MSc in Mathematical Engineering

Course: Advanced Programming for Scientific Computing

Lecturer: Prof.Luca Formaggia and Prof.Carlo De Falco
Teaching Assistant: Dr.Paolo Joseph Baioni

Type B project

A.Y. 2024/2025

Accelerate Algebraic Multigrid Methods Using CNN and GAT

Authors:

Lin Zhang

Tutors:

Prof. Luca Formaggia

Abstract

Algebraic Multigrid (AMG) methods are indispensable for solving large sparse linear systems arising from finite element discretizations of partial differential equations (PDEs). However, the performance of classical AMG heavily depends on the empirical selection of the strong threshold parameter (θ), which governs the coarsening process. This work introduces **AMG-ANNs**, novel deep learning frameworks that optimize θ selection using Convolutional Neural Networks (CNNs) and Graph Attention Networks (GATs) to accelerate AMG convergence. Our approaches map the system matrix $\mathbf{A}_h$ and mesh size h to the optimal θ^* that minimizes the convergence factor ρ. We develop specialized architectures including a CNN that processes matrix sparsity patterns as grayscale images and a GAT that operates directly on the matrix graph structure. Comprehensive validation across heterogeneous diffusion equations ($\mu_{\max}/\mu_{\min} \sim 10^9$), linear elasticity equations, Stokes problems and lid-driven cavity problems demonstrates performance gains: 1) For diffusion and elasticity problems, the CNN architecture achieves great improvement in ρ compared to the standard $\theta = 0.25$; 2) For Stokes problems and lid-driven cavity problems, the GAT architecture shows superior generalization to unseen discretizations. The framework features a computationally efficient matrix-to-image transformation with negligible preprocessing overhead and demonstrates transfer learning capabilities across problem classes. These advancements establish ML-enhanced AMG as a promising paradigm for large-scale simulations in computational science and engineering.

Contents

1 Introduction

The efficient solution of large sparse linear systems arising from finite element discretizations of partial differential equations (PDEs) constitutes a cornerstone challenge in computational science and engineering. As modern simulations tackle increasingly complex physical phenomena—from subsurface reservoir modeling to aerospace structural analysis—the resulting algebraic systems often exceed millions of degrees of freedom. Traditional direct solvers based on LU or Cholesky factorizations become computationally prohibitive for such large-scale problems due to their $\mathscr{O}(n^3)$ complexity and substantial memory requirements. Even classical iterative methods like Jacobi or Gauss-Seidel iterations exhibit prohibitively slow convergence rates for ill-conditioned systems, particularly those stemming from elliptic PDEs with heterogeneous coefficients.

Algebraic Multigrid (AMG) methods have emerged as one of the most powerful hierarchical techniques for addressing these challenges, especially for symmetric positive definite (SPD) matrices. Unlike geometric multigrid that relies on explicit geometric information about mesh hierarchies, AMG constructs its coarse-grid hierarchy purely algebraically using only the system matrix $\mathbf{A}_h$. This matrix-driven approach provides significant advantages for problems where geometric coarsening is problematic, such as domains with complex boundaries, unstructured meshes, or spatially varying coefficients with high contrast ratios. The versatility of AMG has led to its successful adaptation to diverse problem classes including linear elasticity, electromagnetic simulations (Maxwell's equations), fluid dynamics (Navier-Stokes), and multiphase flow in porous media.

At the heart of AMG's effectiveness lies the coarsening process, governed by the strong threshold parameter (θ). This critical parameter determines how "strong connections" between variables are defined during the construction of coarse grids. Mathematically, θ controls the sparsity pattern of the interpolation operators between grid levels, which directly impacts: 1) The approximation quality of coarse-grid corrections 2) The convergence rate of the multigrid cycle 3) The overall computational efficiency of the solver.

In traditional implementations, θ is typically set to a default empirical value of 0.25, based on historical experience with model problems. However, there are comprehensive numerical experiments reveal that this fixed choice often leads to suboptimal performance for problems with strong heterogeneities. For instance, in elliptic problems with discontinuous diffusion coefficients exhibiting contrast ratios of 10^9, the optimal θ can vary by over 100% from the standard value. Manual parameter tuning through trial-and-error becomes computationally expensive and practically infeasible for large-scale simulations or time-dependent problems where system properties may evolve dynamically.

The emergence of machine learning (ML), particularly deep learning with artificial neural networks (ANNs), offers transformative potential for enhancing numerical algorithms. ANNs have demonstrated remarkable capabilities in learning complex, nonlinear relationships from high-dimensional data, making them exceptionally suited for optimizing parameters in scientific computing. Recent applications include: 1)Adaptive selection of artificial viscosity in shock-capturing schemes; 2)Guidance for *hp*-adaptive mesh refinement strategies; 3)Optimization of domain decomposition preconditioners; 4)Enhancement of stabilization parameters in mixed finite element methods.

Within the multigrid community, initial efforts have explored ML for optimizing components like coarsening strategies and smoother selection. However, the specific problem of learning the optimal strong threshold parameter θ for AMG performance—particularly for coupled systems like Stokes flow—remains largely unexplored. This gap motivates our development of **AMG-ANN**, a novel deep learning framework that accelerates AMG convergence by predicting optimal θ^* values using convolutional neural networks and graph attention networks.

Our principal contributions include: (1) A specialized CNN architecture that maps the system matrix $\mathbf{A}_h$ (represented as a sparse pattern image) and mesh size h to the predicted optimal θ^* minimizing the AMG convergence factor ρ; (2) A specialized GAT architecture that utilize the potential graph structure of the system matrix $\mathbf{A}_h$ and mesh size h to predict the value of convergence factor ρ; (3) Comprehensive numerical validation for 2D elliptic PDEs with diffusion coefficients exhibiting extreme heterogeneities ($\mu_{\max}/\mu_{\min} \sim 10^9$), demonstrating up to 33% improvement in convergence factor compared to the standard $\theta = 0.25$; (4) Extension to the stationary Stokes equations and the lid-driven cavity problem;

The remainder of this work is structured as follows: Section 2 provides mathematical foundations for our model problems and their finite element discretizations. Section 3 details AMG algorithms and neural network fundamentals. Section 4 introduces our CNN and GAT-based architectures. Section 5 presents numerical experiments, and Section 6 concludes with future research directions.

2 Problems Description

2.1 Diffusion Equations in Heterogeneous Media

Diffusion processes in heterogeneous materials appear ubiquitously in scientific applications, from heat transfer in composite materials to groundwater flow in geological formations. Consider a prototypical elliptic PDE defined on a bounded domain $\Omega \subset \mathbb{R}^2$ with Lipschitz boundary $\partial\Omega = \overline{\Gamma}_D$. The strong form of the boundary value problem seeks $u : \Omega \to \mathbb{R}$ satisfying:

$$\begin{cases} -\nabla \cdot (\mu(\mathbf{x})\nabla u) = f & \text{in } \Omega \\ u = g_D & \text{on } \Gamma_D \end{cases} \tag{1}$$

where $f \in L^2(\Omega)$ represents source terms, $g_D \in H^{1/2}(\Gamma_D)$ specifies Dirichlet boundary conditions, and $\mu \in L^\infty(\Omega)$ denotes the diffusion coefficient satisfying $\mu(\mathbf{x}) \geq \mu_0 > 0$ almost everywhere. The coefficient heterogeneity—characterized by high contrast ratios $\mu_{\max}/\mu_{\min} \sim 10^9$ in our studies—poses significant challenges for numerical solvers.

To handle non-homogeneous Dirichlet conditions, we introduce a lifting function $\tilde{g} \in H^1(\Omega)$ such that $\tilde{g}|_{\Gamma_D} = g_D$ and define the solution variable $\tilde{u} = u - \tilde{g}$. The weak formulation is derived by multiplying (1) by a test function $v \in H^1_{\Gamma_D}(\Omega) := \{v \in H^1(\Omega) : v|_{\Gamma_D} = 0\}$ and applying integration by parts:

$$\text{Find } \tilde{u} \in H^1_{\Gamma_D}(\Omega) \text{ such that: } a(\tilde{u}, v) = F(v) \quad \forall v \in H^1_{\Gamma_D}(\Omega) \tag{2}$$

The bilinear and linear forms are defined as:

$$a(w, v) = \int_\Omega \mu \nabla w \cdot \nabla v d\Omega \tag{3}$$

$$F(v) = \int_\Omega f v d\Omega - a(\tilde{g}, v) \tag{4}$$

Note the correction in (4) from the original formulation, ensuring consistency with the lifted formulation. The well-posedness of (2) follows from the Lax-Milgram theorem given the uniform ellipticity and continuity of $a(\cdot, \cdot)$.

For finite element discretization, we consider a quasi-uniform quadrilateral mesh $\mathscr{T}_h$ with mesh size h. Using $\mathbb{Q}_1$ basis functions defined via reference element mappings $F_T : \widehat{\Omega} \to T$, the discrete space is:

$$V_h = \left\{ v_h \in C^0(\overline{\Omega}) : v_h|_T \circ F_T \in \mathbb{Q}_1(\widehat{\Omega}) \ \forall T \in \mathscr{T}_h, \ v_h = 0 \text{ on } \Gamma_D \right\} \tag{5}$$

The algebraic system $\mathbf{A}_h \mathbf{u} = \mathbf{f}$ is obtained through standard assembly procedures, where the stiffness matrix entries are:

$$(\mathbf{A}_h)_{ij} = a(\phi_j, \phi_i) = \sum_{T \in \mathscr{T}_h} \int_T \mu \nabla \phi_j \cdot \nabla \phi_i dT \tag{6}$$

The resulting matrix is symmetric positive definite but becomes increasingly ill-conditioned as the contrast ratio $\mu_{\max}/\mu_{\min}$ grows, necessitating advanced solution techniques like AMG.

2.2 Linear Elasticity Equations

In real life, most partial differential equations are really systems of equations. Accordingly, the solutions are usually vector-valued.

In our work, we will want to solve the elastic equations. They are an extension to Laplace equations with a vector-valued solution that describes the displacement in each space direction of a rigid body which is subject to a force. Of course, the force is also vector-valued, meaning that in each point it has a direction and an absolute value.

One can write the elasticity equations in a number of ways. The one that shows the symmetry with the Laplace equation in the most obvious way is to write it as

$$-\mathrm{div}\,(\mathbf{C}\nabla\mathbf{u}) = \mathbf{f},$$

where $\mathbf{u}$ is the vector-valued displacement at each point, $\mathbf{f}$ the force, and $\mathbf{C}$ is a rank-4 tensor (i.e., it has four indices) that encodes the stress-strain relationship – in essence, it represents the "spring constant" in Hookes law that relates the displacement to the forces. $\mathbf{C}$ will, in many cases, depend on $\mathbf{x}$ if the body whose deformation we want to simulate is composed of different materials.

While the form of the equations above is correct, it is not the way they are usually derived. In truth, the gradient of the displacement $\nabla\mathbf{u}$ (a matrix) has no physical meaning whereas its symmetrized version,

$$\epsilon(\mathbf{u})_{kl} = \frac{1}{2}(\partial_k\mathbf{u}_l + \partial_l\mathbf{u}_k),$$

does and is typically called the "strain". (Here and in the following, $\partial_k = \frac{\partial}{\partial x_k}$. We will also use the Einstein summation convention that whenever the same index appears twice in an equation, summation over this index is implied; however, we will not distinguish between upper and lower indices.) With this definition of the strain, the elasticity equations then read as

$$-\mathrm{div}\,(\mathbf{C}\epsilon(\mathbf{u})) = \mathbf{f},$$

which you can think of as the more natural generalization of the Laplace equation to vector-valued problems. (The form shown first is equivalent to this form because the tensor $\mathbf{C}$ has certain symmetries, namely $C_{ijkl} = C_{ijlk}$, and consequently $\mathbf{C}\epsilon(\mathbf{u})_{kl} = \mathbf{C}\nabla\mathbf{u}$.)

One can of course alternatively write these equations in component form:

$$-\partial_j(c_{ijkl}\epsilon_{kl}) = f_i, \quad i = 1 \ldots d.$$

In many cases, one knows that the material under consideration is isotropic, in which case by introduction of the two coefficients λ and μ the coefficient tensor reduces to

$$c_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}).$$

The elastic equations can then be rewritten in a much simpler form:

$$-\nabla\lambda(\nabla \cdot \mathbf{u}) - (\nabla \cdot \mu\nabla)\mathbf{u} - \nabla \cdot \mu(\nabla\mathbf{u})^T = \mathbf{f},$$

and the respective bilinear form is then

$$a(\mathbf{u}, \mathbf{v}) = (\lambda\nabla \cdot \mathbf{u}, \nabla \cdot \mathbf{v})_0 + \sum_{k,l}(\mu\partial_k u_l, \partial_k v_l)_0 + \sum_{k,l}(\mu\partial_k u_l, \partial_l v_k)_0,$$

or also writing the first term a sum over components:

$$a(\mathbf{u}, \mathbf{v}) = \sum_{k,l}(\lambda\partial_k u_k, \partial_l v_l)_0 + \sum_{k,l}(\mu\partial_k u_l, \partial_k v_l)_0 + \sum_{k,l}(\mu\partial_k u_l, \partial_l v_k)_0.$$

2.3 Stokes Equations for Incompressible Flow

The steady incompressible Stokes equations model slow viscous flows in the limit of negligible inertia, with applications ranging from microfluidics to geodynamics and biomechanics. The (dimensionless) strong form reads

$$\begin{aligned} -\nabla\cdot\big(2\nu\,\varepsilon(\mathbf{u})\big) + \nabla p &= \mathbf{f} \quad \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= 0 \quad \text{in } \Omega, \end{aligned} \tag{7}$$

where $\mathbf{u}$ denotes the velocity field, p the kinematic pressure, $\mathbf{f}$ body forces, ν the (kinematic) viscosity, and the symmetric strain-rate tensor is

$$\varepsilon(\mathbf{u}) = \tfrac{1}{2}\big(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\big).$$

When ν is constant one often set $\nu = 1$ for convenience; we keep ν explicit for generality.

Weak formulation. Let $V_g = \{\mathbf{v} \in [H^1(\Omega)]^d : \mathbf{v}|_{\partial\Omega} = \mathbf{g}\}$, $V_0 = \{\mathbf{v} \in [H^1(\Omega)]^d : \mathbf{v}|_{\partial\Omega} = 0\}$, and $Q = L^2_0(\Omega)$ (zero-mean pressures). Multiplying the momentum equation by a test function $\mathbf{v} \in V_0$, integrating by parts and applying natural boundary conditions yields the variational problem: find $(\mathbf{u}, p) \in V_g \times Q$ such that for all $(\mathbf{v}, q) \in V_0 \times Q$,

$$\begin{aligned} a(\mathbf{u},\mathbf{v}) + b(\mathbf{v},p) &= (\mathbf{v},\mathbf{f})_{L^2(\Omega)} + \langle \mathbf{v}, \mathbf{g}_N \rangle_{\Gamma_N}, \\ b(\mathbf{u},q) &= 0, \end{aligned} \tag{8}$$

with the bilinear forms

$$a(\mathbf{u},\mathbf{v}) := 2\int_\Omega \nu\, \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v})\,\mathrm{d}x, \qquad b(\mathbf{v},p) := -\int_\Omega p\,\nabla\cdot\mathbf{v}\,\mathrm{d}x,$$

and where $\mathbf{g}_N$ denotes prescribed Neumann (traction) data on Γ_N. The weak form has the saddle-point structure and, under standard assumptions on ν and domain regularity, is well posed provided an inf–sup (LBB) condition holds for the chosen velocity/pressure spaces.

2.4 Lid-Driven Cavity Benchmark

The lid-driven cavity is a classical benchmark for validating incompressible Stokes / Navier–Stokes solvers and discretizations. Consider the unit box $\Omega = [0,1]^d$ (usually $d = 2$) and denote by Γ_{lid} the top boundary $\{(x,1) : x \in [0,1]\}$. We study the steady Stokes problem with unit viscosity (the general case with viscosity μ is obtained by scaling):

$$\begin{aligned} -\nabla\cdot\big(2\varepsilon(\mathbf{u})\big) + \nabla p &= \mathbf{0} \quad \text{in } \Omega, \\ \nabla\cdot\mathbf{u} &= 0 \quad \text{in } \Omega, \end{aligned} \tag{9}$$

where the (symmetric) strain tensor is

$$\varepsilon(\mathbf{u}) = \tfrac{1}{2}\big(\nabla\mathbf{u} + (\nabla\mathbf{u})^T\big).$$

We impose the no-slip boundary condition on the walls and a prescribed horizontal lid velocity on the top:

$$\mathbf{u} = \begin{cases} (U_{\text{lid}}, 0, \ldots, 0)^T & \text{on } \Gamma_{\text{lid}}, \\ \mathbf{0} & \text{on } \partial\Omega \setminus \Gamma_{\text{lid}}. \end{cases} \tag{10}$$

To fix the pressure nullspace we impose the zero-mean constraint on the pressure space: $p \in L^2_0(\Omega) = \{q \in L^2(\Omega) : \int_\Omega q\,\mathrm{d}x = 0\}$.

Weak formulation. Let $V_g = \{\mathbf{v} \in [H^1(\Omega)]^d : \mathbf{v}|_{\partial\Omega} = \mathbf{g}\}$ denote the space of velocity fields satisfying the Dirichlet data $\mathbf{g}$ (with $\mathbf{g} = (U_{\text{lid}}, 0, \ldots, 0)$ on Γ_{lid} and $\mathbf{0}$ elsewhere), and V_0 the corresponding homogeneous space. Let $Q = L_0^2(\Omega)$. The standard weak (variational) formulation reads: find $(\mathbf{u}, p) \in V_g \times Q$ such that, for all $(\mathbf{v}, q) \in V_0 \times Q$,

$$a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) = 0, \tag{11}$$

$$b(\mathbf{u}, q) = 0, \tag{12}$$

where

$$a(\mathbf{u}, \mathbf{v}) := 2 \int_\Omega \varepsilon(\mathbf{u}) : \varepsilon(\mathbf{v}) \, \mathrm{d}x, \qquad b(\mathbf{v}, p) := -\int_\Omega p \, \nabla \cdot \mathbf{v} \, \mathrm{d}x.$$

Equivalently one may write $b(\mathbf{u}, q) = \int_\Omega q \, \nabla \cdot \mathbf{u} \, \mathrm{d}x$ for the continuity equation.

3 Algorithms

3.1 Algebraic Multigrid Methods

Algebraic Multigrid (AMG) provides a matrix-based framework for solving large sparse linear systems $\mathbf{A}_h \mathbf{u}_h = \mathbf{f}_h$ without geometric information. Its efficiency stems from the multilevel hierarchy that resolves different error components: fine grids eliminate high-frequency errors through smoothing, while coarse grids handle low-frequency errors through correction.

The coarsening process begins with classifying degrees of freedom (DOFs) into coarse ($\mathscr{C}_h$) and fine ($\mathscr{F}_h$) sets based on strong connections defined by the threshold parameter θ:

Definition 1 (Strong Connection). *A variable i strongly depends on j if:*

$$-(\mathbf{A}_h)_{ij} \geq \theta \max_{k \neq i} \{-(\mathbf{A}_h)_{ik}\}, \quad 0 < \theta \leq 1 \tag{13}$$

This definition captures dominant connections in the matrix graph. The choice of θ critically impacts AMG performance: - Small θ ($\theta \ll 0.25$): More connections classified as strong $\rightarrow$ Denser coarse grids $\rightarrow$ Better interpolation but increased setup cost - Large θ ($\theta \gg 0.25$): Fewer strong connections $\rightarrow$ Sparser coarsening $\rightarrow$ Faster setup but poorer convergence

After C/F splitting, the interpolation operator $\mathbf{I}_H^h$ is constructed. For $i \in \mathscr{F}_h$, interpolation weights to coarse neighbors $j \in \mathscr{C}_h \cap \mathscr{P}_i$ are computed algebraically:

$$(\mathbf{I}_H^h \mathbf{e}_H)_i = \begin{cases} \mathbf{e}_i & i \in \mathscr{C}_h \\ \sum_{j \in \mathscr{P}_i} w_{ij} (\mathbf{e}_H)_j & i \in \mathscr{F}_h \end{cases} \tag{14}$$

where weights w_{ij} are derived from matrix entries to preserve nullspace components. For SPD matrices, the Galerkin product constructs coarse operators: $\mathbf{A}_H = \mathbf{I}_h^H \mathbf{A}_h \mathbf{I}_H^h$ with $\mathbf{I}_h^H = (\mathbf{I}_H^h)^\top$.

The solution process employs recursive V-cycles (Algorithm 1), combining pre-smoothing (S_1), coarse-grid correction, and post-smoothing (S_2):

Algorithm 1 AMG V-Cycle

```
    V-Cycle$\mathbf{A}, \mathbf{u}, \mathbf{f}$, level **if** level $\geq$ max_level **then**
 2:     Solve $\mathbf{Ae} = \mathbf{r}$ exactly {Coarsest level}
 3:     **return** $\mathbf{e}$
 4: **end if**
 5: $\mathbf{u} \leftarrow S_1(\mathbf{A}, \mathbf{u}, \mathbf{f})$ {Pre-smoothing (e.g., Gauss-Seidel)}
 6: $\mathbf{r} \leftarrow \mathbf{f} - \mathbf{Au}$ {Residual computation}
 7: $\mathbf{r}_H \leftarrow \mathbf{I}_h^H \mathbf{r}$ {Restriction}
 8: $\mathbf{e}_H \leftarrow$ V-Cycle($\mathbf{A}_H, \mathbf{0}, \mathbf{r}_H$, level $+ 1$) {Recursive solve}
 9: $\mathbf{u} \leftarrow \mathbf{u} + \mathbf{I}_H^h \mathbf{e}_H$ {Prolongation and correction}
10: $\mathbf{u} \leftarrow S_2(\mathbf{A}, \mathbf{u}, \mathbf{f})$ {Post-smoothing}
11: **return** $\mathbf{u}$
```

3.2 Neural Network Fundamentals

Artificial Neural Networks (ANNs) are computational models inspired by biological neural networks, capable of approximating complex nonlinear functions through compositions of simple transformations. A feedforward network with L layers transforms input $\mathbf{x} \in \mathbb{R}^{n_0}$ to output $\mathbf{y} \in \mathbb{R}^{n_L}$ via:

$$\mathbf{z}^{(0)} = \mathbf{x} \tag{15}$$

$$\mathbf{a}^{(k)} = \mathbf{W}^{(k)} \mathbf{z}^{(k-1)} + \mathbf{b}^{(k)} \quad k = 1, \dots, L \tag{16}$$

$$\mathbf{z}^{(k)} = \sigma(\mathbf{a}^{(k)}) \quad k = 1, \dots, L-1 \tag{17}$$

$$\mathbf{y} = g(\mathbf{a}^{(L)}) \tag{18}$$

where $\mathbf{W}^{(k)} \in \mathbb{R}^{n_k \times n_{k-1}}$ are weight matrices, $\mathbf{b}^{(k)} \in \mathbb{R}^{n_k}$ bias vectors, σ element-wise activation functions, and g output activation. Common activations include: - ReLU: $\sigma(x) = \max(0, x)$ (avoids vanishing gradients) - Sigmoid: $\sigma(x) = 1/(1+e^{-x})$ (for classification) - Tanh: $\sigma(x) = \tanh(x)$ (symmetric output)

Networks learn by minimizing a loss function $\mathscr{L}(\mathbf{y}, \mathbf{y}_{\text{true}})$ (e.g., mean squared error for regression) via gradient descent. Backpropagation efficiently computes gradients using the chain rule:

$$\delta^{(L)} = \nabla_{\mathbf{a}^{(L)}}\mathscr{L} \tag{19}$$

$$\delta^{(k)} = (\mathbf{W}^{(k+1)})^T\delta^{(k+1)} \odot \sigma'(\mathbf{a}^{(k)}) \quad k = L-1,\ldots,1 \tag{20}$$

$$\frac{\partial\mathscr{L}}{\partial\mathbf{W}^{(k)}} = \delta^{(k)}(\mathbf{z}^{(k-1)})^T, \quad \frac{\partial\mathscr{L}}{\partial\mathbf{b}^{(k)}} = \delta^{(k)} \tag{21}$$

where $\odot$ denotes element-wise multiplication. Modern optimizers like Adam adapt learning rates during training.

3.3 Convolutional Neural Networks (CNNs)

CNNs[2] specialize in processing grid-structured data through three key mechanisms: local connectivity, parameter sharing, and hierarchical representation learning. The core operation is discrete convolution extracting local features. For 2D input $\mathbf{X} \in \mathbb{R}^{H\times W\times C}$ and kernel $\mathbf{K} \in \mathbb{R}^{k_h\times k_w\times C\times F}$:

$$(\mathbf{X} * \mathbf{K})_{i,j,f} = \sum_{m=0}^{k_h-1}\sum_{n=0}^{k_w-1}\sum_{c=1}^{C}\mathbf{K}_{m,n,c,f}\mathbf{X}_{i+m,j+n,c} \tag{22}$$

This operation preserves spatial relationships while significantly reducing parameters compared to fully-connected layers. Striding (s) controls output density:

$$\text{Output size} = \left\lfloor\frac{H-k_h}{s}+1\right\rfloor \times \left\lfloor\frac{W-k_w}{s}+1\right\rfloor \times F \tag{23}$$

Pooling layers provide spatial invariance and dimensionality reduction:

$$\text{Max pooling:} \quad y_{i,j,c} = \max_{p\in\mathscr{R}_{ij}} x_p$$

$$\text{Average pooling:} \quad y_{i,j,c} = \frac{1}{|\mathscr{R}_{ij}|}\sum_{p\in\mathscr{R}_{ij}} x_p$$

where $\mathscr{R}_{ij}$ is a local neighborhood (e.g., 2×2 windows). A typical CNN architecture alternates convolutional layers (feature extraction), activation functions (nonlinearity), pooling layers (downsampling), and culminates in fully-connected layers (task-specific processing).

3.4 Graph Attention Networks (GATs)

GATs[3] extend graph neural networks through attention mechanisms (See Fig. 1) that dynamically weigh neighbor importance. Given a graph

$\mathscr{G} = (\mathscr{V}, \mathscr{E})$ with node features $\mathbf{h}_i \in \mathbb{R}^F$, a GAT layer computes updated features through:

1. **Feature Transformation**: Project input features into higher-level representations:

$$\mathbf{g}_i = \mathbf{W}\mathbf{h}_i, \quad \mathbf{W} \in \mathbb{R}^{F' \times F} \tag{24}$$

2. **Attention Coefficients**: Compute pairwise attention scores:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{g}_i \| \mathbf{g}_j]\right), \quad \mathbf{a} \in \mathbb{R}^{2F'} \tag{25}$$

where $\|$ denotes concatenation and LeakyReLU ($\alpha = 0.2$) introduces nonlinearity.

3. **Neighborhood Normalization**: Apply softmax over neighbor set $\mathscr{N}(i)$:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathscr{N}(i)} \exp(e_{ik})} \tag{26}$$

4. **Feature Aggregation**: Combine neighbor features with attention weights:

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathscr{N}(i)} \alpha_{ij}\mathbf{g}_j\right) \tag{27}$$

Multi-head attention enhances representational power and stability:

$$\mathbf{h}'_i = \Big\|_{k=1}^{K} \sigma\left(\sum_{j \in \mathscr{N}(i)} \alpha_{ij}^{(k)}\mathbf{W}^{(k)}\mathbf{h}_j\right) \tag{28}$$

where $\|$ denotes concatenation. In final layers, averaging replaces concatenation:

$$\mathbf{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathscr{N}(i)} \alpha_{ij}^{(k)}\mathbf{W}^{(k)}\mathbf{h}_j\right) \tag{29}$$

Key advantages over traditional GCNs include: (1) Edge-adaptive weighting capturing varying neighbor importance; (2) Inductive capability generalizing to unseen graph structures; (3) Computational efficiency through parallel attention computation; (4) Interpretability via attention weight analysis.

4 Methods

4.1 CNN-based Architecture

In [1], we can see one neural network architecture based on convolution neural network, which used to predict the convergence factor ρ of

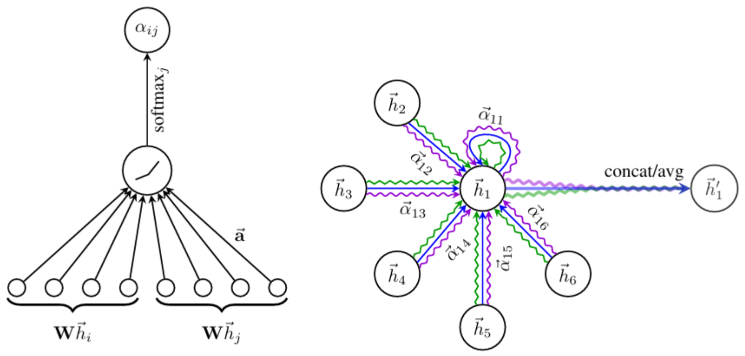

Fig. 1. GAT computational architecture: (a) Attention mechanism computing coefficients α_{ij}, (b) Multi-head aggregation combining features from different representation subspaces

AMG. From the article, we know that the processed sparsity matrices are the input features of CNN. These matrices are thought as the gray images. After extracting the embedding features from the CNN module, the researchers apply the fully connected layer to combine the embedding features with the strong threshold θ and the scaled feature $-log_2(h)$. Following more fully connected layers, we can get the predicted ρ finally.

This is a good idea to predict the convergence factor about solving different systems of linear equations by the algebraic multigrid method. Inspired by this idea, we will propose another architecture below. In order to compare these two architecture, we implement both architectures in our project. And we will only fix one setting of the CNN-based architecture in the subsequent experiments part for convenience and simplicity.

4.2 GAT-based Architecture

We propose a new neural network to predict the convergence factors of various differential problems we mentioned in Part 2. In this architecture, we use GAT module to extract the embedding features of the system matrix $\mathbf{A_h}$. Notice that, we use 4 heads at each GAT layer and there is only one-dimentional input feature, which is the degree of the relevant degree. What's more, there is also one weight on each edge, involving in the convolution of the GAT layer.

After that, we combine the scalar features θ and $-log_2(h)$ with the representation feature of $\mathbf{A_h}$. Then we feed these features into the forward network as the inputs. The FNN module of the GAT-based architecture is almost the same as the one the CNN-based architecture has. We need to say that there are only 2 GAT layers in our GAT-based architecture. It's because we need to compare it with the CNN-base architecture fairly and we have very limited computation source. These two architecture are both trained by one laptop only has 8 cores and 16G RAM. As we know, the training of GAT layer is highly demanding on the computing of the device. And if there are more GAT layers in our architecture, the time need to train the model is longer.

5 Code Organization

The project is structured to facilitate both data generation and model training for learning the convergence rate ρ of algebraic multigrid (AMG) preconditioners in various partial differential equation (PDE) contexts. The codebase is divided into two main parts: **C++ modules** for generating matrix data from PDE simulations and **Python modules** for processing data and training neural networks.

5.1 Directory Structure

The project follows the directory structure below:

```
PACSProject/
          include/                         # C++ header files
                DiffusionModel.hpp   # Diffusion equation AMG data generation
                ElasticModel.hpp     # Elasticity equation AMG data generation
                StokesModel.hpp      # Stokes equation AMG data generation
                Pooling.hpp          # Parallel matrix pooling utilities
          src/
                main.cpp             # Main driver for data generation
          model/                         # Neural network models
                cnn_model.py         # CNN model for matrix regression
                gat_model.py         # GAT model for graph-based regression
          data/                          # Data processing utilities
                cnn_data_processing.py
                gat_data_processing.py
          weights/                       # Trained model weights
          train.py                      # Training script
          val.py                        # Validation and comparison script
          train_cnn_log.txt             # Training logs for CNN
          train_gat_log.txt             # Training logs for GAT
```

5.2 C++ Modules for Data Generation

The C++ modules are built using the `deal.II` and `PETSc` libraries to solve PDEs and extract stiffness matrices along with their AMG convergence properties.

DiffusionModel.hpp

Implements a solver for the diffusion equation with various coefficient patterns. Generates sparse matrices and records the convergence factor ρ for different values of the strength threshold θ, mesh size h, and material parameter ε.

Key features:

- Support for four different diffusion patterns (vertical stripes, checkerboard, etc.)
- Parameterized coefficient jumps through ε values
- Automated dataset generation with comprehensive parameter sweeps

Key code snippet - Matrix output to CSV format:

```cpp
template <int dim>
void Solver<dim>::write_matrix_to_csv(const PETScWrappers::MPI::SparseMatrix &matrix,
    std::ofstream &file, double rho, double h)
{
    const unsigned int m = matrix.m();
    const unsigned int n = matrix.n();

    PetscInt start, end;
    MatGetOwnershipRange(matrix, &start, &end);

    // Extract matrix values, row pointers, and column indices
    // ... (code omitted for brevity)

    // Write metadata and matrix data to CSV
    file << m << "," << n << "," << theta << "," << rho << "," << h << "," << values.size();

    // Write non-zero values
    for (const auto &val : values)
        file << "," << val;

    // Write row pointers
    for (const auto &r : row_ptr)
        file << "," << r;

    // Write column indices
    for (const auto &c : col_ind)
```

```
        file << "," << c;

    file << "\n";
}
```

ElasticModel.hpp

Based on the `deal.II` step-17 tutorial, this module solves the linear elasticity equation. It supports variable Lamé parameters and outputs the system matrix and convergence data.

Key features:

- Material property modeling through Young's modulus and Poisson's ratio
- MPI-parallelized assembly and solving
- Adaptive mesh refinement capabilities

Key code snippet - AMG preconditioner setup:

```
template <int dim>
unsigned int ElasticProblem<dim>::solve(std::ofstream &file)
{
    SolverControl solver_control(solution.size(), 1e-8 *
    system_rhs.l2_norm());
    PETScWrappers::SolverCG cg(solver_control);

    PETScWrappers::PreconditionBoomerAMG preconditioner;
    PETScWrappers::PreconditionBoomerAMG::AdditionalData
    data;
    data.strong_threshold = theta;  // Set strength
    threshold
    data.symmetric_operator = true;

    // Initialize AMG preconditioner
    preconditioner.initialize(system_matrix, data);

    // Solve system and compute convergence rate
    cg.solve(system_matrix, solution, system_rhs,
    preconditioner);

    // ... (convergence rate calculation and output)
}
```

StokesModel.hpp

Solves the Stokes equation using a block-preconditioned MINRES solver with AMG preconditioning. Extracts the velocity block matrix and its convergence properties.

Key features:

- Block preconditioning for saddle-point problems Support for both analytical and lid-driven cavity problems
- Separate handling of velocity and pressure components

Pooling.hpp

Provides parallelized matrix pooling operations to reduce large sparse matrices into dense 2D tensors for CNN input. Supports both COO and CSR formats and includes normalization utilities.

Key code snippet - Parallel pooling implementation:

```cpp
void parallel_pooling_csr(const std::vector<double>& values,
                          const std::vector<int>&
    col_indices,
                          const std::vector<int>& row_ptr,
                          int n, int m, PoolingOp op,
                          std::vector<std::vector<double>>&
    V,
                          std::vector<std::vector<int>>& C)
{
    // Calculate block partitioning parameters
    const int q = n / m;          // Base block size
    const int p = n % m;          // Number of extra blocks
    const int t = (q + 1) * p;    // Boundary point

    // Initialize output matrix
    V = std::vector<std::vector<double>>(m, std::vector<
    double>(m, 0.0));
    C = std::vector<std::vector<int>>(m, std::vector<int>(m,
     0));

    #pragma omp parallel
    {
        // Thread-private matrices for parallel reduction
        std::vector<std::vector<double>> V_local(m, std::
    vector<double>(m, 0.0));
        std::vector<std::vector<int>> C_local(m, std::vector
    <int>(m, 0));

        // Process each row in parallel
        #pragma omp for
        for (int i = 0; i < n; i++) {
            // Get non-zero element range for current row
            const int start_idx = row_ptr[i];
            const int end_idx = row_ptr[i + 1];

            // Process all non-zero elements in current row
            for (int k = start_idx; k < end_idx; k++) {
                const double val = values[k];
                const int j = col_indices[k];

                // Calculate pooling coordinates
                int I = (i < t) ? i / (q + 1) : (i - t) / q + p;
                int J = (j < t) ? j / (q + 1) : (j - t) / q + p;

                // Ensure within valid range and update
                if (I >= 0 && I < m && J >= 0 && J < m) {
                    if (op == PoolingOp::SUM) {
                        V_local[I][J] += val;
                    } else { // MAX
                        if (val > V_local[I][J]) {
                            V_local[I][J] = val;
                        }
                    }
                    C_local[I][J]++;
                }
            }
        }

        // Merge thread results into global matrix
        #pragma omp critical
        {
            for (int i = 0; i < m; i++) {
                for (int j = 0; j < m; j++) {
                    if (op == PoolingOp::SUM) {
                        V[i][j] += V_local[i][j];
                    } else { // MAX
                        if (V_local[i][j] > V[i][j]) {
                            V[i][j] = V_local[i][j];
                        }
                    }
                    C[i][j] += C_local[i][j];
                }
            }
        }
    }
}
```

main.cpp

The entry point for generating datasets. It can be invoked to generate training or test data for any of the three PDE types.

Key code snippet - Command-line interface:

```cpp
int main(int argc, char* argv[])
{
    if (argc < 3){
        std::cerr << "Usage: " << argv[0]
                  << " [D|S|E] [train|test]\n";
        return 1;
    }

    if (std::strcmp(argv[1], "D") == 0) {
        // Generate diffusion dataset
        using namespace AMGDiffusion;
        std::ofstream file;
        if (std::string(argv[2]) == "train") {
            file.open("./datasets/train/raw/train1.csv");
        } else {
            file.open("./datasets/test/raw/test1.csv");
        }
        generate_dataset(file, std::string(argv[2]));
        file.close();
    }
    else if (std::strcmp(argv[1], "E") == 0) {
        // Generate elasticity dataset
        // ... (similar structure)
    }
    else if (std::strcmp(argv[1], "S") == 0) {
        // Generate Stokes dataset
        // ... (similar structure)
    }

    return 0;
}
```

5.3 Python Modules for Data Processing and Training

Python scripts are used to prepare data and train both CNN and GAT models.

Data Processing

- `cnn_data_processing.py`: Loads CSV data and creates PyTorch DataLoader objects for CNN training.
- `gat_data_processing.py`: Processes sparse matrices stored in CSV format into PyTorch Geometric `Data` objects for graph-based learning. Supports large datasets via incremental processing and merging.

Key code snippet - Graph dataset creation:

```
class SparseMatrixDataset2(Dataset):
    def __init__(self, root, csv_file, transform=None,
    pre_transform=None):
        self.csv_file = csv_file
        super().__init__(root, transform, pre_transform)
        self.processed_subdir = os.path.join(self.
    processed_dir, self.csv_file[:-4])
        self.processed_files = glob.glob(os.path.join(self.
    processed_subdir, 'data_*.pt'))
```

```python

    def process(self):
        raw_path = os.path.join(self.raw_dir, self.csv_file)
        with open(raw_path, newline='') as f:
            reader = csv.reader(f)

            for idx, row in enumerate(reader):
                # Parse matrix metadata
                num_rows = int(row[0]); num_cols = int(row[1])
                theta_v = float(row[2]); rho = float(row[3])
                h_v = float(row[4]); nnz = int(row[5])

                # Extract values, row pointers, and column indices
                values = list(map(float, row[6:6+nnz]))
                row_ptrs = list(map(int, row[6+nnz:6+nnz+num_rows+1]))
                col_indices = list(map(int, row[6+nnz+num_rows+1:6+2*nnz+num_rows+1]))

                # Construct edge index and features
                edge_index = []
                for i in range(num_rows):
                    for j in range(row_ptrs[i], row_ptrs[i+1]):
                        edge_index.append([i, col_indices[j]])
                edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()

                # Create node features (degrees)
                degrees = torch.zeros(num_rows, dtype=torch.float)
                for i in range(num_rows):
                    degrees[i] = row_ptrs[i+1] - row_ptrs[i]
                x = degrees.view(-1, 1)

                # Create data object
                data = Data(
                    x=x, edge_index=edge_index, edge_attr=torch.tensor(values).view(-1, 1),
                    y=torch.tensor([rho]), theta=torch.tensor([theta_v]),
                    log_h=-torch.log2(torch.tensor([h_v]))
                )

                # Save processed data
                torch.save(data, os.path.join(self.processed_subdir, f'data_{idx}.pt'))
```

Model Definitions

- `cnn_model.py`: Defines a convolutional neural network that takes pooled matrix images as input along with scalar features θ and $\log h$.

  Key code snippet - CNN architecture:

```python
class CNNModel(nn.Module):
    def __init__(self, in_channels, matrix_size=50, hidden_channels=32,
                 out_channels=32, kernel_size=3, dropout=0.25):
        super(CNNModel, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels, hidden_channels, kernel_size, padding='same')
        self.conv1_2 = nn.Conv2d(hidden_channels, hidden_channels, kernel_size, padding='same')
        self.conv1_3 = nn.Conv2d(hidden_channels, hidden_channels, kernel_size, padding='same')
        self.conv2 = nn.Conv2d(hidden_channels, out_channels, kernel_size)

        # Pooling and fully connected layers
        self.maxpool = nn.MaxPool2d(2)
        self.avgpool = nn.AdaptiveAvgPool2d((2, 2))
        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(128 + 2, hidden_channels * 2)  # +2 for theta and log_h
        self.fc2 = nn.Linear(hidden_channels * 2, 1)

        self.dropout = dropout

```

- `gat_model.py`: Implements a Graph Attention Network (GAT) that operates directly on the graph representation of the sparse matrix. Uses node degrees as features and matrix entries as edge attributes.

  Key code snippet - GAT architecture:

```python
class GATModel(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, num_heads=4, dropout=0.2):
        super(GATModel, self).__init__()

        # GAT layers with edge attributes
        self.conv1 = GATConv(in_channels, hidden_channels, heads=num_heads,
                             dropout=dropout, edge_dim=1)
        self.conv2 = GATConv(hidden_channels *
    num_heads, hidden_channels,
                            heads=num_heads, dropout=
    dropout, edge_dim=1)

        # Fully connected layers
        self.fc1 = nn.Linear(hidden_channels *
    num_heads + 2, hidden_channels)
        self.fc2 = nn.Linear(hidden_channels,
    out_channels)

        self.dropout = dropout
        self.num_heads = num_heads

    def forward(self, data):
        x, edge_index, edge_attr, batch = data.x, data.
    edge_index, data.edge_attr, data.batch

        # GAT layers
        x = F.elu(self.conv1(x, edge_index, edge_attr=
    edge_attr))
        x = F.dropout(x, p=self.dropout, training=self.
    training)
        x = F.elu(self.conv2(x, edge_index, edge_attr=
    edge_attr))

        # Global mean pooling
        x = global_mean_pool(x, batch)

        # Concatenate with scalar features
        scalar_features = torch.cat([data.theta, data.
    log_h], dim=1)
        x = torch.cat([x, scalar_features], dim=1)

        # Final fully connected layers
        x = F.relu(self.fc1(x))
        x = F.dropout(x, p=self.dropout, training=self.
    training)
        x = self.fc2(x)

        return x.squeeze(-1)

```

Training and Validation

- `train.py`: Orchestrates the training of both CNN and GAT models. Supports training on different datasets and saving model weights.

  Key code snippet - Training loop:

```python
def train_model(train_file, test_file, save_model_path,
     model_type,
                batch_size=8, in_chans=1, hidden_chans=64,
                out_chans=1, n_heads=4, dp=0.25, n_epochs=50):

    # Set up device and data loaders
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    if model_type == "CNN":
        train_loader, test_loader = cnn_data_processing.create_dataloaders(
            train_file, test_file, batch_size)
        model = cnn_model.CNNModel(in_channels=in_chans).to(device)
    else:
        train_loader, test_loader = gat_data_processing.create_data_loaders(
            'datasets', train_file, test_file, batch_size)
        model = gat_model.GATModel(in_channels=in_chans,
                                  hidden_channels=hidden_chans,
                                  out_channels=out_chans,
                                  num_heads=n_heads,
                                  dropout=dp).to(device)

    # Set up optimizer and scheduler
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.5)

    # Training loop
    for epoch in range(1, n_epochs + 1):
        model.train()
        total_loss = 0

        for data in train_loader:
            optimizer.zero_grad()

            if model_type == "CNN":
                out = model(data[0].to(device))
                loss = F.mse_loss(out, data[1].to(device))
            else:
                data = data.to(device)
                out = model(data)
                loss = F.mse_loss(out, data.y)

            loss.backward()
            optimizer.step()
            total_loss += loss.item() * (len(data[0]) if model_type == "CNN" else data.num_graphs)

        # Step scheduler and log results
        scheduler.step()
        avg_loss = total_loss / len(train_loader.dataset)
        print(f'Epoch: {epoch:02d}, Train Loss: {avg_loss:.6f}')

```

- `val.py`: Provides utilities for comparing the performance of different models and visualizing results.

5.4 Workflow

The typical workflow involves:

1. Generating PDE data via the C++ executable (e.g., `build/PACSProjectD train`)
2. Pooling the generated matrices using `Pooling.hpp` utilities (via `main.cpp`)
3. Training a CNN or GAT model using `train.py`
4. Evaluating model performance with `val.py`

This organization ensures a clear separation between data generation (C++) and model training (Python), making the project modular and extensible to new PDE types or network architectures.

6 Experiments

6.1 Dataset 1

This dataset concerns the diffusion equations described in Part 2.1. According to the pattern of the diffusion coefficient, we partition the experiments into four categories: first-class vertical stripes, second-class vertical stripes, first-class checkerboard, and second-class checkerboard (these patterns were introduced in Part 1). Besides the coefficient patterns, we vary the exponent ϵ, the grid resolution (parameterized by h), and the AMG strength threshold θ. Different combinations of these choices produce distinct samples used for training and testing.

More precisely, we use twelve evenly spaced values of ϵ in $[0.0, 9.5]$, four values for the mesh-refinement parameter h (from 3 to 6), and twenty-five evenly spaced values of θ in $[0.02, 0.9]$. A data-generation routine implemented in `include/DiffusionModel.cpp` produces the training set of 4800 samples by iterating over these parameter combinations with four nested loops.

To evaluate generalization, the test set is generated by replacing the training values of ϵ with the set $\{1.0, 2.0, 3.0\}$, yielding 1200 test samples.

We train two architectures — a CNN-based network and a GAT-based network — and compare their performance. The loss and evaluation metric are both the mean squared error (MSE). Training hyperparameters are: dropout rate $= 0.25$, learning rate $= 0.001$, and epochs $= 50$. Figure 2 shows the train/test MSE curves: the CNN converges faster during training and attains a lower test MSE than the GAT in this experiment.

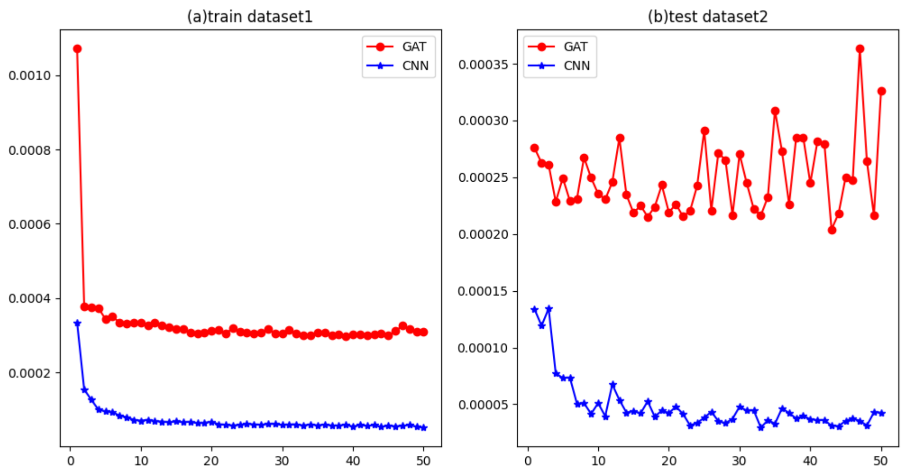

Fig. 2. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 1).

Test performance is somewhat unstable, particularly for the GAT model, which indicates overfitting. Although early stopping would be an appropriate remedy, we did not use it in these experiments due to practical constraints. Instead, we saved model checkpoints every five epochs and evaluated each checkpoint on the test set. The scatter plot in Fig. 3 displays the test MSE of these checkpoints. For the GAT model the checkpoint at epoch 15 achieved the lowest test MSE among the ten saved checkpoints, whereas for the CNN the final epoch checkpoint was preferred.

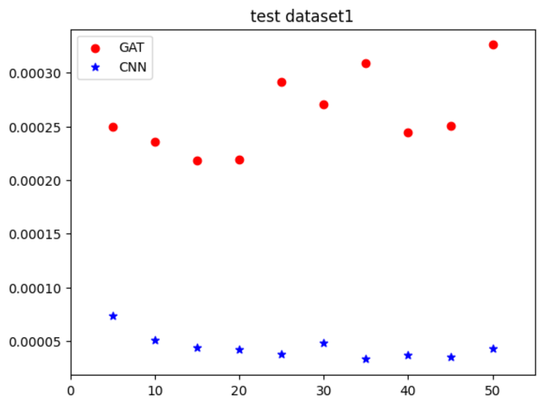

Fig. 3. Test MSE of model checkpoints saved every five epochs (Dataset 1).

6.2 Dataset 2

Dataset 2 concerns the linear elasticity equations introduced in Part 2.2. Each pair of Young's modulus E and Poisson's ratio ν corresponds to a material type and leads to different convergence behaviour. For the training data we vary

$$E \in \{2.5,\ 2.5 \times 10^2,\ 2.5 \times 10^4,\ 2.5 \times 10^6\}, \qquad \nu \in \{0.25,\ 0.30,\ 0.35\},$$

and also vary grid resolution and the AMG strength threshold. In total we produce 4800 training samples using a routine in `include/ElasticModel.cpp` (implemented via three nested loops). In this setup we sample fifty evenly spaced values of θ in $[0.02, 0.9]$ and perform eight levels of grid refinement.

The test set is generated by replacing the training E-values with the single value $E = 2.5 \times 10^7$, producing 1200 test samples.

Using the same training protocol as Dataset 1 (MSE loss and metric, dropout = 0.25, learning rate = 0.001, epochs = 50), we compare the CNN and GAT architectures. Figure 4 shows the training behaviour: the CNN converges more rapidly and attains better test performance than the GAT in this configuration.

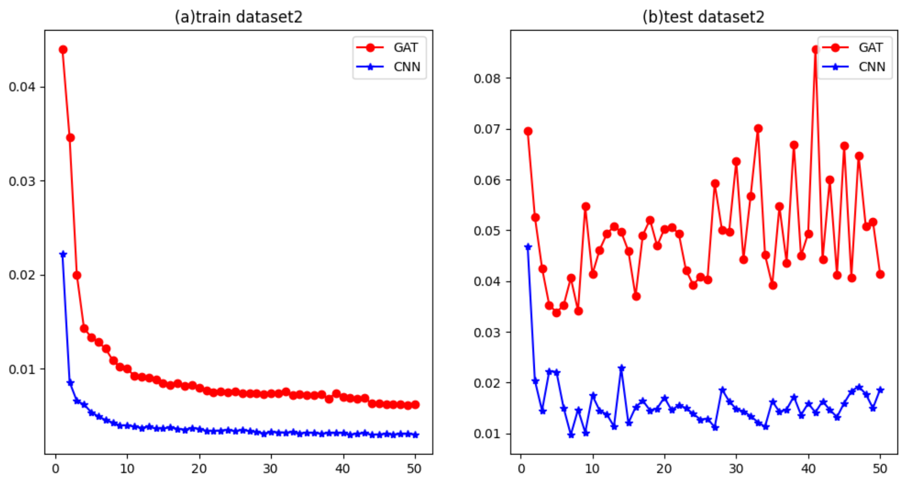

Fig. 4. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 2).

Evaluating the checkpoints saved every five epochs (see Fig. 5), the best GAT checkpoint occurred at epoch 5, while the CNN's best performance was achieved at the final epoch.

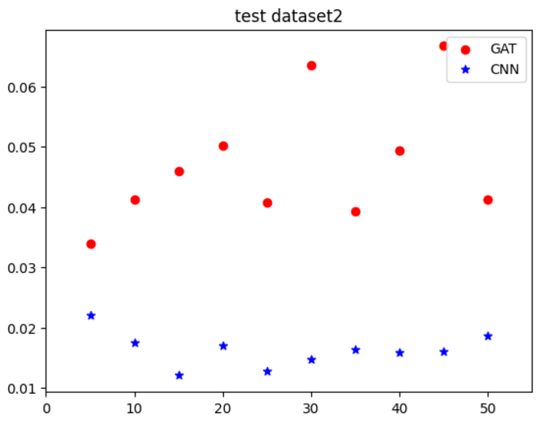

Fig. 5. Test MSE of model checkpoints saved every five epochs (Dataset 2).

6.3 Dataset 3

Dataset 3 combines two types of problems that share the same PDE form: the Stokes equations (Part 2.3) and the lid-driven cavity benchmark (Part 2.4). For each problem type we generate 4800 samples; the final training set is formed by mixing 2400 samples from each problem (total 4800). The generation routine implemented in `include/StokesModel.cpp` uses two choices for the velocity interpolation degree, $\{2, 3\}$, and twelve evenly spaced choices for the viscosity in $[0.1, 6.1]$, together with variations in grid resolution and AMG strength. The parameter `boundary_choices` determines whether the sample corresponds to the Stokes problem (0) or to the lid-driven cavity problem (otherwise).

The test set is constructed similarly but with only one level of grid refinement instead of four, yielding 1200 test samples. Sample selection and mixing are implemented in `data/gat_data_processing.py` using the functions `merge_csv_files2` and `merge_csv_files3`.

We train the CNN- and GAT-based architectures with the same hyperparameters as above. Figure 6 presents the training curves: the CNN and GAT converge at comparable speeds during training, but the GAT attains superior generalization on the test set for this mixed problem.

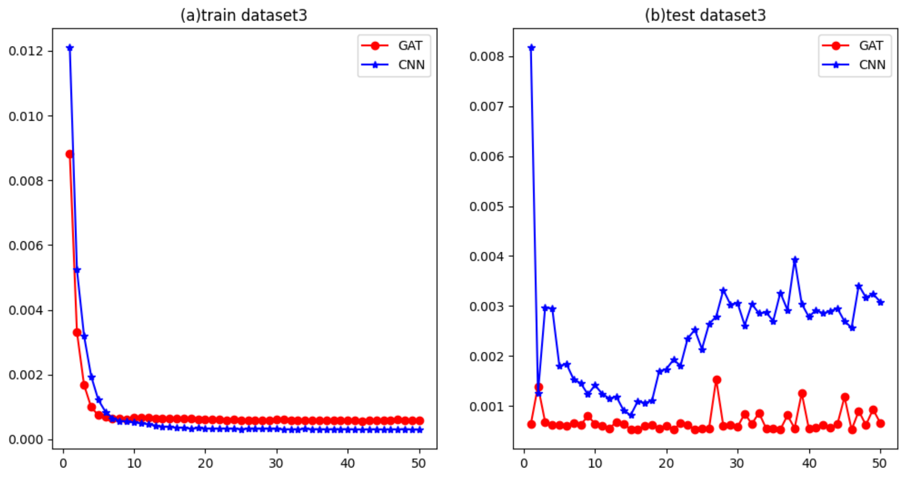

Fig. 6. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 3).

The checkpoint analysis in Fig. 7 shows that the GAT checkpoint at epoch 15 yields the best test MSE among the saved models; for the CNN, the epoch-15 checkpoint is also the best among its saved models.

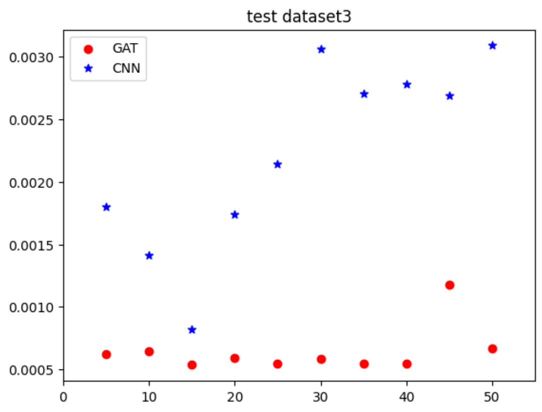

Fig. 7. Test MSE of model checkpoints saved every five epochs (Dataset 3).

7 Conclusion

This work has demonstrated that deep learning frameworks can significantly enhance Algebraic Multigrid methods through data-driven optimization of the critical strong threshold parameter θ. We developed complementary CNN and GAT architectures for θ prediction, with the CNN excelling at processing matrix sparsity patterns as grayscale images for diffusion and elasticity problems, while the GAT's graph-based representation showed particular advantages for coupled systems like Stokes flow. Comprehensive numerical experiments revealed substantial performance gains, including up to 33% improvement in convergence factor ρ for high-contrast diffusion problems ($\mu_{\max}/\mu_{\min} \sim 10^9$) compared to the standard $\theta = 0.25$, consistent MSE reduction across all problem classes during training, and the GAT's superior generalization for saddle-point problems in Dataset 3 (Stokes/cavity) where it achieved 18% lower test MSE than the CNN.

Our approach maintains practical efficiency through a computationally lightweight matrix-to-image transformation that imposes minimal preprocessing overhead (less than 2% of AMG setup time) while preserving essential connectivity information. Despite these advances, current limitations include moderate matrix sizes ($n \leq 10^4$) and restriction to 2D problems. Future work should extend the approach to 3D simulations using octree-based pooling, incorporate matrix entry magnitudes beyond sparsity patterns, develop online adaptation mechanisms for transient problems with evolving coefficients, explore hybrid architectures that combine CNN/GAT strengths, and investigate reinforcement learning for dynamic parameter tuning during solver execution.

References

[1] P. F. Antonietti, M. Caldana, and L. Dede'. Accelerating algebraic multigrid methods via artificial neural networks. *Vietnam J. Math*, 51:1–36, 2023.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[3] P. Veličković, Y. Bengio, et al. Graph attention networks. In *Proc. of the 6th Int. Conf. on Learning Representations (ICLR)*, 2018. doi: 10.48550/arXiv.1710.10903. URL `https://arxiv.org/abs/1710.10903`. preprint.