Graph Neural Preconditioners for Iterative Solutions of Sparse Linear Systems

Jie Chen

MIT-IBM Watson AI Lab, IBM Research. chenjie@us.ibm.com

Background: Linear systems and preconditioning

Krylov methods are commonly used to solve large, sparse linear systems

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{A} \in \mathbb{R}^{n\times n}.$$

The effectiveness of Krylov methods heavily depends on the conditioning of the matrix $\mathbf{A}$. Preconditioning amounts to using a matrix $\mathbf{M} \approx \mathbf{A}^{-1}$ (which we extend to a neural network) to solve an easier problem $\mathbf{A}\mathbf{M}\mathbf{u} = \mathbf{b}$ with the new unknown $\mathbf{u}$ and recover the solution $\mathbf{x} = \mathbf{M}\mathbf{u}$.

In some cases (e.g., solving PDEs), the problem structure provides additional information that aids the development of $\mathbf{M}$. In other cases, little information is known beyond the matrix $\mathbf{A}$ itself. In these cases, a general-purpose preconditioner is desirable; examples include ILU, approximate inverse, and AMG.

Related work: PINN, Neural Operator, etc

PINN/NO approaches use neural networks to approximate $\mathbf{A}^{-1}$

- Drawback: They need PDEs and exploit additional information (e.g., spatial coordinates, smoothness, decay of Green's function, etc).

Neural incomplete-factorization approaches use neural networks to parameterize the nonzero entries of the incomplete factors of $\mathbf{A}$

- Same drawback as incomplete factorization: nonzero factorization error.

We aim at a general-purpose preconditioner: No PDE; purely algebraic.

Nonlinear operator $\mathbf{M}$ needs flexible preconditioning

Algorithm 1 FGMRES(m) with $\mathbf{M} \approx \mathbf{A}^{-1}$ being a nonlinear operator

1: Let $\mathbf{x}_0$ be given. Define $\overline{\mathbf{H}}_m \in \mathbb{R}^{(m+1)\times m}$ and initialize all its entries h_{ij} to zero
2: **loop** until `maxiters` is reached
3: Compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\beta = \|\mathbf{r}_0\|_2$, and $\mathbf{v}_1 = \mathbf{r}_0/\beta$
4: **for** $j = 1, \ldots, m$ **do**
5: Compute $\mathbf{z}_j = \mathbf{M}(\mathbf{v}_j)$ and $\mathbf{w} = \mathbf{A}\mathbf{z}_j$
6: **for** $i = 1, \ldots, j$ **do**
7: Compute $h_{ij} = \mathbf{w}^\top \mathbf{v}_i$ and $\mathbf{w} \leftarrow \mathbf{w} - h_{ij}\mathbf{v}_i$
8: **end for**
9: Compute $h_{j+1,j} = \|\mathbf{w}\|_2$ and $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$
10: **end for**
11: Define $\mathbf{Z}_m = [\mathbf{z}_1, \ldots, \mathbf{z}_m]$ and compute $\mathbf{x}_m = \mathbf{x}_0 + \mathbf{Z}_m\mathbf{y}_m$ where $\mathbf{y}_m = \operatorname{argmin}_{\mathbf{y}} \|\beta\mathbf{e}_1 - \overline{\mathbf{H}}_m\mathbf{y}\|_2$
12: If $\|\mathbf{b} - \mathbf{A}\mathbf{x}_m\|_2 <$ `tol`, exit the loop; otherwise, set $\mathbf{x}_0 \leftarrow \mathbf{x}_m$
13: **end loop**

Flexible GMRES converges linearly

Theorem: Assume that FGMRES is run without restart and without breakdown. On completion, let $\mathbf{H}_n$ be diagonalizable, as in $\mathbf{Y}^{-1}\mathbf{H}_n\mathbf{Y} = \mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_n)$. Then, the residual norm satisfies

$$\|\mathbf{r}_m\|_2 \le \kappa_2(\mathbf{Y})\ \epsilon^{(m)}(\mathbf{\Sigma})\ \|\mathbf{r}_0\|_2,$$

where κ_2 denotes the 2-norm condition number, $\mathbb{P}_m$ denotes the space of degree-m polynomials, and

$$\epsilon^{(m)}(\mathbf{\Sigma}) = \min_{p\in\mathbb{P}_m,\, p(0)=1} \max_{i=1,\ldots,n} |p(\sigma_i)|.$$

Corollary: Assume that all the eigenvalues σ_i of $\mathbf{\Sigma}$ are enclosed by an ellipse $E(c, d, a)$ which excludes the origin, where c is the center, d is the focal distance, and a is major semi-axis. When m is large,

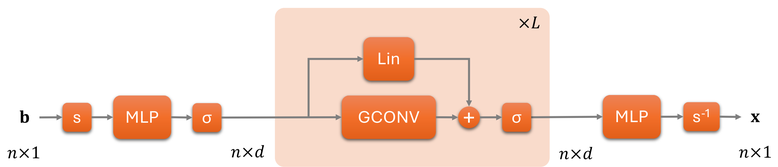
$$\|\mathbf{r}_m\|_2 \lessapprox \kappa_2(\mathbf{Y}) \left(\frac{a + \sqrt{a^2 - d^2}}{c + \sqrt{c^2 - d^2}}\right)^m \|\mathbf{r}_0\|_2.$$

Main question

Can we design a neural network $\mathbb{R}^n \to \mathbb{R}^n$ to approximate $\mathbf{A}^{-1}$?

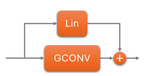
Proposal: Graph neural preconditioner (GNP)

Treat $\mathbf{A}$ as the graph adjacency matrix. Use graph convolution (GCONV) to propagate graph signal $\mathbf{b}$ and obtain $\approx \mathbf{A}^{-1}\mathbf{b}$.



Normalize $\mathbf{A}$ by an upper bound of the spectral radius:

$$\widehat{\mathbf{A}} = \mathbf{A}/\gamma \quad \text{where} \quad \gamma = \min\left\{\max_i\left\{\textstyle\sum_j |a|_{ij}\right\}, \max_j\left\{\textstyle\sum_i |a|_{ij}\right\}\right\}$$



Use residual connections to stack a deeper network:

$$\text{Res-GCONV}(\mathbf{X}) = \text{ReLU}(\mathbf{X}\mathbf{U} + \widehat{\mathbf{A}}\mathbf{X}\mathbf{W})$$

Ensure scale-equivariance $\mathbf{M}(\alpha\mathbf{b}) = \alpha\mathbf{M}(\mathbf{b})$:

$$s(\cdot) = \tfrac{\sqrt{n}}{\tau}\cdot \quad \text{and} \quad s^{-1}(\cdot) = \tfrac{\tau}{\sqrt{n}}\cdot, \quad \text{where } \tau = \|\mathbf{b}\|_2$$

How to generate training data?

Option 1: $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$. **Drawback:** Input space $\mathbf{b} \sim \mathcal{N}(\mathbf{0}, \mathbf{A}\mathbf{A}^\top)$ misses data along the bottom eigen-subspace of $\mathbf{A}\mathbf{A}^\top$. Easy to train but hard to generalize.

Option 2: $\mathbf{b} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$. **Drawback:** Output space $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{A}^{-1}\mathbf{A}^{-\top})$ is skewed; hard to train.

Our proposal:

1. Run m-step Arnoldi without preconditioner $\mathbf{A}\mathbf{V}_m = \mathbf{V}_{m+1}\overline{\mathbf{H}}_m$
2. Perform SVD $\overline{\mathbf{H}}_m = \mathbf{W}_m\mathbf{S}_m\mathbf{Z}_m^\top$
3. Define $\mathbf{x} = \mathbf{V}_m\mathbf{Z}_m\mathbf{S}_m^{-1}\boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_m)$

Consequence:

$$\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_m^{\mathbf{x}}), \qquad \mathbf{\Sigma}_m^{\mathbf{x}} = (\mathbf{V}_m\overline{\mathbf{H}}_m^{+})(\mathbf{V}_m\overline{\mathbf{H}}_m^{+})^\top,$$
$$\mathbf{b} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_m^{\mathbf{b}}), \qquad \mathbf{\Sigma}_m^{\mathbf{b}} = (\mathbf{V}_{m+1}\mathbf{W}_m)(\mathbf{V}_{m+1}\mathbf{W}_m)^\top.$$

In practice: Half the batch $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_m^{\mathbf{x}})$ and half the batch $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$.

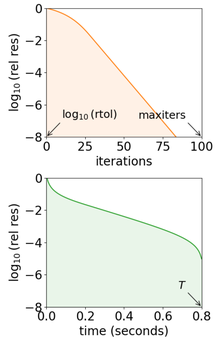
How to evaluate a general-purpose preconditioner?

We evaluate GNP on many matrices ($O(10^3)$). This is a much broader coverage than usual PDE papers.

However, good evaluation metrics are hard to define!

- Comparing the iteration count is insufficient (per-iteration time of each method is different)
- Comparing time? If using `rtol` as a stopping criterion, hard to set the same `rtol` for diverse problems
- If using `timeout` as a stopping criterion, hard to compare reaching `maxiters` with poor residual versus reaching `timeout` with good residual

Proposed metrics: Area under residual curve.

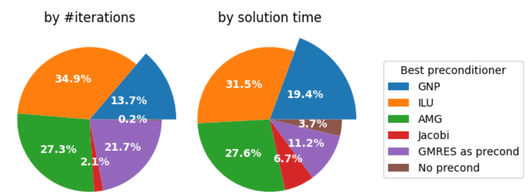


Experiment setting

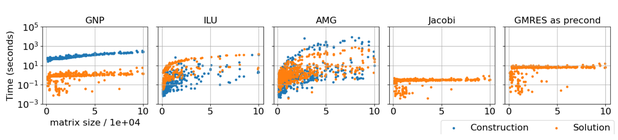
- We compare preconditioners on 867 non-SPD matrices (50 application areas) from the SuiteSparse collection. $1\text{K} \le n \le 100\text{K}$, $\text{nnz} \le 2\text{M}$.
- GNP has 8 layers and hidden dimension 16; trained by Adam for 2K epochs.
- We use one Tesla V100(16GB) GPU.

Experiment findings

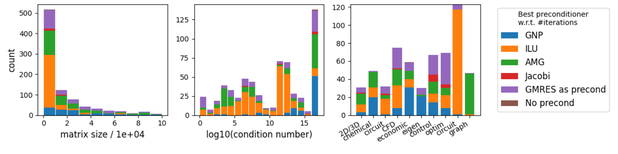
GNP performs the best for a substantial portion of the problems.



The construction time of GNP is predictable, while that of ILU and AMG is not. GNP sometimes is much faster to construct than ILU and AMG. GNP is faster in execution than GMRES as a preconditioner.



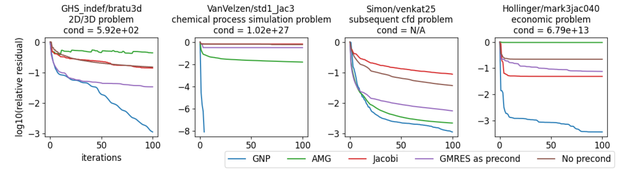
GNP performs competitively for ill-conditioned problems. GNP is useful in some areas: chemical process simulation, economics, and eigenvalue/model reduction.



GNP is very robust, while ILU and AMG often fail.

	GNP	ILU	AMG	Jacobi	GMRES as precond
Construction failure	0 (0.00%)	348 (40.14%)	62 (7.15%)	N/A	N/A
Solution failure	1 (0.12%)	61 (7.04%)	5 (0.58%)	53 (6.11%)	2 (0.23%)

Example convergence histories:



What is next?

- Preconditioning SPD matrices
- Proving universal approximation theorems for GNNs
- Preconditioning a sequence of evolving matrices
- Scaling GNP implementation to real-life applications