Predicting Block Structure in sparse Matrices

Anna-Valentina Hirsch
Toni Johann Schulze Dieckhoff

July 2024

1 Background

1.1 Computational Fluid Dynamics

Many real-world phenomena are continuous in nature and characterised by a multitude of variables that interact in complex ways. Such phenomena are typically modelled using sets of partial differential equations (PDEs) which often do not have closed-form solutions [1]. A prime example of this complexity is found in Computational Fluid Dynamics (CFD), where the behaviour of fluids is simulated accounting for factors such as viscosity, turbulence, and pressure gradients [2]. In CFD, the continuous flow of fluids is modelled using the Navier-Stokes equations, a set of PDEs describing the motion of fluid substances [3]. These equations, while elegant in their continuous form, are extremely difficult to solve analytically for all but the simplest cases [4] [1]. To address this challenge, numerical methods such as Finite Volume (FV), Finite Element (FE), or Finite Difference (FD) are employed to discretise the continuous domain into a finite number of points or cells, allowing for the computation of approximate solutions [6–8].

The discretisation process transforms the continuous fluid flow problem into a large system of algebraic equations, which can be represented as a matrix equation [9]:

$$Ax = b \tag{1}$$

Where:

- A is a large, sparse n x n coefficient matrix
- x is a vector of unknown variables (e.g., velocity, pressure)
- b is a vector of known quantities (often related to boundary conditions and source terms)

Each row in the matrix A typically corresponds to an equation for a specific cell in the grid, while the columns represent the influence of neighbouring cells. For a realistic CFD simulation - i.e. modelling airflow around an entire aircraft - millions of cells are needed to accurately capture the fluid behaviour. This results in a matrix with millions of rows and columns. However, because each cell primarily interacts with its immediate neighbours, most entries in this matrix are zero, leading to what is called a "sparse" matrix [7]. Sparse matrices can be beneficial with regard to computational efficiency, as specialised storage formats can be employed to reduce memory requirements, allowing for the handling of much larger problems than would be feasible with dense matrix representations [1].

[1]The analytical solution of the Navier-Stokes equations in three dimensions is one of the Millennium Prize Problems in mathematics [5]

Due to the multiple variables associated with each grid cell, A often exhibits a block-diagonal structure which can be represented as:

$$A = \begin{bmatrix} B_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & B_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & B_{nn} \end{bmatrix} \tag{2}$$

where each B_{ii} is a square block matrix corresponding to the coupling between variables within a single cell or a small group of cells, while the remaining A_{ij} are sparse.

2 Motivation

Although much research goes into optimising existing algorithms to solve systems of linear equations efficiently, the complexity of direct solving methods, i.e. Gaussian elimination or LU factorisation, can be as high as $O(N^3)$ [10–13]. For large systems, this can lead to significant computational costs and limitations in terms of memory usage. In FE and FV simulations, a common approach is to approximate the solution of the matrices by iteratively refining an initial guess until a predefined convergence criterion is satisfied [14,15]. For iterative solving algorithms such as GMRES, which is introduced in Section 3, each iteration has a complexity of $O(N^2)$. Hence, depending on the size of the matrix and the number of iterations required, these methods can offer substantial computational savings over direct methods, particularly if the matrices are sparse [14]. If the matrices are ill-conditioned, however, the number of iterations required to find a solution within an acceptable error margin can quickly become computationally prohibitive. This is often the case in CFD, where scale differences and coupled phenomena (e.g., velocity-pressure coupling, fluid-thermal interactions) result in an unfavourable distribution of eigenvalues and high condition numbers [10, 14, 16].

2.1 Convergence Acceleration through Preconditioning

To accelerate the convergence of an iterative solver, a preconditioner matrix P can be applied to both sides of the equation, where $P \approx A^{-1}$. Thus, the original system $Ax + b$ is transformed into a system $PAx = Pb$, whereby PA and Pb are ideally cheap to compute and have a more favourable eigenvalue distribution than the original matrix A [14]. We then solve the resulting system $A' \cdot x = b'$.

If P is cleverly chosen, the number iterations for solving this system will be significantly smaller. The goal is to cluster the eigenvalues of PA around 1 and away from zero, thereby reducing the condition number and improving the convergence rate of the iterative solver [14].

2.2 Block-Jacobi Preconditioner

The Block-Jacobi preconditioner is a natural extension of the classical Jacobi method, tailored to handle the block structure often encountered in CFD problems [14]. It is particularly well-suited for systems where variables are tightly coupled within local regions but less so across the entire domain. Although more sophisticated preconditioners like Incomplete LU factorisation (ILU) or Algebraic Multigrid (AMG) outperform Jacobian methods with regard to their effectiveness [17], the inherent parallelism of the block-Jacobi preconditioner allows for efficient distribution of computational workload across multiple processors or nodes in a high-performance computing cluster [14, 18, 19]. This positions the latter at an advantage for large-scale CFD problems.

The Block-Jacobi preconditioner P can be constructed as:

$$P = \begin{bmatrix} A_{11}^{-1} & 0 & \cdots & 0 \\ 0 & A_{22}^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nn}^{-1} \end{bmatrix} \tag{3}$$

where each A_{ii}^{-1} represents the inverse of a diagonal block of the original matrix A.

In a parallel implementation, each processor can be assigned one or more blocks, computing the local inverse and applying it to the corresponding part of the vector without needing to communicate with other processors. This locality of computation significantly reduces inter-processor communication overhead, which is often a bottleneck in parallel algorithms. Moreover, the Block-Jacobi preconditioner aligns well with domain decomposition strategies commonly used in CFD, where the computational domain is divided into subdomains. Each subdomain can naturally correspond to a block in the preconditioner, preserving the physical and numerical relationships within the local region [14].

2.3 Problem Statement

Traditionally, matrix block structures are determined using domain-specific knowledge or heuristics. For matrices from unknown sources, however, their inherent block structures may not be immediately apparent due to ordering issues or noise elements, making the identification of related blocks challenging. Building on the work of [18], our goal is therefore to forecast diagonal block locations in a collection of large sparse matrices in order to efficiently build Block-Jacobi preconditioners and ultimately improve the convergence of the Generalised Minimal Residual (GMRES) solver. The GMRES solver is frequently employed in conjunction with the Block-Jacobi preconditioner [14, 20], and aims to solve sparse systems of linear equations of the form $Ax = b$, where A is a non-singular $n \times n$ matrix, and x and b are vectors of length n. Specifically, GMRES operates by iteratively minimising the residual norm over expanding Krylov subspaces until it falls below the predefined convergence threshold.

Similar to [18], we use predictive techniques to determine the location of diagonal blocks inside our sparse matrices. This enables us to rapidly identify and implement the Block-Jacobi preconditioner. Thus, our initial step involves replicating the findings outlined in their publication [18] using a Convolutional Neural Network (CNN). Subsequently, we extend their research by conducting a comparative analysis of various methods to determine the blocks within the matrices, encompassing graph representations. The objective is for every matrix to accurately predict the size and position of each block. Therefore, we represent the matrices' diagonal structures as binary arrays indicating whether or not a new block starts at each position. Given that a matrix can consist of several blocks, we are faced with a multi-label binary classification problem.

This paper is organised as follows: To begin, we present a comprehensive summary of existing research on the topic. Next, we elucidate our experimental setup as well as some technical details relevant to our work. Subsequently, we proceed to explore each stage of the process methodologically, beginning with data generation and concluding with the modelling process. Finally, we analyse our findings and outline potential avenues for future research.

3 Related Work

The application of machine learning (ML) techniques, particularly CNNs, to enhance the solving of large sparse linear systems has gained significant traction in recent years. This approach aims to leverage the pattern recognition capabilities of Neural Networks (NN) to either directly solve linear systems or, more commonly, to generate effective preconditioners that accelerate the convergence of traditional iterative methods.

Demonstrating the potential of CNNs in this domain, [21] compared scalar-based and image-based methods to identify the optimal preconditioners for the Conjugate Gradient (CG) solver [2]. By representing matrix sparsity patterns in RGB and feeding them into a CNN, the authors were able to extract topological characteristics and identify the most effective preconditioners from a pool of ten options. This approach outperformed scalar feature methods in both accuracy (32% improvement) and runtime (25% reduction).

Focusing on the Helmholtz equation, [22] proposed a U-Net CNN architecture to generate preconditioners. Their method, which integrates CNN-based components with multi-grid techniques, showed promising results in terms of solution quality and computational efficiency, especially when leveraged on GPU hardware.
In a similar vein, [23] explored various NN architectures for preconditioning the Helmholtz equation, including Physics-Informed NNs, DeepONet, Fourier Neural Operator (FNO), and U-Net. Focusing on the preconditioners' robustness across different domain sizes and equation parameters, the potential of U-Net and FNO architectures as generalisable preconditioners were highlighted. Nevertheless, the author also reported notable difficulties including numerical instability and exploding gradients, which arise when differentiating through multiple solver iterations. By implementing a replay buffer, a method known from reinforcement learning, the training process was stabilised and generalisation was enhanced.

While the aforementioned works employ ML techniques to enhance the matrices' conditions, another field of research focuses on developing neural sparse linear solvers (NSLSs) to solve equations directly. These approaches offer different trade-offs between accuracy, speed, and applicability. [24] introduced a deep learning framework that represents a sparse symmetric linear system as an undirected weighted graph, exploiting the graph's permutation-equivariant and scale-invariant properties. Their Graph Convolutional Network (GCN) approach efficiently predicts approximate solutions, and therefore excels in tasks requiring real-time solutions where speed is prioritised over precision. However, NSLSs are generally less accurate than classical algorithms.

Striking a balance between the advantages of ML and the need for numerical accuracy, [25] formulate the task of finding an effective preconditioner as an unsupervised learning problem: The authors employ a CNN to learn a function f that maps an input matrix A to a preconditioner M^{-1}. The learning process involves minimising the condition number of the preconditioned system AM^{-1} across a training set of sparse Symmetric Positive definite (SPD) matrices. The model receives the lower triangular component and diagonal of system matrix A as input and transforms them to create an SPD preconditioner. This approach of using a CNN-generated preconditioner alongside the traditional Conjugate Gradient algorithm works effectively for a variety of applications, including solving the Pressure Poisson Equation (PPE) in CFD, while guaranteeing a certain level of numerical accuracy [25]. Nevertheless, the computational cost of calculating condition numbers during training may limit its performance advantage.

The existing research demonstrates the growing potential of ML in enhancing the solution of large sparse linear systems. Particularly, the works of [21–23, 25] have highlighted the effectiveness of CNNs in identifying optimal preconditioners from sparse matrix representations. Nevertheless, our work distinguishes itself in several key aspects: Instead of selecting the optimal preconditioner from a set of options, we specifically target the Block-Jacobi preconditioner. By exploiting its inherent parallelisability for efficient GPU acceleration (see [18, 26]), we aim to bridge the gap between numerical linear algebra and high-performance computing. Building upon [18]'s work, our objective is therefore to identify the locations of blocks within sparse matrices as the basis for constructing the preconditioner. By employing ML techniques to detect these blocks, we can

[2]Similar to GMRES, CG belongs to the Krylov subspace methods, but it is specifically designed for symmetric positive-definite (SPD) matrices.

align our preconditioning strategy more closely with the matrix's intrinsic properties, potentially handling a wider variety of matrix types more effectively.

4 Experimental Setup

In order to assess the efficacy and scalability of the generated Block-Jacobi preconditioners, we conducted a series of experiments in a controlled environment. Using the matrices from our test dataset, we evaluate the convergence rates of the GMRES solver under different scenarios: without using a preconditioner, using a preconditioner generated from the actual labels, using a preconditioner generated from the predicted labels, and using a preconditioner generated from supervariable blocking (SVB) algorithm, a non-machine learning technique for detecting block patterns [26].

4.1 Hardware and Software Configuration

Python 3.9 is used for all experiments. We employ an A100 GPU equipped with 25GB of RAM to train our model. Additionally, we utilise TensorFlow 2.15.0 in conjunction with Keras 2. A requirements file containing all the necessary libraries and versions is provided for easy replication of our experiments.

4.2 Data Generation (MatrixKit Package)

Due to the insufficient availability of labelled real-world matrices, we developed a Python package that creates custom matrices specifically designed for our machine learning experiments. The library is called matrixkit and can be installed by running `pip install matrixkit`. It contains various functionalities to produce realistic synthetic data by first constructing a set of zero matrices with specified dimensions and then gradually adding complexity.

At first, random background noise is added to the matrices based on user-defined parameters for noise density and value ranges. Following that, two types of block structures are added to the matrices; noise blocks and data blocks. The sizes of these blocks are determined using a truncated normal distribution, allowing for controlled variability within specified ranges. As desired, blocks can be inserted with random gaps based on a given probability. Within each block, values are added according to a specified density. The library's design allows for fine-tuning of various parameters, including matrix dimensions, noise levels, block sizes, and densities. By adjusting these parameters, researchers can create matrices that closely resemble those encountered in their specific domains of study. Throughout the generation process, the library maintains detailed metadata about the matrices, including the user-specified parameters as well as the actual block start positions. The latter corresponds to the labels of the matrices that are used for model training. For further details, please consult the package documentation [27].

5 Modelling

5.1 Convolutional Neural Network

5.1.1 Model Architecture

We trained a CNN with similar architecture to [18] to predict the block starts of our synthetic matrices. The model consists of three main components: a bottleneck unit, a corner detection module, and a fully-connected predictor. The bottleneck unit comprises two convolutional layers with 32 and 128 filters respectively, using SELU activation, batch normalisation, and L2 regularisation. An additive skip connection is employed to facilitate gradient flow. The corner detection module involves zero padding followed by two convolutional layers with tanh activation. The fully-connected predictor flattens the output and applies two dense layers with sigmoid activation, incorporating dropout for regularisation.

Weight Initialisation and Regularisation In addition to the architecture proposed by [18], we also implemented different weight initialisation techniques to maintain constant variance of activations: Xavier weight initialisation [28] was employed across the layers with tanh or sigmoid activations, and LeCun initialisation was used for layers with SELU activation. Weight initialisation ensures that input variance $\approx$ output variance, contributing to the model's stability and quick convergence. Figure 1 displays the (simplified) architecture of the model.

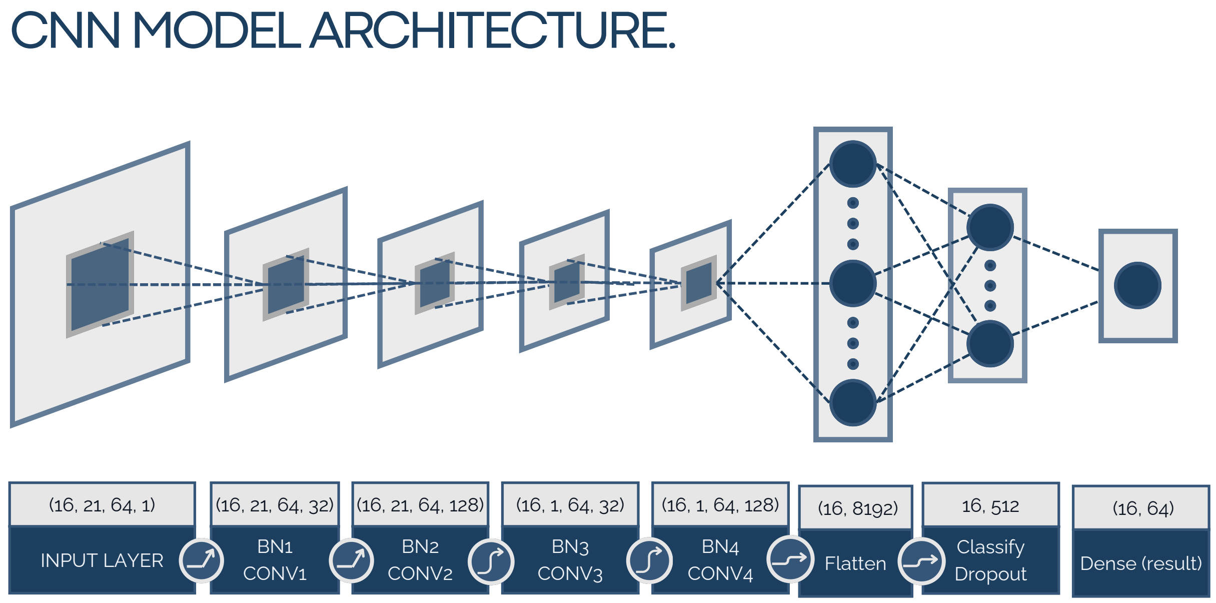

Figure 1: CNN Architecture

5.1.2 Model Training

The training process comprised 200 epochs with a batch size of 16. Early stopping is implemented with patience set to ten epochs. To ensure reproducibility of results and keep track of the model's progress, checkpoints are saved every 10 epochs, and Tensor-Board logging is used for real-time performance monitoring. Similar to [18], we use the Nadam optimiser, which combines Adam (adaptive moment estimation, see [29]) with Nesterov momentum. This optimiser calculates the gradient at a look-ahead position, potentially leading to improved convergence rates. Additionally, a learning rate scheduler was implemented to dynamically adjust the learning rate during training. During training, the model tries to minimise the loss function. Since we are dealing with imbalanced classes, where roughly 10 percent of the matrices represent block starts, a custom loss function named "weighted binary cross entropy" was implemented. The function assigns different weights to classes based on their frequency in the dataset. Therefore, it calculates the standard binary cross-entropy for each element, and subsequently applies class-specific weights to the loss values. The minority class (block start) is given higher weights, forcing the model to pay more attention to these occurrences. The function guarantees the correct arrangement of inputs, trims prediction values to prevent numerical instability, and applies the weighting scheme individually to each element. By calculating the average weighted loss for each batch, it ensures that the training signal is balanced, which in turn helps the model achieve good performance across all classes, regardless of their frequency in the training data. The training function returns the trained model, as well as the loss histories for training and validation.

5.2 Graph Convolutional Network

5.2.1 Model Architecture

Our Graph Convolutional Network (GCN) model is designed to capture the structural relationships within the input data. The architecture consists of several key components: graph convolution layers, graph attention layers, and a combination of residual connections and dense layers for feature extraction and prediction. The model takes two inputs: a feature matrix $X \in \mathbb{R}^{n \times f}$, where n is the number of nodes and f is the number of features per node, and an adjacency matrix $A \in \mathbb{R}^{n \times n}$ representing the graph structure. To maintain consistency with the CNN, we only consider the matrix bands and not the full matrices. Hence, in our specific case:

- Number of nodes (n) = 64 (corresponding to the columns of the original matrix)
- Number of features per node (f) = 21 (corresponding to the rows in the band).

This results in a feature matrix $X \in \mathbb{R}^{64 \times 21}$ and an adjacency matrix $A \in \mathbb{R}^{64 \times 64}$. The model can be configured to use either graph attention or standard graph convolution, which are explained below.

Graph Convolution Layer: The graph convolution layer is implemented as a custom layer that performs the following operation:

$$H = \sigma(AXW + \beta) \tag{4}$$

where σ is an activation function (e.g., ReLU), $W \in \mathbb{R}^{f \times d}$ is the learnable weight matrix (with d being the number of output features), and $\beta \in \mathbb{R}^{d}$ is the bias term [30]. The layer applies the weight matrix W to transform the node features X, then aggregates the features using the adjacency matrix A, capturing information from neighbouring nodes. For a more detailed description of convolutions on graphs, please refer to [30]. L2 regularisation is applied to the weights to prevent overfitting, ensuring that the model generalises well to unseen data.

Graph Attention Layer: To enhance the model's ability to focus on important parts of the graph, we implemented a graph attention mechanism (GraphAttention). This layer uses multi-head attention, allowing the model to jointly attend to information from different representation subspaces. The attention mechanism is defined as:

$$e_{ij} = \text{LeakyReLU}\ (a^T [W h_i || W h_j]) \tag{5}$$

where $a \in \mathbb{R}^{2d \times 1}$ is the attention kernel, $W \in \mathbb{R}^{f \times d}$ is the weight matrix, and h_i, h_j are the feature vectors of nodes i and j [31]. The concatenation of Wh_i and Wh_j (denoted by $||$) represents the combined feature representation of node pair (i, j). The resulting score e_{ij} indicates the importance of node j's features to node i. The attention coefficients are then normalised using the softmax function:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{6}$$

where $\mathcal{N}(i)$ denotes the set of neighbouring nodes of node i [31]. The normalised attention coefficients α_{ij} are used to compute a weighted sum of the node features, effectively allowing the model to focus on the most relevant parts of the graph.

Residual Connections: We also added residual connections, introduced by He et al. [32], which are designed to facilitate the training of deep NNs by alleviating the vanishing gradient problem and improving gradient flow. These connections, also known as "shortcut connections" or "skip connections", enable the model to learn a residual function $F(x)$—which is essentially the difference between the desired output $H(x)$ and the input x—rather than learning the entire mapping function $H(x)$ from scratch. In practice, this means that the output from one layer is directly added to the output of a later layer, bypassing one or more intermediate layers. This bypassed output (i.e., the original input that skips the intermediate layers) is then added to the output of the subsequent layers instead of replacing them. This mechanism helps to preserve the original information while allowing the model to learn more complex features. In our GCN model, these residual connections specifically help maintain the fidelity of the original input features while allowing the network to learn increasingly abstract representations, which is particularly important for capturing both local and global structural information in the sparse matrices.

Initialisation and Regularisation: We use Xavier uniform initialisation for the weight matrices and zero initialisation for biases. L2 regularisation is applied to all weight matrices to prevent overfitting. Dropout is incorporated in the graph attention layers for further regularisation.

Output Layer: The final output layer is a dense layer with a sigmoid activation function, producing a binary classification output for each node. The output labels indicate whether each column in the matrix corresponds to the start of a block, with the final output being reshaped to match the desired output shape of $(\text{batch\_size}, 64)$.

6 Model Performance Evaluation

For evaluation, we calculate the weighted binary cross-entropy loss of the test set and several key metrics. These include element-wise accuracy, which measures the proportion of correctly classified individual elements across all samples and classes. A classification report provides precision, recall, and F1-score for each class. The confusion matrix offers a detailed breakdown of true positives, false negatives, true negatives, and false positives, enabling a thorough analysis of the model's classification performance.

The models' performance is evaluated using confusion matrix metrics, yielding 1408 true positives, 563 false negatives, 16776 true negatives, and 453 false positives for our best model, the CNN. This translates into an overall accuracy score of .947 and an F1-Score of .73 for the block starts. These metrics can be calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{1408 + 16776}{1408 + 16776 + 453 + 563} \approx .947 \tag{7}$$

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{1408}{1408 + 453} \approx 0.757 \tag{8}$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{1408}{1408 + 563} \approx .714 \tag{9}$$

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \approx .73 \tag{10}$$

While the accuracy score is high (.947), it can be misleading in the context of imbalanced datasets, which is the case in our block structure detection task. With less than ten percent of the values representing block starts, a model could achieve high accuracy simply by predicting the majority class (no block start) most of the time, while performing poorly on the minority class of interest (block start). The F1-score provides a more balanced measure of the model's performance, especially for the minority class. It is the harmonic mean of precision and recall, giving equal weight to both metrics. Precision measures the accuracy of positive predictions, while recall measures the model's ability to find all positive instances. By combining these, the F1-score provides a single score that balances both the precision and recall of the model. In our case, an F1-score of .73 for block starts indicates that the model has found a good balance between precisely identifying block starts (minimising false positives) and capturing a high proportion of the actual block starts (minimising false negatives). This is particularly important in our application, where both missing actual block starts and incorrectly identifying non-block starts as blocks could lead to sub-optimal preconditioner generation. Therefore, while we report both accuracy and F1-score, we prioritise the F1-score as a more reliable indicator of our model's performance in this imbalanced classification task.

The CNN's and GCN's block detection capabilities were also compared to traditional methods, specifically the SVB technique proposed by [26]. Therefore, the algorithm described in their paper was implemented in python to assess the relative effectiveness of the AI-based approach compared established methods. The CNN clearly outperformed the GCN and the SVB method. The performance results of all block-detection methods are summarised in 1.

Model	Accuracy	TP	FP	TN	FN	Precision	Recall	F1-score
CNN	.9470	1408	453	16776	563	.76	.71	.73
GCN	.8837	356	561	10955	928	.39	.28	.32
SVB	.8475	66	734	10782	1218	.08	.05	.06

Table 1: Performance comparison of block start detection methods.

7 GMRES Convergence Comparison

Table 2 shows the results from the different GMRES experiment runs. On average, the systems converged after 2.567 iterations when used without a preconditioner, and after 263 iterations when used with Block-Jacobi preconditioners created from the true labels. This is a speed-up of $\approx$ 100x. As anticipated, the convergence improvement is smaller when using a preconditioner constructed from the predicted block starts, which we explain by the deviation of the predictions from the true block starts. Correspondingly, the performance is further affected when creating the preconditioner using the GCN or the SVB technique, although there is still an improvement regarding the average required iteration counts compared to not using a preconditioner at all. It should be noted that there is a significant difference between the mean and median iterations, which is explained by the presence of ill-conditioned matrices in the dataset (outliers) that converged very slowly.

Preconditioner Type	Convergence		Iterations			
	#	%	Mean	Median	Max	Min
No Preconditioner	181/200	90.5	2566.98	1700.0	10381	168
Preconditioner from True Block Starts	190/200	95.0	262.62	111.0	3632	20
Preconditioner from CNN Predictions	184/200	92.0	258.21	107.0	5116	19
Preconditioner from Supervariable Blocking	180/200	90.0	368.43	119.5	5740	19
Preconditioner from GCN Predictions	176/200	88.0	271.42	109.5	3756	16

Table 2: Performance comparison of different preconditioners on original (unprocessed) matrices.

8 Conclusion

In our project, we were able to largely replicate the results reported in [18], showcasing the efficacy of using machine learning to accelerate the convergence of iterative solving algorithms. For this purpose, we also developed a small Python package for easy synthetic matrix generation, which is publicly available and can be installed via pip.

When comparing several methods for predicting matrix block structures, including a CNN, a GCN, and supervariable blocking, we found that neural network approaches yield higher accuracy and precision compared to supervariable blocking. However, we were not able to beat the results from the initial CNN yet.

Investigating the potential for forecasting block structures and expanding it to forecast the complete preconditioner could yield fascinating results in future research focused on GCNs. Furthermore, more research is required to improve the tuning of hyperparameters in order to fully utilise the potential of graph networks when applied to real-world datasets.

9 References

[1] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[2] Robert A. Brown. Chapter 1 fundamentals of fluid dynamics, 1991.

[3] Justin W. Garvin. *A Student's Guide to the Navier-Stokes Equations (Student's Guides)*. Cambridge University Press, Cambridge, February 2023. Publication Date: 2023-02-09.

[4] R.K. Michael Thambynayagam. A class of exact solutions of the navier–stokes equations in three and four dimensions. *European Journal of Mechanics - B/Fluids*, 100:12–20, 2023.

[5] Clay Mathematics Institute. The navier-stokes equation, n.d. Accessed: 2024-07-15.

[6] Timothy Barth and Mario Ohlberger. Finite volume methods: Foundation and analysis. Accessed: 2024-07-15.

[7] Seongjai Kim. Numerical methods for partial differential equations, December 11 2023. Department of Mathematics and Statistics, Mississippi State University, Mississippi State, MS 39762 USA. Email: skim@math.msstate.edu. Accessed: 2024-07-15.

[8] Volker John, Gunar Matthies, and Joachim Rang. A comparison of time-discretization/linearization approaches for the incompressible navier–stokes equations. *Computer Methods in Applied Mechanics and Engineering*, 195(44):5995–6010, 2006.

[9] Arnold Reusken. Numerical methods for the navier-stokes equations, January 6 2012. Chair for Numerical Mathematics, RWTH Aachen. Accessed: 2024-07-15.

[10] Mario Botsch, David Bommes, and Leif Kobbelt. Efficient linear system solvers for mesh processing. In Ralph Martin, Helmut Bez, and Malcolm Sabin, editors, *Mathematics of Surfaces XI*, pages 62–83, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[11] Marcin Skotniczny, Anna Paszyńska, Sergio Rojas, and Maciej Paszyński. Complexity of direct and iterative solvers on space–time formulations and time-marching schemes for h-refined grids towards singularities. *Journal of Computational Science*, 76:102216, 2024.

[12] Ulrich Langer and Marco Zank. Efficient direct space-time finite element solvers for parabolic initial-boundary value problems in anisotropic sobolev spaces. *SIAM Journal on Scientific Computing*, 43:A2714–A2736, 08 2021.

[13] Gary Knott. Gaussian elimination and lu-decomposition. 01 2012.

[14] R. Hoekstra. Parallel block jacobi preconditioned gmres for dense linear systems, 2022.

[15] M. Embree. How descriptive are gmres convergence bounds?, 1999. Unspecified.

[16] S. Eisenträger, E. Atroshchenko, and R. Makvandi. On the condition number of high order finite element methods: Influence of p-refinement and mesh distortion. *Computers Mathematics with Applications*, 80(11):2289–2339, 2020. High-Order Finite Element and Isogeometric Methods 2019.

[17] Yao Zhu and Ahmed H. Sameh. *How to Generate Effective Block Jacobi Preconditioners for Solving Large Sparse Linear Systems*, pages 231–244. Springer International Publishing, Cham, 2016.

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

[19] Aditi Ghai, Cao Lu, and Xiangmin Jiao. A comparison of preconditioned krylov subspace methods for nonsymmetric linear systems. *Numerical Linear Algebra with Applications*, 26, 2016.

[20] Nick Brown, Jonathan Mark Bull, and Iain Bethune. A highly scalable approach to solving linear systems using two-stage multisplitting. *CoRR*, abs/2009.12638, 2020.

[21] Michael Souza, Luiz Mariano Carvalho, Douglas Augusto, Jairo Panetta, Paulo Goldfeld, and José Roberto Pereira Rodrigues. A comparison of image and scalar-based approaches in preconditioner selection. *arXiv preprint arXiv:2312.15747*, v1:23 pages, 8 figures, 9 tables, Dec 2023. Submitted on 25 Dec 2023.

[22] Yael Azulay and Eran Treister. Multigrid-augmented deep learning preconditioners for the helmholtz equation. *SIAM Journal on Scientific Computing*, 45(3):S127–S151, 2023.

[23] Maksym Shpakovych. Neural network preconditioning of large linear systems. Technical Report RT-0518, Inria Centre at the University of Bordeaux, October 2023.

[24] Luca Grementieri and Paolo Galeone. Towards neural sparse linear solvers, 03 2022.

[25] Johannes Sappl, Laurent Seiler, Matthias Harders, and Wolfgang Rauch. Deep learning of preconditioners for conjugate gradient solvers in urban water related problems. *ArXiv*, abs/1906.06925, 2019.

[26] Edmond Chow, Hartwig Anzt, Jennifer Scott, and Jack Dongarra. Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.

[27] Toni Dieckhoff and Anna-Valentina Hirsch. Matrixkit: Advanced synthetic matrix generation for machine learning and scientific computing. https://pypi.org/project/matrixkit/0.1.0/, 2024. Accessed: 2024-07-26.

[28] Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[31] Guangxin Su, Hanchen Wang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Simple and deep graph attention networks. *Knowledge-Based Systems*, 293:111649, 2024.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.