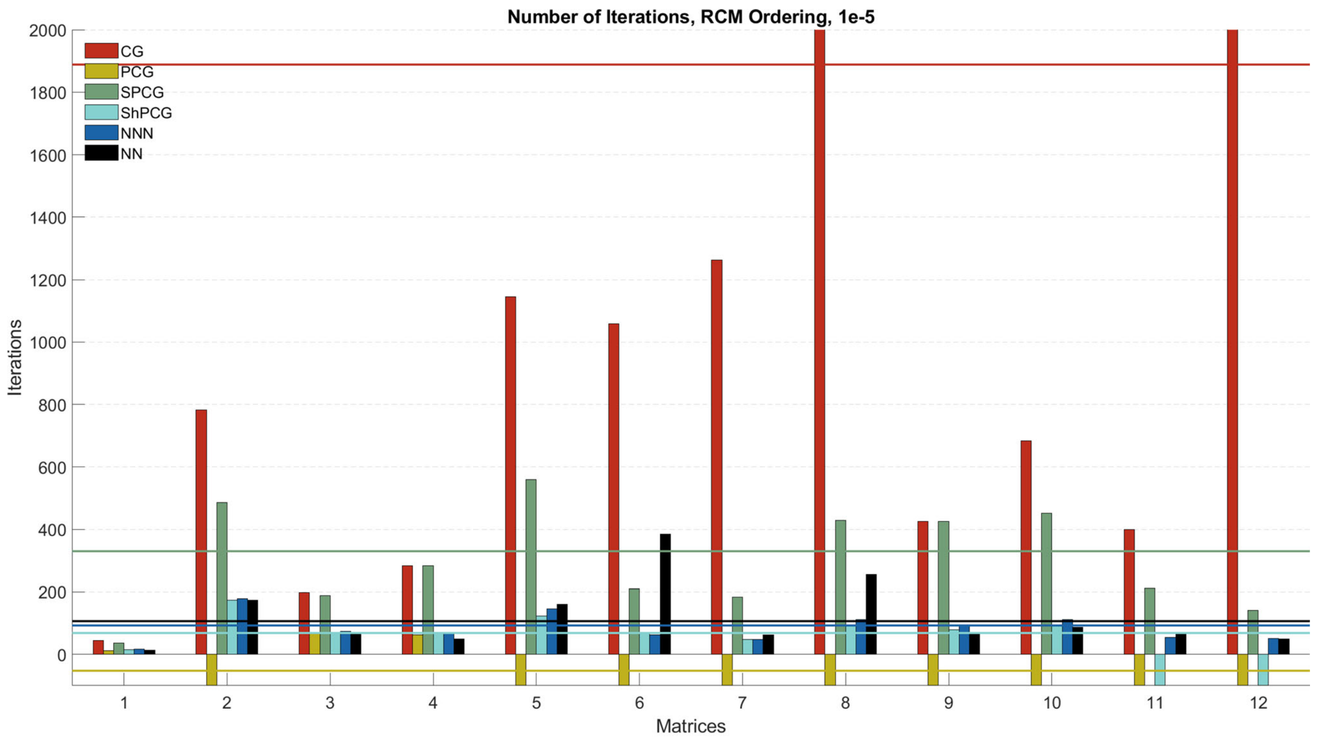
In this section, we evaluate the quality of the incomplete Cholesky factorization generated using the neural network model. To evaluate the quality, we consider the number of iterations for PCG to converge. While the timing performance of PCG is important, this is primarily dominated by the number of iterations, and therefore, the number of iterations is a sufficient measure of performance even in parallel execution.

We consider the quality of two different neural network models. The first is trained with normalized sample vectors and an AdaGrad parameter of $\alpha = N^{3/2}/20000$. We denote this one as `NNN`. The second is trained using non-normalized sample vectors and an AdaGrad parameter of $\alpha = 0.1$. We denote this one as `NN`. Data normalization is very common in training large neural networks. For our model, we desire to test its impact, and the significance is twofold. The first is that normalized sample vectors would provide a guaranteed larger space that is spanned by the $\sqrt{N}$ sample vectors and it may impact the training time as we are considering the transformation onto this vector in each training iteration. The second is that normalized sample vectors would provide a smaller distribution of values that are more ideal to the numerical properties of optimization methods used to train neural networks like AdaGrad. However, this normalization takes a small amount of time and may not capture some larger effect (e.g., extreme scaling found in matrices from multiphysics problems).

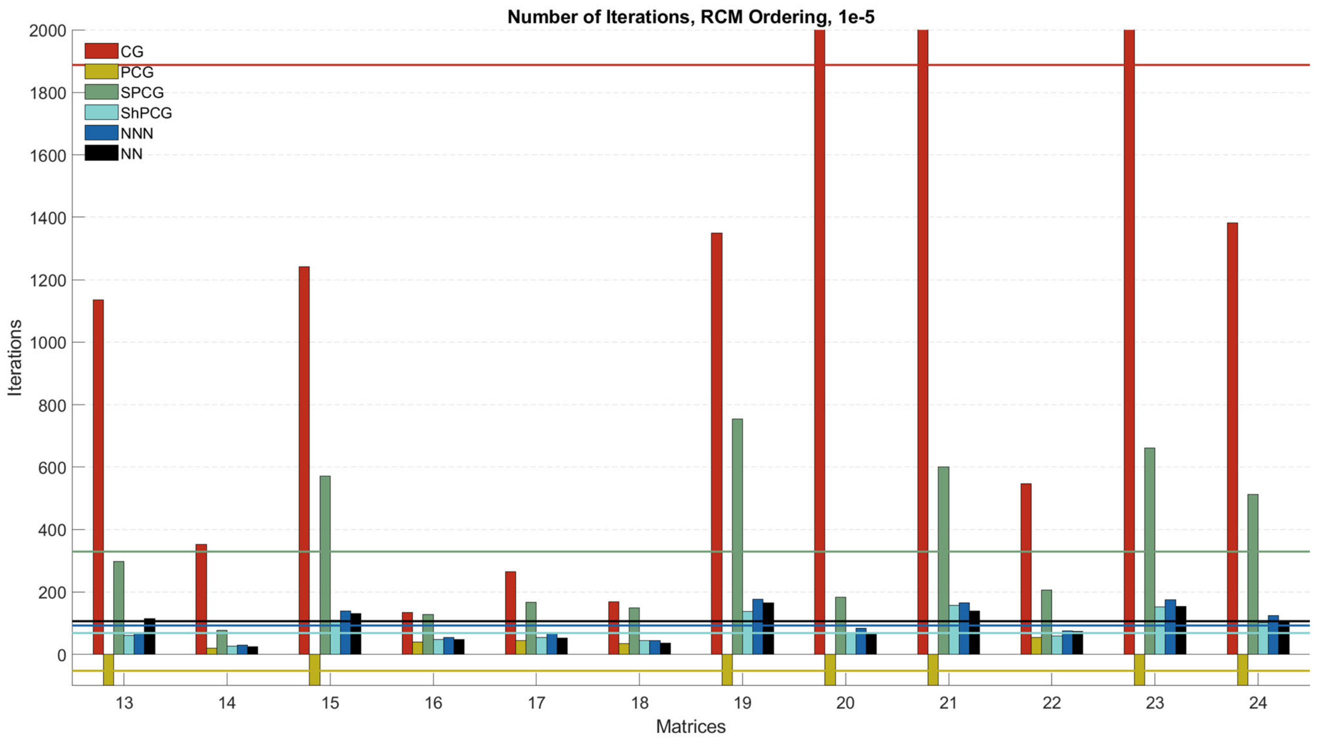
We compare our two models against those shown in Table 3. All PCG methods utilize a *ICHOL*(0), i.e., all preconditioners have the same nonzero pattern as the input matrix. This is also why the number of iterations is a sufficient measure of quality, as the same amount of floating-point operations is done in each iteration for all methods except for `CG`, which does not utilize a preconditioner. Our scaled methods (`SCG` and `ShCG`) utilize a scaling based on the diagonal entries. We used a value of 0.2 for a diagonal shift after exploring multiple others, and found that this works the best on average for the whole test suite. We also consider all these methods for two values of relative tolerance (Tol) for convergence, i.e., 1*e*-5 and 1*e*-7. We tested up to a maximum of 10,000 iterations, though we cut off our figures at 2000 due to space. Lastly, we also consider using the natural and RCM orderings.

Figures 4 and 5 provide the iteration count for each of the methods with natural ordering. Figure 4 provides the iteration counts for converging to a relative tolerance of 1*e*-5, and Fig. 5 provides the same for converging to a relative tolerance of 1*e*-7. In cases where the method would not converge, the number of iterations is reported as -100. The figure also provides lines plotting the average number of iterations for all sparse matrices. In particular, these values are: `CG` $\sim$ 1874; `SCG` $\sim$ 333; `ShCG` $\sim$ 72; `NNN` $\sim$ 95; `NN` $\sim$ 110 for a relative tolerance of 1*e*-5. We mark `PCG` below the line as it fails to converge for almost all sparse matrices, and the average of converging cases provides an inaccurate visual account of its performance. We first notice that our LL^T neural network models (i.e., *NN* and *NNN*) are the only preconditioning method besides `SCG` that converges for all sparse matrices in the test suite. As mentioned in the background (Sect. 2), the development of a preconditioner is as much an art as a science that is guided by expert experience and trials (e.g., `ShCG` failing for sparse matrices 11 and 12 despite being the best method for many of the other sparse matrices). This demonstrates that the use of neural acceleration can help convert this artful practice into a standard function call.

Not only does our neural acceleration method convert it to a simple standard function call that will converge, but it also provides a high-quality preconditioner. The only method that does better in terms of average iteration count than the two neural acceleration-generated preconditioners is the scaled shifted method (`ShCG`). However, this method has its own issues that include not converging for two sparse matrices and having to find an α value that works [10]. For a couple of matrices (i.e., 4 and 9), the LL^T neural network model does better than `ShCG`. However, there are a couple of cases where the neural network models can be worse. These cases include matrices 6 (`vanbody`) and 20 (`StocF-1465`). In the case of matrix 6 (`vanbody`), a model with the non-normalized sample vectors does not do well, i.e., `NN` $> 4,000$ compared to `CG` $> 10,000$ and `ShCG`<600 when the tolerance is 1e-7. However, the model with normalized sample vectors is about on par with `ShCG`. This might indicate a scaling



(a) First Half, NNZ < 5,000k



(b) Second Half, NNZ > 5,000k

◀ **Fig. 6** Number of iterations to converge to a solution when the sparse matrix is ordered in the RCM ordering with a relative tolerance of $1e$-5. The bars represent the raw number of iterations and the lines represent the average iteration for the method across all 24 matrices. We notice that the ordering does not seem to impact the number of iterations required by our method

issue with the optimization method for training. On the other hand, for matrix 20 (`StocF-1465`), one neural network model (`NNN`) does poorly and one does well (`NN`).

When comparing the two neural network methods (i.e., the one trained with normalized samples (`NNN`) and the one trained without normalized samples (`NN`)), there is little difference on average. However, if you look at the performance case by case, you can exclude matrices 6 (`vanbody`) and 13 (`consph`) and notice that `NN` generally does better than `NNN`. This is a nice finding for two particular reasons. First, normalization does not need to be done, thus saving time in training. Second, the training of `NN` is much less sensitive. In particular, a simple parameter of $\alpha = 0.1$ works well with AdaGrad for training, while the training of `NNN` is much more difficult. In the next section, we will compare the time to train the models against the standard methods.

For completeness, we also consider the case when the sparse matrices are reordered with RCM. Figures 6 and 7 present the number of iterations for sparse matrices ordered with RCM. Overall, the iteration counts for the natural and RCM orderings are about the same, even though we know in theory this may not always be true [9]. With RCM ordering, the averages are: `CG` $\sim$ 1889; `SCG` $\sim$ 330; `ShCG` $\sim$ 69; `NNN` $\sim$ 93; `NN` $\sim$ 106. We do note that the time per iteration may be less for sparse matrices ordered with RCM because of their spatial locality in memory [33, 34]. However, this does demonstrate that, for our test suite, the quality of the neural network models seems to depend on ordering. In future work, we will dive deeper into this because of the importance it might have in GPU computations. In particular, many GPUs require formats and orderings that are different from those on multicore CPUs to achieve high performance [34]. If ordering does not matter with these preconditioners, they may be better on GPUs than methods that are dependent on ordering.

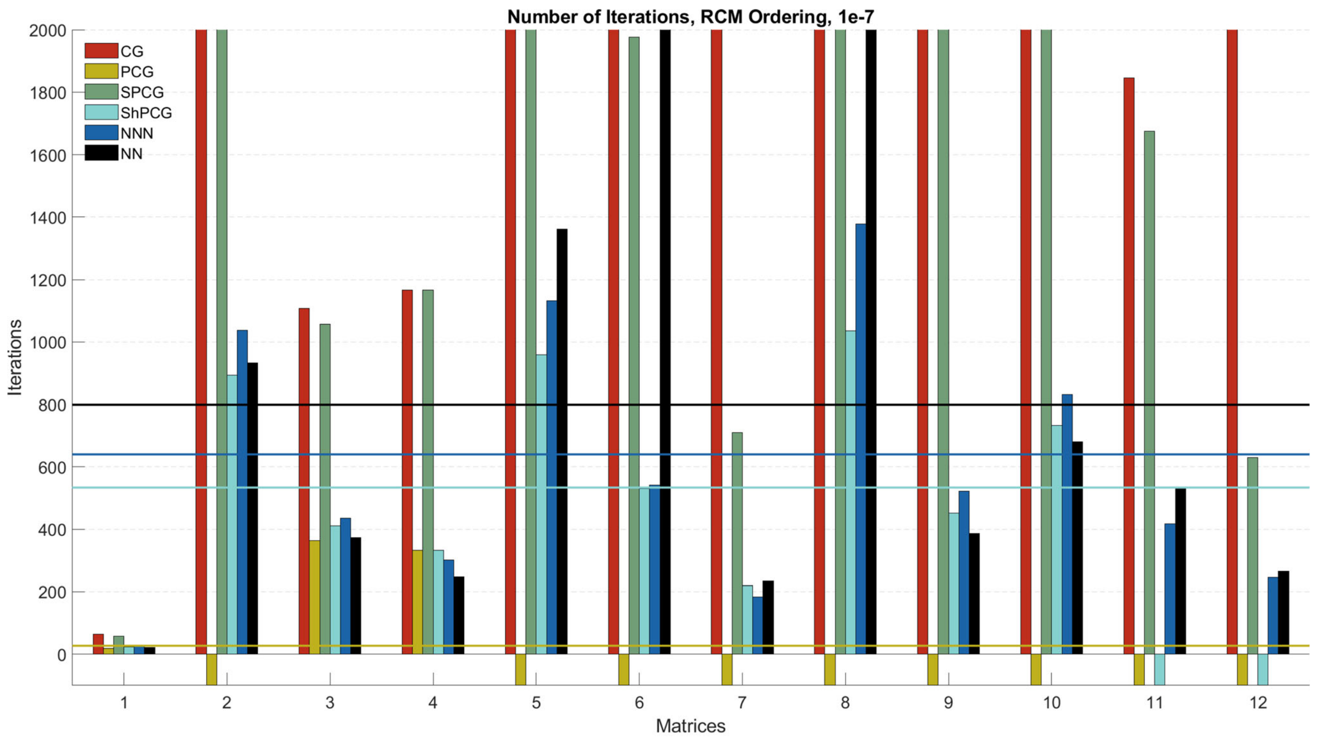
6 Experimental evaluation of cost for LL^T

Next, we evaluate the cost of generating the preconditioners using the neural acceleration method in comparison with the traditional methods. The question is about what is the timing cost of training such a preconditioner. Normally the cost of utilizing a neural network is dominated by two factors: (a) time for training; and (b) time for searching for the correct set of hyperparameters utilized in backpropagation and regularization. Since our neural network model is so simple, a search of the hyperparameter space is not needed, and only training matters. However, this type of training can be very expensive as it requires numerical optimization. We note that many of the current GPU accelerators are optimized for this type of calculation with new progress coming for sparse applications due to graph neural networks. To gauge the cost, we use the following metric:

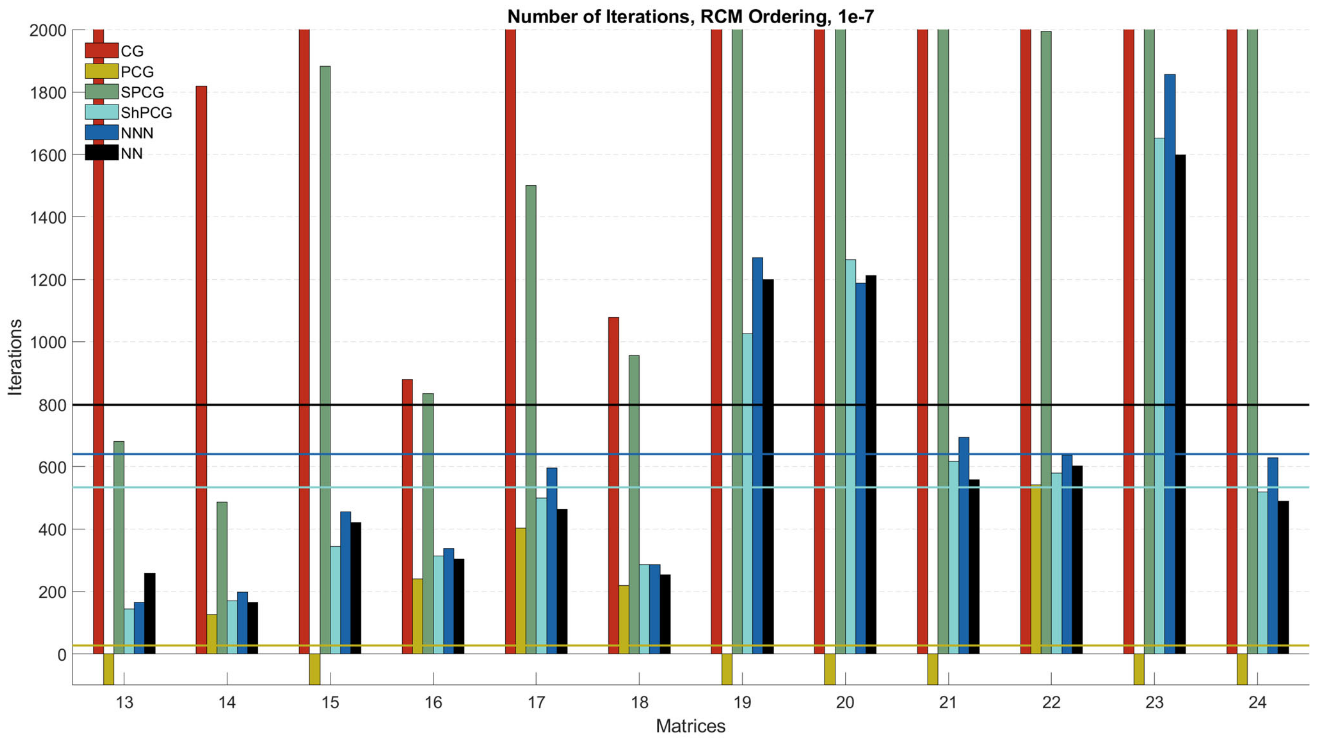
$$NFacts(M) = \frac{TimeNN(M)}{TimeIChol(M)}. \qquad (2)$$

This metric aims to gauge the number of incomplete factorizations for a sparse matrix (M) that can be completed in the time for training the neural network ($TimeNN(M)$) if the time per incomplete factorization is $TimeIChol(M)$. The value of $TimeIChol(M)$ is taken from a modified version of Javelin [3] for the fastest case that will factor without losing numerical stability. The Javelin package is a highly efficient package for incomplete factorization that utilizes threads in a shared memory environment. We have modified Javelin to do $ICHOL(0)$ in place of being designed for incomplete LU. Moreover, we report the time for using four threads with Javelin as this was the largest number of threads that did not suffer from Amdahl's law for all matrices in the test suite. We do not consider the time for moving data in both $TimeNN(M)$ and $TimeIChol(M)$. We justify the use of this metric as follows. The neural acceleration method using our neural networks would only train one model, while someone trying to find a "good" preconditioner utilizing different methods of scaling and parameters for shifting would try multiple preconditioners to match their input case. The metric judges how many of these cases could the user try in the time that our neural acceleration method trains the preconditioner, without user input. Note that there are a couple of factors that may make this metric less than realistic. The first is that a failed incomplete factorization may take less time. The second is that a successful incomplete factorization still might not be as optimal. The only way to test is to apply PCG and observe the number of iterations.

Figure 8 presents the results of using this metric for our test suite. We note that on average the cost is about 69.3 incomplete factorizations. In some cases, e.g., matrix 14 (`G3_circuit`), this value can be much higher. On the other hand, several matrices have much smaller cost, e.g., matrix 1 (`minsurfo`), matrix 11 (`cant`), matrix 12 (`shipsec5`), and matrix 22 (`inline_1`). One factor that seems to dominate this trade-off is the density of the matrix. Since we utilized a less-than-optimal optimizer (i.e., our version of AdaGrad), a great deal more work is



(a) First Half, NNZ $<$ 5,000k



(b) Second Half, NNZ $>$ 5,000k

Fig. 7 Number of iterations to converge to a solution when the sparse matrix is ordered in the RCM ordering with a relative tolerance of $1e$-7. The bars represent the raw number of iterations and the lines represent the average iteration for the method across all 24 matrices. We notice that the ordering does not seem to impact the number of iterations required by our method

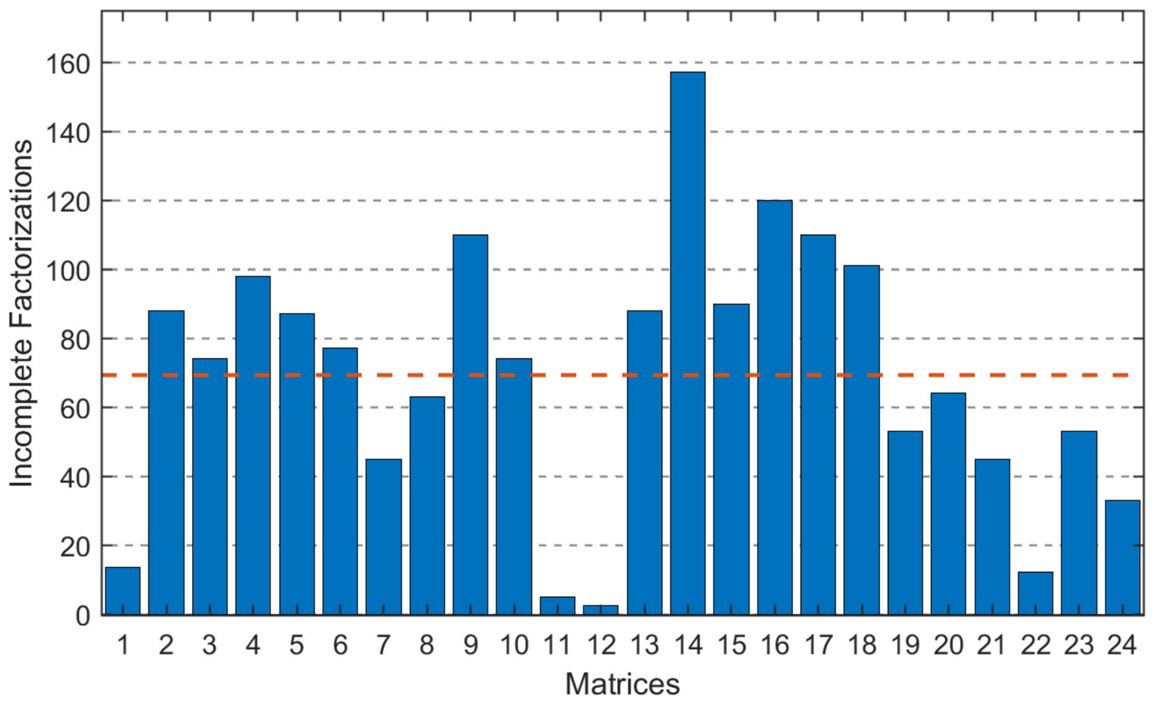


Fig. 8 Evaluation of the cost in terms of the number of factorizations (*NFacts*(*M*)). Each bar represents the value of *NFacts* for the particular sparse matrix and the dotted line represents the average

Table 4 Methods tested for quality for GMRES

Name	Description	Tol
`GMRES`	GMRES	1*e*-5 / 1*e*-7
`ILU`	GMRES with *ILU*(0)	1*e*-5 / 1*e*-7
`IMR`	GMRES with *ILU*(0) preserving row sum	1*e*-5 /1*e*-7
`IMC`	GMRES with *ILU*(0) preserving column sum	1*e*-5 / 1*e*-7
`NN-LU`	GMRES with NN generated *LU*	1*e*-5 / 1*e*-7
`NN-LL`	GMRES with NN generated LL^T	1*e*-5 / 1*e*-7

All but `GMRES` have the same number of floating-point operations per iteration

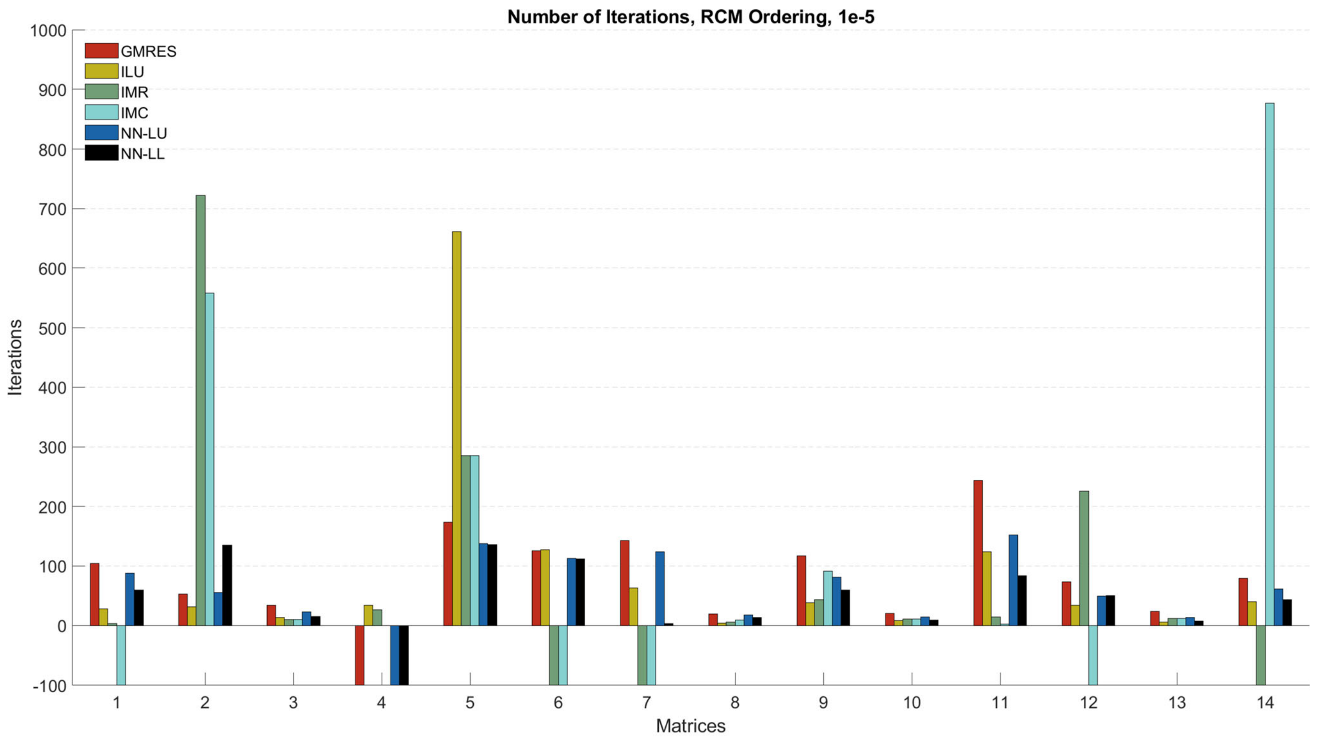
being done for the more sparse cases with the optimizer than for the sparse incomplete factorization. Therefore, the neural acceleration method may consider the density of the input and available hardware (e.g., does the current GPU support sparse tensor operation well?) to determine if a neural network model or the original function call should be used.

Despite many cases where a traditional search method with *ICHOL* could be faster, our method is successful as a neural acceleration method overall. The reason is that a high-quality approximation is found with little to no user interaction in a time that would fit into the compile time of a large scientific application. We note that a large scientific application could take more than 5 min to compile considering large frameworks like Trilinos.[3] We find that the maximum time to generate a preconditioner using our method is relatively small (i.e., ~ 117 second) and the average time is less than a minute (i.e., ~ 53 seconds) using our GPU.

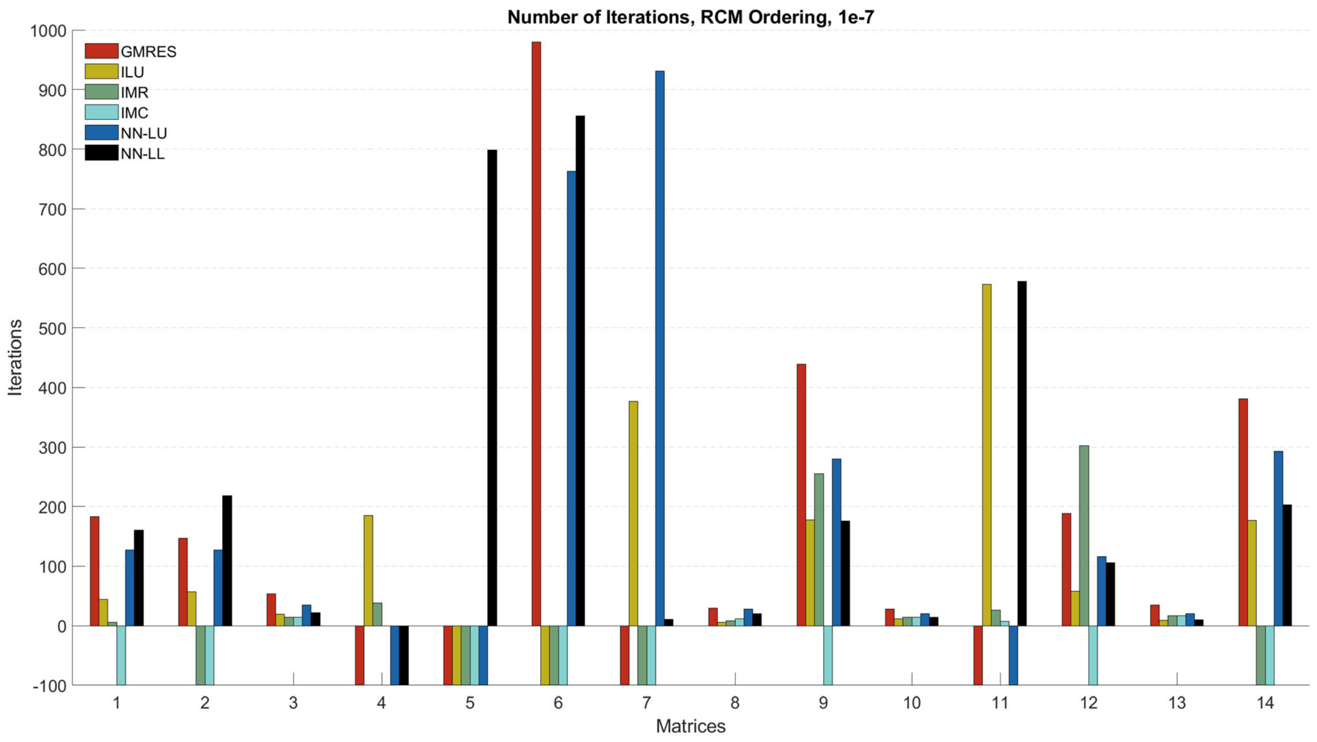
7 Experimental evaluation of quality for *LU*

After understanding that the generation of incomplete LL^T can be done successfully for SPD systems with our neural acceleration method in Sect. 5, we next want to explore this for the more general case of *ILU* utilized by GMRES. However, this is a much more difficult problem due to numerical stability as pointed out previously. Table 4 provides a table of all the methods tested. These methods include the standard *ILU*(0) method along with two common other methods (i.e., `IMR` and `IMC`). These two methods preserve the row (i.e., `IMR`; $Ae = LUe$) and column (i.e., `IMC`; $e^T A = e^T LU$) sum of the incomplete factorization, where e is a vector of all ones. These are found to be better than plain *ILU*(0) in very specific cases. Additionally, we only consider our neural network models where samples are not normalized as we have already shown little difference in this case for LL^T. We test these methods up to 2000 iterations as the dimension of the

[3] https://trilinos.github.io/.



(a) GMRES 1e-5



(b) GMRES 1e-7

Fig. 9 Number of GMRES iterations to converge to a solution when the sparse matrix is ordered in the RCM ordering with relative tolerances of $1e$-5 and $1e$-7. The bars represent the raw number of iterations

matrices in this test suite is smaller due to the number of training parameters in *LU*.

Figure 9 reports the number of iterations for our methods with relative tolerances of 1*e*-5 and 1*e*-7. We do not report averages here as many cases do not converge. Overall the observed performance is more complex and variable than the case of LL^T with SPD matrices. In general, `ILU` is a better preconditioner over our test suite than any others. However, there are a couple of places where `ILU` fails to be the best. In particular, it fails to the reduce iteration count of GMRES (e.g., matrix 5 (`cvxbq1`) which is an SPD optimization problem) and does not do as well as our incomplete LL^T neural network model for matrix 7 (`scircuit`). Our *LU* neural network model does work as a preconditioner as it reduces the iteration count compared to the non-preconditioned `GMRES` case for all matrices. However, there are two interesting cases to point out. The first case is when the input matrix is SPD (e.g., matrices 3, 5, & 11). We noticed that the *LU* model does a fairly good job capturing a preconditioner when the relative tolerance is 1*e*-5, but this becomes worse with the lower relative tolerance of 1*e*-7. This means that model choice is important to match the structure of the initial data, and that training alone cannot make up for picking the wrong model. The second case is that the LL^T model tends to be a better choice than the *LU* model for most of the test suite, even for matrices that are nonsymmetric. This may be due to the simplification of the number of training parameters. There are a couple of places where this is not the case, e.g., matrices 1, 2, & 6. However, for most matrices, the simpler model LL^T trains much faster and provides a better preconditioner. In fact, for matrix 7 (`scircuit`), the LL^T model is the best preconditioner by a wide margin, and the only one to converge for matrix 5 (`cvxbqp1`) with relative tolerance of 1*e*-7.

Based on these observations, we believe that there is merit in using these neural acceleration models as preconditioners for general matrices with GMRES. Interestingly enough, the recommendation would be to first try using the LL^T model and not the *LU* model even if the input is nonsymmetric.

8 Conclusion

In this work, we developed a neural network modeling method for incomplete factorization that can be utilized for the neural acceleration of preconditioners for sparse iterative solver methods of PCG and GMRES. The goal of this method is to produce a good and inexpensive approximation that could be computed at compile time or in parallel on a GPU during execution. These incomplete factorization methods are ideal for neural acceleration to approximate the input matrix *A*. In doing so, we developed a simple two-layer sparse artificial neural network model that utilizes a straightforward implementation of AdaGrad to train. No meta-parameters related to regularization or dropout are needed. As a result, the model is as simple and cheap as expected. In particular, we demonstrated that a model as good as a standard method (i.e., *ICHOL*) could be computed with only $\sqrt{N}$ samples and iterations (i.e., the expected number for this type of training). Not only was the method efficient in training, but it was also the only method that was able to provide a consistent decrease in iteration count for the whole test suite. Additionally, we demonstrate that the time to compute this model is relatively low on GPU (i.e., <1 minute). Moreover, we considered the more general case of *LU* model for GMRES, and we showed that both *LU* and LL^T models work as preconditioners. Surprisingly, we observe that LL^T works well with GMRES even with nonsymmetric sparse matrix. As such, the method works as a black box preconditioner that would be ideal in cases where the application user does not have insight into the problem.

Acknowledgements Thank you to Hu Lee for some of the initial results and funding from NSS-2044633.

Author's contribution JDB was involved in main concept and writing; HS helped in insight on QR, figures, minor writing, initial literature review; TG contributed to initial results related to theoretical problem, background, and figures.

Funding NSF-2044633.

Data availability Sparse matrices are available at the SuiteSparse Matrix Collection (sparse.tamu.edu). The data generated are available from the corresponding author on reasonable request.

Declarations

Conflict of interest There is no conflict of interest.

References

1. Hestenes MR, Stiefel E (1952) Methods of conjugate gradients for solving linear systems. J Res Natl Bureau Stand 49:409–436
2. Saad Y (2003) Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA
3. Booth JD, Bolet G (2020) An on-node scalable sparse incomplete LU factorization for a many-core iterative solver with javelin. Parallel Comput 94–95:102622. https://doi.org/10.1016/j.parco.2020.102622
4. Esmaeilzadeh H, Sampson A, Ceze L, Burger D (2014) Neural acceleration for general-purpose approximate programs. Commun ACM 58(1):105–115
5. Yazdanbakhsh A, Park J, Sharma H, Lotfi-Kamran P, Esmaeilzadeh H (2015) Neural acceleration for gpu throughput

processors. In: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 482–493. https://doi.org/10.1145/2830772.2830810

6. Kiningham K, Levis P, Ré C (2023) Grip: a graph neural network accelerator architecture. IEEE Trans Comput 72(4):914–925
7. Booth JD, Bolet G (2023) Neural acceleration of graph based utility functions for sparse matrices. IEEE Access 11:31619–31635. https://doi.org/10.1109/ACCESS.2023.3262453
8. Saad Y, Schultz M (1986) Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J Sci Stat Comput 7(3):856–869
9. Duff IS, Meurant GA (1989) The effect of ordering on preconditioned conjugate gradients. BIT Numer Math 29:635–657
10. Manteuffel TA (1978) Shifted incomplete cholesky factorization (1)
11. George A, Liu JWH (1978) An automatic nested dissection algorithm for irregular finite element problems. SIAM J Numer Anal 15(5):1053–1069
12. Azad A, Jacquelin M, Buluç A, Ng EG (2017) The reverse Cuthill-Mckee algorithm in distributed-memory. In: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017, pp 22–31. IEEE Computer Society, https://doi.org/10.1109/IPDPS.2017.85
13. Amestoy PR, Davis TA, Duff IS (1996) An approximate minimum degree ordering algorithm. SIAM J Matrix Anal Appl 17(4):886–905
14. Benzi M, Szyld DB, Duin A (1999) Orderings for incomplete factorization preconditioning of nonsymmetric problems. SIAM SISC 20(5):1652–1670
15. Boman EG, Chen D, Hendrickson B, Toledo S (2004) Maximum-weight-basis preconditioners. Numer Linear Algebra Appl 11(8–9):695–721
16. Chen D, Toledo S (2003) Vaidya's preconditioners: implementation and experimental study. Electron Trans Numer Anal 16:30–49
17. Zhang Y, Zhao Z, Feng Z (2020) Sf-grass: solver-free graph spectral sparsification. In: 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp 1–8
18. Hopfield JJ, Tank DW (1985) "neural" computation of decisions in optimization problems. Biol Cybern 52(3):141–152
19. Wang J (1993) Recurrent neural networks for solving linear matrix equations. Comput Math Appl 26(9):23–34. https://doi.org/10.1016/0898-1221(93)90003-E
20. Zhang Y, Li Z, Chen K, Cai B (2008) Common nature of learning exemplified by bp and hopfield neural networks for solving online a system of linear equations. In: 2008 IEEE International Conference on Networking, Sensing and Control, pp 832–836. https://doi.org/10.1109/ICNSC.2008.4525331
21. Cichocki A, Unbehauen R (1992) Neural networks for solving systems of linear equations and related problems. IEEE Trans Circ Syst I: Fund Theory Appl 39(2):124–138. https://doi.org/10.1109/81.167018
22. GÃtz M, Anzt H (2018) Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation. In: SuperComputing, pp 49–56. IEEE, https://doi.org/10.1109/ScalA.2018.00010
23. Chen Y, Davis TA, Hager WW, Rajamanickam S (2008) Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. ACM Trans Math Softw 35(3)
24. Hysom D, Pothen A (2001) A scalable parallel algorithm for incomplete factor preconditioning. SIAM J Sci Comput 22(6):2194–2215. https://doi.org/10.1137/S1064827500376193
25. Chow E, Patel A (2015) Fine-grained parallel incomplete LU factorization. SIAM J Sci Comput 37(2):169–193. https://doi.org/10.1137/140968896
26. Duchi J, Hazan E, Singer Y (2011) Adaptive subgradient methods for online learning and stochastic optimization. J Mach Learn Res 12(61):2121–2159
27. Golub GH, Van Loan CF (1996) Matrix Computations, 3rd edn. Johns Hopkins University Press, USA
28. Davis TA (2011) Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse QR factorization. ACM Trans Math Softw 10(1145/2049662):2049670
29. LeCun Y, Cortes C (1998) The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/
30. Booth JD, Rajamanickam S, Thornquist H (2016) Basker: A threaded sparse LU factorization utilizing hierarchical parallelism and data layouts. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016, pp 673–682. IEEE Computer Society, https://doi.org/10.1109/IPDPSW.2016.92
31. Davis T, Hu Y (2011) The University of Florida sparse matrix collection. ACM Trans Math Softw 10(1145/2049662):2049663
32. Pothen A, Fan C-J (1990) Computing the block triangular form of a sparse matrix. ACM Trans Math Softw 16(4):303–324
33. Kabir H, Booth JD, Raghavan P (2014) A multilevel compressed sparse row format for efficient sparse computations on multicore processors. In: 21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014, pp 1–10. IEEE Computer Society, https://doi.org/10.1109/HiPC.2014.7116882
34. Lane PA, Booth JD (2023) Heterogeneous sparse matrix-vector multiplication via compressed sparse row format. Parallel Comput 115:102997. https://doi.org/10.1016/J.PARCO.2023.102997