Neural incomplete factorization: learning preconditioners for the conjugate gradient method

Paul Häusner[1], **Ozan Öktem**[2], **Jens Sjölund**[1]
[1]Department of Information Technology, Uppsala University, Sweden
[2]Department of Mathematics, KTH Royal Institute of Technology, Stockholm, Sweden
`{paul.hausner,jens.sjolund}@it.uu.se`, `ozan@kth.se`

Abstract

In this paper, we develop a novel data-driven approach to accelerate solving large-scale linear equation systems encountered in scientific computing and optimization. Our method utilizes self-supervised training of a graph neural network to generate an effective preconditioner tailored to the specific problem domain. By replacing conventional hand-crafted preconditioners used with the conjugate gradient method, our approach, named neural incomplete factorization (`NeuralIF`), significantly speeds-up convergence and computational efficiency. At the core of our method is a novel message-passing block, inspired by sparse matrix theory, that aligns with the objective to find a sparse factorization of the matrix. We evaluate our proposed method on both a synthetic and a real-world problem arising from scientific computing. Our results demonstrate that `NeuralIF` consistently outperforms the most common general-purpose preconditioners, including the incomplete Cholesky method, achieving competitive performance across various metrics even outside the training data distribution.

1 Introduction

Solving large-scale systems of linear equations is a fundamental problem in computing science. Available solving techniques can be divided into *direct* and *iterative* methods. While direct methods, which rely on computing the inverse or factorizing the matrix, obtain accurate solutions they do not scale well for large-scale problems. Therefore, iterative methods, which repeatedly refine an initial guess to approach the exact solution, are used in practice when an approximation of the solution is sufficient [16].

The conjugate gradient method is a classical iterative method for solving equation systems of the form $Ax = b$ where the matrix A is symmetric and positive definite. Problems of this form arise naturally in the discretization of elliptical PDEs, such as the Poisson equation, and a wide range of optimization problems, such as quadratic programs and primal-dual interior-point methods [26, 27]. The convergence speed of the method thereby depends on the condition number $\kappa(A)$ of the input matrix A [16]. Therefore, it is common practice to improve the spectral properties of the equation system before solving it. This is typically achieved by preconditioning, where the original system of equations is multiplied with a preconditioning matrix to improve its spectral properties. An approximate inverse of the matrix, $P^{-1} \approx A^{-1}$, is computed and the original equation system is replaced by $P^{-1}Ax = P^{-1}b$, which aims to improve the convergence properties [9]. Despite the critical importance of preconditioners for practical performance, it has proven notoriously difficult to find good designs for a general class of problems [28].

Recent advances in data-driven optimization combine classical optimization algorithms with data-driven approaches. The idea is to replace hand-crafted heuristics with methods parameterized by neural networks that are trained against data [3, 10]. In this work, we are combining these data-driven methods with results known from sparse matrix theory by representing the matrix of the linear equation system as a graph that forms the input to a graph neural network. We then train the network to predict a suitable preconditioner for a given problem with our neural network.

To train our model in a computationally feasible way we are inspired by the incomplete factorization methods. These methods, which include the popular incomplete Cholesky factorization, compute an approximation of the Cholesky factor of the matrix A that can then be used to precondition the linear system [16]. However, these methods can suffer from breakdown and only work well for a small class of problems [6]. Our proposed neural incomplete factorization method (`NeuralIF`) overcomes these limitations by combining the idea of incomplete factorization with data-driven methods. In contrast to the NeuralPCG model, which uses a simple encoder-decoder graph neural network [24], our proposed preconditioner utilizes insight from graph theory in order to motivate the underlying computational framework and align with the objective of our training. This allows us to reduce the number of parameters in the model significantly and reduce inference time, which is critical to amortize the cost of computing the preconditioner. In Figure 1, we can see that the learned preconditioner is able to significantly reduce the number of iterations required for the conjugate gradient method to converge to a given target. Even though it takes additional time to compute the preconditioners (not shown in Figure 1) it can help to reduce the total time required to solve the problem.

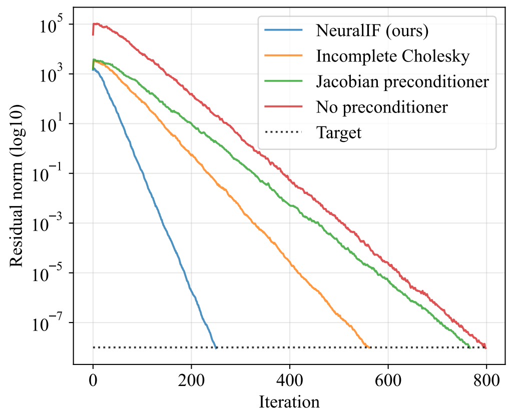


Figure 1: Convergence behaviour of the conjugate gradient method with different preconditioners on a synthetic problem of size $n = 5\,000$.

2 Background

The proposed `NeuralIF` preconditioner is utilized in the conjugate gradient method which is introduced in the following. We then introduce the basics of graph neural networks which form the computational backend for our method.

2.1 Conjugate gradient method

The conjugate gradient method (CG) is a well-established iterative method for solving symmetric positive-definite (spd) systems of linear equations of the form $Ax = b$. The algorithm does not require any matrix-matrix multiplications, making CG particularly effective when dealing with sparse matrices. The method creates a sequence of search directions d_i which are orthogonal in A-norm to each other i.e. $d_i^\top A d_j = 0$ for $i \neq j$. These search directions are used to update the solution iterate x_k. Since it is possible to compute the optimal step size in closed form for each search direction, the method is guaranteed to converge [30].

Convergence The convergence of CG to the optimal solution $x_\star$ depends on the spectral properties of the matrix A. Using the condition number $\kappa(A)$, a linear bound of the error in the number of conjugate gradient steps k is given by

$$||x_\star - x_k||_A \leq 2\,||x_\star - x_0||_A \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k . \tag{1}$$

However, in practice the CG method often converges significantly faster and the convergence also depends on the distribution of eigenvalues and the initial residual. Clustered eigenvalues lead to faster convergence [9]. A common approach to speed up the convergence is to precondition the system to improve its spectral properties.

Preconditioning The underlying idea of preconditioning is to compute a cheap approximation of the inverse of A which is utilized to improve the convergence properties. There are two common ways to achieve this. First, it is possible to directly approximate the inverse of the matrix $M \approx A^{-1}$ called sparse approximate inverse methods. Another possibility is to approximate the original matrix A instead $P \approx A$ while restricting the obtained preconditioner P to be easily invertible. This can for example be achieved by finding a triangular preconditioner which forms an incomplete factorization of matrix A [6]. Given a preconditioner in such a split form, it can be directly applied in the CG method as shown in Algorithm 1.

Finding a good preconditioner requires a trade-off between time required to compute the preconditioner and the resulting speed-up in convergence. A simple way to construct a preconditioner called Jacobian preconditioner is to approximate A with a diagonal matrix. The incomplete Cholesky preconditioner is a more advanced method which is a widely adopted incomplete factorization method. As the name suggests, the idea is to approximate the Cholesky decomposition $A = LL^\mathsf{T}$. The IC(0) preconditioner restricts the non-zero elements in L to non-zero elements in A. Thus, no fill-ins in the factorization are allowed. More general versions allow additional fill-ins of the matrix based on the position or the value of the matrix elements [6].

Finding new preconditioners is an active research area but is often done on a case-by-case basis. Thus, newly developed methods are often tailor-made for specific problem classes and often do not generalize [6, 26]. For PDE problems, for example, incomplete Cholesky and algebraic multigrid methods are widely established [21] while for network flow problems preconditioners based on the underlying spanning tree have been developed [27].

Stopping criterion In practice, due to numerical issues, the residual is only approaching but never reaching zero. Further, the true solution $x_\star$ is not available and therefore, Equation (1) can not be used as a stopping criterion for the algorithm either. Instead, the relative residual error $||r||_A^2$ which is computed recursively in line 7 of Algorithm 1 is widely adopted [9].

Algorithm 1 Split preconditioned conjugate gradient method [28]

```
1: Input: System of linear equations $A \in S_n^{++}, b \in \mathbb{R}^n$, Preconditioner $P = LL^\mathsf{T} \approx A$
2: Output: Solution to the linear equation system $x_\star$
3: Initialize starting guess $x_0$, compute residual $r_0 = Ax_0 - b$, $\hat{r}_0 = L^{-1}r_0$, search direction
   $p_0 = L^{-\mathsf{T}}\hat{r}_0$
4: for $k = 0, 1, \dots,$ until convergence do
5:     $a_k = \langle \hat{r}_k, \hat{r}_k \rangle / \langle Ap_k, p_k \rangle$            ▷ determine step size
6:     $x_{k+1} = x_k + a_k p_k$            ▷ update approximate solution iterate
7:     $\hat{r}_{k+1} = \hat{r}_k - a_k L^{-1} A p_k$            ▷ iteratively compute residual
8:     $\beta_k = \langle \hat{r}_{k+1}, \hat{r}_{k+1} \rangle / \langle \hat{r}_k, \hat{r}_k \rangle$            ▷ compute orthogonalization coefficient
9:     $p_{k+1} = L^{-\mathsf{T}} \hat{r}_{k+1} + \beta_k p_k$            ▷ determine next search direction
10: end for
```

2.2 Graph neural networks

A graph $G = (V, E)$ is a tuple consisting of a set of nodes V and a set of edges E connecting the nodes in the graph $E \subseteq V \times V$. We assign every node $v \in V$ a node feature vector $x_v \in \mathbb{R}^n$ and respectively every edge e_{ij}, connecting nodes i and j, an edge feature vector $z_{ij} \in \mathbb{R}^m$. Mathematically, these features are functions with domain E and co-domain $\mathbb{R}^n$ which map $x : v \mapsto x_v$. With slight abuse of notation we refer to these directly as x_v, instead of $x(v)$. Analogously, we refer to the edge features as z_{ij} directly, instead of explicitly writing $z(e_{ij})$.

Graph neural networks (GNN) belong to an emerging family of neural network architectures that are well-suited to many real-world problems which possess a natural graph structure [34]. GNNs consist of multiple layers updating the node and edge feature vectors iteratively using permutation equivariant aggregations over the neighborhoods and learned update functions. The exact type of aggregation used and the updates determines the exact type of GNN flavor [4].

For a simple GNN, the updated edge features in layer l of the network are computed by

$$z_{ij}^{(l+1)} = \phi_{\theta_z^{(l)}} \left(z_{ij}^{(l)}, x_i^{(l)}, x_j^{(l)} \right), \tag{2}$$

where ϕ is a parameterized function. Next, for each node the features from the neighboring edges are aggregated using a suitable aggregation function. Typical choices include sum, mean and max aggregations. Any such permutation invariant aggregation function is denoted by $\oplus$. The aggregation of edge features over the neighborhood $\mathcal{N}$ of node n_i is computed as

$$m_i^{(l+1)} = \bigoplus_{j \in \mathcal{N}(i)} z_{ji}^{(l+1)}, \tag{3}$$

where the neighbourhood of node i is defined as $\mathcal{N}(i) = \{j | (i,j) \in E\}$. Note that the neighbourhood structure is typically kept fixed in GNNs and no edges are added or removed between the different layers. The final step of the GNN scheme is updating the node features as

$$x_i^{(l+1)} = \psi_{\theta_x^{(l)}} \left(x_i^{(l)}, m_i^{(l+1)} \right). \tag{4}$$

The parameterized update functions ϕ and ψ are typically parameterized using shallow, fully-connected neural networks which parameters are denoted by θ. The equations (2)–(4) describe the iterative scheme of message passing that is implemented by the GNN. By choosing a permutation equivariant function in the neighborhood aggregation step (3) and due to the fact that the update functions only act locally on the node and edge features, the learned function represented by the GNN itself is permutation equivariant and can handle inputs of varying sizes [4]. Furthermore, GNNs have shown to align with many known algorithms making them a common choice to use in a wide range of problems in a sense that they share a common structure with the underlying computations executed. This is often referred to as algorithmic alignment [14].

3 Method

Learning problem Our goal is to learn a mapping $f_\theta : S_n^{++} \to S_n^{++}$ that takes a spd matrix A and predicts a suitable preconditioner P which improves the spectral properties of the system – and therefore also the convergence behavior of the conjugate gradient method.

In order to ensure convergence, the output of the learned mapping further needs to be spd and is in practice often required to be sparse due to resource constraints. In order to ensure these properties of the output, we restrict the mapping to lower triangular matrices with strictly positive elements on the diagonal denoted as $\Lambda_\theta(\cdot)$. This can be achieved by using a suitable activation function and network architecture as discussed later. Another advantage of mapping to a lower triangular matrix is that the matrix is guaranteed to be invertible if the diagonal elements are non-zero. The inversion of the output is, furthermore, computationally easy using forward-backward substitution. The final output of our parameterized function is given by

$$P = \Lambda_\theta(A)\Lambda_\theta(A)^\mathsf{T}. \tag{5}$$

In order to improve the convergence properties we aim to improve the condition number of the preconditioned system. However, the computation for the condition number is scaling very poorly for large problems as it requires $\mathcal{O}(n^3)$ floating-point operations. Therefore, we can not optimize the spectral properties of the matrix directly. Instead, we are inspired by incomplete factorization methods [6]. These methods try to find a sparse approximation of the Cholesky factor of the input matrix while remaining computationally tractable. In order to learn a good approximation model we assume matrices of interest are generated by a matrix-valued random variable $\mathbb{A}$ with an unknown distribution describing the underlying class of problems.

Our objective mimicking the sparse factorization methods with no fill-ins is encoded into an optimization problem as

$$\hat{\theta} \in \underset{\theta}{\operatorname{argmin}}\ \mathbb{E}_{A \sim \mathbb{A}} \left[\|\Lambda_\theta(A)\Lambda_\theta(A)^\mathsf{T} - A\|_F \right] \tag{6}$$

$$\text{s.t. } \Lambda_\theta(A)_{ij} = 0 \text{ for all } i,j \text{ where } A_{ij} = 0. \tag{7}$$

Equation (6) aims to minimize the distance between the learned factorization and the original input matrix using the Frobenius norm. It is, however, also possible to use a different distance metric with this formulation. When considering more advanced preconditioners, the sparsity constraint (7) can be relaxed slightly allowing more non-zero elements in the preconditioner. However, it is desirable that the required storage for the preconditioner is known beforehand. A practical problem with objective (6) is that we cannot compute the expectation since we lack access to the distribution of $\mathbb{A}$. On the other hand, we have access to training data $A_1, A_2, \ldots, A_n \sim \mathbb{A}$, so we consider the empirical counterpart of equation (6) – empirical risk minimization – where the intractable expected value is replaced by the sample mean which we can optimize to obtain an approximation of $\hat{\theta}$.

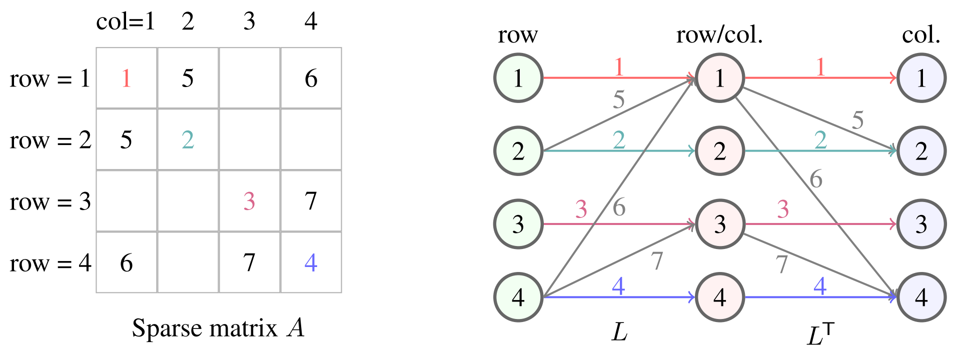


Figure 2: Sparse matrix A and corresponding unrolled graph representation extending the well known Coates graph. Here, the matrix is split into lower and upper triangular parts representing the matrix multiplication LL^T in graph form.

Model architecture Linear equation systems are inherently permutation equivariant and the size of the matrix is variable depending on the underlying problem. Therefore, it is a natural choice to parameterize the function Λ_θ with a graph neural network. Furthermore, the sparsity constraints shown in equation (7) can be implicitly build in the GNN model architecture and therefore do not require additional care.

On a high level, we interpret the matrix A of the input problem as the adjacency matrix of the corresponding graph. Thus, the nodes of the graph represent the columns/rows – the system is symmetric – of the input matrix and are augmented with an additional feature vector. The edges of the corresponding graph are the non-zero elements in the adjacency matrix. This transformation from a matrix to a graph is known as Coates graph [17].

In order to align our message-passing framework to the underlying objective to find a sparse factorization, we introduce a novel message-passing block for graph neural networks which extends the Coates graph representation to execute the computations. The proposed graph structure consisting of two message-passing steps is shown in Figure 2. Instead of using matrix A directly as the underlying scheme for message passing, we only use the lower triangular part in the first step to update node and edge features. In the second step, the message-passing scheme is executed over the upper triangular part of the matrix A. However, the edge features between the two consecutive layers are shared i.e. $z_{ij}^{(n)} = z_{ji}^{(n)}$. After the two message-passing steps, the original matrix A is used to augment the edge features by introducing skip connections. In Figure 2 the unrolled architecture for one block is depicted. The implementation of the block shares the node features across the layer leading to an equivalent Coates graph representation with two different sets of edges L and L^T. For each set of edges a message-passing step in the graph block is added which consists of the node update, neighborhood aggregation, and edge update as described in Section 2.2.

The underlying idea of the message-passing block is the fact that matrix multiplication can be represented in matrix format by concatenating the graph representations of the two factors. Each element in the product matrix can then be computed as the sum of the weights connecting the two corresponding nodes in the concatenated graphs. Furthermore, the skip connections are introduced as they represent an element-wise addition of the two matrices [13, 7].

Our `NeuralIF` model consists of several of these message passing blocks. The final output of our model is then transformed via

$$(\Lambda(A))_{ij} = \begin{cases} \exp(z_{ii}^{(N)}), & \text{if } i = j, \\ z_{ij}^{(N)}, & \text{if } i < j, \end{cases} \tag{8}$$

in order to enforce the positive definiteness of the learned preconditioner, as the activation forces the diagonal elements to be strictly positive. Here, $z_{ij}^{(N)}$ is the edge embedding obtained from the message passing scheme after N such steps. The output is then used to train the network using the empirical counterpart of the training objective (6).

Inference and amortization When the trained model is used to generate preconditioners, the lower triangular output $\Lambda(A)$ is inverted using forward-backward substitution which can be implemented efficiently. The resulting inverse of the lower triangular matrix $\Lambda(A)^{-1}$ is the approximate inverse Cholesky factor and then amortized in the CG method as shown in Algorithm 1. The implementation details can be found in the appendix.

4 Results

All experiments reported are run on a Intel Core i7-11850H CPU @ 2.5GHz in order to ensure a fair comparison in computation time for the different preconditioners. In practice, it is possible to further accelerate our proposed `NeuralIF` method if specialized hardware such as GPUs are used.

The overall goal is to reduce total computational time required to solve the problem up to a given precision measured by the residual as described in Section 3. The time used to compute the preconditioner beforehand therefore needs to be traded-off with the achieved speed-up through the usage of the preconditioner. Therefore, we compare the different methods based on both the time required to compute the preconditioner (P-time) and time needed to solve the preconditioned linear equation system (CG-time) which is directly related to the number of iterations required. The preconditioned conjugate gradient method shown in Algorithm 1 requires additional matrix multiplications compared to the non-preconditioned version of CG. Thus, the preconditioner can only be amortized when the linear system becomes significantly easier to solve. Golub and Van Loan [16] measure how useful a preconditioner is by the strength of the inequality

$$\text{P-time} + (\text{PCG time-per-iteration} \cdot \text{PCG iterations}) < \text{CG time-per-iteration} \cdot \text{CG iterations}. \quad (9)$$

We test our `NeuralIF` method on two different datasets. The first dataset consists of synthetically generated problems. The other one arises from a real-world application in scientific computing. The details for the dataset generation as well as the implementation details of the method can be found in the appendix.

4.1 Synthetic problems

The results for different preconditioners on synthetically generated problems of size $n = 5\,000$ are shown in Table 1. We can see that the `NeuralIF` preconditioner is able to outperform the other methods significantly not only in terms of solving time but also regarding the condition number obtained from the preconditioned system. It is noticeable that neither the Jacobian nor the incomplete Cholesky method are able to outperform the standard CG method even though the preconditioning reduces the condition number and the number of required iterations. This is due to the fact that the cost per iteration is also increased when using a preconditioner as shown in the inequality (9). The time required to compute the preconditioner is comparable for incomplete Cholesky and our learned method and the same sparsity pattern in the preconditioning matrix is obtained. Overall, `NeuralIF` outperforms the other preconditioning methods clearly across all important metrics in our experiments. Figure 3, displays the distribution of solving times for the different preconditioned

Table 1: Results for the synthetic dataset for $n = 5\,000$. First column lists the condition number of the (preconditioned) system, the second the preconditioner's sparsity. Remaining columns list performance-related figures for the preconditioned conjugate gradient method: 4th column lists the computation time for the preconditioner (P-time), 5th lists the time for finding the solution and the number of iterations required running the preconditioned conjugate gradient method (CG-time/Iters). All times are in seconds. Arrows indicate whether higher/lower is better.

Preconditioner	$\kappa \downarrow$	Sparsity $\uparrow$	P-time $\downarrow$	CG-time (Iters) $\downarrow$	Total time $\downarrow$
None	12 171.70	-	-	5.62 (846.65)	5.62
Jacobian	9 949.49	**99.98%**	**0.01**	8.50 (681.26)	8.51
IC(0)	7 094.39	93.24%	0.87	7.00 (575.95)	7.86
NeuralIF (ours)	**1 566.17**	93.24%	0.89	**3.37 (264.92)**	**4.25**

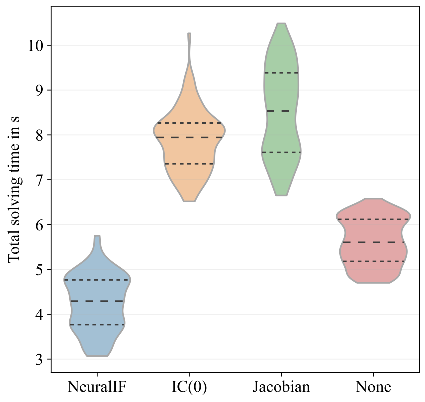


Figure 3: Performance comparison of the conjugate gradient method for different preconditioners on the test set consisting of 100 samples of synthetic problems of size $n = 5\,000$. Here shown is the total time required (preconditioner time and conjugate gradient time combined). The dotted and dashed lines are indicating the quartiles.

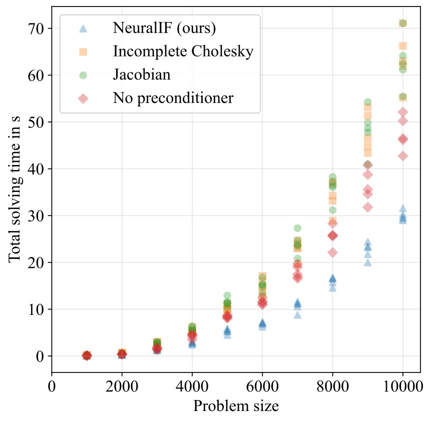


Figure 4: Performance comparison in terms of total solving time of different preconditioners for synthetic samples outside of the training domain. Both the dimension as well as the condition number of the system are changed in comparison to the training data. The `NeuralIF` preconditioner is trained on samples of size $n = 5\,000$.

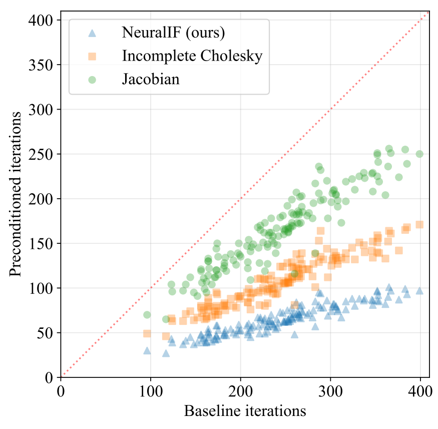
conjugate gradient runs for 100 samples. We can see that `NeuralIF` is the only method able to increase overall performance compared to the non-preconditioned version.

It is important to note that the synthetically generated matrices all have the same size and a similar condition number due to the data generation process leading to a similar performance across the different problem instances which can be seen from the small variance in solving time in Figure 3. In Figure 4, we compare the total time required to solve the problem for different problem sizes and conditioning to show the generalization abilities of our method. We can see that also on samples outside of the training domain, the `NeuralIF` preconditioner remains superior to the other tested preconditioners leading to the fastest convergence time of the method for all tested samples. For a number of problems shown in Figure 4, the incomplete Cholesky method suffers from breakdown where no suitable preconditioner can be obtained from the method due to numerical problems with very ill-conditioned matrices. This further decreasing the performance of the method. Our learned preconditioner does not suffer from these issues.

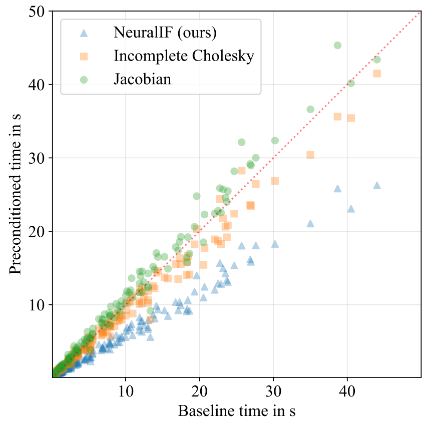
4.2 Poisson equation

Discretizing elliptical partial differential equations such as the Poisson equation using the finite-element method leads to large-scale linear equation systems which are spd and usually very sparse. We generate a dataset of such PDE problems by changing the underlying domain sampling different convex and non-convex 2D grids on which we are solving the boundary value problem arising from the Poisson equation [22].

The results for solving the PDE problems are shown in Figure 5. We can see that all tested preconditioners are able to reduce the number of iterations required to solve the problem. However, in terms of solving time, the Jacobian preconditioner is slightly slower than the baseline CG method. This is due to the pre-compute time as well as the higher cost per iteration since additional matrix-vector multiplications are required each iteration. The incomplete Cholesky method is able to further reduce the number of required iterations and is slightly faster on the test samples which require long solving times. Our learned `NeuralIF` preconditioner is able to reduce both the number of iterations as well as the required solving time significantly both compared to the baseline as well as the other preconditioners.



(a)



(b)

Figure 5: Effect of different preconditioners on the number of required CG iterations (a) and total time (P-time and CG-time) required to solve the problem (b) in comparison to the CG method without any preconditioner. Here shown are 150 samples arising from the discretization of the Poisson PDE problem. Lower-right corner is better.

5 Discussion

One limitation of our proposed method is that additional time is required to train the model before it can be used to accelerate convergence. However, this can be done in an offline fashion and once the model is trained new problems can benefit from the time investment. Furthermore, the typically required manual work by hand-engineering the method is replaced by the neural network training. While the method works well in practice, no guarantees about the convergence speed can be made and it is likely that for some problems the usage of the learned preconditioner leads to an increase in overall solving time. However, as seen in the experiments also classical preconditioners suffer from this problem. Further, for problems where preconditioners can be designed based on domain insights – such as for some PDE problems – the hand-engineered approaches can outperform the general purpose learned approach but require knowledge about the problem domain and manual engineering to design a good preconditioner with the available information.

The `NeuralIF` preconditioner assumes that the input matrix itself is spd. Many real-world problems however do not satisfy this assumption and are therefore solved using other iterative methods such as the GMRES method. In order to be able to utilize the learned preconditioners in these methods additional changes need to be made to the network architecture taking into account the specifics of the underlying solving techniques. While our method is inspired by the class of incomplete factorization methods, another promising approach could be obtaining a learned sparse approximation inverse preconditioner [6]. Furthermore, our approach is limited by the fact that, like the IC(0) preconditioner, only non-zero elements in the original input matrix are considered in equation (7). Extending the current approach to learned preconditioners with additional fill-ins is a promising but also challenging direction because of the problem's inherently discrete structure.

6 Related work

Data driven-optimization or "learning to optimize (L2O)" is an emerging field aiming to accelerate optimization methods by combining them with data-driven techniques [2, 10]. This methodology can be utilized to directly approximate the solution to an optimization problem [17] or to replace some hand-crafted heuristic within a known optimization algorithm [5]. `NeuralIF` follows this latter approach. Graph neural networks form the computational backend of our method and have recently emerged as a novel and promising architecture for a wide range of problems [34].

L2O using GNNs Graph neural networks have recently also been recognized as a suitable computational backend for problems in data-driven optimization. Chen et al. [11] study the expressiveness of GNNs with respect to their power to represent linear programs on a theoretical level and further extend this setting to non-convex mixed-integer linear programs [12]. Here, the graph encodes both objective function and additional constraints as a bipartite graph while for linear equation systems it is sufficient to only encode a single matrix in the graph structure. Using the Coates graph representation, Grementieri and Galeone [17] develop a sparse linear solver for linear equation systems. They represent the linear equation system as the input to a graph neural network which is trained to approximate the solution to the equation system directly. However, no guarantees for the solution can be obtained. In contrast, our method benefits from the convergence properties of the preconditioned conjugate gradient method, since we ensure that the learned preconditioner is spd by an appropriate choice of activation function.

Following an AutoML approach, Tang et al. [33] instead try to predict a good combination of solver and preconditioner from a predefined list of techniques that are readily available for general linear equation systems. The training is executed based on a supervised loss which relies on solving the equation systems in the training data with all available combinations of techniques. The NeuralIF preconditioner instead focuses on further accelerating the existing CG method and does not need any explicit supervision during training. Sjölund and Bånkestad [31] use the König graph representation instead to accelerate low-rank matrix factorization algorithms by representing the matrix multiplication as a concatenation graph. However, their architecture utilizes a graph transformer while our approach works in a fully sparse setting, which avoids the scalability issues of transformer architectures and makes it better suited for large-scale problems.

Learned CG Learned approachs have also been applied to the conjugate gradient method previously. Kaneda et al. [20] suggest to replace the search direction in the conjugate gradient method by the output of a neural network. This approach can, however, not be integrated with existing solutions for accelerating the CG method and does not allow further improvements through preconditioning. Furthermore, in order to ensure convergence all previous search directions need to be saved and the full Gram–Schmidt orthonormalization needs to be computed in every iteration. Our method does not suffer from these problems.

There have also been some earlier approaches to learning preconditioners for the conjugate gradient method following similar ideas as our proposed `NeuralIF` method. Ackmann et al. [1] use a fully-connected neural network to predict the preconditioner for climate modeling using a supervised loss. Sappl et al. [29] use a convolutional neural network (CNN) in order to learn a preconditioner for applications in water engineering by optimizing the condition number directly. However, both of these approaches are only able to handle small-scale problems and their architecture and training is limited due to their poor scailability compared to our suggested approach. Similarly, Götz and Anzt [19] try to predict the sparsity pattern for the block-Jacobian preconditioner using a convolutional neural network (CNN) which limits the usability as only problems suited for this class can benefit from this approach. Our learned method is far more general and can be applied to a significantly larger class of problems. Extending the sparsity pattern prediction using CNNs to general incomplete factorization methods did not lead to performance gains [32]. Even though CNNs are widely adopted, they are not well-suited to represent the underlying problem in contrast to GNNs utilized by `NeuralIF`.

7 Conclusions

In this paper we introduce `NeuralIF`, a novel data-driven preconditioner for the conjugate gradient method to accelerate solving large-scale linear equation systems. Our method learns an approximation of the sparse Cholesky factor of the input matrix following the widespread idea of incomplete factorization preconditioners [6]. We use the matrix A as an input to a graph neural network which aligns with the objective to minimize the distance between the learned Cholesky factor and the input matrix A using the Frobenius norm as a distance measure. Our experiments show that the proposed method is outperforming general purpose preconditioners both on synthetic and real-world problems. Our work shows the large potential of data-driven techniques in combination with numerical optimization and the usefulness of graph neural networks as a natural computational backend for problems from linear algebra.

Acknowledgments and Disclosure of Funding

This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We would like to thank Daniel Gedon and Fredrik K. Gustafsson for their valuable feedback.

References

[1] Jan Ackmann, Peter D Düben, Tim N Palmer, and Piotr K Smolarkiewicz. Machine-learned preconditioners for linear solvers in geophysical fluid flows. *arXiv preprint arXiv:2010.02866*, 2020.

[2] Brandon Amos. Tutorial on amortized optimization for learning to optimize over continuous domains. *Foundations and Trends in Machine Learning*, (accepted for publication), 2023.

[3] Sebastian Banert, Jevgenija Rudzusika, Ozan Öktem, and Jonas Adler. Accelerated forward-backward optimization using deep learning. *arXiv preprint arXiv:2105.05210*, 2021.

[4] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[5] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[6] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[7] Richard A Brualdi and Dragos Cvetkovic. *A combinatorial approach to matrix theory and its applications*. CRC press, 2008.

[8] Chen Cai and Yusu Wang. A simple yet effective baseline for non-attributed graph classification. *ICLR Workshop: Representation Learning on Graphs and Manifolds*, 2019.

[9] Erin Carson, Jörg Liesen, and Zdeněk Strakoš. 70 years of Krylov subspace methods: The journey continues. *arXiv preprint arXiv:2211.00953*, 2022.

[10] Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *arXiv:2103.12828*, 2021.

[11] Ziang Chen, Jialin Liu, Xinshang Wang, Jianfeng Lu, and Wotao Yin. On Representing Linear Programs by Graph Neural Networks. *arXiv:2209.12288*, 2022.

[12] Ziang Chen, Jialin Liu, Xinshang Wang, Jianfeng Lu, and Wotao Yin. On Representing Mixed-Integer Linear Programs by Graph Neural Networks. *arXiv preprint arXiv:2210.10759*, 2022.

[13] Michael Doob. Applications of graph theory in linear algebra. *Mathematics Magazine*, 57(2): 67–76, 1984.

[14] Andrew J Dudzik and Petar Veličković. Graph neural networks are dynamic programmers. *Advances in Neural Information Processing Systems*, 35:20635–20647, 2022.

[15] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[16] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[17] Luca Grementieri and Paolo Galeone. Towards Neural Sparse Linear Solvers. *arXiv:2203.06944*, 2022.

[18] Tom Gustafsson and G. D. McBain. scikit-fem: A Python package for finite element assembly. *Journal of Open Source Software*, 5(52):2369, 2020.

[19] Markus Götz and Hartwig Anzt. Machine learning-aided numerical linear algebra: Convolutional neural networks for the efficient preconditioner generation. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, pages 49–56, 2018.

[20] Ayano Kaneda, Osman Akar, Jingyu Chen, Victoria Kala, David Hyde, and Joseph Teran. A deep gradient correction method for iteratively solving linear systems. *arXiv preprint arXiv:2205.10763*, 2022.

[21] Jelena Koldan, Vladimir Puzyrev, Josep de la Puente, Guillaume Houzeaux, and José María Cela. Algebraic multigrid preconditioning within parallel finite-element solvers for 3-d electromagnetic modelling problems in geophysics. *Geophysical Journal International*, 197(3): 1442–1458, 2014.

[22] Hans Petter Langtangen and Anders Logg. *Solving PDEs in minutes-the FEniCS tutorial I*, volume 3 of *Simula SpringerBriefs on Computing*. Springer, 2016.

[23] Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software*, 31(3):302–325, September 2005.

[24] Yichen Li, Tao Du, Peter Yichen Chen, and Wojciech Matusik. NeuralPCG: Learning preconditioner for solving partial differential equations with graph neural network, 2023. URL https://openreview.net/forum?id=IDSXUFQeZO5.

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[26] John W. Pearson and Jennifer Pestana. Preconditioners for Krylov subspace methods: An overview. *GAMM-Mitteilungen*, 43(4), November 2020. ISSN 0936-7195, 1522-2608.

[27] Florian A. Potra and Stephen J. Wright. Interior-point methods. *Journal of Computational and Applied Mathematics*, 124(1):281–302, 2000. ISSN 0377-0427. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

[28] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2nd edition, 2003. ISBN 978-0-89871-534-7.

[29] Johannes Sappl, Laurent Seiler, Matthias Harders, and Wolfgang Rauch. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems. *arXiv preprint arXiv:1906.06925*, 2019.

[30] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. *Carnegie-Mellon University. Department of Computer Science Pittsburgh*, 1994.

[31] Jens Sjölund and Maria Bånkestad. Graph-based neural acceleration for nonnegative matrix factorization. *arXiv preprint arXiv:2202.00264*, 2022.

[32] Rita Stanaityte. *ILU and Machine Learning Based Preconditioning For The Discretized Incompressible Navier-Stokes Equations*. PhD thesis, University of Houston, 2020.

[33] Ziyuan Tang, Hong Zhang, and Jie Chen. Graph Neural Networks for Selection of Preconditioners and Krylov Solvers. *NeurIPS 2022 New Frontiers in Graph Learning Workshop*, 2022.

[34] Petar Veličković. Everything is connected: Graph neural networks. *Current Opinion in Structural Biology*, 79:102538, 2023.

[35] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

A Dataset details

The (preconditioned) conjugate gradient method is applicable to systems of linear equations of the form

$$Ax = b \tag{10}$$

where A is symmetric and positive definite (spd) i.e. $A = A^T$ and $x^\mathsf{T} Ax > 0$ for all $x \neq 0$. We also write $A \in S_n^{++}$ to indicate that A is a $n \times n$ spd matrix. In this section we specify different problem settings leading to distributions $\mathbb{A}$. Over such problems we are then aiming to train a model using the empirical risk minimization objective shown in the main paper to compute an efficient preconditioner. In Section C, the results presented in the main paper on these problem settings are extended.

In total, we are testing our method on two different datasets but vary the parameters used to generate these datasets. The first problem dataset considers random problem instances. The other problem arises from scientific computing where large-scale spd linear equation systems arise naturally from the discretization of PDEs. The size of the matrices considered in the experiments is between $n = 1\,000$ and $n = 10\,000$ for all datasets in use. Throughout the paper we assume that the problem at hand is (very) sparse and spd. Thus, the preconditioned conjugate gradient method is a natural choice for all these problems. The datasets are summarized in Table 2. In the following, the details for the problem generation are explained for each of the datasets.

Table 2: Summary of the datasets used with some additional statistics. Samples refer to train, validation and test set respectively.

Dataset	Samples	Size	Sparsity
Random	5 000/20/1000	5 000	$80 - 90\%$
Random (OOD)	-/-/50	1 000 - 10 000	$80 - 95\%$
Poisson-PDE	3 000/30/900	5 000 - 20 000	$99\% >$

A.1 Random matrices

The set of random matrices is constructed by first choosing a sparsity parameter p which indicates the expected percentage of non-zero elements in the matrix. It is also possible to specify p itself as a distribution. We tune the non-zero probability such that the resulting spd matrices which are generated with equation (11) have between 80 and 90% total sparsity. We then create the problem by sampling a matrix A with $P(A_{ij} = 0) = p$ and $P(A_{ij} | A_{ij} \neq 0) \sim \mathcal{N}(0, 1)$. The final problem is then given by

$$AA^\mathsf{T} + \alpha I \tag{11}$$

where $\alpha \approx 10^{-2}$ is used to ensure the resulting problem is positive definite. By choosing the α parameter, the condition number of the matrix can be controlled. For each set of matrices of a given size we specify the sample probability p such that the generated problem remains within the desired sparisty bounds. The right hand side of the linear equation system is sampled uniformly $b \sim \mathcal{U}[0, 1]$.

Problems of this form are also arising when minimizing quadratic programs since the equation (10) represents the first-order optimality condition for an unconstrained QP. Thus, the problem represented here ca encountered in various settings.

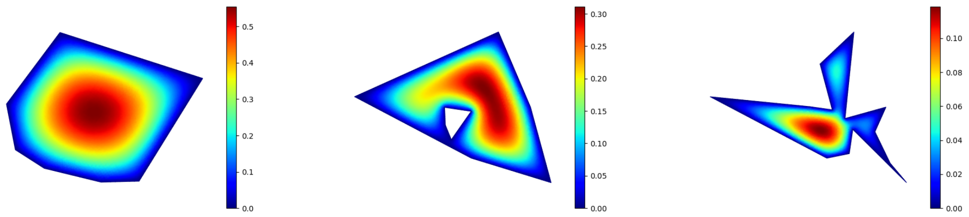
A.2 PDE problem: Poisson equation

The Poisson equation is an elliptical partial differential equation (PDE) and one of the most fundamental problems in numerical methods [22]. The problem is stated as solving the boundary value problem

$$-\nabla^2 u(x) = f(x) \qquad x \in \Omega \tag{12}$$

$$u(x) = u_D(x) \quad x \in \partial\Omega \tag{13}$$

where $f(x)$ is the source function and $u = u(x)$ is the unknown function to be solved for. By discretizing this problem and writing it in matrix form a system of linear equations of the form



(a) Convex (b) Convex with hole (c) Simple polytope

Figure 6: Samples for the three different domains used to generate training data for the Poisson problem.

$Lx = b$ is obtained where the stiffness matrix L is sparse and spd. For large n, this problem therefore can be efficiently solved using the conjugate gradient method given a good preconditioner.

In order to obtain a distribution $\mathbb{A}$ over a large number of Poisson equation PDE problems, we consider a set of meshes that are generated by sampling from a normal distribution and creating a mesh based on the sampled points. We consider three different cases for the mesh generation: (1) we form the convex hull of the sampled points and triangulate the resulting shape, (2) we sample two sets of points and transform them into convex shapes as before. The second samples have a significantly smaller variance. The final shape that is used for triangulation is then obtained as the difference between the two samples. Finally, (3) we generate a simple polytope from the sampled points. An example for each of the generated shapes is shown in Figure 6. The sampled meshes are discretized using the scikit-fem python library [18]. Due to the discretization based on the provided mesh the size of the generated matrix is variable. Based on these discretized problems the `NeuralIF` preconditioner is trained.

B Implementation details

In the following, we describe the details of the network architecture our `NeuralIF` model uses as well as the details of the model training and inference.

Node and edge features In total 8 features are used for each node i.e. $x_i \in \mathbb{R}^8$. The first five node features listed in Table 3 are implemented following the *local degree profile* introduced by Cai and Wang [8]. The dominance and decay features shown in the table are originally introduced by Tang et al. [33]. The final node feature is the position of the node in the matrix. This feature introduced by us breaks the permutation equivariance of the GNN. However, since we only consider the lower triangular output of the position implicitly effects the output either way. By explicitly using the position also as a feature, we make sure the network is able to utilize this information to predict a good preconditioner.

As edge feature, only the scalar value of the matrix entry is used. In deeper layers of the network, when skip connections are introduced, the edges are augmented with the original matrix entries. This effectively leads to having two edge features in these layers: the computed edge embedding of the current layer $z_{ij}^{(l)}$ as well as the original matrix entry a_{ij}. However, from these only a one-dimensional output is computed $z_{ij}^{(l+1)}$ as described in the following.

Message passing block At the core of our method, we introduce a novel message passing block which aligns with the objective of our training. The underlying idea is that matrix multiplication can be represented in graph form by concatenating the two Coates graph representations as described previously [13]. The message passing block consists of two GNN layers as introduced in the background section which are concatenated to mimic the matrix multiplication. To align the block with our objective, we only consider the edges in the lower triangular part of the matrix in the first layer while in the second layer of the block only edges from the upper triangular part of the matrix are used. In the final step of the message passing block, we introduce skip connections and concatenate the edge features in the current layer with the original matrix entries. This is also shown in the model

Table 3: List of node features used in the graph neural network.

Feature name	Description
deg(v)	Degree of node v
max deg(u)	Maximum degree of neighboring nodes
min deg(u)	Minimum degree of neighboring nodes
mean deg(u)	Average degree of neighboring nodes
var deg(u)	Variance in the degrees of neighboring nodes
dominance	Diagonal dominance
decay	Diagonal decaying
pos	Position of node in the matrix

pseudocode in Algorithm 2. Here, we denote the updates after the first message passing layer with the superscript $(l + \frac{1}{2})$ to explicitly indicate that it is an intermediate update step. The final output from the block is then computed based on another message passing step which utilizes the computed intermediate embeddings. Here, $\mathcal{N}_L(i)$ denotes the neighborhood of node i with respect to the edges in the lower triangular part L and $\mathcal{N}_U(i)$ the ones from the upper triangular part U respectivly. There are always at least diagonal elements in the neighborhood and therefore, the message computation is well defined.

Network design Both the parameterzied edge update ϕ and the node update ψ functions in every layer are implemented using 2-layer fully-connected neural networks. The input to the edge update network ϕ is formed by the nodes of the corresponding edge and the edge features themselves leading to a total of $8 + 8 + 1 = 17$ inputs in the first layer and one additional input in later layers. The edge network outputs a scalar value. For each network, 16 hidden units are used. To compute the aggregation over the neighborhood $\oplus$, the mean aggregation function is used which is applied component-wise to the set of neighboring feature vectors. The node update function ψ takes the aggregation of the neighborhood as well as the current node feature as an input $(8 + 8 = 16)$.

The proposed `NeuralIF` model consists of 3 of the message passing blocks introduced in Section 3 in the main paper. Resulting in a total of 6 GNN message passing steps with 2 skip connections from the input matrix A. In total the `NeuralIF` GNN model has 3 638 learnable parameters. The forward pass through the model is described in Algorithm 2.

Training parameters We are training our model for a total of 100 epochs using a batch size of 1. The Adam optimizer with initial learning rate 0.1 is used. Due to the small batch size and the loss landscape, we utilize gradient clipping to restrict the length of the allowed update steps and reduce the variance during stochastic gradient descent. Additionally, we use early stopping in order to avoid overfitting of the model based on the validation set performance. We use a learning rate scheduler in order to decrease the learning rate based on validation set performance.

Baselines and iterative methods The conjugate gradient method is implemented both in the preconditioned form (as shown in Algorithm 1) and, as a baseline, without preconditioner using PyTorch [25]. The `NeuralIF` preconditioner is implemented using PyTorch Geometric [15]. The incomplete Cholesky preconditioner utilizes the readily available `ilu` functionality of SciPy [35] which internally relies on the efficient implementation provided by the SuperLU library [23]. The Jacobian preconditioner is, due to its computationally simplicity, directly implemented using PyTorch.

Compute In order to save memory, we are using the `torch.float16` (half precision) datatype during the training procedure. This datatype is, however, only supported on CUDA devices. The model training is executed on a single NVIDIA-Titan GPU with 12GB memory. The model inference is compared on a consumer CPU. A single model training takes 12 hours on the given hardware. The inference time for the model is significantly faster. Experiments reported in the result section required 2 hours each ($\approx$ 30 minutes per method). A significant amount of this time was also required to compute the eigendecomposition of the matrices that are reported in the results. Additional experiments, not reported here, used for the architecture search and prototyping were conducted on a smaller dataset consisting of matrices of size $n = 1\,000$. This allows for rapid prototyping since

Algorithm 2 Pseudo-code for `NeuralIF` preconditioner.

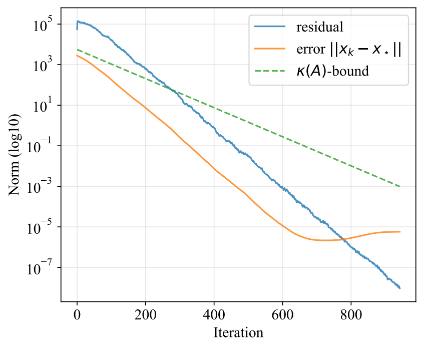
```
1: Input: Graph representing the spd system of linear equations $Ax = b$.
2: Output: Lower-triangular preconditioner for the linear system.
3: Compute node features $x_i$ shown in Table 3.
4: Split graph adjacency matrix into lower $L$ and upper triangular parts $U$
5: for each message passing block $l$ in $0, 1, \ldots, N-1$ do
6:     ▷ update using the lower-triangular matrix part
7:     $z_{ij}^{(l+\frac{1}{2})} \leftarrow \phi_{\theta_{z,1}^l}(z_{ij}^{(l)}, x_i^{(l)}, x_j^{(l)})$ for all $(i,j) \in L$
8:     $m_i^{(l+\frac{1}{2})} \leftarrow \frac{1}{|\mathcal{N}_L(i)|} \sum_{j \in \mathcal{N}_L(i)} z_{ji}^{(l+\frac{1}{2})}$
9:     $x_i^{(l+\frac{1}{2})} \leftarrow \psi_{\theta_{e,1}^l}(x_i^{(l)}, m_i^{(l+\frac{1}{2})})$
10:    ▷ share the computed edge updates between the layers
11:    $z_{ji}^{(l+\frac{1}{2})} \leftarrow z_{ij}^{(l+\frac{1}{2})}$
12:    ▷ update using the upper triangular matrix part
13:    $z_{ji}^{(l+1)} \leftarrow \phi_{\theta_{z,2}^l}(z_{ji}^{(l+\frac{1}{2})}, x_j^{(l+\frac{1}{2})}, x_i^{(l+\frac{1}{2})})$ for all $(j,i) \in U$
14:    $m_i^{(l+1)} \leftarrow \frac{1}{|\mathcal{N}_U(i)|} \sum_{j \in \mathcal{N}_U(i)} z_{ji}^{(l+1)}$
15:    $x_i^{(l+1)} \leftarrow \psi_{\theta_{e,2}^l}(x_i^{(l)}, m_i^{(l+1)})$
16:    if not final layer in the network then
17:        ▷ Add skip connections
18:        $z_{ij}^{(l+1)} \leftarrow [z_{ij}^{(l+1)}, a_{ij}]^\mathsf{T}$
19:    end if
20: end for
21: Apply exp-activation function to final edge embeddings of diagonal matrix entries $z_{ii}^{(N)}$.
22: Return lower triangular matrix with elements $z_{ij}^{(N)}$ for $i \leq j$.
```

model training only requires 20 minutes on this dataset. In total, during early stages of the project another 25 hours have been spent on model training.

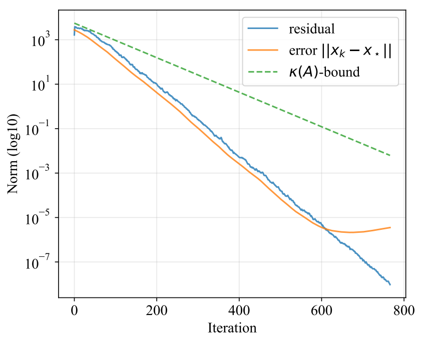
C Additional Results

Random matrices Here, additional results for the random dataset are shown. Figure 7, shows the convergence of the different (preconditioned) CG runs. Figure 8 shows the distribution of the eigenvalues obtained from the preconditioned systems together with the obtained condition number. This directly influences the convergence behavior of the method as the linear $\kappa(A)$-bound can be obtained from the condition number. We can see that the `NeuralIF` preconditioner converges fastest. However, all methods are not able to reduce the actual error $||x_k - x_\star||$ in the solution below $10e-6$ and even though the residual gets smaller The error even increases after a certain number of iterations for all tested methods. This is due to numerical issues with the conjugate gradient method. In practice, the true solution is, however, not available.

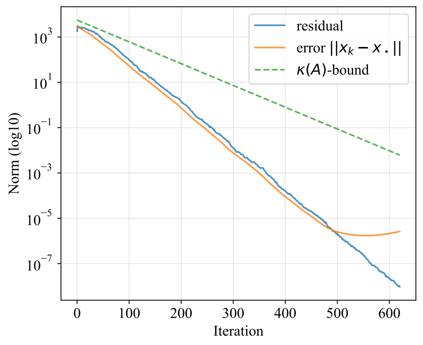
Poisson PDE In Figure 9 and Figure 10, the same information is shown for a matrix system arising from solving a Poisson PDE. The discretized problem shown here is of size $n = 14\,145$. Even though the matrix is significantly larger, the CG method converges faster than in the previously shown random matrix problem. This is due to the fact that the problem is less ill-conditioned and the eigenvalues in the original matrix are already clustered around one. Nevertheless, the `NeuralIF` preconditioner is able to improve the spectral properties and speed up the convergence of the method.



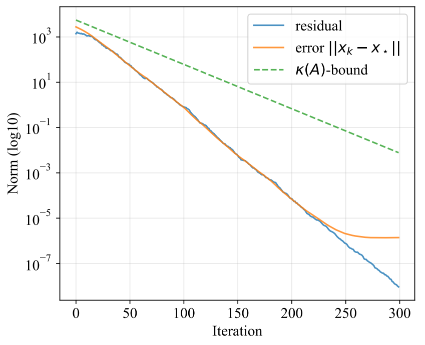
(a) No preconditioner



(b) Jacobian preconditioner

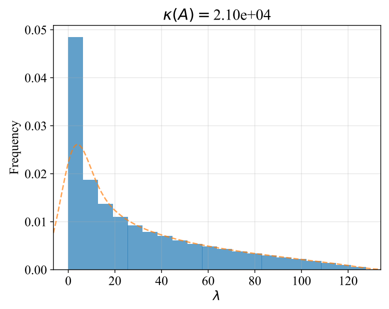


(c) Incomplete Cholesky preconditioner

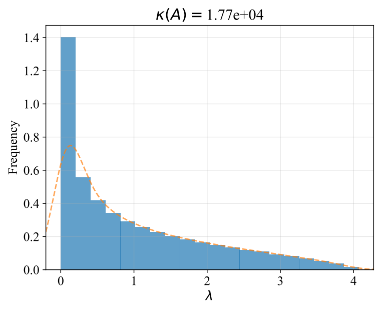


(d) **NeuralIF (ours)**

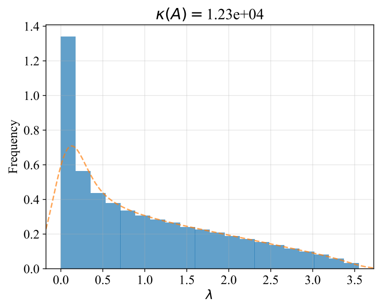
Figure 7: Convergence behavior of the preconditioned conjugate gradient method based on the computed residual for different preconditioner compared with the actual error as well as the $\kappa(A)$-bound for the random dataset with $n = 5\,000$.



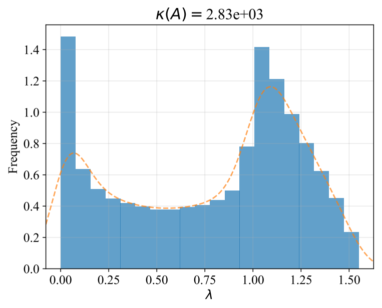
(a) No preconditioner



(b) Jacobian preconditioner

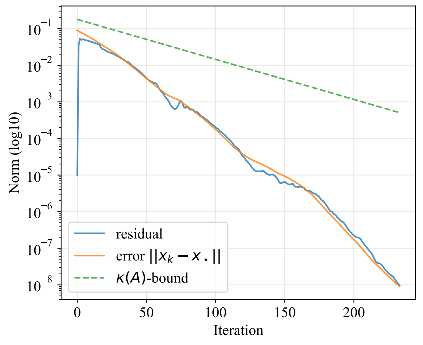


(c) Incomplete Cholesky preconditioner

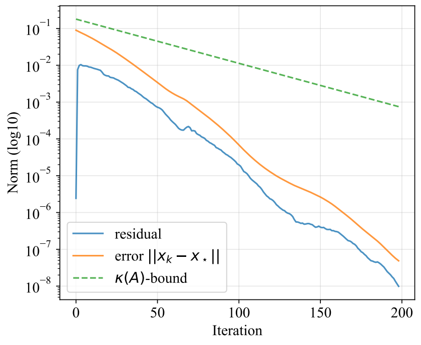


(d) **NeuralIF (ours)**

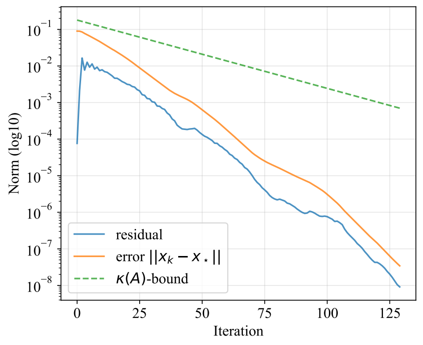
Figure 8: Eigenvalue distribution and condition number of the (preconditioned) linear equation matrix and the corresponding condition number for a sample from the random dataset with $n = 5\,000$.



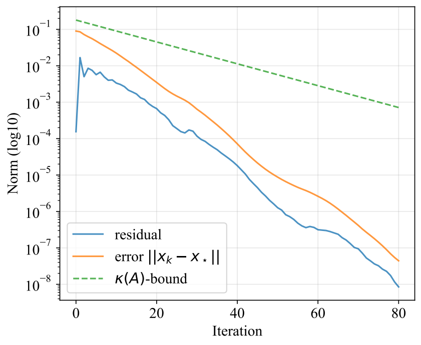
(a) No preconditioner



(b) Jacobian preconditioner

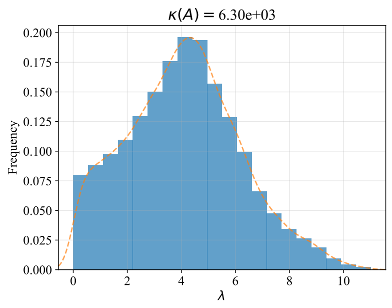


(c) Incomplete Cholesky preconditioner

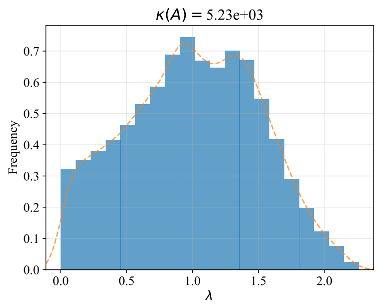


(d) **NeuralIF (ours)**

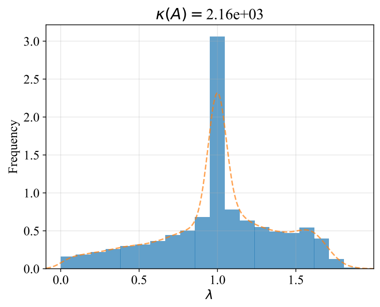
Figure 9: Convergence behavior of the preconditioned conjugate gradient method based on the computed residual for different preconditioner compared with the actual error as well as the $\kappa(A)$-bound for the PDE Poisson problem.



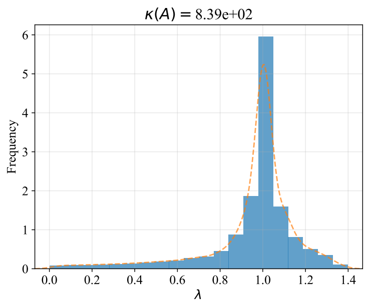
(a) No preconditioner



(b) Jacobian preconditioner



(c) Incomplete Cholesky preconditioner



(d) **NeuralIF (ours)**

Figure 10: Eigenvalue distribution and condition number of the (preconditioned) linear equation matrix and the corresponding condition number for a sample from the PDE Poisson problem.