On Enabling Layer-Parallelism for Graph Neural Networks using IMEX Integration

by

Omer Ege Kara

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2024

Author's Declaration

Abstract

Graph Neural Networks (GNNs) are a type of neural networks designed to perform machine learning tasks with graph data. Recently, there have been several works to train differential equation-inspired GNN architectures, which are suitable for robust training when equipped with a relatively large number of layers. Neural networks with more layers are potentially more expressive. However, the training time increases linearly with the number of layers. Parallel-in-layer training is a method that was developed to overcome the increase in training time of deeper networks and was first applied to training residual networks. In this thesis, we first give an overview of existing works on layer-parallel training and graph neural networks inspired by differential equations. We then discuss issues that are encountered when these graph neural network architectures are trained parallel-in-layer and propose solutions to address these issues. Finally, we present and evaluate experimental results about layer-parallel GNN training using the proposed approach.

Acknowledgments

This master's process that I experienced taught me many things and allowed me to discover many different research areas that I wouldn't have done without attending.

I want to thank my supervisors Hans De Sterck and Jun Liu for all the mentorship and guidance they provided. Your supervision has shaped my perspective on applied mathematics research a lot. Additionally, thanks for giving me a chance to play soccer in the intramural team.

I want to thank Eric C. Cyr for the discussion related to parallel-in-layer training, and maintaining the TorchBraid library which I was a frequent user. Special thanks for allocating time for online meetings, and timely replies to my emails.

I want to thank Giang Tran, and Roberto Guglielmi for reviewing this thesis and being committee members.

I want to thank the University of Waterloo, especially the Applied Math Department members, including students, faculty, and staff for providing such a rich environment for a young mathematician.

I want to thank my family members, my mother, my father, my sister, and my grandmother for showing their support throughout all phases of my education.

Dedication

I dedicate this thesis to all people who apply math to problems for the benefit of humankind.

Table of Contents

List of Figures

Chapter 1

Introduction

Deep learning is a branch of machine learning that aims to solve learning problems by artificial neural network (ANN) models and training algorithms. Recent advances in deep learning research showed that neural networks have achieved tremendous success in problems of various scientific disciplines, such as computer vision [24], natural language processing [37], and applied mathematics [30]. While the models have been achieving promising success, computations that involve neural networks require long training times and expensive hardware resources, which makes using these tools challenging.

In this thesis, we consider one solution to decrease the computing time of neural networks, named parallel-in-layer training. Given a neural network with many layers, parallel-in-layer training aims to distribute stages of forward and backward propagation to multiple processors. It is powered by concepts and numerical algorithms that were invented for the parallel-in-time integration of dynamical systems.

In the literature, parallel-in-layer training was first applied to Residual Networks [16], and a similar parallel-in-time training is applied to Gated Recurrent Units(GRU) [25], for sequence-models where index in a sequence can be interpreted as time. Parallel-in-layer and parallel-in-time approaches are using ordinary differential equation formulations of these architectures. In this thesis, we apply this training algorithm to a graph convolutional network architecture, named PDE-GCN [8], which is inspired by a parabolic partial differential equation. We evaluate the performance of naive parallel-in-layer training in terms of speed-up and accuracy. We also proposed a feasible solution to problems related to numerical stability and accuracy loss when this GNN is trained in a parallel-in-layer manner.

This thesis is divided into six chapters. Background material on preliminary topics such as numerical solution of differential equations, neural networks, and machine learning on graphs is provided in Chapter 2, followed by a discussion of how neural network architectures in the form of differential equations can be trained parallel-in-layer in Chapter 3. Chapter 4 introduces implicit-explicit time integration of dynamical systems, its application to residual networks, and corresponding feasible implementations for PDE-GCN to make this architecture more suitable for parallel-in-layer training. Chapter 5 describes the experiments and results. Chapter 6 concludes this thesis and discusses future work.

Chapter 2

Background

This chapter provides background material for the upcoming chapters. It states definitions, discusses concepts about differential equations, and introduces algorithms for numerical solutions. We then introduce neural networks and describe different layer types. Lastly, definitions of concepts related to graph machine learning are given.

2.1 Differential Equations and Numerical Solutions

Differential equations state the relationship between functions and their derivatives. The language they introduce is useful in modeling laws in science and engineering, and they can appear in many different formats and types in practical problems. This variety led to different ways to classify those equations to develop theories about them. One common classification is according to the dimension of the function domain and the type of derivatives involved, where differential equations are divided into two categories, ordinary differential equations (ODE) and partial differential equations (PDE). In this section, we give definitions and descriptions of concepts related to ODEs and then mention their PDE counterparts.

2.1.1 Ordinary Differential Equations and Initial Value Problems

This subsection defines ordinary differential equations and initial value problems whose solutions are single-variable scalar functions, $x : \mathbb{R} \to \mathbb{R}$. Definitions for equations whose solutions are vector-valued single-variable functions $\vec{x} : \mathbb{R} \to \mathbb{R}^n$, are similar.

Definition 2.1.1 *An **ordinary differential equation** of order n is an equation of the form,*

$$F(t, x(t), \frac{dx}{dt}, \frac{d^2x}{dt^2}, \dots, \frac{d^nx}{dt^n}) = 0,$$

*where $t \in \mathbb{R}$ is an **independent variable**, and x is single-variable scalar function of t called **dependent variable**, and F defines the equation in terms of t, x, and derivatives of x with respect to t.*

If functions $x(t)$ exist that satisfy the differential equation, then they are defined up to some constants. To eliminate this uncertainty in constant values, differential equations arise in practical problems with initial conditions. In definition 2.1.2, we define initial value problems with first-order ODEs.

Definition 2.1.2 *A first-order differential equation with an initial condition specified at t=0 forms a **first-order initial value problem**:*

$$\frac{dx}{dt} = f(x, t), \tag{2.1}$$

$$x(0) = x^{(0)}. \tag{2.2}$$

*A function x(t) which satisfies both differential equation (2.1) and initial condition (2.2) is called a **solution** of the initial value problem (IVP).*

Like many problems in mathematics, we can discuss the well-posedness of initial value problems. Three properties are required for the well-posedness of an initial value problem: existence, uniqueness, and stability (i.e. the solution changes continuously with changes in initial conditions) of a solution. Well-posedness is a desirable property, when we don't have a closed-form solution but still want to make inferences about the properties of the solution with approximations.

2.1.2 Numerical Methods for Initial Value Problems

Finding closed-form solutions to initial value problems is often not trivial. For such cases, various numerical methods have been developed to simulate solutions to the problem of interest. The predictions generated by these methods are called **numerical solutions**. Obtaining a numerical solution is an algorithmic process. Therefore, numerical methods make the most sense when implemented in a computer environment.

Within the context of IVPs, one iteration of these numerical methods computes a prediction for the function value at some point t in the time domain, in terms of known values (exact or approximate) for points smaller than t. This procedure has several names, some of which are **time integration**, **time stepping**, and **time marching**.

Many different time marching schemes can be found in the literature. Euler's method [11], Heun's method, and Runge-Kutta-Fehlberg [13] are some well-known examples.

Before employing a time marching scheme, a discrete representation of the time interval has to be generated. Given a time interval $[t_0, t_f]$, and a positive number N, a uniform discretization of the time interval with $N+1$ points, leads to grid points $t^{(k)} = t_0 + k\frac{t_f - t_0}{N}$, $k = 0, \dots, N$. This procedure is **temporal discretization** and the set formed by the $t^{(k)}$ values is named as **temporal grid**.

Two different Euler method formulas are given below for the IVP in Definition 2.1.2 and an appropriate temporal discretization. Equation (2.3), is called the **explicit (forward) Euler scheme**, and Equation (2.4) corresponds to the **implicit (backward) Euler scheme**. Here, $x^{(k)}$ stands for the numerical solution value at time point $t^{(k)}$. Numerical solution values are computed for any $t^{(k)}$, $k > 0$ starting from the known initial value $x(t^{(0)}) = x^{(0)}$:

$$x^{(k+1)} = x^{(k)} + hf(t^{(k)}, x^{(k)}), \text{or} \tag{2.3}$$
$$x^{(k+1)} = x^{(k)} + hf(x^{(k+1)}, t^{(k+1)}), \tag{2.4}$$

where $h > 0$ is called the step size.

Two properties worth mentioning about Euler's time marching schemes are their truncation error and numerical stability.

The **truncation error** quantifies the difference between the numerical solution and the exact solution, in terms of step size h. Both the explicit and implicit Euler schemes have local truncation error $\mathcal{O}(h^2)$, and the global truncation error $\mathcal{O}(h)$. These orders can be derived using Taylor's expansion of the exact solution of $x(t)$. Other than Euler method, Runge-Kutta schemes with multiple steps have higher order truncation errors, i.e. values generated are expected to be closer to the exact solution.

Other than truncation error, another concern for a time marching scheme is whether the exact solution and numerical solution stay close and do not diverge from each other as time goes to infinity for the case in which the exact solution approaches zero, despite the errors accumulated in each iteration. This is relevant to the property known as the **numerical stability** of the time marching scheme. Numerical stability can be understood by considering Example 2.1.1 below, which defines a special ODE named the test equation.

Example 2.1.1 *Consider the initial value problem*

$$\begin{aligned} \frac{dx}{dt} &= \lambda x(t), \\ x(0) &= x^{(0)}, \end{aligned} \tag{2.5}$$

where $\lambda \in \mathbb{R}, \lambda < 0$ *Its solution is* $x(t) = e^{\lambda t}$, *and* $\lim_{x\to\infty} x(t) = 0$.

The criterion for numerical stability of the forward Euler scheme can be obtained by following the steps below. First the forward Euler scheme (2.3) is applied to the problem (2.5). We can write $x^{(k+1)}$ *in terms of the initial condition :*

$$x^{(k+1)} = x^{(k)} + h\lambda x^{(k)} \tag{2.6}$$

$$= (1 + h\lambda)x^{(k)} \tag{2.7}$$

$$= (1 + h\lambda)^{k+1}x^{(0)}. \tag{2.8}$$

Because $\lim_{t\to\infty} x(t) = 0$, *we require for numerical stability that* $\lim_{k\to\infty} x^{(k)} = 0$, *to have the numerical solution remain close to the exact solution as time increases. This is possible if*

$$|1 + h\lambda| < 1, \tag{2.9}$$

$$-1 < 1 + h\lambda < 1, \tag{2.10}$$

$$-2 < h\lambda < 0, \tag{2.11}$$

$$h < \frac{-2}{\lambda}. \tag{2.12}$$

Similarly, the solution sequence generated by the backward Euler scheme in terms of the initial condition can be obtained. Application of the backward Euler scheme (2.4) to problem (2.5) leads to recursive relation (2.15):

$$x^{(k+1)} = x^{(k)} + h\lambda x^{(k+1)}, \tag{2.13}$$

$$= \frac{1}{(1 - h\lambda)}x^{(k)}, \tag{2.14}$$

$$= \frac{1}{(1 - h\lambda)^{k+1}}x^{(0)}. \tag{2.15}$$

Following a similar reasoning as how (2.8) was derived, the numerical stability condition of the backward Euler scheme (2.16) can be obtained:

$$\frac{1}{|1 - h\lambda|} < 1. \tag{2.16}$$

Since $\lambda < 0$, this inequality holds for any $h > 0$. Therefore there is no restriction on the step size h for numerical stability.

As seen above, the backward Euler scheme is stable for any step size h, this is called **unconditional stability** of the backward Euler. However, larger step sizes when used by the forward Euler scheme lead to unstable numerical solutions due to the upper bound in inequality (2.12). Equations sensitive to the choice of step size are called **stiff** differential equations. In general implicit methods allow taking larger time step values, which are preferred for problems with stiffness.

2.1.3 Partial Differential Equations and Diffusion Equation

Other than ODEs, partial differential equations (PDEs) are another type of equation that we will mention. PDEs involve functions with multiple variables and partial derivatives of the functions involved. We will focus on one relevant type of PDEs called diffusion equations, rather than discussing PDEs in general.

A diffusion PDE models heat/substance distribution in some region over time. The heat equation is given by

$$\frac{\partial u}{\partial t} = \alpha \nabla \cdot \nabla u, \tag{2.17}$$

where u is a multivariable function that maps a point $\vec{x} \in \mathbb{R}^n$ and time $t \in \mathbb{R}^+$ to a real scalar value. Also, $\nabla \cdot$ is divergence and ∇u stands for the gradient of u; they only involve partial derivatives of the spatial variables. Finally, $\alpha > 0$ is called the diffusion coefficient, which is assumed to be a constant value.

The form of the diffusion equation in one spatial dimension is given by

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad u(x, 0) = f(x). \tag{2.18}$$

Here, $u(x, 0) = f(x)$ is called the initial condition.

Similar to ODEs, numerical schemes exist for simulating the solution of diffusion PDEs. These numerical schemes are obtained by discretizing time and space to obtain finite difference approximations of derivatives and can be divided into explicit and implicit schemes. For example, the explicit method Forward in Time Central in Space, uses the current state of the system, and the implicit method Backward in Time Central in Space, uses future function values to approximate spatial derivative. The Crank-Nicholson method combines two methods by averaging and it can be considered as an example of the implicit-explicit method.

The explicit method has a condition for numerical stability, which is known as the Courant-Friedrichs-Lewy (CFL) condition [6]. The implicit method and the Crank-Nicholson has no restrictions for stability.

2.2 Machine Learning and Neural Networks

There are many different definitions of what the field of machine learning is concerned with. One can view the task as an approximation of unknown functions by parametrized models. The function is approximated according to data, which provides information about the true function. Formal introduction to machine learning and its problems can be found in resources [34, 36].

Problems of machine learning can be divided into classes. One division is according to whether data is labeled or not. Such division puts problems with labeled data into **supervised learning**, and the ones with unlabeled data into **unsupervised learning**. In this thesis, our interest will be in supervised learning problems and we will assume the dataset consists of ordered pairs and will denote the dataset by $S = \{(X_i, Y_i)\}_{i=1}^{N_D}$, where X_i denotes the feature vector, and Y_i denotes the label of i^{th} sample. The domains X_i and Y_i belong to differ: their components can be binary, real, or in some other data form depending on the problem. In supervised learning the goal is obtaining a parametrized function $\hat{f}(X, \theta)$ such that $\hat{f}(X_i, \theta) \approx Y_i$ for all samples in the dataset S. The rule for computing the parametrized function is defined by a **model** decided to be used. θ stands for all learnable parameters of the model.

Artificial Neural Networks (ANN) are one family of possible model types, which we can use for obtaining $\hat{f}$. They are inspired by mechanisms in the brain and can be interpreted as a directed graph, especially in visuals.

Neural networks consist of parts called layers, each layer has learnable parameters that define the mapping for a given input. Input and output relation for layer i on arbitrary input x can be modeled as a parametrized function similar to ANNs. We will denote the output of function defined by layer i on vector x with $f^{(i)}(x, \theta^{(i)})$, and the parameters $\theta^{(i)} \in \mathbb{R}^p$, will be weights of layer i and $x \in \mathbb{R}^{n_{i-1}}$, in function notation $f : \mathbb{R}^{n_{i-1}} \times \mathbb{R}^{n_p} \to \mathbb{R}^{n_i}$, n_i stand for the output dimension of layer i.

There are various layer types, designed for different purposes. The most primitive layer type is the fully connected layer as used in feedforward neural networks. Computation of a single fully connected layer with an index i involves an application of an affine transformation defined by its parameters, $\theta^{(i)}$ that outputs a hidden state of the layer h_i and application of nonlinear activation function to h_i to compute output. Parameters $\theta^{(i)}$ consists of weight matrix W_i, and bias vector b_i. Example activation functions σ are *Relu* and *tanh*, they introduce nonlinearity to the function. Formulas for the fully connected layer are given in Example 2.2.1 below.

Example 2.2.1 *Let $x^{(i-1)} \in \mathbb{R}^{n_{i-1}}$ be the output of layer $i-1$, $W^{(i)} \in \mathbb{R}^{n_i \times n_{i-1}}$, $b^{(i)} \in \mathbb{R}^{n_i}$ weight matrix and bias vector. The hidden state h_i and output x_i are obtained as follows :*

$$h^{(i)} = W^{(i)} x^{(i-1)} + b^{(i)}, \tag{2.19}$$

$$x^{(i)} = \sigma(h^{(i)}). \tag{2.20}$$

Fully connected layer computations take linear combinations of input vector elements and compute output according to them. They are not designed specifically for recognizing local, stationary, and multiscale patterns, which are properties of natural signals such as sound and images. Convolutional layers are types of layers designed so that these natural signal properties are considered. The convolution operation is linear and has a sparse matrix representation. More detailed information about convolutional can be found in [27].

For a given learning problem, after defining the architecture of the model in terms of layers, the next stage is **training**, where model parameters, that achieve desirable accuracy are obtained. This stage involves defining an optimization problem and performing optimization algorithm iterations to obtain local or global minimum points in learnable parameter space.

In general, the optimization problem for supervised learning problems involves two terms, loss and regularization. The loss term $\frac{1}{N}\sum_{i=1}^{M} L(Y_i, \hat{f}(X_i, \theta))$ measures how close the neural networks approximate the to true labels. The regularization term $\lambda R(\theta)$'s presence prevents overfitting, i.e. it enhances generalization to unseen data. The combined loss function results in the following optimization problem:

$$\text{minimize} \quad J(\theta) = \frac{1}{N}\sum_{i=1}^{M} L(Y_i, \hat{f}(X_i,\theta)) + \lambda R(\theta). \tag{2.21}$$

The local minimum of the optimization problem can be obtained by performing batch gradient descent; with an update rule given by,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta J(\theta^{(t)}) \tag{2.22}$$

where $\theta_{(t)}$ is the parameter state after iteration t, and α is the **learning rate**. In practice, to shorten computing time, stochastic gradient-based iterations are used. Some examples of stochastic iterations that are used for training neural networks are stochastic gradient descent (SGD), ADAM [22], SGD with Momentum [26], and RMS-Prop [35].

2.3 Machine Learning on Graphs

Standard neural network layers, such as feed-forward layers, convolutional layers, and recurrent neural networks define their operations on input that belongs to an Euclidian domain, where components of input have some spatial position. For example, image pixels have a specific coordinate that defines their position, or each word in a text has an order that defines its position in a text.

However, not all data are encoded in the appropriate format for Euclidian domains. One example type of data is data that can be represented as a graph. Standard layer operations are not suitable for data belonging to domains that are non-Euclidian. This deficiency requires layer formulations different from standard ones, but at the same time, they should have some common design with standard layers to make the transferability of theory possible.

In this section, we first introduce definitions of graphs. We then define graph neural network operations in terms of primitive definitions. Afterward, we describe an example graph machine learning task called node classification by providing details about the dataset CORA [33]. We end the section by describing Graph Convolutional Network (GCN) [23], which is relevant to future chapters.

2.3.1 Basic Definitions about Graphs

Graph theory is the branch of mathematics that deals with mathematical structures called graphs. They have applications in modeling many different situations. Social networks, geographic maps, and chemical substances are some examples that are suitable for modeling with graphs. The formal definition of a graph is given in the definition below:

Definition 2.3.1 *A* ***graph*** *G is a pair of two sets, (V, E), vertices and edges. E can either contain 2-element subsets or 2-tuples of vertices. If E consists of 2-element subsets the graph is* ***undirected****, otherwise it is* ***directed****.*

Nodes in the graph are individual elements and, edges are connections between those elements. In a social network example, profiles can be viewed as nodes, and having a friendship relation between two profiles can be seen as the presence of an edge between corresponding nodes.

For a node in a graph, it can be important to talk about what other nodes it is connected to. The definition for neighbors of a node is given below:

Definition 2.3.2 *Let $G = (V, E)$ be an undirected graph. The* ***neighboring nodes*** *of a vertex $v \in V$, denoted by $N(v)$, are the vertices that share an edge with v in G. Formally,*

$$N(v) = \{u \in V \mid \{v, u\} \in E\}.$$

The definition for the directed graph is similar, by replacing 2-element subsets in the definition with tuples.

In social media examples, for a given profile, neighboring nodes consist of all friends of this profile.

There are characteristic matrices for graphs, constructed according to the vertex (V) and edge (E) sets. In Definitions 2.3.3, 2.3.4, 2.3.5 and 2.3.6 below, we state the rules to construct some matrices relevant to this thesis. During construction, it is assumed that nodes in V are indexed from 1 to N, where $N = |V|$ and nodes of the original graph G do not have any self-connections. Additionally, we define an indicator function $e : N \times N \to \{0, 1\}$, where the value of $e(i, j)$ is 1 if there is an edge between nodes with indexes i and j, and otherwise it is 0.

Definition 2.3.3 *Let $G = (V, E)$ be an undirected graph. The* ***adjacency matrix*** *$\boldsymbol{A} = (a_{ij}) \in \mathbb{R}^{N \times N}$, and its entries are defined as*

$$a_{ij} = e(i, j).$$

From the adjacency matrix A, the ***adjacency matrix with self connections*** *is obtained as $\tilde{A} = A + I_N$. It is simply an adjacency matrix of a modified G where the self-edge for each node is added.*

Definition 2.3.4 *Let $G = (V, E)$ be an undirected graph. The* ***degree matrix D*** $= (d_{ij}) \in \mathbb{R}^{N \times N}$*, is a diagonal matrix whose entries are defined according to the rule below:*

$$d_{ij} = \begin{cases} \sum_{k=1}^{N} e(i,k) & \text{if } i = j, \\ 0, & \text{otherwise.} \end{cases}$$

Similar to $\tilde{A}$, the ***degree matrix with self connections*** *is obtained by $\tilde{D} = D + I_N$.*

Definition 2.3.5 *Let $G = (V, E)$ be an undirected graph. The* ***incidence matrix*** $B = (b_{ij}) \in \mathbb{R}^{N \times M}$ *is a matrix whose entries are defined as follows:*

$$b_{ij} = \begin{cases} 1, & \text{if edge } e_j \text{ is incident to vertex } v_i, \\ 0, & \text{otherwise.} \end{cases}$$

Definition 2.3.6 *Let $G = (V, E)$ be an undirected graph. The* ***graph Laplacian*** *L of G is defined as,*

$$L = D - A.$$

The ***symmetric normalized graph laplacian with self-connections*** *$\tilde{L}$ is defined by*

$$\tilde{L} = I - \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}.$$

There is a connection between the Laplacian operator $\Delta = \nabla \cdot \nabla$ that appears in equation (2.17), and matrix operation on graphs. Matrix operator L acts as a discrete version of the continuous Laplacian operator to functions defined on nodes. Additionally, there are connections between the gradient and the incidence matrix B, and between divergence and B^T. These connections can be seen by considering the following example:

Example 2.3.1 *A grid discretizing a line segment with 5 points can be represented with the undirected graph $G = (V, E)$ below:*

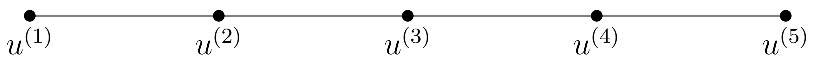

Figure 2.1: Graph for a 1-D grid with 5 grid points

The column vector of u values is given by:

$$u = \begin{bmatrix} u^{(1)} & u^{(2)} & u^{(3)} & u^{(4)} & u^{(5)} \end{bmatrix}^T$$

The adjacency (A), degree (D), and Laplacian (L) matrices of the graph are:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

The Laplacian on the 1-D grid corresponds to an operator that approximates the second partial derivative of the spatial variable, assuming the grid spacing $h = 1$ without loss of generality. Let Q be the matrix representation of the Laplacian PDE operator and for an interior grid point with index i, application of the operator to interior grid points results in the following computation:

$$(Qu)^{(i)} = u^{(i+1)} - 2u^{(i)} + u^{(i-1)}. \tag{2.23}$$

Another way to obtain this expression is by performing a matrix-vector product involving L and the vector of u_i values.

$$(Lu)^{(i)} = 2u^{(i)} - u^{(i+1)} - u^{(i-1)}. \tag{2.24}$$

The expression in equation (2.24) is equal to (2.23) with the opposite sign:

The same operation can be expressed in terms of incidence matrices. For the grid in Figure 2.1, incidence matrix B and its transpose is given by:

$$B = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}, \qquad B^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

The matrix B corresponds to a gradient operation (∇); it maps vectors whose dimension is equal to the number of nodes to vectors whose dimension is equal to the number of edges. The matrix B_1^T acts like a divergence operator ($\nabla\cdot$); it maps vectors with dimension equal to the number of edges to vectors with size number of nodes.

Similar to the continuous case, the Laplacian L is nothing but the application of the gradient operation, B followed by divergence, B^T as illustrated by the following equality:

$$Q = -L = -B^T B. \tag{2.25}$$

Similar to this example in 1-D, this interpretation also holds in higher dimensions, if a grid is viewed as a graph.

2.3.2 Node Classification as a Learning Task

Graphs are involved in various types of learning tasks and one that we tested our approach with is node classification. In the node classification problem instance relevant to this work, there is a single graph $G = (V, E)$, where each node belongs to a class, and each node has a feature vector. Node features are completely given and node classes are partially given to the model. The learning task is training a model that aims to predict unseen node classes, by using feature vectors, seen node classes, and underlying edge relations in the training part.

The name of the dataset we used in our experiments is CORA [33]. The type of graph the CORA dataset includes is called the citation network. There are 2708 nodes, each corresponding to a computer science publication, and 5429 undirected edges representing citations. Each node belongs to one of 7 classes according to the category of the paper, and a binary feature vector with 1433 entries describes each publication, and each entry indicates whether a keyword appears in the paper.

In the semi-supervised learning task, the training set consists of 140 nodes, 20 per class. The validation and test sets consist of 500 and 1000 nodes respectively.

2.3.3 Graph Neural Networks

Graph neural networks are the general name of neural networks that can operate on data represented as graphs. The key idea in designing graph neural networks is to include message passing in layers. Message passing is the general name of an operation that computes node features of outputs as a combination of its feature vector and neighboring node feature vectors. The message passing formulation that we will consider is the one employed in graph convolutional network (GCN) layers [23]; other variants of message passing can be seen in [32, 19, 38].

The update rule of a graph convolutional layer is given by

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}). \tag{2.26}$$

Here, $\tilde{A}$ and $\tilde{D}$ are the matrices defined in Definitions 2.3.3 and 2.3.4, $H^{(l)} \in \mathbb{R}^{N \times K}$ is the node activation of layer l, each row belonging to a node, matrix K is the dimension of the node activation vectors, i.e., the channel number of node feature vectors in layer l, $W^{(l)}$ is the trainable weight matrix of layer l, and σ is the chosen activation function.

The multiplication with $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ from the left acts as a message passing operation, multiplication with $W^{(l)}$ from the right mixes channels for each node vector and application of σ introduces nonlinearity.

Chapter 3

Parallel-in-Layer Training of Neural Networks

This chapter includes detailed background about parallel-in-layer training of neural networks. It first details the multigrid-reduction-in-time (MGRIT) algorithm, a crucial component for achieving layer-parallelism. Afterward, we introduce neural differential equations and end the chapter by giving the formulations on parallel-in-layer training of residual networks.

3.1 Multigrid-Reduction-in-Time Algorithm

Classical methods for simulating initial value problems, which we mentioned in Chapter 2, generate values of the numerical solution sequentially. Obtaining the solution value at an arbitrary time point requires previously computed values at earlier time points, and this dependency on past values puts a barrier to the trivial parallelization of time-marching computations. However, modern computing technology favors increasing the number of computing units instead of developing faster processors. To use modern computers with many cores, numerical integration algorithms that are suitable for parallelization with multiple processors are proposed to replace sequential time-marching schemes, and the general name of these methods is parallel-in-time integration [15].

The parallel-in-time integration algorithm that we consider for introducing layer-parallelism in neural networks is the multigrid-reduction-in-time (MGRIT) algorithm [12]. A general form of time-dependent ODE initial value problem that MGRIT can be used for, is given by

$$\frac{dx(t)}{dt} = f(t, x(t)), \quad x(0) = x_0, \quad t \in (0, t_f]. \tag{3.1}$$

It is possible to apply MGRIT to problems with PDEs as well.

To estimate values of the solution $x(t)$ at certain points in time, we discretize the time interval with a grid of uniformly distributed points $t^{(i)} = i\Delta t$, $i = 0, 1..., N_t$, where $\Delta t = t_f/N_t$. This discretization divides the interval into N_t equally spaced time intervals. The approximate solution at $t^{(i)}$ is $x^{(i)} \approx x(t^{(i)})$ for $i = 1..., N_t$, with $x^{(0)} = x(0) = g_0$. Given this discretization of the time interval, the time integration of the initial value problem of equation (3.1) is computed by the sequence

$$x^{(i)} = \mathbf{\Phi}(x^{(i-1)}), \quad i = 1, 2, \ldots, N_t, \tag{3.2}$$

, where $\mathbf{\Phi}$ is the time propagation rule that depends on the time integration scheme of choice. The variants of the Euler scheme in equations (2.3) and (2.4) we mentioned in Chapter 2 are examples of formulas that define different time propagation rules. Time integrator $\mathbf{\Phi}$ can have an index as well if different rules are used at different time points, and such dependency can be useful to represent adaptive solvers. Throughout this thesis, we will assume the choice of time propagation rule is global, independent of the time point index.

The $x^{(i)}$ values satisfying the relation (3.2) define a system with $N_t + 1$ algebraic equations whose representation is given by

$$\mathcal{A}(\vec{x}) \equiv \begin{bmatrix} x^{(0)} \\ x^{(1)} - \mathbf{\Phi}(x^{(0)}) \\ \vdots \\ x^{(N_t)} - \mathbf{\Phi}(x^{(N_t-1)}) \end{bmatrix} = \begin{bmatrix} g^{(0)} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \equiv \vec{g}. \tag{3.3}$$

Conventional time-stepping methods solve this system sequentially, by computing $x^{(i)}$ values one after another according to the propagation rule (3.2). This sequential forward scheme obtains all $x^{(i)}$ values after N_t time steps, with time complexity $\mathcal{O}(N_t)$.

The MGRIT algorithm follows an iterative procedure to generate a sequence converging to the solution of the system (3.3). MGRIT starts with an initial guess for the solution and after each iteration, the application of MGRIT is expected to update the current guess with a better approximation. The pseudocode that outlines the operations of one MGRIT iteration is given in Algorithm 1, this example is for 2-level MGRIT; pseudocode for arbitrary level numbers are similar with recursion.

Algorithm 1 MGRIT-FAS($\mathcal{A}, \vec{x}, \vec{g}$)

```
1: Apply $F$-relaxation or $FCF$-relaxation to $\mathcal{A}_0(\vec{x}_0) = \vec{g}_0$
2: Inject the approximation and its residual to the coarse grid:
       $\vec{x}_1 = \mathbf{R}_I(\vec{x}_0)$,
       $\vec{g}_1 = \mathbf{R}_I(\vec{g}_0 - \mathcal{A}_0(\vec{x}_0))$
3: Solve $\mathcal{A}_1(\vec{v}_1) = \mathcal{A}_1(\vec{x}_1) + \vec{g}_1$
4: Compute the error approximation: $\vec{e} = \vec{v}_1 - \vec{x}_1$
5: Correct using ideal interpolation: $\vec{x}_0 = \vec{x}_0 + \mathbf{P}(\vec{e})$
```

In Algorithm 1 FAS stands for Full-Approximation Storage, and it is used for nonlinear problems, but the algorithm is still valid for linear problems. We will describe the algorithm details in the following paragraphs.

Iterations of MGRIT achieve speedup by performing operations on multiple temporal grids. Each temporal grid has a level, associated with it. The grid with the smallest level, i.e., level 0, is the finest grid, it contains all time points of interest.

Given the fine grid, coarser grids are constructed according to an integer parameter called coarsening factor, which we will denote with c_f. Points on the fine grid are divided into two classes, F-points, and C-points. All points that have an index with an integer multiple of c_f form the C-points of the grid, and the remaining points are F-points. The C-points of the fine grid are the points that define the coarser grid on one higher level. Figure 3.1 summarizes the relation between fine and coarse temporal grids. Different coarsening factors can be used for different grid levels, however, for simplicity, we will assume c_f is shared among levels.

It is worth clarifying the notation before describing the steps of Algorithm 1. Lowercase letters with arrows on top are vectors, and bold capital letters are operators that map vectors to vectors. Superscripts in the algorithm denote the level of the grid, where the vector of level 0 contains all data corresponding to points of the fine grid.

The first phases of MGRIT are the relaxations that appear in line 1 of Algorithm 1. Relaxations in MGRIT are local operations that update approximation at temporal points in a parallelizable way. F-Relaxation starts by taking the solution value at a C-point and updates the solution at F-points in future time by applying the propagation rule (3.2). This propagation stops at an F-point whose next point is a C point. Similar to

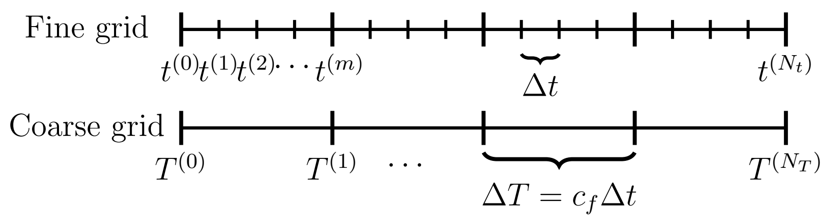

Figure 3.1: Fine and coarse temporal grid obtained by uniform discretization. The coarse grid is created by removing all F- points (represented by short markers) of the fine grid.

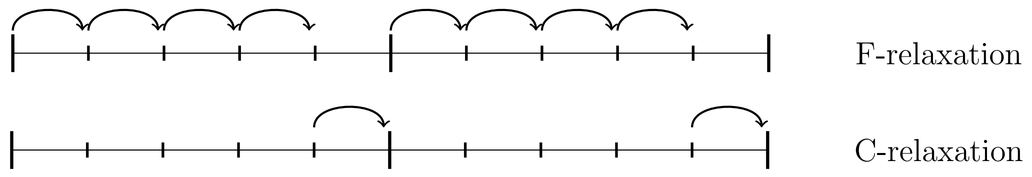

Figure 3.2: F-relaxation and C-relaxation on temporal grids with coarsening factor parameter $c_f = 5$.

F-relaxation, C-relaxation updates the solution value at C points, by applying the rule (3.2) to the solution value located at the preceding F point. The F and C relaxations can be considered primitive relaxations. New relaxations can be defined in terms of these. FC relaxation consists of F relaxation followed by C and FCF-relaxation consists of application of F, C, and F relaxations. Relaxations can be parallelized by distributing the computations of intervals between C-points to different processors. Visuals describing F and C relaxations are provided in Figure 3.2. This relaxation phase updates the fine approximation $\vec{x}_0$ in Algorithm 1.

The phase after relaxation is the restriction of the updated fine grid solution $\vec{x}_0$ and the residual $\vec{g}_0 - \mathcal{A}_0(\vec{x}_0)$ to the coarser grid. There are different ways to define restriction operators; one common choice is injection, denoted with $\mathbf{R}_I$. Restriction with injection transfers updated solution values at C-points to a coarser grid vector without any change. The restricted solution on level 1 is denoted as $\vec{x}_1$ and the restricted right-hand side is $\vec{g}_1$.

After the restriction of the vectors to the coarse grid, the error on the coarse grid has to be calculated. This error is calculated according to the coarse-level time propagator. The coarse-level time propagator has to be different from the one defined for the fine grid because the length of time intervals between grid points is changed from Δt to $c_f \Delta t$. We denote the time propagator function of level l with $\mathbf{\Phi}_l(.)$ and the propagation operator of the coarse grid (on level 1) with $\mathcal{A}_1$. If the fine level contains $N_t + 1$ points with N_t subintervals, the coarse level system contains $N_t/c_f + 1$ points and $N_t/c_f \equiv N_T$ non-overlapping intervals between coarse points. This error on the coarse grid is computed by solving a system of equations similar to the fine level. If the grid is the coarsest one, the exact solution to the coarse system is calculated, otherwise, an approximate using an MGRIT iteration is computed. The equation that needs to be solved on the coarse grid is given in line 3 of Algorithm 1. The operator $\mathcal{A}_1$ is defined according to

$$\mathcal{A}_1(\vec{x}_1) = \begin{bmatrix} x_1^{(0)} \\ x_1^{(1)} - \mathbf{\Phi}_1(x_1^{(0)}) \\ \vdots \\ x_1^{(N_T)} - \mathbf{\Phi}_1(x_1^{(N_T-1)}) \end{bmatrix}. \tag{3.4}$$

where $x_1^{(i)}$ stands for guessed solution value at i^{th} time point of the coarse grid (on level 1).

After error calculation on the coarse level, this error vector is interpolated to the fine level to estimate the fine error. This fine grid error estimate is used to update the fine grid solution, to make it closer to the exact solution of the fine system.

Optionally, error correction can be followed by F-relaxation to propagate values at C points to F points. This concludes one MGRIT iteration.

MGRIT can be seen as a fixed-point method: its iterations are expected to converge to the fixed point of the equation (3.5) below:

$$\vec{x}_{(k)} = \text{MGRIT-FAS}(\mathcal{A}, \vec{x}_{(k-1)}, \vec{g}). \tag{3.5}$$

Here $\vec{x}_{(k)}$ is the approximation computed by the k^{th} MGRIT iteration. This notation will appear in the parallel-in-layer algorithm where MGRIT is used instead of classical time marching.

3.2 Neural Differential Equations

With the increase in popularity of deep learning in the last decade, there have been several works to define connections between the theory of dynamical systems and neural network architectures [39, 18, 4]. These ideas enabled the application of prior knowledge from differential equations to deep learning. Parallel-in-layer training's main inspiration relies on this differential equation interpretation of neural networks. In this section, we will focus on the ODE viewpoint of residual networks (ResNets) [21]. We will describe the foundations of parallel-in-layer algorithms with the help of this viewpoint. In Chapter 4, we will describe PDE-GCN [8], a graph neural network architecture, whose layers are inspired by the numerical solution of diffusion PDEs which we introduced in equation (2.17) before.

ResNets are neural network architectures that are designed to allow the training of deeper networks with skip connections. Skip connections make it easier to express identity maps in forward propagation from layer to layer. The output of layer $k+1$ is computed in terms of the output of layer k by

$$x^{(k+1)} = x^{(k)} + f(x^{(k)}, \theta^{(k)}). \tag{3.6}$$

Here $x^{(k)}$ denotes the output of layer k, and $\theta^{(k)}$ stands for the trainable parameters of layer k.

For the N-layer residual network this formula is calculated starting from the initial value $x^{(0)}$ and ends when $x^{(N)}$ is obtained. We will assume that before the application of residual layers, input feature vector X_i will be mapped to $x_i^{(0)}$, an initial embedding for the i^{th} sample. The initial embedding is obtained by an application of an embedding layer L_{in} to feature vectors. We will denote this calculation with

$$x_i^{(0)} = L_{in} X_i. \tag{3.7}$$

The forward propagation of residual layers performed for obtaining $x^{(N)}$, can be seen as a sequential evaluation of the forward Euler scheme with time step size $h = 1$, for the initial value problem with parametrized ODE

$$\frac{dx(t)}{dt} = f(x(t), \theta(t)), \quad x(0) = x^{(0)} = L_{in} X, \quad t \in (0, t_N]. \tag{3.8}$$

In this viewpoint $x^{(N)}$ corresponds to the value of the solution at time point t_N, which is $x(t_N)$.

This initial value problem interpretation lets us view the optimization problem for training in (2.21) as an optimization problem of a functional which has constraints in the form of a differential equation. ODE-constrained optimization that corresponds to (2.21) is given by

$$\min \quad J(\theta(t)) = \frac{1}{M}\sum_{i=1}^{M} L(y_i, x_i(t_f)) + \int_0^{t_f} R(\theta(t))\,\mathrm{d}t \tag{3.9}$$

$$\text{subject to} \quad \frac{dx_i(t)}{dt} = f(x_i(t), \theta(t)), \quad t \in (0, t_f], \tag{3.10}$$

$$x_i(0) = L_{in}X_i \qquad \forall i = 1, \dots, M. \tag{3.11}$$

Here, L is the training loss function that measures the difference between predictions $x_i(t_f)$ and labels y_i, and R is a regularization function. It defines a training objective over M samples, where each sample has a different initial condition. The differential equations controlled by θ map initial conditions to output vectors at final time t_f.

Obtaining a solution to the constrained optimization problem in (3.9) is not tractable; discretization of the ODE constraint and the parameter function $\theta(t)$ yields the approximate problem

$$\min \quad J(\theta) = \frac{1}{M}\sum_{k=1}^{M} L(Y_k, x_k^{(N)}) + \lambda \sum_{i=0}^{N-1} R(\theta^{(i)}) \tag{3.12}$$

$$\text{subject to} \quad x_k^{(i+1)} = \mathbf{\Phi}(x^{(i)}, \theta^{(i)}), \quad \forall i = 0, \dots, N-1. \tag{3.13}$$

$$x_k{}^{(0)} = L_{in}X_k \qquad \forall k = 1, \dots, M. \tag{3.14}$$

It represents a learning problem for residual networks as a constrained optimization problem over a finite-dimensional space of parameters with algebraic constraints. Here N is the number of residual layers and M is the number of training examples. We denote the residual network propagation rule in equation (3.6) with $\mathbf{\Phi}$, where $x^{(k+1)} = \mathbf{\Phi}(x^{(k)}, \theta^{(k)})$.

In addition to constraints (3.13) and (3.14) (called state equations), there are two more groups of constraints which are the adjoint equations and the design equations.

1. *Adjoint equations*

$$\bar{x}_k^{(i)} = \left(\frac{\partial \mathbf{\Phi}(x_k^{(i)}, \theta^{(i)})}{\partial x}\right)^T \bar{x}_k^{(i+1)}, \qquad \forall i = 0, \dots, N-1, \tag{3.15}$$

$$\text{with} \quad \bar{x}_k^{(N)} = \frac{1}{M}\left(\frac{\partial L(Y_K, x_k^{(N)})}{\partial x}\right)^T, \qquad \forall k = 1, \dots, M. \tag{3.16}$$

2. *Design equations*

$$0 = \sum_{k=1}^{M} \left(\frac{\partial \mathbf{\Phi}(x_k^{(i)}, \theta^{(i)})}{\partial \theta^{(i)}} \right)^T \bar{x}_k^{(i+1)} + \left(\frac{\partial R}{\partial \theta^{(i)}} \right)^T \quad , \forall i = 0, \ldots, N-1. \tag{3.17}$$

Here, the $\bar{x}_k^{(i)}$s are the adjoint state variables. Adjoint constraints and design constraints are derived using the method of Lagrange multipliers, which is a common method to solve optimization problems with smooth equality constraints. Adjoint constraints are a result of the forward propagation constraints in (3.13) and (3.14), and the design equation derivation comes from optimality conditions, i.e. forcing partial derivatives of the Lagrangian to zero.

Each constraint satisfaction and optimality condition has a corresponding phase in neural network training. The forward propagation phase corresponds to obtaining values that satisfy the state equation (3.13). Backpropagation computes the loss gradient which corresponds to generating the adjoint variable sequence that satisfies (3.15). Finally, the right-hand side of the design equations is nothing but the analytical gradient of the loss and regularization terms with respect to network parameters. Taking gradient descent steps corresponds to solving the design equations.

3.3 Parallel-in-Layer Training of Neural Networks with MGRIT

Layer-parallel training aims to distribute the phases of forward and backward propagation to multiple processors and execute those phases in parallel so that total execution time can be shortened.

In [16], the authors achieve parallelism by replacing serial forward/backward propagation with MGRIT iterations applied to state/adjoint systems and then computing gradients with output of the MGRIT iterations. The reason for the applicability of MGRIT to the state/adjoint conditions is that the rule generates the solution of these conditions, and has the form of sequential time marching. Like the state equations, the adjoint equations can be interpreted as time integration starting from $\bar{u}_N$ and ending at $\bar{u}_0$. The applicability of MGRIT can be understood better when considering the system of equations for the adjoint variables:

$$\mathcal{A}(\vec{x}_{adj}) \equiv \begin{bmatrix} \bar{x}^{(N)} \\ \bar{x}^{(N-1)} - (\frac{\partial \mathbf{\Phi}(x^{(N)},\theta^{(N)})}{\partial x})^T \bar{x}^{(N)} \\ \vdots \\ \bar{x}^{(0)} - (\frac{\partial \mathbf{\Phi}(x^{(1)},\theta^{(1)})}{\partial x})^T \bar{x}^{(1)} \end{bmatrix} = \begin{bmatrix} \frac{1}{M}\left(\frac{\partial L(Y,x^{(N)})}{\partial x}\right)^T \\ 0 \\ \vdots \\ 0 \end{bmatrix} \equiv \vec{g}_{adj}. \tag{3.18}$$

The linear system in equation (3.18) is in a form that is suitable for MGRIT iterations, i.e., a $\mathbf{\Phi}$ rule can be constructed for it. Therefore MGRIT can be utilized to approximate its solution to obtain an approximation for the adjoint variables. We can denote the relation between approximate solutions of the adjoint system with MGRIT iterations with a relation stated in equation (3.5). Similar to the state equations, MGRIT converges to the fixed point of this sequence.

After obtaining approximations of the state and adjoint variables, the approximate gradient can be calculated and used for a gradient descent step, and this is the end of one epoch. The pseudocode of one epoch of training is given in Algorithm 2.

Algorithm 2 Layer-Parallel-Epoch

```
1: Compute state variables with T1 iterations $\vec{x} = MGRIT - FAS(A, \vec{x}, \vec{g})$
2: Compute adjoint variables with T2 iterations $\vec{x}_{adj} = MGRIT - FAS(A_{adj}, \vec{x}_{adj}, \vec{g}_{adj})$
3: Assemble Gradient of Loss $J$ w.r.t. weights, $\nabla_\theta J$, using (3.17)
4: Perform gradient descent update on trainable parameters, $\vec{\theta} = \vec{\theta} - \alpha \nabla_\theta J$
```

The main motivation for utilizing MGRIT for neural network training is to achieve speed-up by eliminating the bottleneck introduced by serial forward and backward propagations. The potential for speed-up in training becomes more apparent when the layer number is large and the number of MGRIT iterations per epoch is small.

Chapter 4

IMEX Schemes Applied to PDE-GCN

In this chapter, we introduce PDE-GCN-D, the graph neural network architecture that we choose to evaluate the performance of the parallel-in-layer training algorithm's performance introduced in Section 3.3. We then mention potential numerical instability that arises due to stepsizes of different lengths when multigrid iterations of more than one level are performed. In the final part, we discuss a potential solution for numerical instability called IMEX integration and outline various implementation choices for parallel-in-layer training.

4.1 Dynamical Systems, GNNs, and PDE-GCN

Similar to neural network architectures that operate on data belonging to the Euclidian domain, there have been works to design new graph neural network architectures inspired by differential equations [5, 3, 31, 9, 8]. The general idea of differential equation-inspired graph neural networks is picking up a dynamical system whose evolution in time was studied, defining graph neural network layers accordingly, and explaining the strengths of the chosen dynamical system as an inductive bias for a particular graph machine learning problem. As examples of dynamical system-inspired models, GRAND [3] and PDE-GCN-D [8] are inspired by diffusion PDEs. Their training dynamics are similar to GCNs [23]. PDE-GCN-H [8] and Graph-CON [31] employ training dynamics where energy is preserved, and ADR-GNN [9] includes an advection term in its architecture and achieves feature transportation with directionality. Broad information about GNN architectures inspired by dynamical systems can be found in the survey paper [20].

Papers on the architectures GRAND, PDE-GCN-D, and Graph-CON discuss over-smoothing problems that appear in the training of earlier GNN architectures [23, 38], using a dynamical systems point of view. Over-smoothing in earlier architectures makes it hard to train GNN models with many layers. Architectures in [3, 31, 8] reported they were able to maintain accuracy at a desirable level for a large number of layers, where earlier GNN architectures suffer. The appearance of GNN architectures that use a large number of layers makes those graph neural networks suitable for speedup if trained with the parallel-in-layer algorithm that we mentioned in Chapter 3.

The rest of this section is about PDE-GCN, the architecture we used in our computational experiments. It describes the fundamentals of diffusion equations on manifolds, PDE-GCN-D layer operations (where D stands for diffusion), and details about training PDE-GCN-D on semisupervised node classification learning tasks.

PDE-GCN-D is formulated as a discretization of diffusion PDEs on manifolds. Let M be a manifold and $f : M \to \mathbb{R}^n$ be a vector-valued function that maps points on this manifold M. Continuous differential operators gradient ∇, divergence $\nabla\cdot$, and Laplacian Δ are defined similarly to corresponding operators on functions over $\mathbb{R}^n$. Given these differential operators, the initial value problem for the nonlinear PDE equation is given by

$$\frac{\partial f}{\partial t} = \nabla \cdot K^* \sigma(K \nabla f), \quad f(t=0) = f^{(0)}, \quad t \in [0, T]. \tag{4.1}$$

Here K is a coefficient matrix that is assumed to be time-dependent, K^* is its conjugate transpose, and σ is a non-linear activation function. This PDE in equation (4.1) can be interpreted as an operator on the space of vector-valued functions defined on M. It maps initial value $f^{(0)}$ to the solution value at T, $f^{(T)}$, where both of them can be viewed as time-independent functions, i.e. functions only dependent on points of the manifold.

Similar to the operator interpretation of the PDE in (4.1), graph neural networks can be considered as operators defined on the space of vector-valued functions over graph nodes. Feature values of input data defined for each node can be regarded as an initial value, $\mathbf{f}^{(0)}$, and the GNN model maps this function to output features of its final layer. We first define graph analogs of differential operators and then define the time integration of a PDE on the graph.

Let $G = (V, E)$ be a directed graph with n nodes and m edges; we can consider it as a discretization of a manifold M to a finite space. Let $\mathbf{f} : V \to \mathbb{R}^c$ be a function that maps each node of the graph to a vector of $\mathbb{R}^c$, where c denotes the number of channels or features. We will assume nodes have indices from 0 to $n-1$, and the vector that the function $\mathbf{f}$ maps node i to will be denoted with $\mathbf{f}_i$. We also consider the matrix $\mathbf{f} \in \mathbf{R}^{n\times c}$ that stores the feature vectors for the n nodes in its rows. This means $\mathbf{f}_i$ can be seen as the i^{th} row of matrix $\mathbf{f}$.

In the setting described above, one can define discrete gradient operator $\mathbf{G} \in \mathbb{R}^{m\times n}$, which maps node functions to edge functions, i.e., functions whose domain is the set of nodes to functions whose domain is the set of edges. For a node function, $\mathbf{f}$, and an edge between nodes i and j, the edge function value obtained by the gradient operation is defined by

$$(\mathbf{G}\mathbf{f})_{ij} = \mathbf{W}_{ij}(\mathbf{f}_i - \mathbf{f}_j), \tag{4.2}$$

where the subscripts ij denotes the edge between nodes i and j and $\mathbf{W}_{ij}$ is the weight of an edge between node i and node j. The weights $\mathbf{W}_{ij}$ can be defined in many ways; if $\mathbf{W}_{ij} = d_{ij}^{-1}$, where d_{ij} is the distance between nodes i and j, then the discrete gradient operator becomes a second-order approximation of the true gradient. The choice that is used in our experiments is $\mathbf{W}_{ij} = \gamma_{ij}^{-1}$, where γ_{ij} is the geometric mean of the degrees of nodes i and j. The discrete gradient maps vector functions of nodes, to vector functions of edges. In terms of linear algebra, it maps matrices of size $n \times c$ to matrices of size $m \times c$. Therefore, because the gradient operator acts on each channel individually, it can be represented as an $m \times n$ matrix. The incidence matrix is the relevant matrix for the unweighted discrete gradient, we described this relation in example 2.3.1.

From the definition of discrete gradient, it is possible to define the discrete divergence which maps edge functions to node functions. Discrete divergence is defined in a way that would make sense in terms of inner products between vectors. For an arbitrary graph, if we consider a vector-valued edge function $\mathbf{q} \in \mathbb{R}^{m\times c}$ and a discrete gradient applied to node function $\mathbf{f} \in \mathbb{R}^{n\times c}$, it is possible to define an inner product between them. This inner product should satisfy the property

$$(\mathbf{q}, \mathbf{G}\mathbf{f}) = \mathbf{q}^T\mathbf{G}\mathbf{f} = \mathbf{f}^T\mathbf{G}^T\mathbf{q} \tag{4.3}$$

This property induces a natural definition for discrete divergence as $\nabla\cdot \approx -\mathbf{G}^T$. This definition also resembles when we consider it in terms of dimensions, it can be represented as a matrix with dimension $m \times n$.

Given these discrete differential operator definitions, the propagation rule for a PDE-GCN-D layer is defined as,

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - h\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T. \tag{4.4}$$

It corresponds to an explicit time integration of a diffusion PDE defined over the graph.

In this formula, $\mathbf{f}^{(l)}$ is the output of l^{th} PDE-GCN layer, $h > 0$ is step size, $\mathbf{K}_l$ is a 1×1 trainable convolution operation, and $\mathbf{K}_l^T$ is the transpose of this convolution operation. The differential operators $\mathbf{G}$ and $\mathbf{G^T}$ do message passing between neighboring nodes, and the convolution operators act on individual node/edge vectors and mix values in the channels. The only trainable parameter in this equation is the convolution operator $\mathbf{K}_l$.

In practice, the models use the propagation rule (4.4) as their intermediate layers. These model architectures have two additional parts, the (opening) embedding layer, and the (closing) embedding layer.

The (opening) embedding layer is where the input vertex features $\mathbf{u}_v$ are mapped to the initial feature vectors $\mathbf{f}_0$. Input and output relations are defined by $\mathbf{f}_0 = \mathbf{K}_o\mathbf{u}_v$, where $\mathbf{K}_o$ is a 1×1 trainable convolution operator.

The (closing) embedding layer maps the output of the final PDE-GCN-D layer, $\mathbf{f}^{(L)}$, to the network output, $\mathbf{u}_{out}$. The network output is used for postprocessing operations, like loss computation, or representation learning. Similar to the (opening) embedding layer, it consists of K_c, a 1×1 trainable convolution layer, and the input-output relation is defined by $\mathbf{u}_c = \mathbf{K}_c\mathbf{f}^{(L)}$.

The reason behind our choice of PDE-GCN-D, among other examples, is that it has distinct parameter definitions for each layer, i.e., there are no learnable parameters shared among layers. It is known that PDE-GCN-D can achieve relatively high accuracy when trained with 64 layers. Training a standard GNN with 64 layers is a challenging task due to over-smoothing, so it can be said that the PDE-GCN-D architecture addresses this issue. Training a neural network with many layers is required to obtain speed up with layer-parallel training.

4.2 IMEX Net in Residual Networks Setting

Parallel-in-layer training when performed on multiple levels requires integration with larger timesteps. When explicit integration is performed, this leads to instabilities due to the limited stability regions of explicit integration schemes, like forward Euler in (2.3). One way to obtain stability is by picking an implicit scheme. However, for problems that involve their nonlinearity, their derivation and implementation are not trivial. In this section, we describe an alternative method, called implicit-explicit (IMEX) integration, that results in relatively simple equations for implementation compared to using fully implicit integration.

IMEXnet [17] is a neural network architecture whose main motivation was addressing the limited receptive field problem of convolutional neural networks. It introduces additional terms to the ODE corresponding to ResNet and states a new forward propagation rule inspired by the idea of IMEX time integration of dynamical systems.

IMEX schemes have been studied to integrate spatially discretized time-dependent PDEs with two distinct terms in the spatial derivative part of the PDE[1, 2]. One example of PDEs suitable for which IMEX schemes are commonly used is the advection-diffusion equation. Their use is convenient when the implicit scheme is not feasible for the one term, and explicit time integration of the remaining term is sensitive to h, i.e., the term is stiff. Under such conditions, picking an explicit discretization for the first term, and implicit discretization for the second term leads to the **implicit-explicit (IMEX)** time integration scheme.

IMEXnet can be considered as a variant of a ResNet which differs in the forward propagation formulas. In derivations of IMEXNet formulas, first the ResNet ODE in equation (3.8) is modified by adding and subtracting a term $Lx(t)$. Here, $x(t)$ is multiplied with an invertible matrix and an equivalent ODE (IMEX ODE) is obtained:

$$\frac{dx(t)}{dt} = f(x(t), \theta(t)) + Lx(t) - Lx(t). \tag{4.5}$$

The motivation behind the addition of the L term is to obtain a form in which the IMEX scheme can be utilized rather than explicit Euler integration. IMEXNet does this by discretizing the first two terms on the right-hand side explicitly in time, and the remaining term implicitly in time. The derivation to obtain $x^{(k+1)}$ in terms of $x^{(k)}$ is given below:

$$\frac{x^{(k+1)} - x^{(k)}}{h} = f(x^{(k)}, \theta^{(k)}) + Lx^{(k)} - Lx^{(k+1)}, \tag{4.6}$$

$$x^{(k+1)} - x^{(k)} = hf(x^{(k)}, \theta^{(k)}) + hLx^{(k)} - hLx^{(k+1)}, \tag{4.7}$$

$$x^{(k+1)} = (I + hL)^{-1}(x^{(k)} + hLx^{(k)} + hf(x^{(k)}, \theta^{(k)})), \tag{4.8}$$

$$x^{(k+1)} = x^{(k)} + (I + hL)^{-1}hf(x^{(k)}, \theta^{(k)}). \tag{4.9}$$

In [17], the authors discuss the choice of the L term. For their tasks, they design an operator of the form $L = B^T B$, where B is a group convolution, which leads to a positive definite invertible IMEX term. The authors mention that the applicability of identities involving the Fourier transform is computationally cheaper in inverse calculations. Reasoning about this choice is more relevant to increasing the receptive field of standard convolutional layers, rather than having a stable time integration scheme which will be our concern in this thesis.

In addition to discussing the design decisions on L, the authors of [17] provided a theoretical analysis for the simplified case where $L = \alpha I$, $\alpha > 0$. Their theorem about stability is provided below.

Theorem 4.2.1 (from [17]) *Let J be a given matrix and consider the linear dynamical system $\dot{Y}(t) = JY(t)$. Assume that the eigenvalues of J have non-positive real parts, i.e., can be written as $\lambda = -\lambda_{real} + i\lambda_{imag}$, where $\lambda_{real} \geq 0$. Then, if we choose α such that*

$$|\lambda|^2 + 2\lambda_{real} - \frac{1}{h} \leq \alpha, \tag{4.10}$$

for all λ, the magnification factor between layers in the IMEX method is

$$\left| \frac{1 + h\lambda + h\alpha}{1 + h\alpha} \right| \leq 1, \tag{4.11}$$

for all λ, and the method is stable.

Although this theorem does not cover exactly the case for the numerical stability of ResNet's forward propagation, the matrix J can be considered as the Jacobian of any layer. An exhaustive analysis was not provided because the ResNet ODE is non-autonomous and nonlinear.

4.3 PDE-GCN with IMEX Term

In this section, we discuss the IMEX idea for PDE-GCN-D in (4.4) and describe different choices for the linear operator L.

We start with adding and subtracting a linear term to the continuous PDE in (4.1). This operation yields an equivalent continuous PDE,

$$\frac{\partial f}{\partial t} = \nabla \cdot K^* \sigma(K \nabla f) + \mathcal{L} f - \mathcal{L} f. \tag{4.12}$$

Here $\mathcal{L}$ is nothing but a linear differential operator that acts on the function values at a given time. Example $\mathcal{L}$ choices will be mentioned throughout the text.

Like in the case of ResNet, IMEX discretization for PDE-GCN results in the following derivation:

$$\frac{\mathbf{f}^{(l+1)} - \mathbf{f}^{(l)}}{h} = -\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T + \mathcal{L}\mathbf{f}^{(l)} - \mathcal{L}\mathbf{f}^{(l+1)}, \tag{4.13}$$

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - h\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T + h\mathcal{L}\mathbf{f}^{(l)} - h\mathcal{L}\mathbf{f}^{(l+1)}, \tag{4.14}$$

$$\mathbf{f}^{(l+1)} = (\mathbf{I} + h\mathcal{L})^{-1}(\mathbf{f}^{(l)} - h\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T + \mathcal{L}\mathbf{f}^{(l)}), \tag{4.15}$$

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - (\mathbf{I} + h\mathcal{L})^{-1}h\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T. \tag{4.16}$$

Here $\mathcal{L}\mathbf{f}^{(l)}$ can be written as a matrix multiplication involving two matrices $\mathbf{L}_1 \in \mathbb{R}^{n\times n}$ and $\mathbf{L}_2 \in \mathbb{R}^{c\times c}$, with resulting equality $\mathcal{L}\mathbf{f}^{(l)} = \mathbf{L}_1\mathbf{f}^{(l)}\mathbf{L}_2$.

We are using equation (4.16) in our implementations. One important detail to consider here is that $\mathbf{f}^{(l)}$ is a matrix defined by all feature vectors of nodes produced by the output of layer l, it lies in $n \times c$ dimensional space. It is possible to define $\mathcal{L}$ whose matrix representation has dimensions $(nc) \times (nc)$ where $\mathbf{f}^{(l)}$ is viewed as a vector in $\mathbb{R}^{nc}$. This is in contrast to the ResNet setting, where the linear term L in equation (4.5) has dimension $c \times c$ if $x(t) \in \mathbb{R}^c$.

After obtaining a general IMEX formula for PDE-GCN-D, we investigated various potential IMEX terms which resulted in a feasible forward propagation rule, i.e., terms not involving computationally expensive procedures. The first variant was choosing $\mathbf{L}_1$ as a scalar value multiplied by the identity matrix, expressed as $\mathbf{L}_1 = \alpha_l\mathbf{I}$, and $\mathbf{L}_2 = \mathbf{I}$ We will call it the **shared diagonal** IMEX term since all channels share the same α per layer then (4.16) becomes

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - \frac{h}{1 + h\alpha_l}(\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T). \tag{4.17}$$

Here α_l is the coefficient for the IMEX term in layer l. We use a learnable α term for each layer in some of our experiments.

The second IMEX term we considered is the **channel-wise diagonal** IMEX term. In this IMEX variant, $\mathcal{L}$ is still diagonal with the possibility of having different values for each channel. In this case, $\mathbf{L}_1 = \mathbf{I}$ and $\mathbf{L}_2$ is a diagonal matrix in $\mathbb{R}^{c\times c}$. Having different values on the diagonal for each channel makes sense: when $\mathcal{L}$ is learnable, this can tune the information propagation with a different rate for each channel. The idea of having a different propagation rate per channel in GNNs was mentioned in [10], and its benefit is shown for the over-smoothing problem. Time marching with channel-wise diagonal IMEX term yields

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - h(\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T)(diag(\mathbf{I} + h\mathbf{L}_2))^{-1}. \tag{4.18}$$

Here the IMEX term is constructed according to

$$(\mathbf{L}_2)_{ij} = \begin{cases} 0 & \text{if } i \neq j, \\ \alpha_i & \text{if } i = j \end{cases}, \tag{4.19}$$

After considering these diagonal operators, we also considered IMEX terms which are relatively more expensive to invert. We questioned whether we could remove the nonlinearity in the diffusion equation and obtain an IMEX term that has a diffusive behavior. This idea leads to a **linear-diffusion** IMEX term, given by:

$$\mathcal{L}\mathbf{f}^{(l)} = \mathbf{G}^T\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}\mathbf{K}^T \approx \mathbf{P}\mathbf{f}^{(\mathbf{l})}\mathbf{W}. \tag{4.20}$$

Here multiplication with $\mathbf{G}^T\mathbf{G}$ has an effect of message passing between neighboring nodes, and $\mathbf{K}\mathbf{K}^T$, where $\mathbf{K} \in \mathbb{R}^{c\times c}$ is a 1×1 convolution (optionally trainable), mixes values in each channel. Another way to derive a similar formula is to neglect the nonlinearity in the GCN formula in (2.26) so that $\mathbf{P}$ in (4.20) is a graph Laplacian and $\mathbf{W}$ is a trainable weight matrix.

Although the operations in (4.20) are linear, its matrix representation $\mathcal{L}$ has a dimension $nc \times nc$, where n is node count, and c is the number of channels for each feature vector. Inverting the linear operator whose terms involve this operator introduces a challenge regarding the memory and computation required. To obtain a compact matrix representation, we replaced the convolution operation $\mathbf{K}$ with a positive constant k, resulting in a simpler IMEX term:

$$\mathcal{L}\mathbf{f}^{(l)} = \mathbf{G}^T\mathbf{G}\mathbf{f}^{(l)}k^2, \tag{4.21}$$

Because k^2 is constant, we are free to change its position in the multiplication and obtain the formula (4.22):

$$\mathcal{L}\mathbf{f}^{(l)} = k^2\mathbf{G}^T\mathbf{G}\mathbf{f}^{(l)}. \tag{4.22}$$

Picking the formula (4.22) as the IMEX term, and plugging it into (4.16) leads to the final linear-diffusion formula that is used in our implementations below:

$$\mathbf{f}^{(l+1)} = \mathbf{f}^{(l)} - (\mathbf{I} + hk^2\mathbf{G}^T\mathbf{G})^{-1}(h\mathbf{G}^T\sigma(\mathbf{G}\mathbf{f}^{(l)}\mathbf{K}_l)\mathbf{K}_l^T). \tag{4.23}$$

Because $\mathbf{G}^T\mathbf{G}$ is symmetric positive definite, (4.23) can be computed efficiently by a few iterations of a conjugate gradient solver.

The IMEX terms mentioned in this chapter are implemented and their performance in terms of stability and accuracy in parallel-in-layer training is evaluated in the following chapter.

Chapter 5

Experiments

This chapter includes the results from the experiments we performed and their discussion. The chapter is divided into three parts. The first part includes experiments that are done for PDE-GCN without employing MGRIT. We assessed the performance of different IMEX terms. After the evaluation of IMEX scheme performance on a single level, experiments are presented for MGRIT-trained networks with different MGRIT parameters. In the final part, results from experiments with multiple processors are presented to evaluate layer-parallelism and our approach's efficiency in terms of speed-up and accuracy.

Before performing the experiments mentioned in the following sections, the hyperparameters of PDE-GCN [8] were tuned with the help of code provided as supplementary material of that paper. The task we consider is semi-supervised node classification on the CORA dataset; we mentioned details about this task in subsection 2.3.2.

Throughout all experiments, PyTorch [28], PyTorch Geometric [14], and TorchBraid [7] were the main libraries that we used.

5.1 Experiments for Stability

The requirement for achieving stability on a coarse level in MGRIT is to have a propagation scheme that allows the network to take relatively large time steps. To see whether the IMEX schemes make a difference in the stable step size range for PDE-GCN, we did experiments with the IMEX terms we introduced with coefficient choices and large stepsize h. We took the stepsize value which we obtained by hyperparameter tuning for 64 layers, and trained different architectures by changing the stepsize and the number of layers while keeping their product constant; This can be considered as keeping the final time constant for each grid level. Below we provide plots from these experiments.

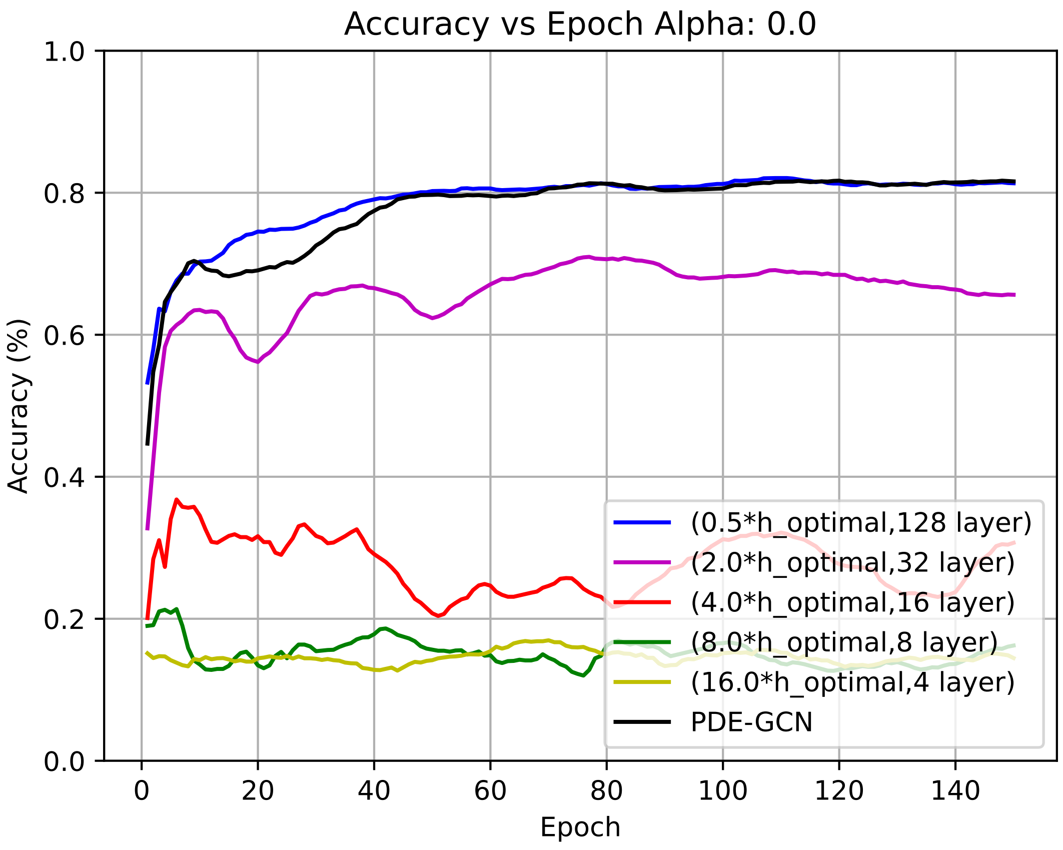

Figure 5.1: Accuracy comparison for different step sizes and number of layers with constant $\alpha = 0$, i.e., without any IMEX term involved. Each curve is obtained by averaging the accuracy values from three runs

In Figures 5.1, 5.2, and 5.3, the black curve stands for the original PDE-GCN where no IMEX term is involved. The other lines stand for architectures equipped with different step sizes and layer numbers. By looking at the three figures collectively, it can be stated that the original PDE-GCN does not allow training with large stepsizes. This is potentially due to the stability of the initial weight configuration. This argument can explain why there are vertical gaps between the curves in Figure 5.1. Introducing the IMEX term to training

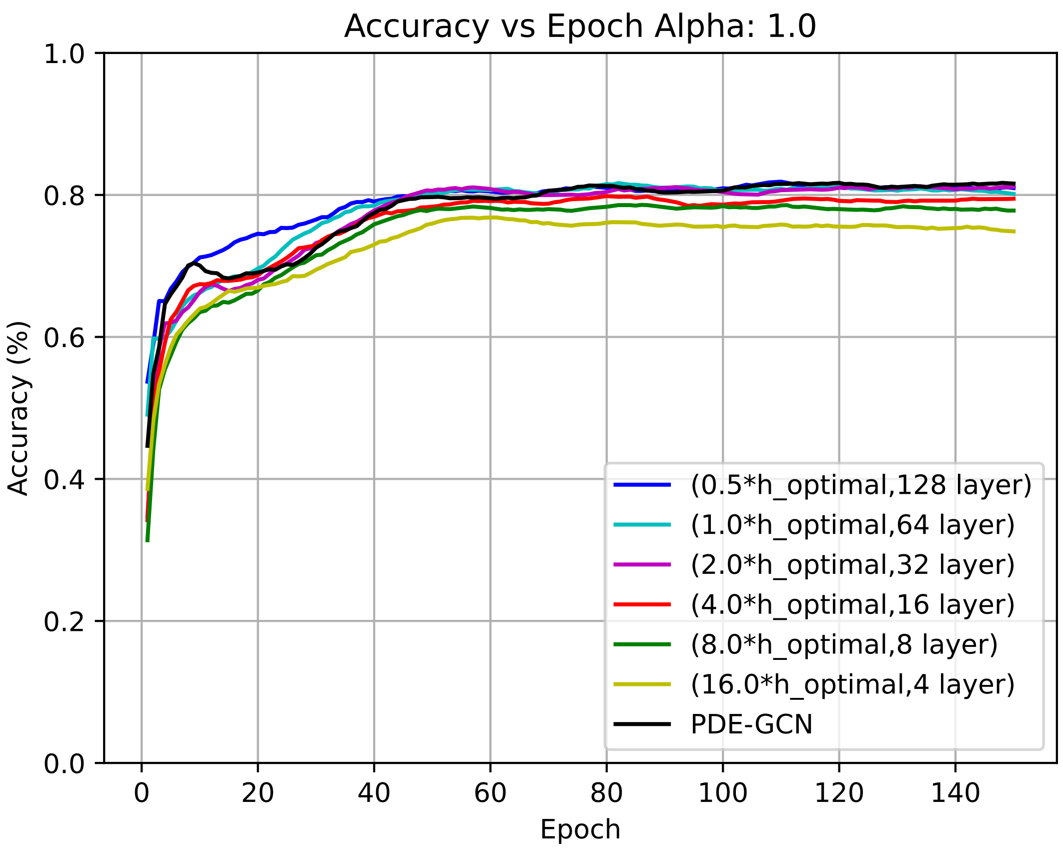

Figure 5.2: Accuracy comparison for different (step size, layer number) pairs with $\alpha = 1$, i.e. , IMEX terms $\mathbf{L}_1 = \mathbf{I}$, $\mathbf{L}_2 = \mathbf{I}$

helps the networks achieve comparable accuracies with the original PDE-GCN. This fact can be seen in Figure 5.2, where the training curves are close to each other. Having a relatively large weight for the IMEX term results in accuracy loss as seen in Figure 5.3: differences between training curves are increased compared to the $\alpha = 1$ case. This is potentially due to the IMEX term introducing diffusive errors. From the numerical PDE point of view, the IMEX term introduces a discretization error, which makes the numerical solution less accurate compared to actual dynamics.

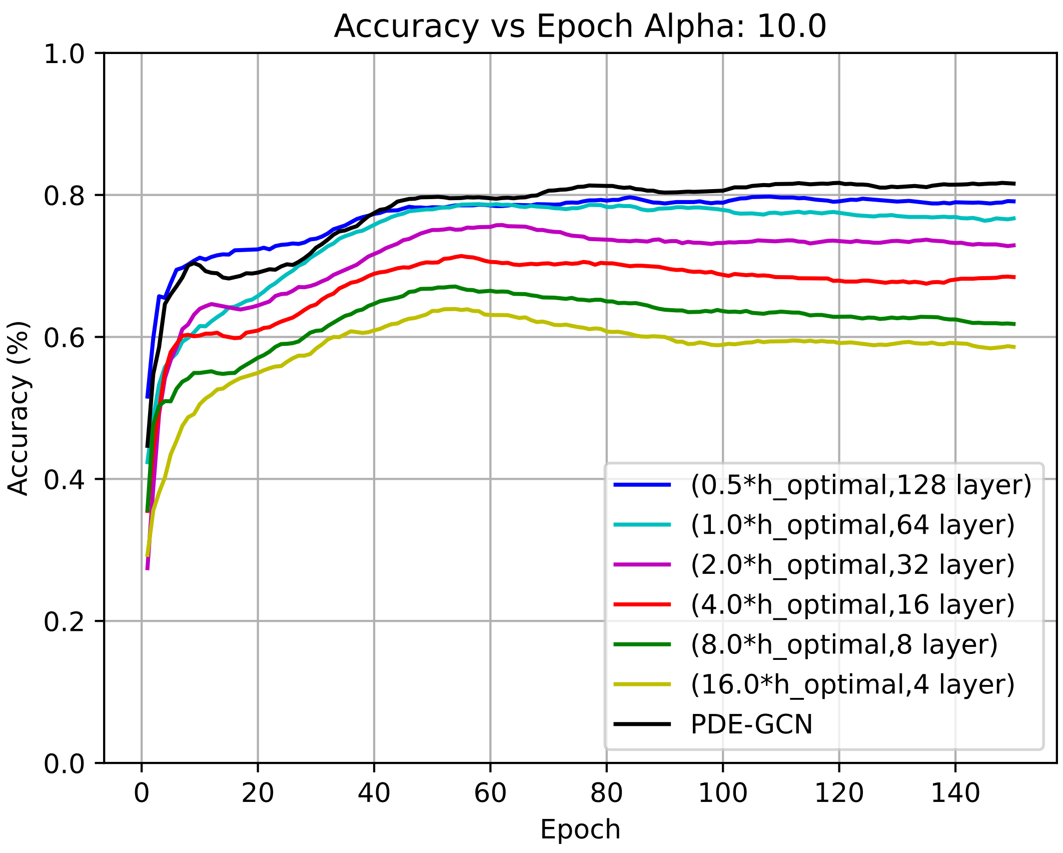

Figure 5.3: Accuracy comparison for different (step size, layer number) pairs with $\alpha = 10$, i.e., IMEX terms $\mathbf{L}_1 = 10\mathbf{I}$, $\mathbf{L}_2 = \mathbf{I}$

In addition to comparing each pair in one plot for different α values, we generated plots for comparing different α in the same plots for specific pairs. Two plots comparing α values are given in Figures 5.4, and 5.5. Our reason for generating those plots is to have a reference for MGRIT experiments because MGRIT propagation on coarse levels corresponds to solving the same PDE with different step sizes and step number parameters. We see that for each of the stepsize-depth pairs, the best performing α value is 1.0.

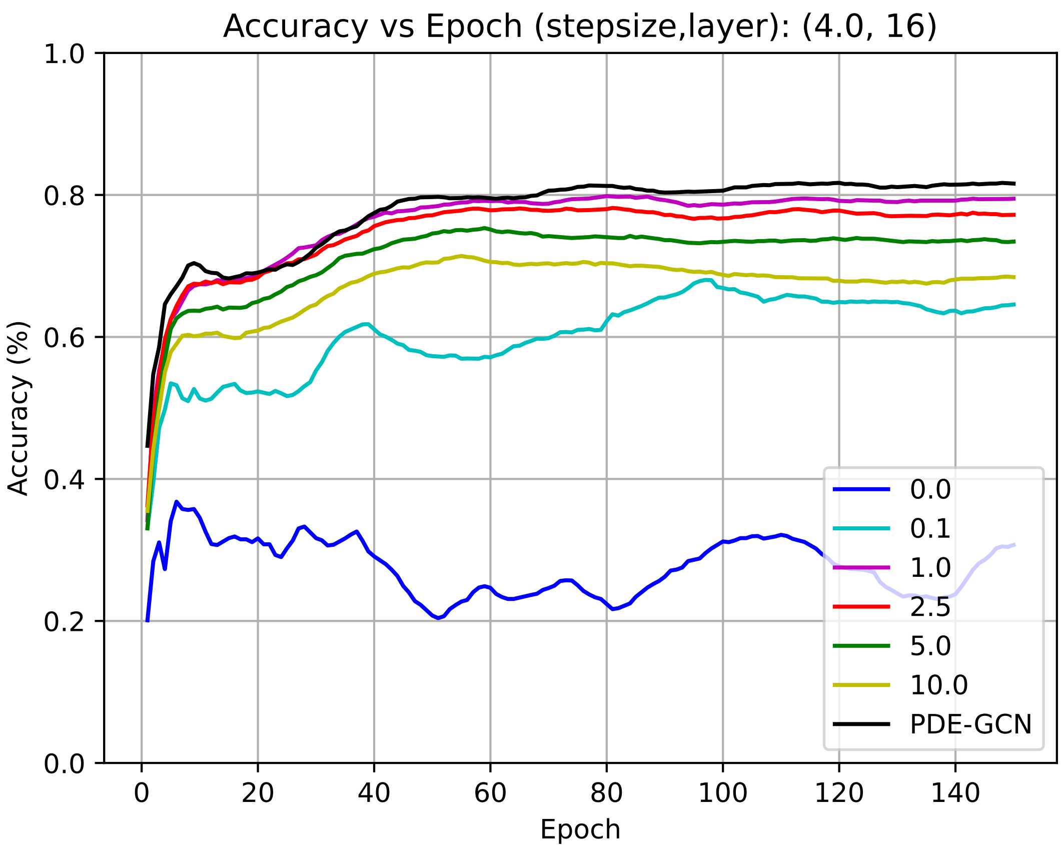

Figure 5.4: Training curves for 16 layer network, where the tuned ”optimal” stepsize multiplied by 4. The best-performing alpha is 1.0.

Next, we implemented the IMEX scheme and ran experiments to see how efficient variants of the IMEX scheme are in terms of addressing instabilities while maintaining accuracy. Within this experiment, we evaluated whether there is a potential advantage to having a learnable stepsize. We did not see any significant advantages between choosing among shared alpha, channel-wise diagonal, and GCN-1 IMEX terms, therefore we did not put any plots regarding them.

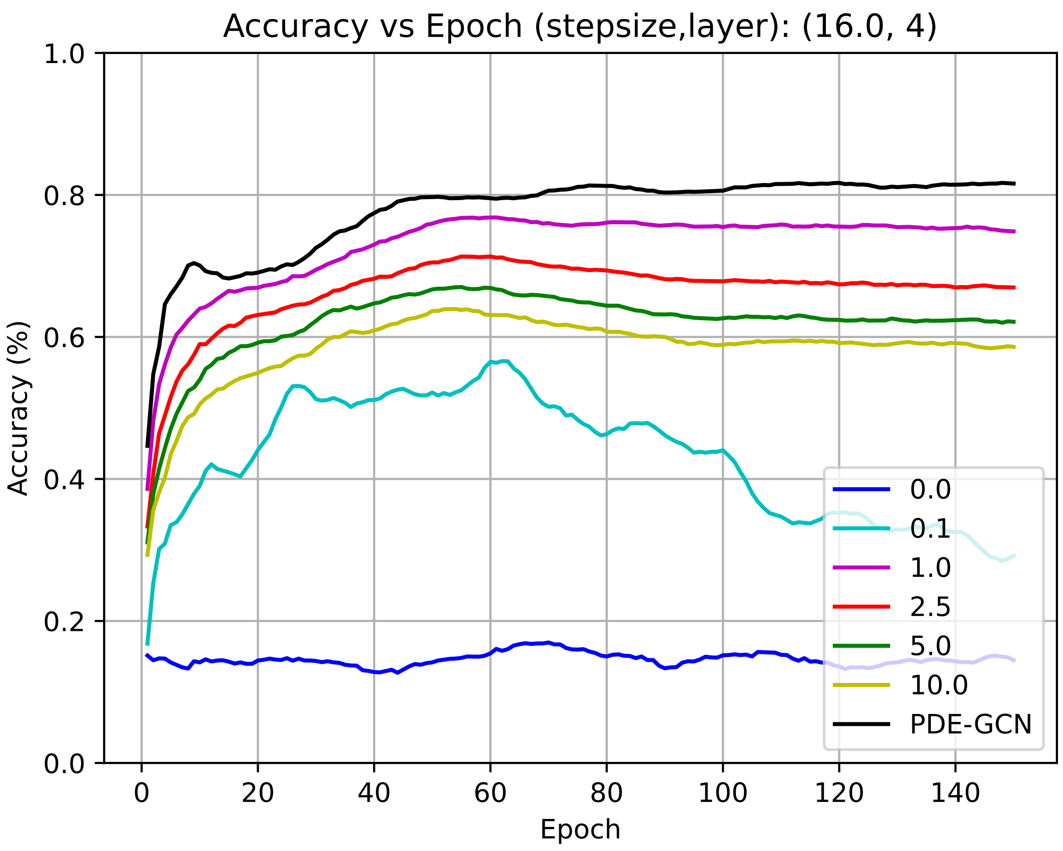

Figure 5.5: Training curves for 4 layer network, where the tuned stepsize is multiplied by 16. The best-performing initial alpha is 1.0.

5.2 Experiments with MGRIT Propagation

After testing IMEX with one-level propagation and comparing outputs of different alpha values, and IMEX terms in one-level PDE-GCN, we performed MGRIT training experiments on multiple levels. Our plots are limited to an architecture with 64 layers and a constant shared diagonal IMEX term.

Figures 5.6, 5.7, 5.8 and 5.9 demonstrate the performance of different α values when training is equipped with a different number of levels and different coarsening factor choices. Each figure's individual commentary is provided in the caption. It can be seen that the

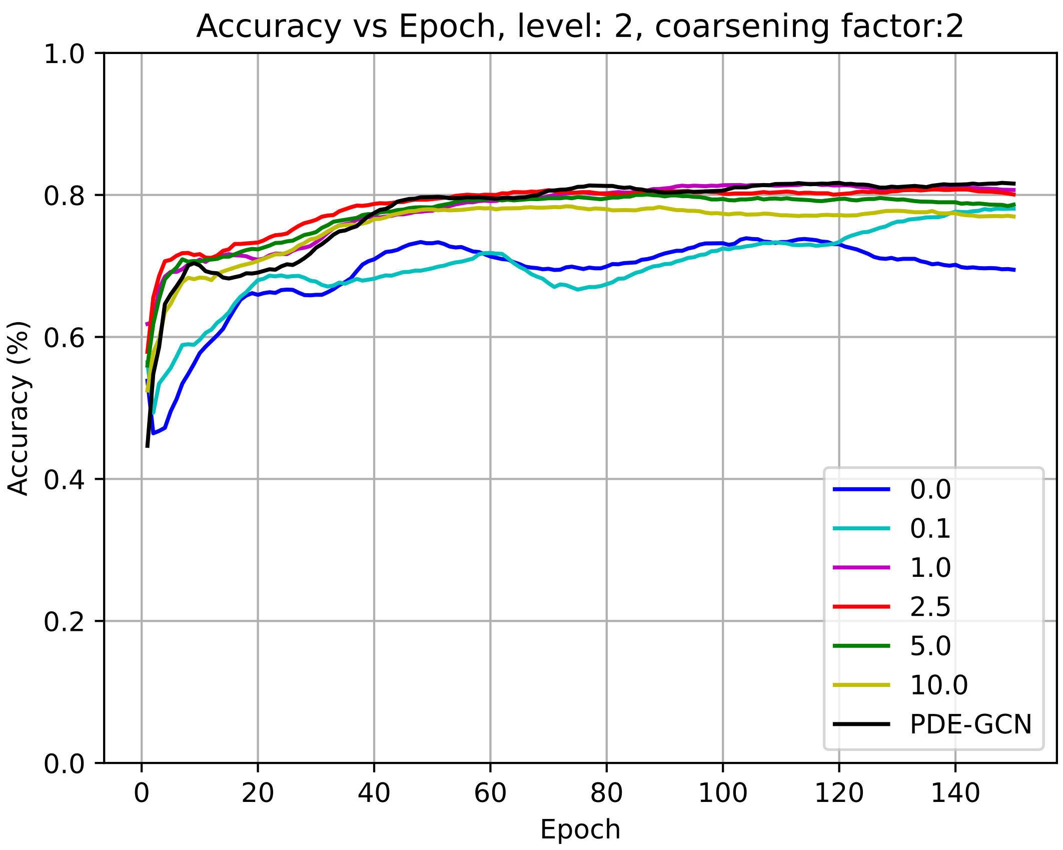

Figure 5.6: Networks with different alpha values are trained with MGRIT using 2 levels and coarsening factor 2. Except for the case without IMEX ,i.e., $\alpha = 0$, all training curves end close to each other, and the best-performing α value is 1.0.

magnitude of the best IMEX term coefficient is positively correlated with the number of levels and the coarsening factor. An increase in both parameters results in a requirement with time marching with larger time step values.

Another design decision apart from the choice of alpha for the IMEX terms is picking IMEX terms with different alpha for various levels. To test whether this design choice has an advantage, we trained networks that propagate information without IMEX on the fine level but introduce IMEX with best-performing coefficients depending on the level and coarsening factor. Results corresponding to this experiment are provided in Figure 5.10,

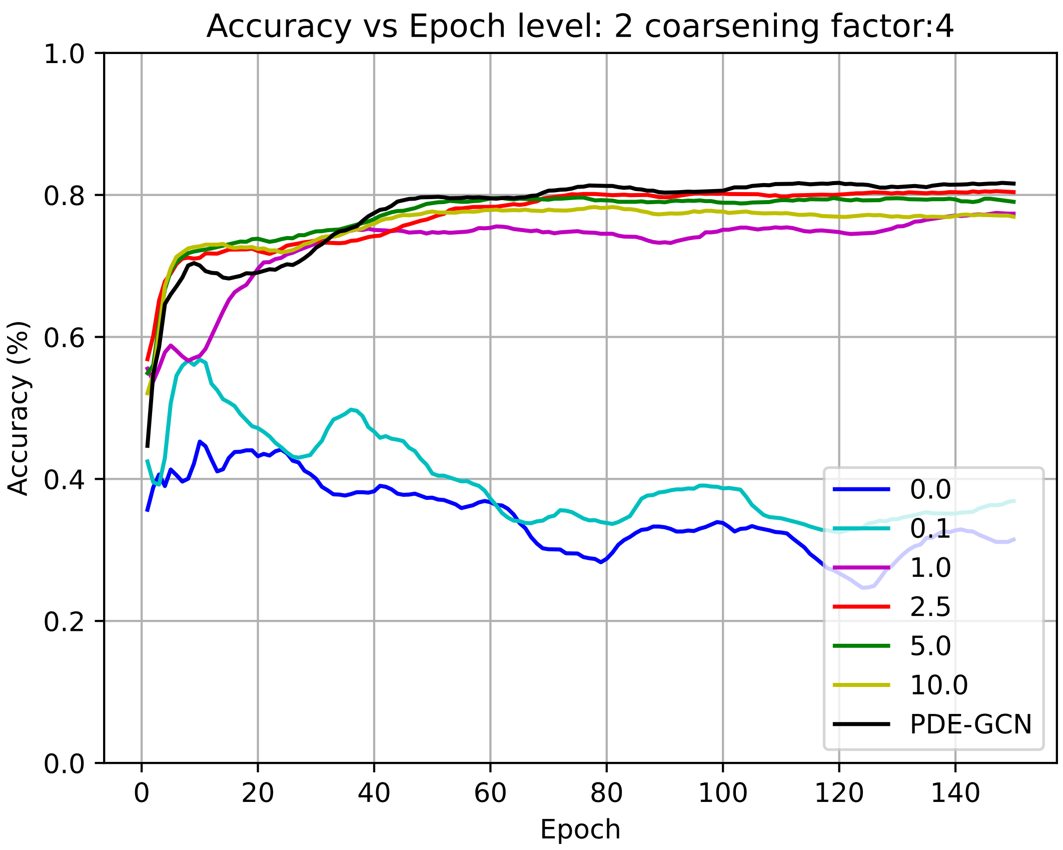

Figure 5.7: Networks with different alpha values are trained with MGRIT using 2 levels and coarsening factor 4. Except for the case without IMEX i.e. $\alpha = 0$ and $\alpha = 0.1$, all training curves end close to each other, and the best performing α value is 2.5.

where solid lines are from training that uses different IMEX parameters between the finest and coarser levels, and the lines with markers are from runs in which IMEX parameters shared among levels. Solid lines are generally positioned below the marker lines, so we can say this design choice is not helping our case. This can potentially be different if more MGRIT iterations are performed. In our experiments, however, we always use 2 iterations for forward and 1 iteration for backward propagation calculations.

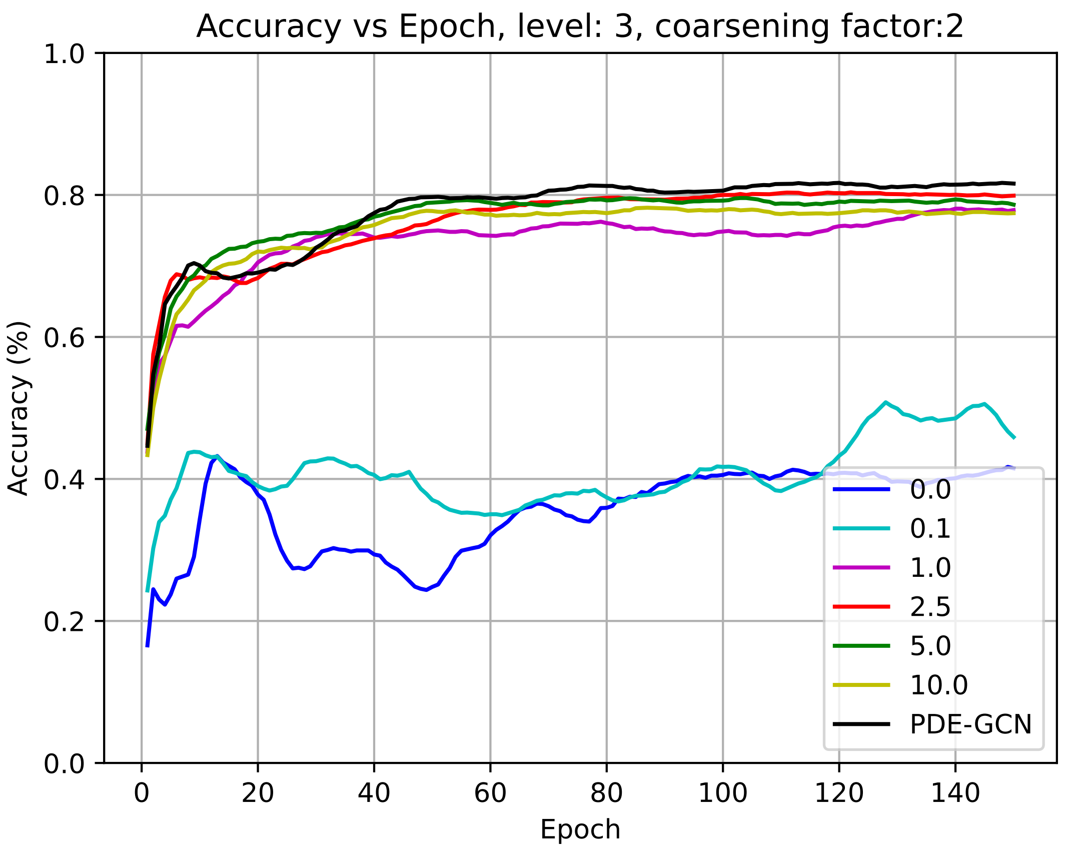

Figure 5.8: Networks with different alpha values are trained with MGRIT using 3 levels and coarsening factor 2. Except for the case without IMEX i.e., $\alpha = 0$ and $\alpha = 0.1$, all training curves end close to each other, and the best performing α value is 2.5.

5.3 Experiments with Multiple Processors

After evaluating the performance of IMEX using single and multiple levels on one processor, we perform parallel tests with 150 epochs using the best alpha value for each MGRIT parameter combination and having alpha in all levels. Our plots and commentary regarding the performance of parallel-in-layer training on multiple processors are divided into two, a comparison of training times vs accuracy with 64-layered networks, and experiments with 256-layered networks.

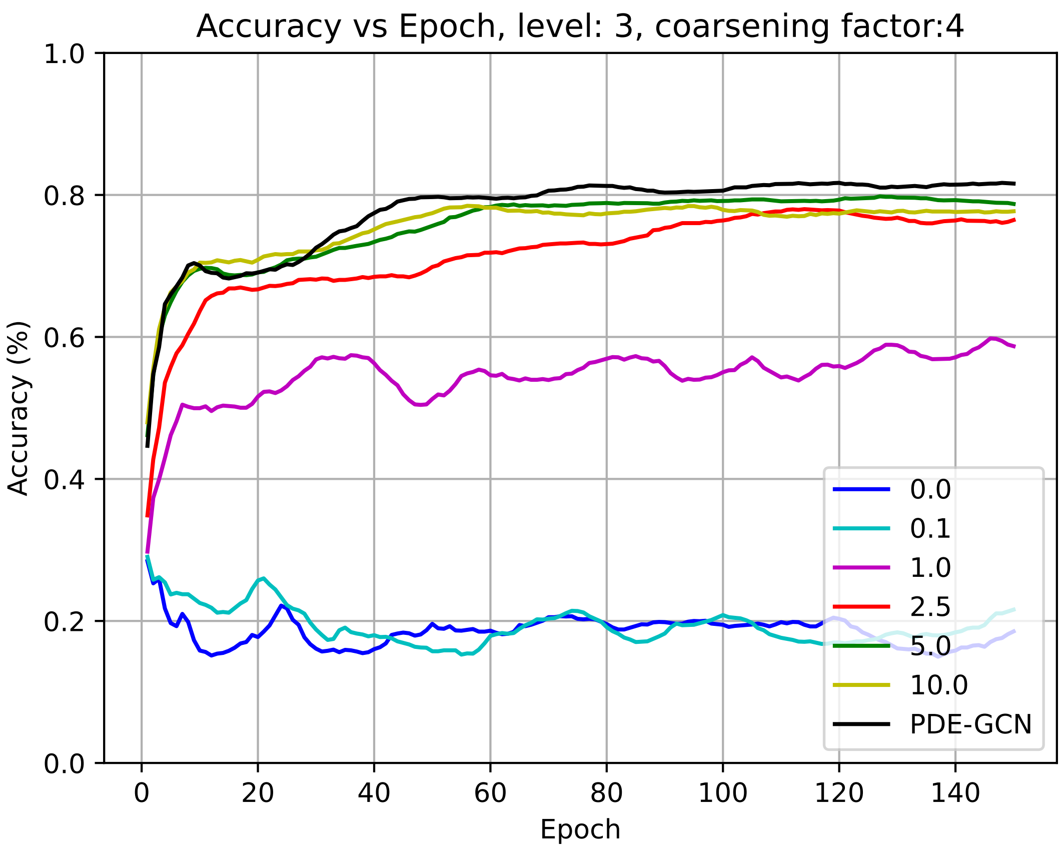

Figure 5.9: Networks with different alpha values are trained with MGRIT using 3 levels and coarsening factor 4. Except for the case without IMEX i.e. $\alpha = 0$, and; cases $\alpha = 0.1$ and $\alpha = 1.0$ all training curves end close to each other, and the best performing α value is 5.0.

The results from the parallel training of a 64-layer network are shown in Figure 5.11, and the results from the parallel training of a 256-layer network are given in Figure 5.12 and Figure 5.13.

For 64-layer network experiments, we can see that the training curve with 2 levels, and coarsening factor parameter 2, achieved comparable accuracy with the serial training in a shorter time, however, this choice is insufficient to have a speed-up for a fixed number of epochs. The training time of the network equipped with 3 levels, and coarsening factor 4

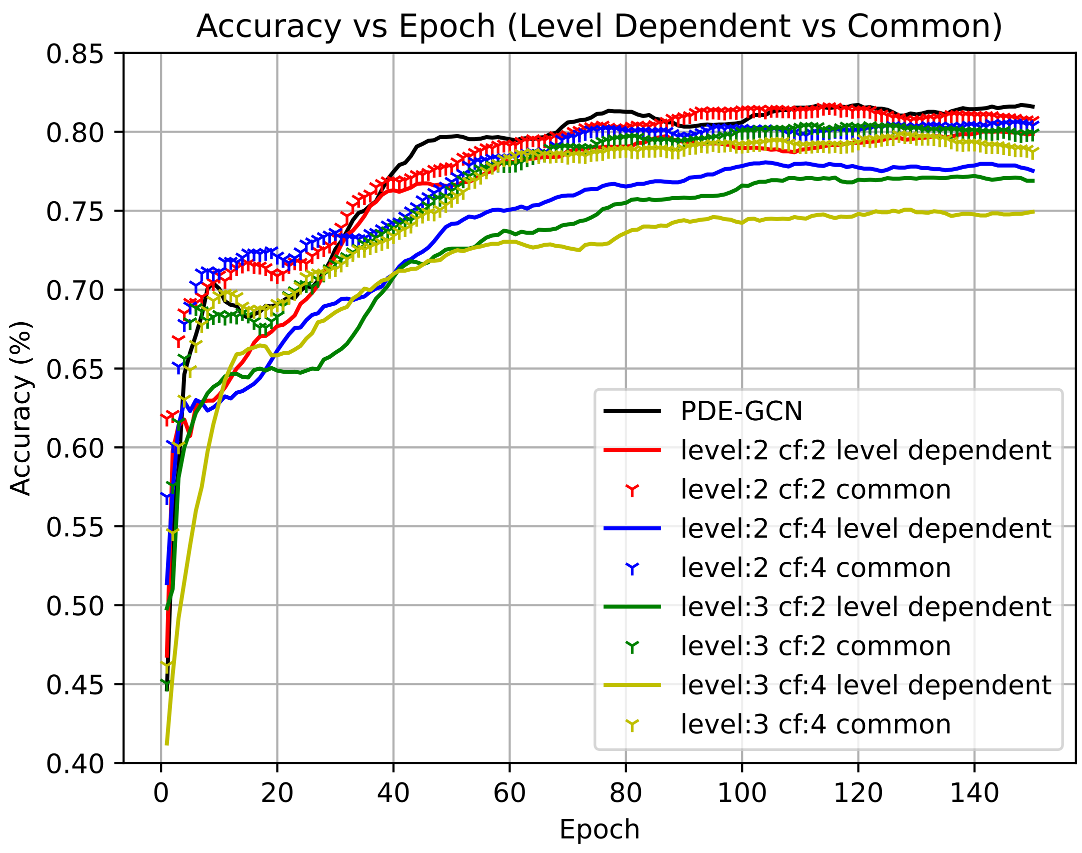

Figure 5.10: Training curves to compare different IMEX parameters on different levels (solid lines), and using the same choice for α on all levels (with markers).

(cyan curve) is the smallest. However, larger MGRIT parameters require larger α values, which introduces a loss in accuracy and prevents the network from attaining final accuracy comparable with the serial training (black curve).

For 256-layer network experiments, the network equipped with 3 levels, and coarsening factor 4 (cyan curve) produces a training curve in which speed-up is apparent. When all the networks are trained for a duration of the same length, it can be a cyan curve that belongs to parallel training with 3 levels, and coarsening factor 4 has achieved a steep section in a shorter time, and final accuracy values are comparable.

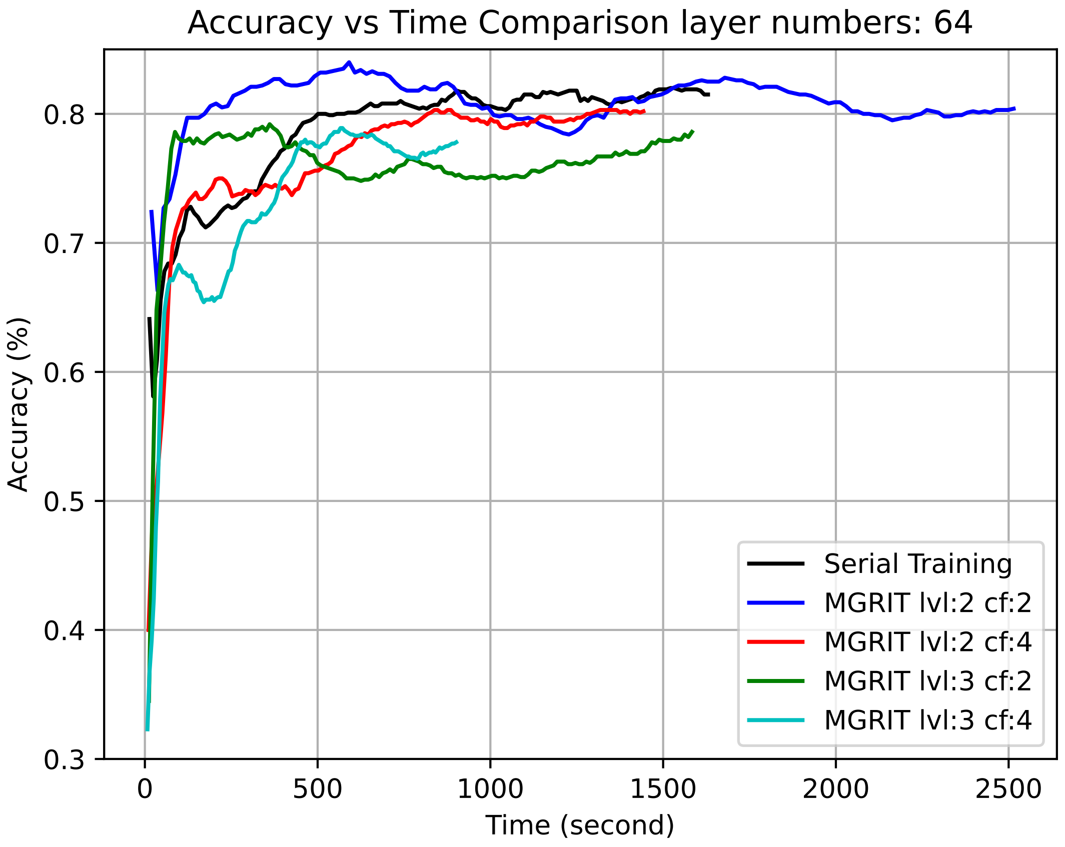

Figure 5.11: Training curves for a different number of levels and different coarsening factor parameters when a 64-layer network is trained with 32 processors. The black curve is serial PDE-GCN without IMEX. The best-performing IMEX parameter choice is employed for each pair. This experiment is done with 32 MPI processes. We used the same alpha value in fine and coarser levels. For the blue curve $\alpha = 1.0$, for the red and green curves $\alpha = 2.5$, and for the cyan curve $\alpha = 5.0$ values are used.

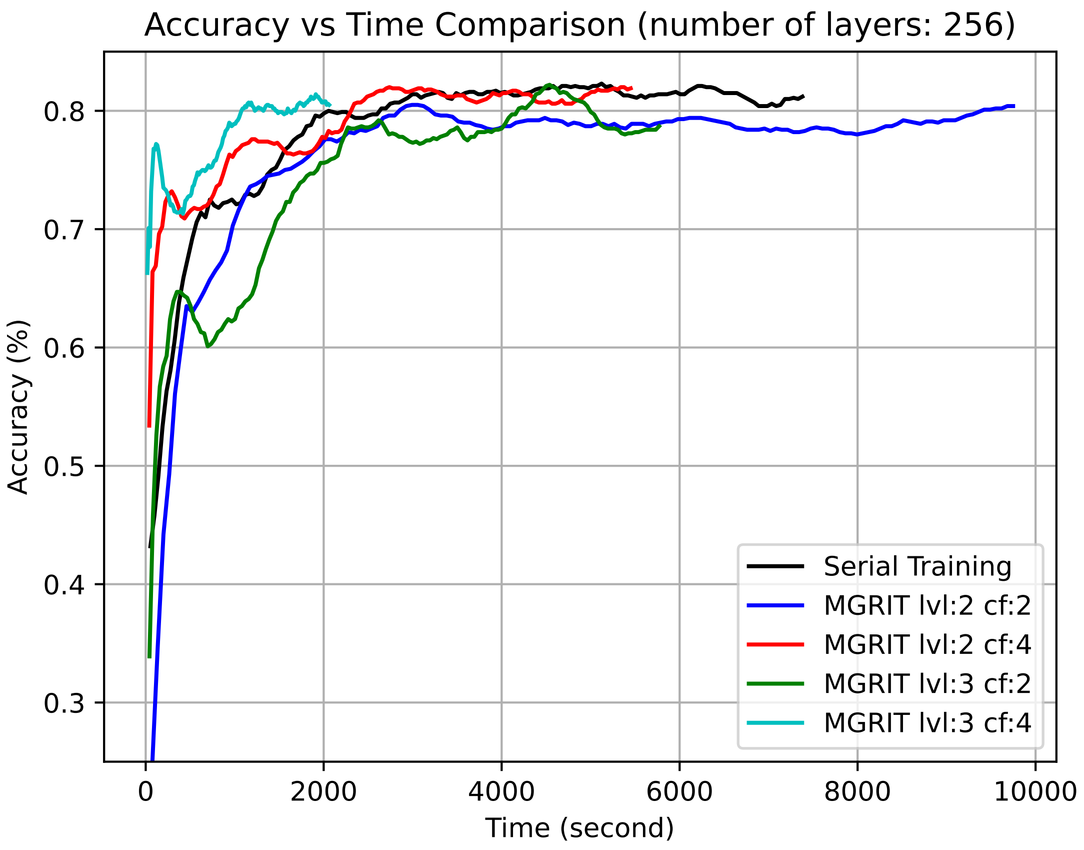

Figure 5.12: Training curves for 256 layer network experiments. Each curve is from training with different MGRIT parameter pairs. We can state that network with 3 levels, and coarsening factor 4, had completed 150 epochs around 3.5 times earlier (around 2000 seconds) than the serial-trained network (around 7000 seconds).The number of processors used is 64.

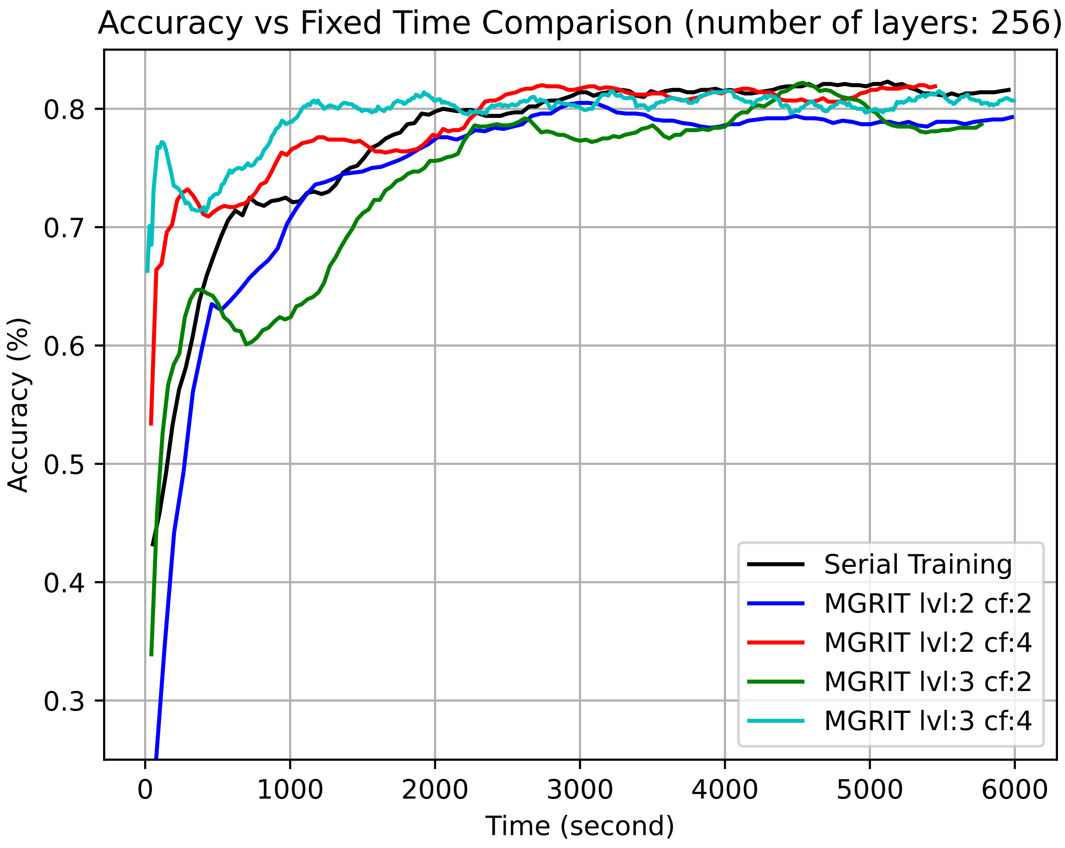

Figure 5.13: In this experiment, we fixed the training end time to compare training curves. It can be seen the parallel-trained network with a cyan curve reached the peak accuracy in a shorter time when compared with the serial-trained network. Achieving comparable accuracies for serial training in a shorter time is a result of the IMEX modification that we did. The number of processors used is 64.

Chapter 6

Conclusion and Discussion

This work explores parallel-in-layer training of graph neural networks using MGRIT, potential solutions to stability problems on coarse grids, and experimental results that evaluate the effectiveness of those potential solutions. In this chapter, we summarize our comments about the experiments and mention potential future work that can be relevant to this thesis.

6.1 Numerical Results

We can evaluate the plots in Chapter 6 collectively. It can be stated that having stability at coarser levels is critical for achieving competitive performance with parallel-in-layer training. The idea of introducing an additive term, with IMEX integration, improves the performance of parallel-in-layer training in terms of accuracy and makes training networks closer in accuracy to serial training.

The choice of the IMEX parameter α is also an important factor. Introducing IMEX terms with small quantities contributions to achieving coarse propagation stability likely to be minimal, if the IMEX term introduced dominates the original network equation, this might result in a small final accuracy.

Experimental results also show that for 2 forward and 1 backward iteration, using IMEX only on coarser levels is not achieveing competitive accuracy compared to choosing same α on all levels.

6.2 Potential Future Work

Extensions of the work performed within this thesis are possible in several directions

Spatial coarsening on graph data can be a direction to explore. Spatial coarsening is a technique to accelerate numerical PDE algorithms that can be coupled with time parallelization. New techniques inspired by PDEs can be introduced to graph machine learning, which might accelerate graph neural network training.

Theoretical justification for the optimal IMEX parameter α, given the level, and the coarsening factor, can be another potential direction of interest, where numerical PDE and parallel-in-time knowledge can be useful.

References

[1] Uri M Ascher, Steven J Ruuth, and Raymond J Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[2] Uri M Ascher, Steven J Ruuth, and Brian TR Wetton. Implicit-explicit methods for time-dependent partial differential equations. *SIAM Journal on Numerical Analysis*, 32(3):797–823, 1995.

[3] Ben Chamberlain, James Rowbottom, Maria I Gorinova, Michael Bronstein, Stefan Webb, and Emanuele Rossi. Grand: Graph neural diffusion. In *International Conference on Machine Learning*, pages 1407–1418. PMLR, 2021.

[4] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[5] Jeongwhan Choi, Seoyoung Hong, Noseong Park, and Sung-Bae Cho. Gread: Graph neural reaction-diffusion networks. In *International Conference on Machine Learning*, pages 5722–5747. PMLR, 2023.

[6] Richard Courant, Kurt Friedrichs, and Hans Lewy. On the partial difference equations of mathematical physics. *IBM journal of Research and Development*, 11(2):215–234, 1967.

[7] Eric Cyr, Gordon Moon, Jacob Schroder, Stephanie Guenther, University of New Mexico, Lawrence Livermore National Laboratory, and USDOE. Torchbraid, version 0.1, 6 2020.

[8] Moshe Eliasof, Eldad Haber, and Eran Treister. PDE-GCN: Novel architectures for graph neural networks motivated by partial differential equations. In A. Beygelz-

imer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[9] Moshe Eliasof, Eldad Haber, and Eran Treister. Feature transportation improves graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 11874–11882, 2024.

[10] Moshe Eliasof, Lars Ruthotto, and Eran Treister. Improving graph neural networks with learnable propagation operators. In *International Conference on Machine Learning*, pages 9224–9245. PMLR, 2023.

[11] Leonhard Euler. *Institutiones calculi integralis*, volume 4. Academia Imperialis Scientiarum, 1794.

[12] Robert D Falgout, Stephanie Friedhoff, Tz V Kolev, Scott P MacLachlan, and Jacob B Schroder. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing*, 36(6):C635–C661, 2014.

[13] Erwin Fehlberg. Classical fourth-and lower order runge-kutta formulas with stepsize control and their application to heat transfer problems. *Computing*, 6:61–71, 1970.

[14] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[15] Martin J Gander. 50 years of time parallel time integration. In *Multiple Shooting and Time Domain Decomposition Methods: MuS-TDD, Heidelberg, May 6-8, 2013*, pages 69–113. Springer, 2015.

[16] Stefanie Gunther, Lars Ruthotto, Jacob B Schroder, Eric C Cyr, and Nicolas R Gauger. Layer-parallel training of deep residual neural networks. *SIAM Journal on Mathematics of Data Science*, 2(1):1–23, 2020.

[17] Eldad Haber, Keegan Lensink, Eran Treister, and Lars Ruthotto. Imexnet a forward stable deep neural network. In *International Conference on Machine Learning*, pages 2525–2534. PMLR, 2019.

[18] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse problems*, 34(1):014004, 2017.

[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[20] Andi Han, Dai Shi, Lequan Lin, and Junbin Gao. From continuous dynamics to graph neural networks: Neural diffusion and beyond. *arXiv preprint arXiv:2310.10121*, 2023.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[23] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[25] Euhyun Moon and Eric C Cyr. Parallel training of GRU networks with a multi-grid solver for long sequences. In *International Conference on Learning Representations*, 2022.

[26] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k** 2). *Doklady Akademii Nauk SSSR*, 269(3):543, 1983.

[27] Keiron O'shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[29] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on pattern analysis and machine intelligence*, 12(7):629–639, 1990.

[30] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

[31] T Konstantin Rusch, Ben Chamberlain, James Rowbottom, Siddhartha Mishra, and Michael Bronstein. Graph-coupled oscillator networks. In *International Conference on Machine Learning*, pages 18888–18909. PMLR, 2022.

[32] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[33] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[34] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.

[35] Tijmen Tieleman. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26, 2012.

[36] Vladimir Vapnik. *The nature of statistical learning theory.* Springer science & business media, 2013.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[38] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[39] Ee Weinan. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 1(5):1–11, 2017.