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 by Two-stage Deep Learning Methods

---

Authors:

Lin Zhang

Tutors:

Prof. Luca Formaggia

Abstract

Algebraic Multigrid (AMG) methods provide efficient solutions for large sparse linear systems arising from partial differential equation (PDE) discretizations. However, the performance of classical AMG depends critically on algorithmic parameters that are typically chosen empirically. This work presents a two-stage deep learning framework to optimize AMG performance by learning optimal parameters from problem characteristics. In Stage 1, we train neural networks to predict the optimal strong threshold parameter θ that controls the coarsening process. We compare two architectures: a Convolutional Neural Network (CNN) that processes pooled matrix representations as images, and a Graph Neural Network (GNN) that operates directly on the sparse matrix graph. In Stage 2, we learn improved prolongation matrix values using a message-passing graph network architecture. We validate our approach on two problem classes: heterogeneous diffusion equations and graph Laplacian problems generated from random Delaunay triangulations. Our data generation pipeline leverages efficient binary formats that accelerate data training compared to standard text-based formats. We provide a complete implementation including C++ modules for finite element assembly and graph generation, Python training scripts for both stages, and evaluation tools for convergence analysis.

Contents

Abstract **1**

1 Introduction **4**

2 Problems Description **5**
2.1 Diffusion Equations in Heterogeneous Media 5
2.2 Linear Elasticity Equations 6
2.3 Stokes Equations for Incompressible Flow 7
2.4 Graph Laplacian Problems 8
2.5 Spectral Clustering 9

3 Algorithms **9**
3.1 Algebraic Multigrid Methods 9
3.2 Neural Network Fundamentals 11
3.3 Convolutional Neural Networks (CNNs) 11
3.4 Graph Neural Networks (GNNs) 12
3.5 Graph Attention Networks (GATs) 13

4 Methods **14**
4.1 Stage 1: Learning Optimal Strong Threshold 14
4.1.1 CNN Architecture for Stage 1 15
4.1.2 GNN Architecture for Stage 1 15
4.2 Stage 2: Learning Prolongation Values 16
4.3 Two-Stage Training Protocol 17

5 Code Organization **17**
5.1 Project Structure 17
5.2 Data Generation Pipeline 18
5.3 Neural Network Training 18
5.4 Key Implementation Details 19

6 Experiments **19**
6.1 Data Generation Efficiency 20
6.2 Experimental Protocol 20
6.2.1 Problem Classes 20
6.2.2 Training Configuration 21
6.3 Experiment 1: Diffusion + Stage 1 GNN 21
6.4 Experiment 2: Diffusion + Stage 1 CNN 21
6.5 Experiment 3: Diffusion + Stage 2 P-value 22
6.6 Experiment 4: Graph Laplacian + Stage 1 GNN 23
6.7 Experiment 5: Graph Laplacian + Stage 1 CNN 23
6.8 Experiment 6: Graph Laplacian + Stage 2 P-value 24
6.9 Summary of Results 25
6.9.1 Key Findings 25

7 Conclusion **28**

Bibliography **29**

A Previous Experiments **30**
A.1 Dataset 1 30
A.2 Dataset 2 31
A.3 Dataset 3 32

1 Introduction

The efficient solution of large sparse linear systems arising from finite element discretizations of partial differential equations (PDEs) constitutes a fundamental challenge in computational science and engineering. As modern simulations tackle increasingly complex physical phenomena, 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 $\mathcal{O}(n^3)$ complexity and substantial memory requirements. Even classical iterative methods like Jacobi or Gauss-Seidel iterations exhibit slow convergence rates for ill-conditioned systems, particularly those stemming from elliptic PDEs with heterogeneous coefficients.

Algebraic Multigrid (AMG) methods have emerged as powerful hierarchical solvers for addressing these challenges, especially for symmetric positive definite (SPD) matrices. Unlike geometric multigrid that relies on explicit mesh hierarchies, AMG constructs its coarse-grid hierarchy purely algebraically using only the system matrix $\mathbf{A}_h$. This matrix-driven approach provides 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 application to diverse problem classes including linear elasticity, electromagnetic simulations, fluid dynamics, and graph-based problems.

At the heart of AMG lies the coarsening process, which partitions degrees of freedom into coarse and fine sets. This partition is governed by the strong threshold parameter θ, which defines "strong connections" between variables. The parameter θ controls the sparsity pattern of the interpolation operators between grid levels, directly impacting the approximation quality of coarse-grid corrections, the convergence rate of the multigrid cycle, and the overall computational efficiency.

In traditional implementations, θ is typically set to a default empirical value of 0.25 based on historical experience with model problems. However, 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 high contrast ratios, the optimal θ can vary significantly from the standard value. Manual parameter tuning through trial-and-error becomes computationally expensive and practically infeasible for large-scale simulations.

Beyond the coarsening threshold, AMG performance also depends critically on the prolongation operator $\mathbf{P}$ that transfers coarse-grid corrections to fine grids. Classical AMG constructs $\mathbf{P}$ using fixed algebraic formulas based on matrix entries. Recent research has shown that learning improved prolongation values from data can further enhance convergence. This observation motivates a two-stage optimization approach: first optimize the coarsening structure via θ selection, then refine the prolongation values within that structure.

The emergence of machine learning, particularly deep learning with neural networks, offers transformative potential for optimizing numerical algorithms. Neural networks can learn complex, nonlinear relationships from high-dimensional data, making them well-suited for parameter optimization in scientific computing. Recent applications include adaptive selection of stabilization parameters, guidance for mesh

refinement strategies, and optimization of preconditioners.

This work develops a two-stage deep learning framework for AMG optimization. In Stage 1, we train neural networks to predict the optimal strong threshold θ that minimizes the convergence factor ρ. We compare two approaches: a Convolutional Neural Network (CNN) that processes pooled matrix representations as images, and a Graph Neural Network (GNN) that operates directly on the sparse matrix graph structure. In Stage 2, we train a message-passing graph network to learn improved prolongation matrix values, building on the coarsening structure from Stage 1.

Our principal contributions include: (1) A two-stage learning framework that sequentially optimizes AMG coarsening and prolongation; (2) Comparison of CNN and GNN architectures for θ prediction on different problem classes; (3) An efficient data generation pipeline leveraging binary formats and text-based formats; (4) Comprehensive validation on heterogeneous diffusion equations and graph Laplacian problems from random Delaunay triangulations; (5) A complete open-source implementation integrating C++ modules for problem generation and Python modules for neural network training.

The remainder of this report is structured as follows. Section 2 provides mathematical foundations for the model problems including diffusion equations, linear elasticity, Stokes equations and graph Laplacians. Section 3 details AMG algorithms and neural network fundamentals. Section 4 describes our two-stage learning approach and network architectures. Section 5 explains the code organization and implementation. Section 6 presents experimental methodology and results, and Section 7 concludes with future 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

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

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

$$-\text{div}\left(\mathbf{C}\epsilon(\mathbf{u})\right) = \mathbf{f},$$

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

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

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 Graph Laplacian Problems

Graph Laplacian matrices arise naturally in many applications including network analysis, image segmentation, and machine learning. For a weighted undirected graph $G = (V, E)$ with vertices V and edges E, the graph Laplacian matrix $\mathbf{L} \in \mathbb{R}^{n \times n}$ encodes the connectivity structure. Given edge weights $w_{ij} > 0$ for each edge $(i, j) \in E$, the Laplacian is defined as

$$\mathbf{L}_{ij} = \begin{cases} \sum_{k \in N(i)} w_{ik} & \text{if } i = j \\ -w_{ij} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where $N(i)$ denotes the neighbor set of vertex i. The matrix $\mathbf{L}$ is symmetric positive semidefinite with a one-dimensional nullspace spanned by the constant vector.

In this work, we generate graph Laplacian problems using random Delaunay triangulations. Given n random points sampled uniformly in the unit square $[0, 1]^2$, the Delaunay triangulation connects points such that no point lies inside the circumcircle of any triangle. This produces a planar graph with favorable connectivity properties. For each edge in the triangulation, we assign weights using the standard finite element approach:

$$w_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}x \tag{10}$$

where ϕ_i and ϕ_j are piecewise linear basis functions associated with vertices i and j. The resulting graph Laplacian $\mathbf{L}$ is the stiffness matrix for the Poisson equation $-\Delta u = f$ discretized on the Delaunay mesh.

These graph-based problems provide an important test case for AMG methods because they lack the regular structure of FEM discretizations on structured meshes. The random geometry and varying node degrees present challenges for coarsening algorithms, making optimal parameter selection particularly important.

2.5 Spectral Clustering

Spectral clustering is a widely used technique for partitioning data into groups based on pairwise similarity measures. The method constructs a graph where vertices represent data points and edge weights encode similarities. Given n data points $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, a common approach builds a k-nearest neighbor (k-NN) graph: each point is connected to its k nearest neighbors with edge weights computed from distances.

For the k-NN graph, typical weight functions include:

$$w_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right) \tag{11}$$

where σ is a bandwidth parameter. The graph Laplacian constructed from these weights encodes the local geometric structure of the data manifold.

Spectral clustering requires solving an eigenvalue problem involving the graph Laplacian. For large-scale problems with thousands to millions of data points, computing the smallest nontrivial eigenvectors becomes computationally demanding. AMG methods provide an efficient approach for this task when used as preconditioners for iterative eigensolvers. The performance of AMG on spectral clustering problems depends on the graph structure and edge weight distribution, making parameter optimization valuable for practical applications.

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 [1]:

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

$$-(\mathbf{A}_h)_{ij} \geq \theta \max_{k \neq i}\{-(\mathbf{A}_h)_{ik}\}, \quad 0 < \theta \leq 1 \tag{12}$$

This definition captures dominant connections in the matrix graph. The choice of θ critically impacts AMG performance. Small values of θ (say $\theta \ll 0.25$) classify more connections as strong, leading to denser coarse grids with better interpolation quality but increased setup cost. Large values of θ (say $\theta \gg 0.25$) classify fewer connections as strong, resulting in sparser coarsening with faster setup but potentially poorer convergence. The standard default value $\theta = 0.25$ represents a balance based on experience with model problems, but optimal values can vary significantly depending on problem characteristics.

After identifying strong connections, the C/F splitting algorithm partitions variables into coarse and fine sets. Classical Ruge-Stuben coarsening proceeds by selecting coarse variables to ensure that each fine variable is strongly connected to at least one coarse variable. This ensures that fine-grid errors can be well-approximated using coarse-grid corrections.

Once the C/F splitting is determined, the prolongation (interpolation) operator $\mathbf{P} = \mathbf{I}_H^h : \mathbb{R}^{n_H} \to \mathbb{R}^n$ is constructed. This operator maps coarse-level vectors to fine-level vectors and plays a central role in the multigrid hierarchy. For $i \in \mathscr{F}_h$, the prolongation weights to coarse neighbors $j \in \mathscr{C}_h \cap \mathscr{P}_i$ are computed algebraically:

$$(\mathbf{P}\mathbf{e}_H)_i = \begin{cases} \mathbf{e}_i & i \in \mathscr{C}_h \\ \sum_{j \in \mathscr{P}_i} w_{ij} (\mathbf{e}_H)_j & i \in \mathscr{F}_h \end{cases} \tag{13}$$

where $\mathscr{P}_i \subset \mathscr{C}_h$ denotes the interpolatory set for fine variable i, typically consisting of coarse variables strongly connected to i.

Classical AMG uses direct interpolation formulas to compute the weights w_{ij}. For symmetric positive definite problems, these weights are designed to preserve certain properties of the error. A standard approach computes:

$$w_{ij} = \frac{-(\mathbf{A}_h)_{ij} + \alpha_i \sum_{m \in \mathscr{S}_i \cap \mathscr{F}_h} \frac{-(\mathbf{A}_h)_{im}}{|\mathscr{C}_i \cap \mathscr{P}_m|}}{\sum_{k \in \mathscr{P}_i} (-(\mathbf{A}_h)_{ik})} \tag{14}$$

where α_i and additional correction terms account for contributions from fine neighbors and weak connections. The quality of these weights directly affects the AMG convergence rate.

For SPD matrices, the restriction operator is chosen as $\mathbf{R} = \mathbf{I}_h^H = \mathbf{P}^\top$, and the Galerkin product constructs coarse operators: $\mathbf{A}_H = \mathbf{R}\mathbf{A}_h\mathbf{P} = \mathbf{P}^\top\mathbf{A}_h\mathbf{P}$. This construction ensures that the coarse-level system inherits the SPD property and that the two-level method converges when the smoothing iteration is convergent.

The convergence of AMG is characterized by the error propagation matrix of the two-grid cycle. With s_1 pre-smoothing and s_2 post-smoothing steps using smoother $\mathbf{S}$, the two-grid error propagation matrix is:

$$\mathbf{M} = \mathbf{S}^{s_2}(\mathbf{I} - \mathbf{P}\mathbf{A}_H^{-1}\mathbf{R}\mathbf{A}_h)\mathbf{S}^{s_1} \tag{15}$$

The convergence factor $\rho = \|\mathbf{M}\|$ (typically the spectral radius) determines how fast errors are reduced per cycle. Smaller values of ρ correspond to faster convergence. Both the choice of θ (affecting the C/F splitting and sparsity of $\mathbf{P}$) and the values of the prolongation weights w_{ij} influence ρ.

The solution process can employ two-level algorithm (Algorithm 1), combining pre-smoothing (S_1), coarse-grid correction, and post-smoothing (S_2):

Algorithm 1 Two-Level Algorithm

1: **Input:** SPD matrix $A \in \mathbb{R}^{n\times n}$, initial approximation $x^{(0)} \in \mathbb{R}^n$, right-hand side $b \in \mathbb{R}^n$, full-rank prolongation matrix $P \in \mathbb{R}^{n\times n_c}$, a relaxation scheme, $k = 0$, residual tolerance δ.
2: **repeat**
3: Perform s_1 relaxation sweeps starting with the current approximation $x^{(k)}$, obtaining $\tilde{x}^{(k)}$.
4: Compute the residual: $r^{(k)} = b - A\tilde{x}^{(k)}$.
5: Project the error equations to the coarser level and solve the coarse-level system: $A_c e_c^{(k)} = P^T r^{(k)}$, with $A_c = P^T AP$.
6: Prolongate and add the coarse-level solution: $\tilde{x}^{(k)} = \tilde{x}^{(k)} + Pe_c^{(k)}$.
7: Perform s_2 relaxation sweeps obtaining $x^{(k+1)}$.
8: $k = k + 1$.
9: **until** $\|r^{(k-1)}\| < \delta$.

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:

$$\mathbf{z}^{(0)} = \mathbf{x} \tag{16}$$

$$\mathbf{a}^{(k)} = \mathbf{W}^{(k)}\mathbf{z}^{(k-1)} + \mathbf{b}^{(k)} \quad k = 1, \ldots, L \tag{17}$$

$$\mathbf{z}^{(k)} = \sigma(\mathbf{a}^{(k)}) \quad k = 1, \ldots, L-1 \tag{18}$$

$$\mathbf{y} = g(\mathbf{a}^{(L)}) \tag{19}$$

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:

$$\delta^{(L)} = \nabla_{\mathbf{a}^{(L)}} \mathscr{L} \tag{20}$$

$$\delta^{(k)} = (\mathbf{W}^{(k+1)})^T \delta^{(k+1)} \odot \sigma'(\mathbf{a}^{(k)}) \quad k = L-1, \ldots, 1 \tag{21}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^{(k)}} = \delta^{(k)}(\mathbf{z}^{(k-1)})^T, \quad \frac{\partial \mathscr{L}}{\partial \mathbf{b}^{(k)}} = \delta^{(k)} \tag{22}$$

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 $\mathbf{K} \in \mathbb{R}^{k_h \times k_w \times C \times F}$:

$$(\mathbf{X} * \mathbf{K})_{i,j,f} = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} \sum_{c=1}^{C} \mathbf{K}_{m,n,c,f} \mathbf{X}_{i+m,j+n,c} \tag{23}$$

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

$$\text{Output size} = \left\lfloor \frac{H - k_h}{s} + 1 \right\rfloor \times \left\lfloor \frac{W - k_w}{s} + 1 \right\rfloor \times F \tag{24}$$

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 (non-linearity), pooling layers (downsampling), and culminates in fully-connected layers (task-specific processing).

3.4 Graph Neural Networks (GNNs)

Graph Neural Networks provide a framework for learning on graph-structured data by iteratively updating node representations through message passing between neighbors. Given a graph $G = (V, E)$ with node features $\mathbf{h}_i \in \mathbb{R}^F$ for $i \in V$ and edge features $\mathbf{e}_{ij} \in \mathbb{R}^{F_e}$ for $(i, j) \in E$, a GNN layer transforms node features based on local graph structure.

The general message-passing framework consists of two key operations: message computation and node update. In each layer, node i receives messages from its neighbors $\mathscr{N}(i)$ and updates its feature representation. A generic GNN layer can be written as:

$$\mathbf{m}_{ij} = \text{MESSAGE}(\mathbf{h}_i, \mathbf{h}_j, \mathbf{e}_{ij}) \tag{25}$$
$$\mathbf{m}_i = \text{AGGREGATE}(\{\mathbf{m}_{ij} : j \in N(i)\}) \tag{26}$$
$$\mathbf{h}'_i = \text{UPDATE}(\mathbf{h}_i, \mathbf{m}_i) \tag{27}$$

where MESSAGE computes messages from neighbors, AGGREGATE combines incoming messages (typically using sum, mean, or max), and UPDATE produces the new node representation.

Different GNN architectures implement these operations in various ways. Graph Convolutional Networks (GCNs) use normalized aggregation:

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in N(i) \cup \{i\}} \frac{1}{\sqrt{d_i d_j}} \mathbf{W} \mathbf{h}_j \right) \tag{28}$$

where d_i denotes the degree of node i, $\mathbf{W}$ is a learnable weight matrix, and σ is an activation function. This normalization helps prevent numerical instabilities in deep networks.

For problems involving edge features (such as sparse matrices with weighted edges), message functions can incorporate edge information. A simple approach defines:

$$\mathbf{m}_{ij} = \text{MLP}([\mathbf{h}_i \| \mathbf{h}_j \| \mathbf{e}_{ij}]) \tag{29}$$

where $\|$ denotes concatenation and MLP is a multi-layer perceptron. This allows the network to learn how edge weights influence message propagation.

After multiple message-passing layers, a graph-level readout function aggregates node features to produce a graph-level representation:

$$\mathbf{h}_{\text{graph}} = \text{READOUT}(\{\mathbf{h}'_i : i \in V\}) \tag{30}$$

Common readout functions include global mean pooling, global max pooling, or more sophisticated learned aggregations.

For AMG applications, sparse matrices are naturally represented as weighted graphs where nodes correspond to degrees of freedom and edges correspond to matrix entries. Node features can encode local properties such as degree or diagonal entries, while edge features encode matrix values. GNNs can then learn to predict optimal AMG parameters by processing this graph representation directly, without requiring pooling or image conversion as in CNN approaches.

3.5 Graph Attention Networks (GATs)

Here we introduce the GATs neural network applied in our previous work. The relevant results of experiments can be seen in the Appendix part. GATs[4] 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:

$$\mathbf{g}_i = \mathbf{W}\mathbf{h}_i, \quad \mathbf{W} \in \mathbb{R}^{F' \times F} \tag{31}$$

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

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T[\mathbf{g}_i \| \mathbf{g}_j]\right), \quad \mathbf{a} \in \mathbb{R}^{2F'} \tag{32}$$

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

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

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathscr{N}(i)} \exp(e_{ik})} \tag{33}$$

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

$$\mathbf{h}'_i = \sigma\left(\sum_{j \in \mathscr{N}(i)} \alpha_{ij} \mathbf{g}_j\right) \tag{34}$$

Multi-head attention enhances representational power and stability:

$$\mathbf{h}_i' = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_j\right) \tag{35}$$

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

$$\mathbf{h}_i' = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_j\right) \tag{36}$$

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.

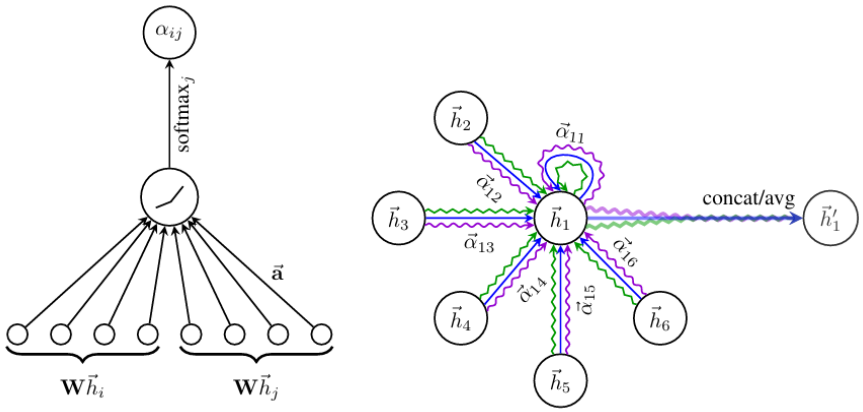

Fig. 1. GAT computational architecture: (a) Attention mechanism computing coefficients α_{ij}, (b) Multi-head aggregation combining features from different representation subspaces

4 Methods

This work develops a two-stage learning framework for optimizing AMG performance. Stage 1 learns the optimal strong threshold parameter θ that determines the coarsening structure. Stage 2 learns improved prolongation matrix values within the structure determined by Stage 1. This sequential approach decouples two key decisions in AMG: which variables to coarsen (governed by θ) and how to interpolate between levels (governed by prolongation values).

4.1 Stage 1: Learning Optimal Strong Threshold

The goal of Stage 1 is to learn a mapping from problem characteristics to the optimal strong threshold θ^* that minimizes the AMG convergence factor ρ. We formulate this as a supervised regression problem:

$$\theta^* = f_{\theta_{\text{NN}}}(\mathbf{A}_h, h) \tag{37}$$

where $f_{\theta_{\text{NN}}}$ is a neural network parameterized by weights θ_{NN}, $\mathbf{A}_h$ is the system matrix, and h is the mesh size parameter.

We compare two architectural approaches for implementing $f_{\theta_{\text{NN}}}$: a CNN-based approach that processes pooled matrix representations and a GNN-based approach that operates directly on the sparse matrix graph.

4.1.1 CNN Architecture for Stage 1

Inspired by the work of Antonietti et al.[1], the CNN approach treats the sparse matrix sparsity pattern as an image. Since the raw matrix is too large for direct CNN processing, we apply a pooling operation that reduces the matrix to a fixed size $m \times m$ (typically $m = 50$).

The pooling operation partitions the $n \times n$ sparse matrix into $m \times m$ blocks and aggregates entries within each block. For block (I, J), the pooled value is computed as:

$$V_{IJ} = \frac{1}{|B_{IJ}|} \sum_{(i,j) \in B_{IJ}} |(\mathbf{A}_h)_{ij}| \tag{38}$$

where B_{IJ} denotes the set of nonzero entries in block (I, J). This produces a dense 50×50 matrix that preserves the overall sparsity pattern and coupling strength distribution.

The CNN architecture processes this pooled matrix as a single-channel grayscale image:

$$\mathbf{X} \in \mathbb{R}^{50 \times 50 \times 1} \quad \text{(pooled matrix)} \tag{39}$$

$$\mathbf{Z}_1 = \text{ReLU}(\text{Conv2D}(\mathbf{X}; 64 \text{ filters})) \tag{40}$$

$$\mathbf{Z}_2 = \text{MaxPool}(\text{ReLU}(\text{Conv2D}(\mathbf{Z}_1; 64 \text{ filters}))) \tag{41}$$

$$\mathbf{Z}_3 = \text{AdaptiveAvgPool}(\text{ReLU}(\text{Conv2D}(\mathbf{Z}_2; 32 \text{ filters}))) \tag{42}$$

$$\mathbf{z}_{\text{flat}} = \text{Flatten}(\mathbf{Z}_3) \in \mathbb{R}^{128} \tag{43}$$

The flattened CNN features are concatenated with scalar features $[\theta, -\log_2(h)]$ and passed through fully connected layers:

$$\mathbf{z}_{\text{combined}} = [\mathbf{z}_{\text{flat}} \| \theta \| - \log_2(h)] \in \mathbb{R}^{130} \tag{44}$$

$$\hat{\rho} = \text{FC}_2(\text{ReLU}(\text{FC}_1(\mathbf{z}_{\text{combined}}))) \tag{45}$$

where FC denotes fully connected layers. The network is trained to minimize mean squared error between predicted $\hat{\rho}$ and true convergence factor ρ:

$$\mathscr{L}_{\text{Stage1}} = \frac{1}{N} \sum_{i=1}^{N} (\hat{\rho}_i - \rho_i)^2 \tag{46}$$

4.1.2 GNN Architecture for Stage 1

The GNN approach processes the sparse matrix directly as a weighted graph without pooling. The graph representation is: nodes correspond to degrees of freedom, edges correspond to nonzero matrix entries, node features are degrees $d_i = |\{j : (\mathbf{A}_h)_{ij} \neq 0\}|$, and edge features are matrix values $(\mathbf{A}_h)_{ij}$.

We use a simple GNN architecture with message-passing layers:

$$\mathbf{h}_i^{(0)} = [d_i] \in \mathbb{R}^1 \quad \text{(initial node features)} \tag{47}$$

$$\mathbf{m}_{ij}^{(l)} = \text{MLP}_{\text{msg}}^{(l)}([\mathbf{h}_i^{(l)} \| \mathbf{h}_j^{(l)} \| (\mathbf{A}_h)_{ij}]) \tag{48}$$

$$\mathbf{h}_i^{(l+1)} = \text{MLP}_{\text{update}}^{(l)}([\mathbf{h}_i^{(l)} \| \frac{1}{|N(i)|} \sum_{j \in N(i)} \mathbf{m}_{ij}^{(l)}]) \tag{49}$$

After L message-passing layers (typically $L = 2$), node features are aggregated using global mean pooling:

$$\mathbf{z}_{\text{graph}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{h}_i^{(L)} \in \mathbb{R}^F \tag{50}$$

The graph embedding is concatenated with scalar features and processed through fully connected layers as in the CNN approach.

4.2 Stage 2: Learning Prolongation Values

Stage 2 builds on Stage 1 by learning improved prolongation matrix values. Given a system matrix $\mathbf{A}_h$ and the strong threshold θ, we first compute the C/F splitting and the sparsity pattern of the prolongation matrix $\mathbf{P}$ using classical AMG algorithms. Stage 2 then learns to fill in the nonzero values of $\mathbf{P}$ to minimize the two-grid convergence factor.

We adopt the encode-process-decode architecture from graph network literature [3]. The network operates on an augmented graph that includes both the original matrix structure and the prolongation pattern. Key inputs include: the system matrix $\mathbf{A}_h$, C/F splitting (binary label for each node), strength matrix $\mathbf{S_m}$ (strong connections), and baseline prolongation $\mathbf{P}_{\text{base}}$ (classical AMG formula).

The architecture consists of three components:

Encoder: Maps raw inputs to latent representations. Node features encode whether a node is coarse or fine using one-hot vector. Edge features encode matrix entries and whether an edge is in the prolongation pattern using one-hot representation.

Processor: Applies multiple rounds of message passing to propagate information through the graph. Each round updates node features, edge features, and global graph features using learned update functions. This allows the network to reason about long-range dependencies in the multigrid hierarchy.

Decoder: Maps updated edge features to predicted prolongation values. The decoder outputs corrections Δw_{ij} to the baseline weights: $w_{ij} = w_{ij}^{\text{base}} + \Delta w_{ij}$.

The training objective minimizes the two-grid convergence factor:

$$\mathscr{L}_{\text{Stage2}} = \|\mathbf{M}(\mathbf{P})\|_F^2 \tag{51}$$

where $\mathbf{M}(\mathbf{P}) = \mathbf{S}^{s_2}(\mathbf{I} - \mathbf{P}\mathbf{A}_H^{-1}\mathbf{P}^\top \mathbf{A}_h)\mathbf{S}^{s_1}$ is the error propagation matrix and $\mathbf{A}_H = \mathbf{P}^\top \mathbf{A}_h \mathbf{P}$. In practice, we use the Frobenius norm as a surrogate for the spectral radius since it is differentiable and easier to compute.

4.3 Two-Stage Training Protocol

Next, we propose an overall training procedure. At first, train Stage 1 model (CNN or GNN) to predict optimal θ for 50-100 epochs using MSE loss on convergence factors. Then save the best Stage 1 model based on validation performance. Thirdly, freeze the Stage 1 model and use it to generate C/F splittings for Stage 2 training data. After that, train Stage 2 model for 50-100 epochs using two-grid loss on convergence factors. Finally, deploy the complete two-stage pipeline: for a new problem, Stage 1 predicts θ^*, classical AMG performs C/F splitting and computes baseline prolongation, then Stage 2 refines the non-zero values of the prolongation matrix.

This sequential approach allows each stage to focus on a specific aspect of AMG optimization while maintaining computational tractability. Note that it is a proposal that needs to be implemented in the future.

5 Code Organization

The project implements a complete pipeline for AMG optimization through deep learning, integrating C++ modules for efficient data generation and Python modules for neural network training. The codebase is organized to support both stages of the learning framework.

5.1 Project Structure

The project follows this directory structure:

```
amg-learning/
          include/                          # C++ header files
                DiffusionModel.hpp            # Diffusion FEM solver
                ElasticModel.hpp              # Elasticity FEM solver
                StokesModel.hpp               # Stokes FEM solver
                GraphLaplacianModelEigen.hpp # Graph Laplacian via Delaunay
                AMGOperators.hpp              # C/F split, prolongation, strength
                Pooling.hpp                   # Parallel pooling for CNN
                NPYWriter.hpp                 # NumPy .npy format writer
                NPZWriter.hpp                 # NumPy .npz format writer
                UnifiedDataGenerator.hpp    # Orchestrates data generation
          src/
                generate_amg_data.cpp         # Unified dataset generator
          model/                             # Neural network models
                cnn_model.py                    # CNN for Stage 1 (theta)
                gnn_model.py                    # GNN for Stage 1 (theta)
                graph_net_model.py              # EncodeProcessDecode for Stage 2
                unified_model.py                # Two-stage combined model
          data/                               # Data processing (legacy CSV)
                cnn_data_processing.py
                gnn_data_processing.py
```

```
        data_loader_npy.py              # High-performance NPZ loaders
        utils/                          # Training utilities
            multigrid_utils.py           # Two-grid convergence analysis
            amg_utils.py                 # AMG operators in Python
            training_utils.py            # Checkpointing, logging
        train_stage1.py                 # Train Stage 1 models
        train_stage2.py                 # Train Stage 2 models
        evaluate.py                     # Model evaluation
        datasets/                       # Generated data
            unified/
                train/raw/              # Training data (NPZ format)
                test/raw/               # Test data (NPZ format)
        weights/                        # Trained model checkpoints
        logs/                           # Training logs
        results/                        # Evaluation results
```

5.2 Data Generation Pipeline

The C++ data generation module is built on deal.II and Eigen libraries. The unified generator (`generate_amg_data.cpp`) supports five problem types through a command-line interface:

```
# Generate diffusion training data in all formats
./build/generate_amg_data -p D -s train -f all -c medium

# Generate graph Laplacian test data
./build/generate_amg_data -p GL -s test -f theta_gnn -c large
```

Problem types include Diffusion (D), Elasticity (E), Stokes (S), Graph Laplacian (GL), and Spectral Clustering (SC). Output formats include `theta_cnn` for CNN training (pooled matrices), `theta_gnn` for GNN training (sparse graphs), and `p_value` for Stage 2 training (full AMG data).

A key innovation is the efficient NPZ binary format. Compared to text-based CSV format, NPZ provides faster loading speed during the data training. Each NPZ file contains a complete sample with arrays for matrix entries, graph structure, and metadata. The `NPZWriter` class handles compression and efficient storage of sparse matrices in CSR format.

For each sample, the generation process follows these steps: (1) Generate or discretize the problem to obtain system matrix $\mathbf{A}_h$. (2) For a range of θ values, compute the AMG convergence factor ρ using PETSc's BoomerAMG. (3) Store the matrix and corresponding (θ, ρ, h) pairs. (4) For `p_value` format, additionally compute and store C/F splitting, strength matrix, and prolongation matrix.

5.3 Neural Network Training

The training pipeline is implemented in PyTorch and PyTorch Geometric. Stage 1 training (`train_stage1.py`) accepts flexible command-line arguments:

```
# Train GNN model on graph Laplacian data
python train_stage1.py --model GNN --dataset datasets/unified \
  --train-file train_GL --test-file test_GL --use-npy \
```

```
  --epochs 50 --batch-size 32 --lr 0.001 --hidden-channels 64 \
  --save-dir weights/GL_stage1_gnn

# Train CNN model on diffusion data
python train_stage1.py --model CNN --dataset datasets/unified \
  --train-file train_D.csv --test-file test_D.csv \
  --epochs 50 --batch-size 64 --save-dir weights/D_stage1_cnn
```

The `-use-npy` flag enables high-performance NPZ data loading through `data_loader_npy.py`, which provides `GNNThetaDatasetNPY` and `PValueDatasetNPY` classes. These classes use memory mapping for efficient large-scale dataset access.

Stage 2 training (`train_stage2.py`) can start with the command line like the following:

```
python train_stage2.py --dataset datasets/unified \
  --train-file train_GL --test-file test_GL --use-npy \
  --epochs 100 --batch-size 16 --lr 0.003 \
  --latent-size 64 --num-layers 4 --num-message-passing 3 \
  --save-dir weights/GL_stage2_pvalue
```

The training loop implements early stopping, learning rate scheduling, and checkpointing. Models are evaluated on validation sets every epoch, and the best model (lowest validation loss) is saved. Training logs are written to TensorBoard-compatible format for visualization.

5.4 Key Implementation Details

Parallel Pooling: The `Pooling.hpp` module implements OpenMP-parallelized pooling to reduce sparse matrices to dense 50×50 images. The algorithm partitions rows across threads and uses block-wise aggregation to ensure load balancing.

AMG Operators: The `AMGOperators.hpp` module provides C++ implementations of strength matrix computation, Ruge-Stuben C/F splitting, and direct interpolation. These operators generate ground truth prolongation matrices for Stage 2 training.

Two-Grid Analysis: The Python `multigrid_utils.py` module computes two-grid error propagation matrices and convergence factors. This is used both for generating training labels and for computing the Stage 2 loss function.

Data Processing: NPY/NPZ writers support storaging data in NPZ format, which save storage space and data loading time. It's useful for the following training work.

This modular architecture enables independent development and testing of each component while maintaining end-to-end integration for the complete learning pipeline.

6 Experiments

This section presents our experimental methodology and results for the two-stage AMG optimization framework. We begin by evaluating data generation efficiency, then describe the comprehensive six-experiment protocol that validates both stages across different problem classes and architectures.

6.1 Data Generation Efficiency

A critical component of any machine learning pipeline is efficient data generation and loading. We compare two data formats: text-based CSV and binary NPZ (compressed NumPy format).

The CSV format stores each matrix sample as a text row containing metadata, matrix values, row pointers, and column indices. While human-readable, CSV requires string parsing and type conversion during loading, creating performance bottlenecks for large datasets.

The NPZ format stores each sample as a compressed binary file containing NumPy arrays. Matrix data is stored directly in memory-mapped format, eliminating parsing overhead. For sparse matrices, we use CSR representation with separate arrays for values, row pointers, and column indices.

We measured generation speed for different sizes of datasets from the graph Laplacian problem on a system with Intel i7-8750H CPU and 16GB RAM. The table 1 shows different generation speeds with 2 formats (CSV and NPZ) and 5 choices of the number of threads for OpenMP.

Table 1. Data Generation Speed for the Graph Laplacian Problem (samples/s)

Scale	**auto Threads**	**1 Thread**	**2 Threads**	**4 Threads**	**8 Threads**
small	368.72 (156.48)	445.05 (235.56)	**586.66** (**260.87**)	534.21 (252.69)	349.75 (169.46)
medium	384.18 (138.25)	216.74 (144.36)	368.61 (**203.20**)	**491.14** (200.42)	382.81 (135.77)
large	470.28 (180.05)	472.54 (246.68)	671.95 (**281.16**)	**696.03** (281.03)	422.89 (156.24)
xlarge	106.24 (**78.94**)	25.54 (24.32)	50.11 (44.65)	88.80 (73.55)	**113.30** (63.68)

Note: Values are in samples per second (samples/s). Higher is better. Numbers outside parentheses are for CSV format; numbers inside parentheses are for NPZ format. **Bold** numbers indicate the fastest speed for a given scale and format (CSV or NPZ).

The CSV format can achieve over 2-fold speedup in most generation and loading. However, the loading of CSV files is time-consuming and costs much storage of CPU or GPU during the training phases. Hence, for the remainder of our experiments, we use the NPZ format exclusively.

6.2 Experimental Protocol

We conduct six experiments following a 2×3 factorial design: two problem classes (Diffusion and Graph Laplacian) crossed with three approaches (Stage 1 with GNN, Stage 1 with CNN, and Stage 2 with P-value learning).

6.2.1 Problem Classes

Diffusion Problems (D): We generate heterogeneous diffusion equations on the unit square $[0, 1]^2$. Training data includes 12 values of the contrast parameter

$\epsilon \in [0, 9.5]$, 2 mesh refinement levels, and 50 values of $\theta \in [0.02, 0.9]$, yielding approximately 1200 training samples. Test data uses unseen ϵ values to evaluate generalization.

Graph Laplacian Problems (GL): We generate random graphs via Delaunay triangulation of uniformly sampled points in the unit square. Each graph has 64-512 nodes depending on problem size. The graph Laplacian matrix is assembled using standard finite element weights. Training data includes 5000 samples with 64×64 nodes and a given random seed. Test data uses different random seeds to evaluate robustness to graph topology variations.

6.2.2 Training Configuration

All models are trained on a laptop with Intel Core i7-8750H and 16GB RAM.

Stage 1 training parameters: 50 epochs, batch size 32 for both GNN and CNN, learning rate 0.001 with ReduceLROnPlateau scheduler (patience 5, factor 0.5), dropout 0.25, hidden channels 64, early stopping with patience 10 on validation loss.

Stage 2 training parameters: 50 epochs, batch size 32, learning rate 0.001, latent size 64, 4 layers, 3 message-passing rounds, patience 15 for early stopping.

6.3 Experiment 1: Diffusion + Stage 1 GNN

This experiment trains a GNN to predict optimal θ for diffusion problems. The GNN processes sparse matrices directly as graphs with node features (degrees) and edge features (matrix values).

The training exhibits significant instability. Initial validation loss shows extreme spikes exceeding 10^9, suggesting numerical overflow or gradient explosion in early epochs. While training loss decreases steadily from 15,704 in epoch 1 to 0.062 in epoch 50, validation loss remains erratic with large fluctuations. The model achieves best validation loss of 3.997×10^{-4} at epoch 50, but test performance reveals fundamental problems.

Final test metrics show the model fails to learn meaningful predictions: MSE of 4.00×10^{-4}, MAE of 0.0156, and critically, a negative R^2 score of -14.645. The negative correlation coefficient (-0.856) indicates predictions are inversely related to true values. Mean predictions (0.0357) differ substantially from mean targets (0.0204), showing systematic bias.

This failure likely stems from GNN architecture challenges with the diffusion problem structure. The highly regular connectivity pattern from FEM discretization may not provide sufficient graph-structural information for the message-passing mechanism to exploit.

6.4 Experiment 2: Diffusion + Stage 1 CNN

This experiment trains a CNN on pooled 50×50 matrix images for diffusion problems. The pooling operation aggregates the sparse matrix while preserving overall structure.

The CNN demonstrates excellent performance and stable training. Training loss decreases smoothly from 7.92×10^{-4} in epoch 1 to 1.79×10^{-6} in epoch 37, where

early stopping triggers. Validation loss follows a similar smooth trajectory, reaching a best value of 1.02×10^{-7} at epoch 26 without the erratic behavior observed in the GNN experiment.

Test metrics confirm exceptional prediction quality: MSE of 1.02×10^{-7} (nearly 4000 times better than GNN), MAE of 0.0002, relative error of only 1.15%, and R^2 score of 0.996. The correlation coefficient of 0.999 indicates near-perfect linear relationship between predictions and targets. Mean predictions (0.0203) closely match mean targets (0.0204), showing minimal bias.

This strong performance demonstrates that CNNs can effectively learn structural patterns in pooled matrix representations for regular FEM discretizations. The 50×50 pooled image captures sufficient information about the diffusion coefficient structure and mesh topology while providing a regular input format well-suited to convolutional processing. The CNN's success on diffusion problems, contrasted with GNN's failure, suggests that the problem structure favors image-based representation over graph-based message passing for this problem class.

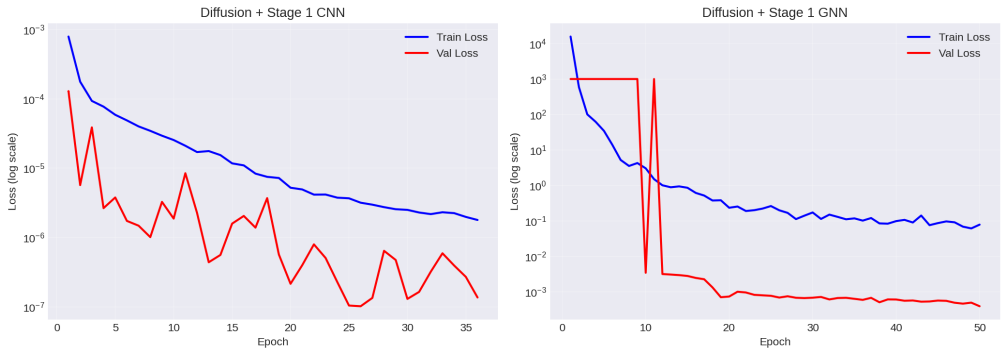

Fig. 2. Training curves for Diffusion Stage 1 experiments. Left: CNN shows smooth convergence with training and validation losses decreasing steadily. Right: GNN exhibits severe instability with validation loss spikes exceeding 10^9 in early epochs, demonstrating training failure despite decreasing training loss.

6.5 Experiment 3: Diffusion + Stage 2 P-value

This experiment trains the Stage 2 model to learn improved prolongation values for diffusion problems, using the optimal θ from Stage 1 as input.

Unfortunately, this experiment reveals fundamental training difficulties with the Stage 2 architecture. Training loss hovers around 69.0 throughout all 50 epochs with minimal variation, indicating the optimizer fails to find improving directions. More critically, validation loss remains completely constant at exactly 69.69 for all 50 epochs, demonstrating no learning progress whatsoever.

Test metrics confirm complete failure: MSE of 220.76, MAE of 5.69, and catastrophic R^2 of -2155.09. The negative R^2 indicates predictions are over 2000 times worse than simply predicting the mean. Mean predictions (2.81) differ drastically from mean targets (0.16), and the negative correlation (-0.236) shows predictions are weakly anti-correlated with true values.

Analysis of the two-grid convergence metric (mean 69.70, std 23.96, range 38.4-100.2) shows substantial variation across samples, but the model fails to exploit

this information. The constant validation loss throughout training strongly suggests gradient flow issues in the prolongation matrix reconstruction process. While the forward pass constructs matrices correctly, gradients either vanish or produce numerically unstable updates that prevent learning.

This failure represents a significant limitation of the current Stage 2 implementation and requires fundamental architectural revision before P-value learning can be considered viable for practical applications.

6.6 Experiment 4: Graph Laplacian + Stage 1 GNN

This experiment applies the GNN architecture to random graph Laplacian problems. The irregular graph structure makes this a natural fit for GNN processing.

The GNN shows stable training on graph Laplacian problems, contrasting sharply with its failure on diffusion. Training converges after only 16 epochs with early stopping, as both training and validation losses decrease smoothly. Training loss drops from 1.63×10^{-3} to 7.69×10^{-5}, while validation loss decreases from 5.83×10^{-4} to 4.13×10^{-4} at the best epoch (16).

Test performance is modest but reasonable: MSE of 3.51×10^{-4}, MAE of 0.0147, relative error of 4.45%, and R^2 of 0.238. The correlation coefficient (0.488) indicates moderate positive relationship between predictions and targets. Mean predictions (0.326) closely match mean targets (0.326), showing good calibration without systematic bias.

The GNN's success on graph Laplacian problems versus its failure on diffusion suggests that the irregular connectivity patterns from Delaunay triangulation provide informative graph structure that message-passing can exploit. Random graphs exhibit varying node degrees and local topology that the GNN architecture can learn to interpret, whereas the regular FEM mesh structure from diffusion problems may be too uniform to benefit from graph convolutions.

6.7 Experiment 5: Graph Laplacian + Stage 1 CNN

This experiment applies the CNN architecture to graph Laplacian problems. The pooling operation must handle highly irregular sparsity patterns from random Delaunay triangulations.

The CNN shows performance very similar to the GNN on graph Laplacian problems. The training loss decreases smoothly, and the test loss also decreases in trend with early stopping at the 19th epoch. The test metrics reveal: MSE of 3.59×10^{-4}, MAE of 0.0149, relative error of 4.51%, and R^2 of 0.222. These results are nearly identical to the GNN (MSE 3.51×10^{-4}, R^2 0.238), with the GNN showing a slight edge.

The correlation coefficient (0.483) is marginally lower than GNN's 0.488, and both models achieve similar calibration with mean predictions (0.325) close to mean targets (0.326). The comparable performance suggests that both the pooled image representation and graph-based processing capture relevant structural information for graph Laplacian problems, though neither architecture achieves the exceptional performance seen with CNN on diffusion problems.

The similarity in results indicates that for graph Laplacian problems, architectural choice matters less than for structured FEM problems. Both representations -

pooled matrix images and sparse graph structure - provide sufficient information for modest generalization, but neither exploits problem-specific structure as effectively as CNN does for regular diffusion discretizations.

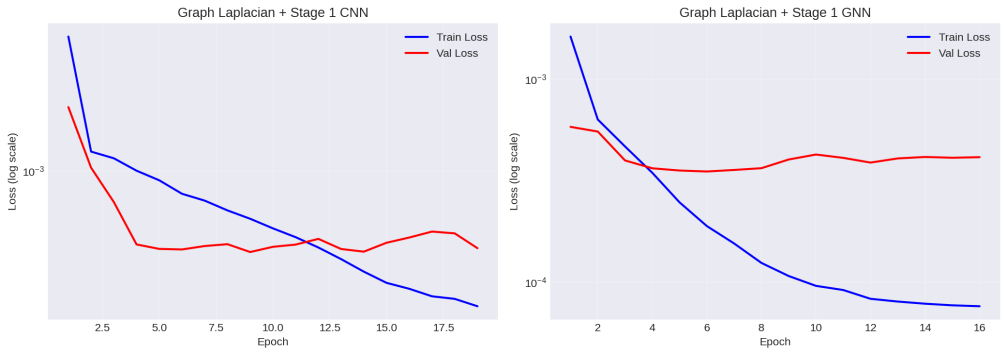

Fig. 3. Training curves for Graph Laplacian Stage 1 experiments. Both CNN and GNN show stable convergence on irregular graph structures, with smooth decreases in training and validation losses. The GNN converges faster (16 epochs) than typical diffusion experiments, demonstrating better fit for graph-structured data.

6.8 Experiment 6: Graph Laplacian + Stage 2 P-value

This experiment trains Stage 2 on graph Laplacian problems to refine prolongation values.

Similar to Diffusion Stage 2, this experiment exhibits complete training failure, though manifested differently. Training loss remains essentially constant around 57.05 for all 50 epochs (range 57.04-57.06), indicating no gradient-based improvement. Validation loss is exactly constant at 22.00 throughout training, identical to the Diffusion Stage 2 behavior.

Test metrics reveal a degenerate solution: MSE of 0.167, MAE of 0.227, and R^2 of -0.447. Most critically, the standard deviation of predictions is exactly 0.0, meaning the model outputs a constant value (-0.043) for all test samples regardless of input. This represents complete failure to learn any input-output relationship.

The two-grid convergence metric (mean 22.00, std 0.39, range 20.9-22.7) shows remarkably low variation compared to diffusion (std 23.96), yet the model still fails to exploit even this limited signal. The identical validation loss pattern (constant 22.00) in both Stage 2 experiments strongly suggests a systematic architectural problem rather than problem-specific issues.

The constant-prediction behavior indicates the network has collapsed to outputting the bias term from the final layer, with all learned weights effectively zeroed out or producing negligible contributions. This could result from vanishing gradients through the prolongation reconstruction, numerical instabilities in the two-grid loss computation, or fundamental incompatibility between the edge-prediction formulation and the matrix-valued target.

6.9 Summary of Results

The experimental results reveal distinct performance characteristics across problem classes and architectural choices. Table 2 summarizes Stage 1 results for theta prediction, while Table 3 presents Stage 2 results for P-value prediction.

Table 2. Stage 1 Experimental Results (Theta Prediction)

Experiment	MSE	MAE	Rel. Error	R^2
Diffusion + CNN	1.02×10^{-7}	0.0002	1.15%	0.996
Diffusion + GNN	4.00×10^{-4}	0.0156	96.58%	-14.645
Graph Lap. + CNN	3.59×10^{-4}	0.0149	4.51%	0.222
Graph Lap. + GNN	3.51×10^{-4}	0.0147	4.45%	0.238

Table 3. Stage 2 Experimental Results (P-Value Prediction)

Experiment	MSE	MAE	TwoGrid Loss	R^2
Diffusion + Stage 2	2.21×10^{2}	5.69	69.70 ± 23.96	-2155.09
Graph Lap. + Stage 2	1.67×10^{-1}	0.23	22.00 ± 0.39	-0.447

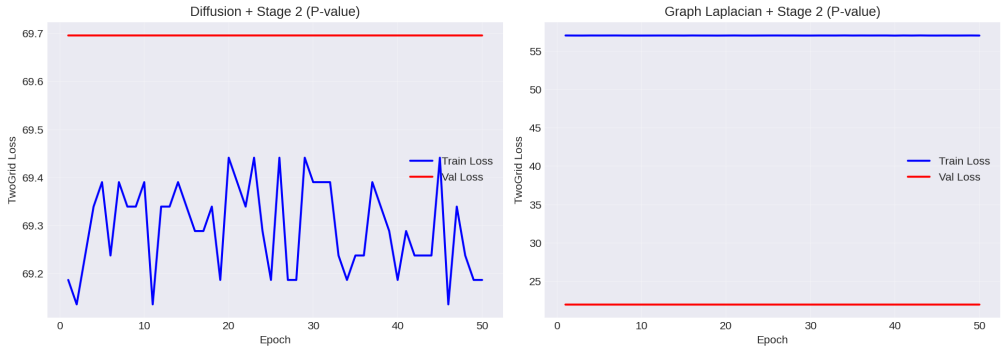

Fig. 4. Training and curves for Stage 2 experiments. Both Diffusion (left) and Graph Laplacian (right) show constant validation loss throughout training (exactly 69.69 and 22.0 for all epochs, respectively), while training loss oscillates around 69.3 and 57.0 without meaningful decrease, respectively. This pattern indicates complete failure of gradient-based learning, likely due to vanishing gradients in the prolongation reconstruction process.

6.9.1 Key Findings

The CNN architecture demonstrates exceptional performance on diffusion problems, achieving an R^2 score of 0.996 and extremely low MSE (1.02×10^{-7}). This strong result reflects the CNN's ability to capture structural patterns in the pooled matrix representation for regular FEM discretizations. In contrast, the GNN shows poor performance on diffusion problems with negative R^2 (-14.645), indicating training instability and failure to converge to meaningful predictions.

For graph Laplacian problems, both architectures show modest but reasonable performance (R^2 around 0.22-0.24) with very similar error metrics. The GNN slightly outperforms CNN on this problem class (MSE 3.51×10^{-4} vs 3.59×10^{-4}), suggesting that graph-structured data may be more amenable to direct graph neural network processing than pooling-based approaches.

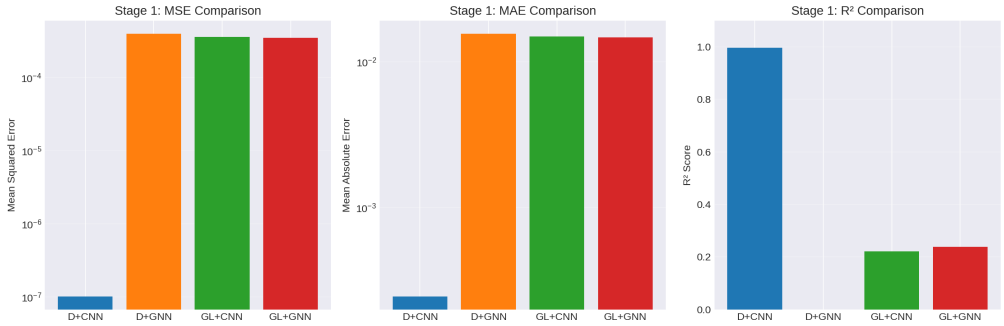

Fig. 5. Stage 1 performance comparison across all four experiments. Left: MSE comparison (log scale) shows Diffusion+CNN achieves orders of magnitude better error than other approaches. Center: MAE comparison confirms CNN's exceptional performance on diffusion. Right: R^2 scores demonstrate CNN's near-perfect fit (0.996) for diffusion, while GNN fails (negative R^2), and both architectures show modest performance on graph Laplacian problems.

Both Stage 2 experiments exhibit severe training difficulties. The negative R^2 scores (-2155.09 for diffusion, -0.447 for graph Laplacian) indicate that the models perform worse than simply predicting the mean value. Analysis of training logs reveals that validation loss remains constant throughout training, suggesting gradient flow issues in the prolongation matrix reconstruction process. The Graph Laplacian Stage 2 model outputs constant predictions (standard deviation of predictions is exactly 0.0), indicating complete failure to learn meaningful patterns.

These Stage 2 failures likely stem from challenges in maintaining gradient flow through the sparse matrix reconstruction and two-grid operator formation. The current implementation, while theoretically differentiable, appears to suffer from vanishing gradients or numerical instabilities that prevent effective learning. This represents a significant limitation requiring further architectural investigation and alternative training strategies.

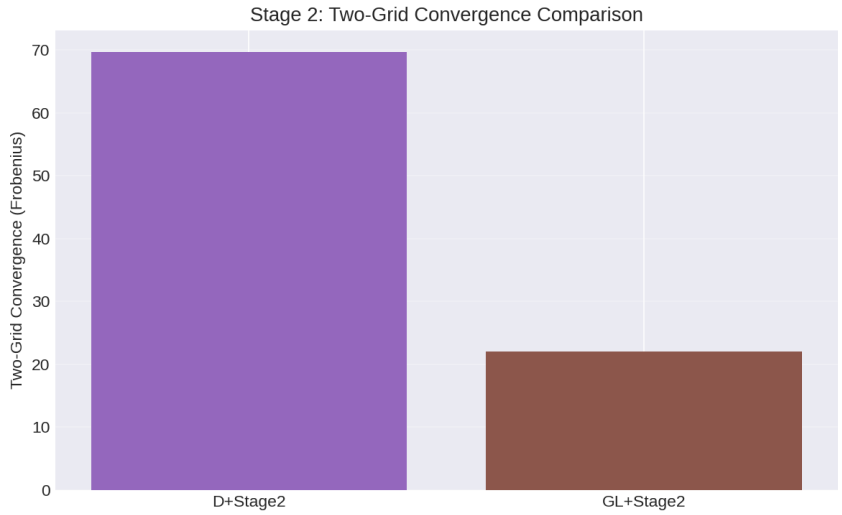

Fig. 6. Two-grid convergence metrics for Stage 2 experiments. Despite substantial variation in true convergence factors (Diffusion: 69.70 ± 23.96, Graph Laplacian: 22.00 ± 0.39), the models fail to learn these patterns due to gradient flow issues. The high values indicate poor convergence, which the learning algorithm cannot optimize.

Stage 1 training converges within 30-50 epochs (15-90 minutes on GPU). The CNN typically requires fewer epochs than GNN due to faster convergence on the structured pooled representation. Stage 2 training runs for 50 epochs (4-5 hours) but shows no meaningful learning progress, making the computational cost unjustified given current results.

CNNs provide superior stability and performance for structured problems arising from regular FEM discretizations. GNNs show potential for irregular graph structures but require careful hyperparameter tuning and exhibit less stable training dynamics. The NPZ format proves to be essential for efficient experimentation, providing a speedup that enables rapid iteration during model development.

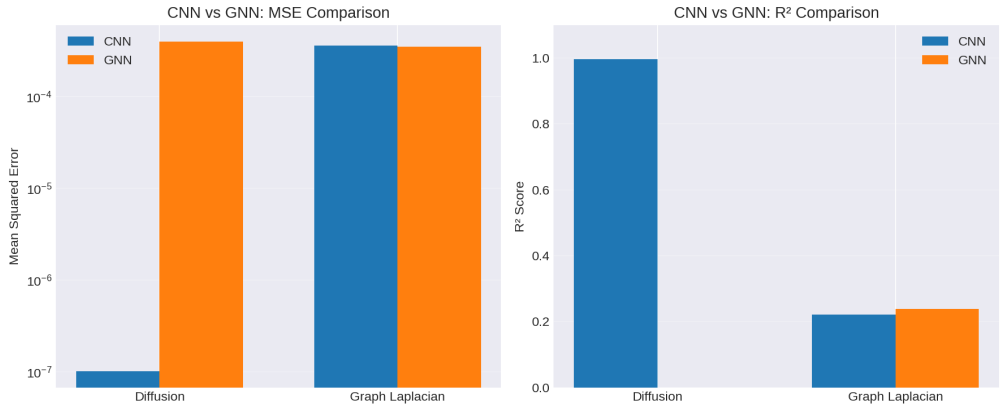

Fig. 7. Direct CNN vs GNN architecture comparison across both problem classes. Left: MSE comparison (log scale) shows CNN dominates on Diffusion problems (4000× better) while both architectures perform similarly on Graph Laplacian. Right: R^2 comparison reveals CNN achieves near-perfect fit (0.996) on diffusion while GNN fails catastrophically (negative R^2). For Graph Laplacian, both show modest but positive R^2 (0.22-0.24), with GNN holding a slight edge.

7 Conclusion

This work presents a two-stage deep learning framework for optimizing Algebraic Multigrid performance through learned parameter selection. Stage 1 learns the optimal strong threshold parameter θ that governs coarsening, while Stage 2 aims to learn improved prolongation matrix values. The experimental evaluation reveals both successes and significant challenges in this approach.

For theta prediction, the CNN architecture demonstrates exceptional performance on heterogeneous diffusion equations. This validates that learned threshold selection can substantially outperform fixed heuristic values for structured FEM discretizations. Both CNN and GNN show modest performance on graph Laplacian problems from random Delaunay triangulations, suggesting that these irregular structures present greater generalization challenges.

Both Stage 2 experiments exhibit complete training failure. Validation loss remains constant throughout all epochs, and predictions collapse to constant values. The identical failure pattern across both problem classes indicates systematic architectural problems rather than problem-specific issues. Investigation suggests vanishing gradients in the prolongation reconstruction process or numerical instabilities in two-grid loss computation. The current Stage 2 implementation is not viable for practical applications and requires fundamental architectural revision.

Beyond Stage 2 failures, current limitations include restriction to 2D problems and moderate matrix sizes (up to approximately 10,000 DOFs). The framework has been validated on diffusion and graph Laplacian problems only. Training instability issues (particularly GNN on diffusion) require careful hyperparameter tuning and show sensitivity to problem characteristics.

Immediate priorities include diagnosing and resolving Stage 2 gradient flow issues, potentially through alternative loss formulations, architectural modifications to stabilize gradient propagation, or reformulation of the P-value prediction task. For

Stage 1, investigating hybrid CNN-GNN architectures could combine pooling efficiency with graph flexibility. Longer-term extensions include scaling to 3D problems through 3D convolutions, developing online adaptation for transient problems, and exploring reinforcement learning for end-to-end solver optimization.

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] I. Luz, M. Galun, H. Maron, R. Basri, and I. Yavneh. Learning algebraic multigrid using graph neural networks. *arXiv preprint arXiv:2003.05744*, 2020.

[4] 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.

A Previous Experiments

These experiments are done in our previous work, referring to the Github repository (https://github.com/tianxingjian2791/PACSProject/tree/v1.0).

A.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 `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 8 shows the train/test MSE curves: the CNN converges faster during training and attains a lower test MSE than the GAT in this experiment.

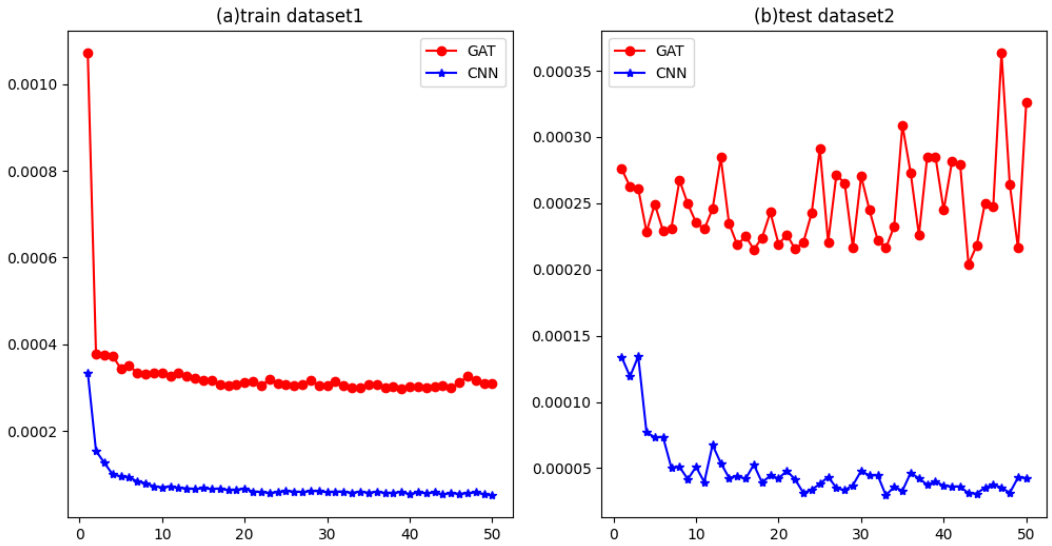

Fig. 8. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 1).

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. 9 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.

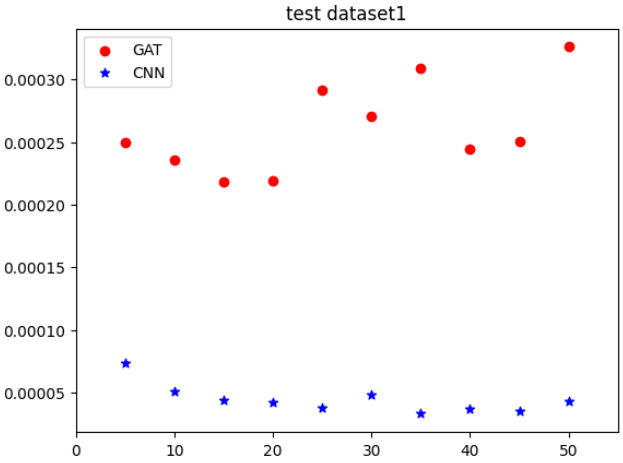

Fig. 9. Test MSE of model checkpoints saved every five epochs (Dataset 1).

A.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 10 shows the training behaviour: the CNN converges more rapidly and attains better test performance than the GAT in this configuration.

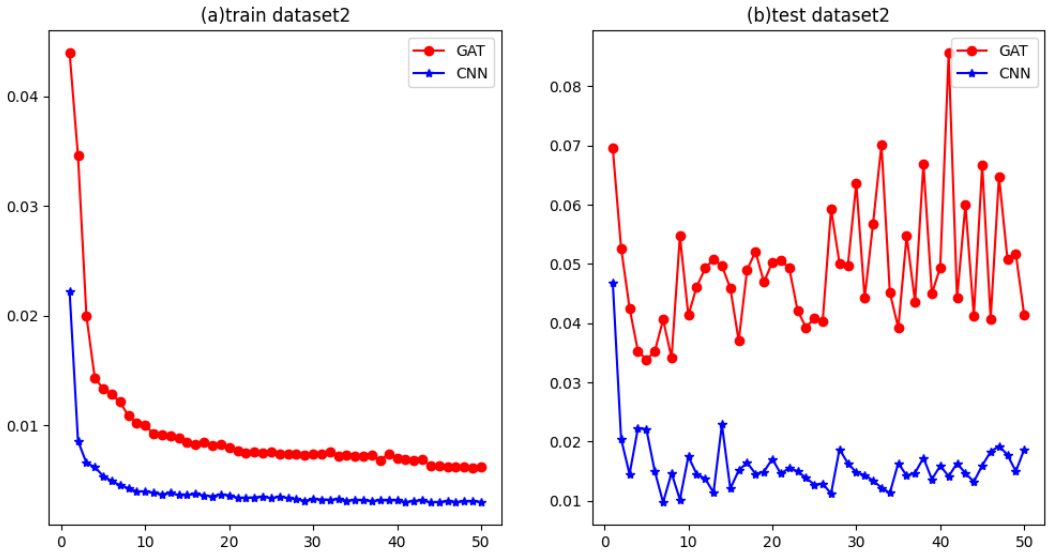

Fig. 10. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 2).

Evaluating the checkpoints saved every five epochs (see Fig. 11), the best GAT checkpoint occurred at epoch 5, while the CNN's best performance was achieved at the final epoch.

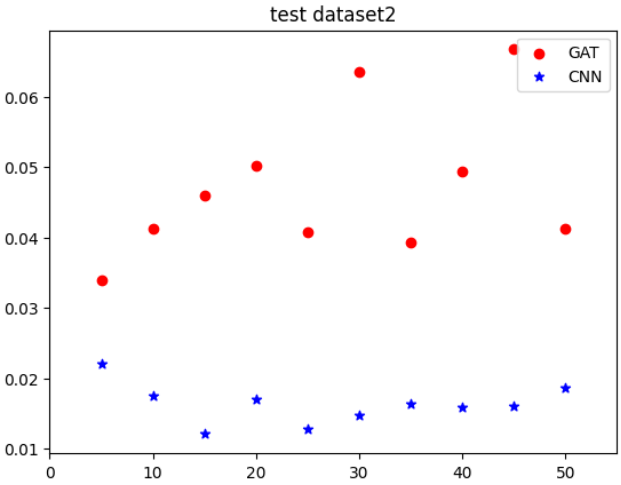

Fig. 11. Test MSE of model checkpoints saved every five epochs (Dataset 2).

A.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. 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 12 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.

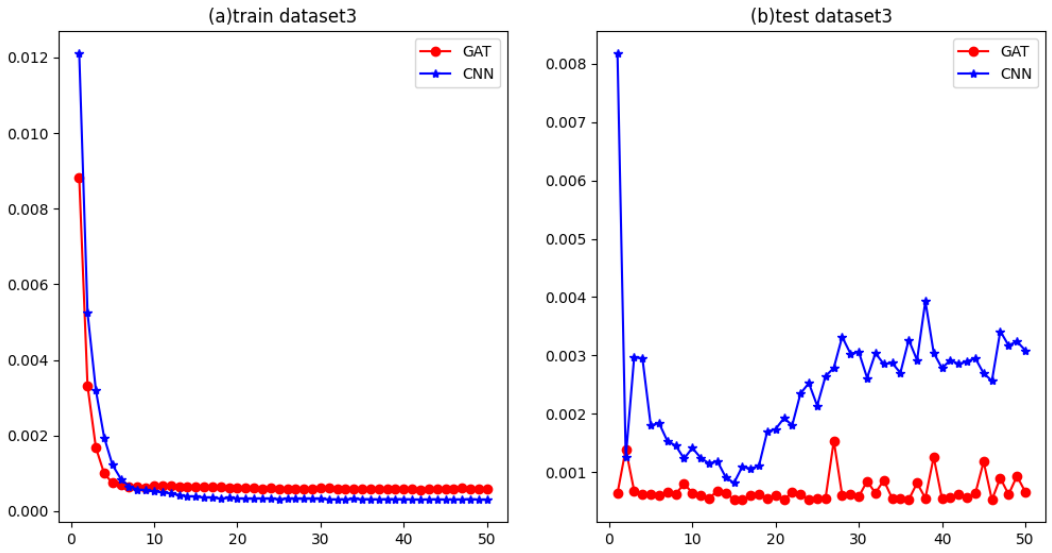

Fig. 12. Train and test MSE of the GAT-based and CNN-based networks during training (Dataset 3).

The checkpoint analysis in Fig. 13 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.

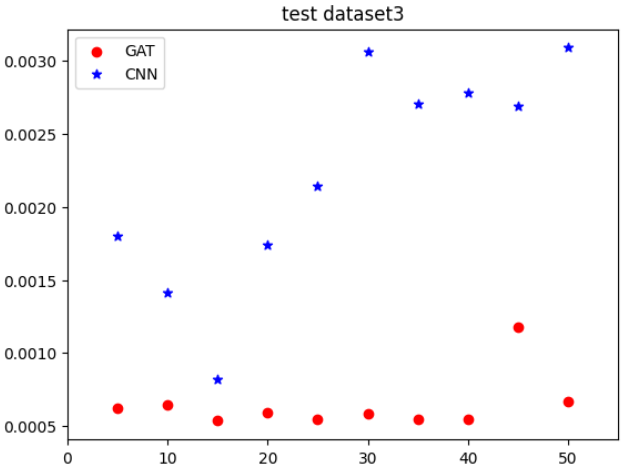

Fig. 13. Test MSE of model checkpoints saved every five epochs (Dataset 3).