UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

SIMPLE AND EFFICIENT GRAPH NEURAL NETWORKS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS DE LA INGENIERÍA, MENCIÓN ELÉCTRICA

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

HO JIN KANG KIM

PROFESOR GUÍA:
Felipe Tobar Henríquez
PROFESOR CO-GUÍA:
Jorge Pérez Rojas

MIEMBROS DE LA COMISIÓN:
Hans Löbel Díaz
Jorge Silva Sánchez

Este trabajo ha sido parcialmente financiado por:
Beca de Magíster Nacional ANID

SANTIAGO DE CHILE
2022

RESUMEN DE LA TESIS PARA OPTAR
AL GRADO DE MAGÍSTER EN CIENCIAS
DE LA INGENIERÍA, MENCIÓN ELÉCTRICA
Y MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO
POR: HO JIN KANG KIM
FECHA: 2022
PROF. GUÍA: FELIPE TOBAR HENRÍQUEZ

REDES NEURONALES DE GRAFOS SIMPLES Y EFICIENTES

El éxito de los modelos de aprendizaje profundo en datos Euclideanos es innegable. Esto, en conjunto con la prevalencia de datos no Euclideanos ha resultado en una importante alza en popularidad de modelos de aprendizaje profundo para este tipo de datos. En particular, las redes neuronales de grafos (GNN) se han vuelto uno de los principales modelos de aprendizaje automático para grafos. Debido a su gran popularidad, se han utilizado en aplicaciones relacionadas a redes de citaciones, redes sociales y estructuras moleculares. La popularidad y rápido desarrollo de GNNs ha resultado en modelos cada vez más complejos, que utilizan una gran variedad de trucos para alcanzar el estado del arte. Debido al enfoque que se le ha dado a modelos complejos, los investigadores no han estudiado GNNs más simples en detalle. En el siguiente trabajo proponemos GNNs más simples que: i) logran resultados comparables al estado del arte en varios conjuntos de datos y tareas; ii) requieren un menor número de cómputos que modelos comprables. Los resultados muestran que los componentes básicos de las GNNs se deberían estudiar en mayor detalle para entender qué componentes contribuyen a su rendimiento. Los resultados invitan a enfocar más investigación de GNNs en la simplicidad de los modelos, buscando entender sus componentes, en vez de agregar complejidad a éstos para obtener mejoras marginales de rendimiento.

RESUMEN DE LA TESIS PARA OPTAR
AL GRADO DE MAGÍSTER EN CIENCIAS
DE LA INGENIERÍA, MENCIÓN ELÉCTRICA
Y MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELÉCTRICO
POR: HO JIN KANG KIM
FECHA: 2022
PROF. GUÍA: FELIPE TOBAR HENRÍQUEZ

SIMPLE AND EFFICIENT GRAPH NEURAL NETWORKS

The undeniable success of deep learning models in Euclidean data, combined with the recent prevalence of non-Euclidean data has resulted in a surge in popularity of deep learning models for this type of data. In particular, graph neural networks (GNN) have gained significant popularity for graph-structured data, finding use in a variety of areas such as citation networks, social networks and molecules. The hasty development of GNNs has resulted in increasingly complex models that use a variety of tricks to obtain state-of-the-art results. Due to their infatuation with more complex models, researchers have glossed over simpler models, both in terms of number of parameters and computational complexity. In the following work, we propose simpler GNN models that: i) achieve comparable performance to state-of-the-art models in a variety of benchmarks; ii) require less computation than comparable models. The results invite more GNN research to focus on simplicity and understanding of the models rather than adding complexity for marginal gains.

Don't panic

Agradecimientos

A todos los profesores que me brindaron un sinfín de enseñanzas durante este proceso. A mis profesores guía por la orientación que me brindaron durante estos años. Al Dr. Jorge Pérez por todas las enseñanzas, tanto en lo académico como en lo personal, que me han llevado a crecer durante estos últimos años. Al Dr. César Azurdia por introducirme al mundo de la academia. Por su buena voluntad y simpatía durante mis primeros años de investigación.

A mi familia por aguantarme durante todos estos años. Por todo el cariño incondicional que me han brindado. Por siempre brindarme el apoyo y felicidad que necesito. A mi padre por todos los sacrificios que ha hecho para que pueda estar aquí. Por siempre ponernos primero y hacerlo siempre con una sonrisa en el rostro. A mi madre por volverme la persona que soy hoy. Por ser la persona que más me ha enseñado en la vida. Por enseñarme a ser amable con las personas, a pensar por mí mismo y a empujarme a alcanzar lo que puedo. A mi hermana, simplemente por ser la mejor hermana que alguien podría desear.

A todos los compañeros con los que interactué durante mi tiempo en la Universidad. Aprendí mucho de cada uno de ustedes. Les deseo todo el éxito del mundo. A Javier por siempre ser un gran amigo. Por aguantar mi desorden y caprichos durante todos estos años de Universidad. A Jou-Hui por enseñarme tanto en tan poco tiempo. Por siempre mostrarme una perspectiva nueva de las cosas. Por todos los días que sufrimos investigando juntos. Por ser mi modelo a seguir en muchos aspectos de la vida. A Eduardo por su compañía durante todos estos años. Por todos esos trabajos en los que sufrimos juntos y toda la felicidad que compartimos al terminarlos. Por siempre haber estado ahí durante todo este proceso.

A cada una de las personas con las que hablé durante este tiempo. Cada una de esas conversaciones me han vuelto la persona que soy hoy.

Table of Content

1. Introduction **1**
- 1.1. Motivation . . . 1
- 1.2. Hypothesis . . . 1
- 1.3. General Objectives . . . 2
- 1.4. Specific Objectives . . . 2

2. Preliminaries **3**
- 2.1. Graphs . . . 3
- 2.2. Graph spectral theory . . . 4
  - 2.2.1. Convolutions in Euclidean space . . . 4
  - 2.2.2. Convolutions in graphs . . . 6
- 2.3. Graph Neural Networks . . . 9
  - 2.3.1. Spatial Graph Neural Networks . . . 9
  - 2.3.2. Spectral Graph Neural Networks . . . 10
  - 2.3.3. Literature review . . . 13
- 2.4. Dynamic graphs . . . 17
- 2.5. Temporal Graph Neural Networks . . . 17
  - 2.5.1. Literature review . . . 19

3. Methods **23**
- 3.1. Polynomial Subspace Net (PSN) . . . 23
  - 3.1.1. Further restrictions over the base filter . . . 24
- 3.2. Online Graph Nets (OGN) . . . 26
  - 3.2.1. Complexity analysis . . . 28
  - 3.2.2. On-edge prediction . . . 29

4. Results and discussion **30**
- 4.1. Polynomial Subspace Net (PSN) . . . 30
  - 4.1.1. Graph-level prediction tasks . . . 30
  - 4.1.2. Node classification . . . 31
- 4.2. Where is the gain coming from? . . . 32
  - 4.2.1. Ablation study . . . 33
  - 4.2.2. The impact of restricting the parameters . . . 34
  - 4.2.3. Spectral analysis . . . 34
  - 4.2.4. Measuring oversmoothing . . . 35
  - 4.2.5. Limitations . . . 35
- 4.3. Online Graph Nets (OGN) . . . 36
  - 4.3.1. Temporal link prediction . . . 36

4.3.2. Node classification 37
4.4. Ablation Studies 38
4.4.1. Temporal GNNs 38
4.4.2. OGN 39

5. Conclusions 42

Bibliography 44

Annex A. PSN 48
A.1. Implementation details 48
A.1.1. Datasets 48
A.1.2. Evaluation setup 49
A.1.3. Hardware 49
A.2. Additional experiments 51
A.2.1. Complexity 51
A.2.2. Ablation study 51
A.2.3. Measuring oversmoothing 52
A.2.4. Gradients 54
A.3. Proof of Proposition 1 54
A.4. Proof of Proposition 2 55
A.5. Proof of Proposition 3 56
A.6. Proof of Proposition 4 57

Annex B. OGN 58
B.1. Causal Anonymous Walk (CAW) 58
B.1.1. The Bug: Attention over the batch instead of the walks 58
B.2. Datasets 59
B.2.1. Twitter dataset 60
B.3. Implementation details 60
B.3.1. Evaluation setup and hyperparameters 61
B.4. Further Ablation Study 61
B.4.1. Hardware 62
B.5. More experiments 62
B.6. Further intuition behind the weighted average 62
B.7. Proof: tractable aggregation 63

List of Tables

3.1. Complexity analysis and parameter count. In the table, n and m denote the number of nodes and edges in an input graph, K the degree of the polynomial filter, and d the input feature dimension. 26

4.1. Graph-level prediction on OGB datasets and ZINC. PSN reaches the best result for ZINC and MOLPCBA, and is competitive against PNA in MOLHIV. For ZINC, the lower the better. Boldface indicates the best average result in each dataset, and underline the second best. 31

4.2. Graph classification on TU datasets. PSN shows better overall performance than other methods. 31

4.3. Node classification on citation networks. Despite its simple formulation, PSN is competitive with recently proposed methods, such as DAGNN and GCNII. . . . 32

4.4. Ablation study. For each model, we report the number of layers inside parenthesis. 33

4.5. Oversmoothing for different GNNs (higher is better). Entries represent normalized mean distance between pairs of neighbors, averaged over all graphs. Overall, PSN shows the most consistent results. 35

4.6. **Results in average precision (AP) on link prediction for datasets with edge features.** In all datasets, OGN is either the best performing method or is closely behind it. * Corresponds to the original standard deviation 0.04 rounded to the first decimal. 36

4.7. **Results for node classification (AUC)**. OGN is the best performing model on Wikipedia and obtains the second best AUC on Reddit. 38

4.8. **Results for TGN and TGAT with different attention modules in the transductive setting.** Using attention either brings marginal gains or winds up hurting performance (TGAT on MOOC). Superscripts * and ** denote a standard deviation of 0.04 and 0.03 rounded to the first decimal, respectively. . . 40

4.9. **Ablation study for OGN**. Neighborhood state and edge information play a crucial role on the model's performance. Discretized timestamps generally lead to better performance. 40

A.1. Summary of graph classification and regression datasets. 48

A.2. Summary of node classification datasets. 48

A.3. Models used for graph classification datasets. For the cases marked as * the result that we presented in the Experiments section was obtained by a search over 2,3,4, and 5 layers by [46]. 50

A.4. Models used for node classification datasets. 51

A.5. Complexity analysis and parameter count for the ZINC dataset with 16 layers, $K = 4$ and $d = 128$. We empirically measure the number of parameters and the seconds per epoch both in CPU and GPU. 51

A.6. **Ablation study for a varying number of layers.** For each dataset, boldface indicates the best average result for each PSN variant. An additional underline highlights the best overall model. 52
B.1. Results of CAW in transductive setting (average precision). 59
B.2. Summary statistics for temporal link prediction datasets. * Corresponds to edge features filled with zero values. 60
B.3. Hyperparameters for CAW. 61
B.4. Results in average precision (AP) on link prediction for datasets that do not contain edge features. 62

List of Figures

2.1. Visual representation for the Laplacian of a function $f(x, y)$. First the gradient $\nabla f(x, y)$ is computed. Then the divergence is applied resulting in the Laplacian $\Delta f(x, y)$. [15] 5

2.2. **T-GNNs: general framework to compute temporal node representations.** For clarity, we illustrate an example for node u with $L = 2$. To predict for an event between nodes u and v at time t, T-GNNs (*i*) iteratively sample a L-hop temporal neighborhood of u (Sample), (*ii*) encode the continuous timestamps (TimeEnc), (*iii*) iteratively aggregate the neighborhood information (Aggregate), and (*iv*) combine the resulting neighborhood information vector with the state vector to obtain the final embedding for node u (Combine). . . . 18

3.1. Polynomial Subspace Nets (PSNs). For clarity, we only denote the first layer ($\boldsymbol{H}^{(0)} = \boldsymbol{X}$) and omit layer indexes in the model parameters. 24

3.2. **OGN vs SOTA T-GNNs on Reddit (+500k interactions).** The horizontal axis shows the relative training time for each method as a multiple of OGN's running time. The vertical shows average precision. OGN clearly outperforms the SOTA but runs approximately 10 times faster than TGN [34] and 374 times faster than CAW [35]. 26

3.3. **Online Graph Nets (OGN).** OGN maintains a state vector and a neighborhood summary for each node. For a node u, to predict for a new event with node v, OGN updates the state vector of u using the neighborhood summary $r_u^{(n-1)}$ and the information of the event ($\boldsymbol{e}^{(n)}$ and $\boldsymbol{t}^{(n)}$), then updates its neighborhood information using the updated state vector of v. OGN performs this update for both nodes, and combines the resulting state vectors through an MLP for prediction purposes. 28

4.1. Filter responses for PSN in different datasets. We show the individual filters at each layer (red) and the combined filter response (blue). PSN layers combine to generate diverse filters: high-pass (ZINC), band-stop (REDDIT-B), and band-pass (MOLHIV). On Citesser, PSN learns a low-pass filter, meeting well-known efficient filter designs for node classification tasks [47]. 33

4.2. Normalized std of gradients. Using tanh leads to more stable training as gradient values fluctuate less. 34

4.3. **Time/epoch vs average precision.** We normalize the execution time per epoch of all models with respect to the execution time of OGN. In all cases, OGN nearly matches or surpasses the SOTA. Also, OGN is the fastest method overall. For instance, OGN is more than two orders of magnitude faster than CAW and at least one order of magnitude faster than TGAT. 37

4.4. (**a**) **Effect of using regularly spaced timestamps ($\mathcal{U}$-time) on TGAT, TGN and CAW.** In general, using $\mathcal{U}$-TIME ($\triangle$ markers) does not harm performance. In fact, TGN and TGAT benefit from time discretization. Also, the models show smaller standard deviations overall. (**b**) **Predictions of TGN with shifted timestamps on test data for Wikipedia.** Even after applying a 0.99 relative lag, the logits remain rather similar to those from the original no-shifted timestamps (red line). 39
A.1. **Measuring oversmoothing: mean average distance for a varying number of layers.** Among all methods, PSN is the least affected by oversmoothing when depth increases. Higher is better. 53
A.2. **Performance as a function of network depth.** Overall, deep PSNs (32 layers) outperform the competing methods. 53
A.3. **Normalized std of gradients per epoch for the TOX21, REDDIT-B, and DD datasets.** Overall, using tanh results in smaller gradient fluctuations during training. 54
B.1. Removing timestamps can hurt the performance of T-GNNs. Both TGN (NO-TIME) and TGAT (NO-TIME) experience a significant performance drop on the UCI dataset. 61

Chapter 1

Introduction

1.1. Motivation

During the last few decades, deep learning has found success in areas that used to rely in handcrafted features, such as computer vision and natural language processing [1, 2, 3]. The success of these methods has been fueled by advances in hardware and a significant increase of available training data. The newfound success of deep learning methods has also piqued the interest of researchers around the world, who look to extract representations from Euclidean data, i.e., data that is represented in an Euclidean space.

While the success of deep learning in Euclidean data is undeniable, non-Euclidean data, or data with an underlying structure that is not Euclidean, is becoming ever more prevalent. This results in the need for deep learning models that focus on this type of data. Particularly, models that operate on graph-structured data, known as graph neural networks (GNN) [4, 5] have gained great popularity, being applied to a wide variety of different data sources such as social networks [6, 7], citation networks [8] and molecules [9, 10].

Recently, research in GNNs has focused on increasingly complex models that mix a variety of tricks to obtain state-of-the-art results [11, 12, 13]. We believe that simpler GNN models, both in terms of number of parameters and computational complexity, have not been sufficiently studied and could provide an alternative to state-of-the-art GNN models, by being more efficient while achieving similar performance to its more complex counterparts. With this in mind, we pose the following hypothesis for this thesis.

1.2. Hypothesis

“Simple GNN models, in terms of number of parameters and computational complexity, can achieve results that are competitive with more complex state-of-the-art models.”

1.3. General Objectives

In order to evaluate the hypothesis, we propose the following general objectives for the work.

- Propose simple GNN models that have lower computational complexity than state-of-the-art methods while being competitive in terms of performance.
- Study the advantages and limitations of the proposed methods with respect to more complex alternatives.

1.4. Specific Objectives

The general objective captures the overarching goal of the thesis. Alongside the general objective, these are the main specific results achieved in the thesis.

- Propose and implement a spectral GNN for static graphs that has lower computational complexity and number of parameters than other such models, while outperforming state-of-the-art methods in graph classification tasks and obtaining comparable performance in node classification tasks.
- Propose and implement a temporal GNN for dynamic graphs that has lower computational complexity and memory usage than other such methods. The method should present significant speedup over other state-of-the-art methods while achieving comparable performance or even outperforming them. It should also be a fully online streaming method to enable its usage in real world applications.
- Analyze the different components of the proposed methods to understand how they achieve competitive performance with state-of-the-art methods while maintaining simplicity.
- Study the limitations of the proposed methods when compared to more complex alternatives.

Chapter 2

Preliminaries

In order to understand how neural networks are applied to graph-structured data, we need to introduce some key concepts about graphs. Further, due to the content of the work we also need to introduce concepts for spectral graph theory and temporal graphs. The following section serves the purpose of introducing these key concepts and presenting the notation that is used throughout the work.

2.1. Graphs

Graphs are a type of data structure that is represented as a set of nodes connected by edges [14]. This is denoted as $G = (V, E)$, where G is the graph, V is the set of vertices and E is the set of edges. We use $v_i \in V$ to denote that a node is part of the graph, and $e_{ij} = (v_i, v_j) \in E$ to denote that there is an edge from node v_i to v_j in the graph. The number of nodes and edges in the graphs are expressed as $n = |V|$ and $m = |E|$ respectively. Each node in the graph has a neighborhood, which we denote by $\mathcal{N}(v_i) = \{v_j \in V : (i, j) \in E\}$, and contains all of the nodes that share an edge with v_i.

An alternate way to represent a graph is through its adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$. This matrix takes the value $\boldsymbol{A}_{ij} = 1$ if $e_{ij} \in E$, and takes the value $\boldsymbol{A}_{ij} = 0$ if $e_{ij} \notin E$. A special case is when $\boldsymbol{A}$ is symmetric, in which case the graph is called undirected. In these type of graphs, if edge (i, j) is in the graph, the inverse direction (j, i) is also in the graph. We can also define the degree matrix, $\boldsymbol{D} \in \mathbb{R}^{n \times n}$, as the matrix that contains the number of neighbors that node v_i has in $\boldsymbol{D}_{ii}$. The elements of $\boldsymbol{D}$ can be computed as $\boldsymbol{D}_{ii} = \sum_{j, v_j \in \mathcal{N}(v_i)} \boldsymbol{A}_{ij}$. Elements outside of the diagonal of $\boldsymbol{D}$ are zero.

Graph data often includes node and edge features. We denote the node feature for node $v_i \in V$ as $\boldsymbol{h}_i \in \mathbb{R}^d$, with d the size of node features. The edge feature for for edge $e_{ij} \in E$ is denote by $\boldsymbol{h}_{ij} \in \mathbb{R}^c$, with c the size of edge features. The features can also be represented in a matrix, where each row contain a single feature vector. For node features we call the matrix $\boldsymbol{H}_n \in \mathbb{R}^{n \times d}$, and for edge features we call the matrix $\boldsymbol{H}_e \in \mathbb{R}^{m \times c}$. Frequently, datasets contain node features, but no edge features. If this is the case, we denote the node features simply by $\boldsymbol{H} \in \mathbb{R}^{n \times d}$.

2.2. Graph spectral theory

2.2.1. Convolutions in Euclidean space

To understand and motivate graph neural networks, it is necessary to introduce concepts of Graph Spectral Theory. In particular, we look to introduce the concept of convolutions on graph-structured data. To do this, we begin by introducing the concept of a convolution in a Euclidean Space and then generalizing this concept for graphs. We denote the convolution between two functions $f, g : \mathbb{R}^d \to \mathbb{R}$ as $f \star g$. The convolution outputs a function $h : \mathbb{R}^d \to \mathbb{R}$ which is defined as follows.

$$h(\boldsymbol{x}) = (f \star g)(\boldsymbol{x}) = \int_{\mathbb{R}^d} f(\boldsymbol{y}) g(\boldsymbol{x} - \boldsymbol{y}) d\boldsymbol{y}. \tag{2.1}$$

A key characteristic of the convolution operator is that it becomes an element-wise multiplication when the Fourier transform is applied. Formally, if we denote the Fourier transform by $\mathcal{F}$, the following equality holds.

$$\mathcal{F}\left((f \star g)(\boldsymbol{x})\right) = \mathcal{F}(f(\boldsymbol{x})) \odot \mathcal{F}(g(\boldsymbol{x})), \tag{2.2}$$

where $\odot$ denotes the element-wise multiplication, and the Fourier transform is defined as follows.

$$\mathcal{F}(f)(\boldsymbol{s}) = \hat{f}(\boldsymbol{s}) = \int_{\mathbb{R}^d} f(\boldsymbol{s}) e^{-j2\pi(\boldsymbol{x} \cdot \boldsymbol{s})} d\boldsymbol{x}. \tag{2.3}$$

We use the property provided in Equation 2.2 as a key component to define convolutions on graphs. The second key definition we require is the Laplacian in an Euclidean Space. Let $f : \mathbb{R}^d \to \mathbb{R}$ be a smooth function, we can define the Laplacian of f, denoted as Δf, as follows.

$$\Delta f = \nabla \cdot (\nabla f) = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}. \tag{2.4}$$

Note that in Equation 2.4 we use ∇f to denote the gradient of function f and $\nabla \cdot$ to denote the divergence operator. The gradient and the divergence are of great importance to generalize the Fourier Transform in graph-structured data, so we explain them in detail.

The gradient of a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ returns a vector field, let us call it F, that points in the direction of steepest ascent for the function f. Further, the magnitude at a point $\boldsymbol{x}$ is proportional to the change of function f at point $\boldsymbol{x}$. We denote the value for the gradient of f at a point $\boldsymbol{x}$ as $F(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$, and it takes the value $F(\boldsymbol{x})_i = \frac{\partial f}{\partial x_i}(\boldsymbol{x})$. A visual representation for the gradient can be seen in Figure 2.1.

On the other hand, the divergence operator takes a vector field and returns a scalar value for every position $\boldsymbol{x}$ in the vector field. Intuitively, the divergence for a point $\boldsymbol{x}$ describes the flux of the vector field in a given point. If the flux is coming out of the point, the divergence has a value larger than 0, and if its going into the point the divergence has a value smaller than

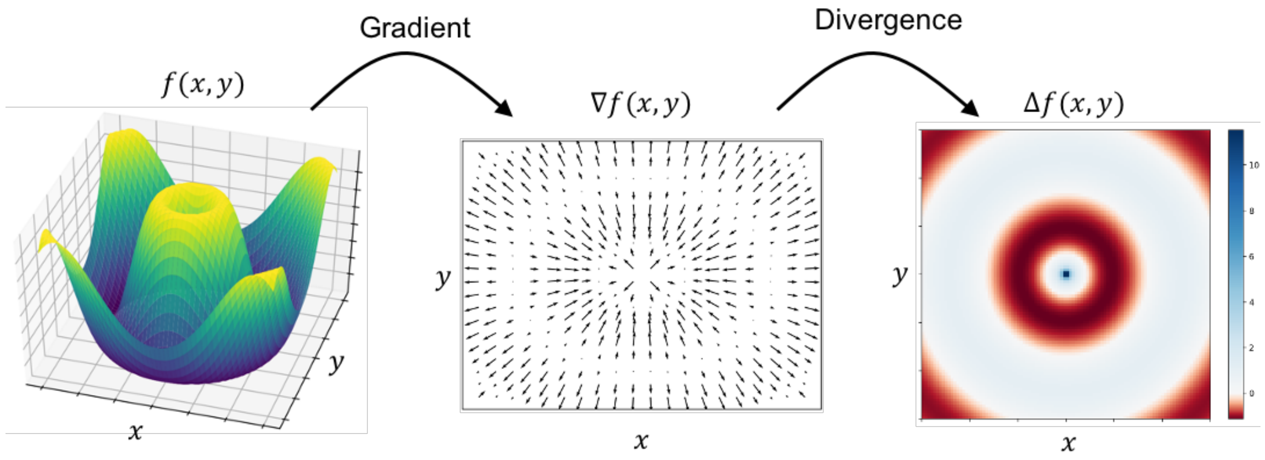


Figure 2.1: Visual representation for the Laplacian of a function $f(x, y)$. First the gradient $\nabla f(x, y)$ is computed. Then the divergence is applied resulting in the Laplacian $\Delta f(x, y)$. [15]

0. The magnitude of the divergence in a point measures the magnitude of the flux in the point. Formally, the divergence of a vector field F at a point $\boldsymbol{x}$ is given by $\nabla \cdot F(\boldsymbol{x}) = \sum_{i=1}^{d} \frac{\partial F}{\partial x_i}(\boldsymbol{x})$. A visual representation of applying the divergence to a vector field is shown in Figure 2.1, where the divergence is applied to the vector field resulting from applying the gradient to a function $f(x, y)$.

Going back to the Laplacian operator, an interesting property is that its eigenfunctions are the complex exponentials used in the Fourier Transform. The eigenfunctions of an operator D are defined as the as the functions f that satisfy the following equality.

$$Df = \lambda f, \tag{2.5}$$

where λ is a scalar called the eigenvalue. It is easy to see that using the Laplacian as the operator and the complex exponentials as the function, the equality in Equation 2.5 holds.

$$\Delta(e^{-j2\pi xs}) = \frac{\partial^2(e^{-j2\pi xs})}{\partial x^2} = -(2\pi s)^2 e^{-j2\pi xs}. \tag{2.6}$$

Using $\lambda = -(2\pi s)^2$ we can clearly see that Equation 2.5 holds for this particular case. As mentioned previously, this indicates that the eigenfunctions of the Laplacian are the modes of the frequency domain in the Fourier Transform, which is a characteristic we will use when defining the Fourier Transform in graph-structured data.

To summarize, we want to generalize the convolution operation to graph-structured data in a way that:

1. We want the frequency modes of our Fourier transform to be the eigenbasis of the Laplacian operator defined in graph-structured data, as in Equation 2.6.
2. The Fourier transform of the convolution, results in the element-wise multiplication of the respective Fourier transforms, as in Equation 2.2.

2.2.2. Convolutions in graphs

To generalize the concept of convolutions to graphs, we begin by defining the Laplacian operator for graphs. To do this, we will use the definition for the Laplacian operator that we presented in Equation 2.4, and use it for graph-structured data by defining the gradient and divergence in this domain.

To introduce the analogy for gradients and divergences in graphs, we consider that every vertex is analogous to a position in an Euclidean Space. With this in mind, we define a function in a graph, denoted by f_G as a function that takes a vertex and outputs a real value, $f_G : V \to \mathbb{R}$. This is analogous to a function in an Euclidean Space, which takes a point and outputs a real value.

With the previous definitions in mind, we can define the gradient of a function f_G, as a finite difference, similarly to how it's done in discrete functions. The gradient for graphs is computed for its edges, because it naturally encapsulates neighboring nodes, similarly to neighboring points for discrete functions. With this, the gradient for an edge e_{ij} in the graph is given by the following expression.

$$g_G(e_{ij}) = \nabla f_G(e_{ij}) = f_G(v_i) - f_G(v_j). \tag{2.7}$$

An important point to note is that the direction of the edge defines the order of the difference in Equation 2.7. In undirected graphs, we have both edges e_{ij} and e_{ji}, which results in both directions of the difference being in the gradient.

To generalize the divergence to graph-structured data, we remember the intuition for the divergence in Euclidean Space. The divergence operator can be understood as a measure of the flux that comes out of a point in space. Since in graph-structured data we consider the nodes as the points in space and the edges to vectors around each point, it follows that we compute the divergence as the sum of edges around each node. Taking this into consideration, we can define the divergence for a point v_i as follows.

$$(\nabla \cdot g_G)(v_i) = \sum_{j, v_j \in \mathcal{N}_i} g_G(e_{ij}). \tag{2.8}$$

With the definition of the gradient and divergence for graph-structured data, we can define the Laplacian for graphs as follows:

$$\begin{aligned}
\Delta f_G(v_i) &= \sum_{j, v_j \in \mathcal{N}_i} (f_G(v_i) - f_G(v_j)) \\
&= |\mathcal{N}_i| f_G(v_i) - \sum_{j, v_j \in \mathcal{N}_i} f_G(v_j) \\
&= \boldsymbol{D}_{ii} f_G(v_i) - \sum_{i \in V} \boldsymbol{A}_{ij} f_G(v_j).
\end{aligned} \tag{2.9}$$

Let us remember that $\boldsymbol{A}$ and $\boldsymbol{D}$ in Equation 2.9 denote the adjacency matrix and the degree matrix respectively. From the way that Equation 2.9 is written, it is clear that it can be rewritten as the product between a matrix and a vector. In fact, let us denote $\boldsymbol{f} = [f_G(v_1), f_G(v_2), ...f_G(v_N)]^\top$ and $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$. We can re-write Equation 2.9 as:

$$\Delta \boldsymbol{f} = \Delta f_G(v_i) = (\boldsymbol{D} - \boldsymbol{A})\boldsymbol{f} = \boldsymbol{L}\boldsymbol{f}. \tag{2.10}$$

It is evident from Equation 2.10 that matrix $\boldsymbol{L}$ has a close relation to the laplacian operator for graph-structured data. Because of this, we call matrix $\boldsymbol{L}$ the Laplacian Matrix. Note that this is not the only possible definition of the Laplacian Matrix. Different definitions for the gradient and divergence result in different Laplacian Matrices. One such Laplacian Matrix, that we will use throughout the work is the normalized graph Laplacian, which we denote $\boldsymbol{L}_{norm} = \boldsymbol{D}^{-1/2}\boldsymbol{L}\boldsymbol{D}^{-1/2} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$.

Let us remember that the reason we were looking to generalize the Laplacian operator to graph-structured data, was so we could define the frequency modes and the Fourier transform. Recall from Equation 2.6 that we want the frequency modes to be the eigenbasis of the Laplacian operator. Since the Laplacian operator we defined in graph-structured data is a linear operator with the matrix $\boldsymbol{L}$, we can define our eigenbasis as the eigevectors for the matrix.

To define the frequency modes, let us begin by assuming that the graph G we are considering is undirected. In this type of graphs, $\boldsymbol{A}$ is symmetric. Since $\boldsymbol{D}$ is a diagonal matrix, and thus always symmetric, $\boldsymbol{L}$ is also symmetric. Further, $\boldsymbol{L}$ is real valued, because $\boldsymbol{A}$ and $\boldsymbol{D}$ are real valued. With this in mind, we can use the following well known theorem for real symmetric matrices.

Theorem 1 (See Corollary 2.5.11 in [16]) Any square, real-valued symmetric matrix $\boldsymbol{B} \in \mathbb{R}^{n\times n}$ admits an eigendecomposition of the form $\boldsymbol{B} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$, where $\boldsymbol{U} \in \mathbb{R}^{n\times n}$ is an orthogonal matrix and its columns $\boldsymbol{u}_1, ..., \boldsymbol{u}_n$ are the eigevectors for $\boldsymbol{B}$. Furthermore, the matrix $\boldsymbol{U}$ is orthonormal, which implies that $\boldsymbol{U}^{-1} = \boldsymbol{U}^\top$. $\boldsymbol{\Lambda} \in \mathbb{R}^{n\times n}$ is a diagonal matrix that contains the eigenvalues for $\boldsymbol{B}$.

The matrix $\boldsymbol{U}$ defined in Theorem 1 is specially useful since its columns form an eigenbasis for the matrix. We can then apply this eigendecomposition to the Laplacian matrix $\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$ and use the eigenbasis as the frequency modes of the Fourier transform that we define in graph-structured data. Further, we also consider the eigenvalues $\boldsymbol{\Lambda}$ as the spectrum of the Fourier transform. Taking this into account, we can define the Fourier transform for graph-structured data, $\mathcal{F}_G$ as follows:

$$\mathcal{F}_G(\boldsymbol{h}) = \boldsymbol{U}^\top \boldsymbol{h}. \tag{2.11}$$

$\boldsymbol{U} \in \mathbb{R}^{n\times n}$ denotes the eigenbasis for the graph Laplacian and $\boldsymbol{h} \in \mathbb{R}^n$ denotes a signal in the graph. From Theorem 1 the inverse of matrix $\boldsymbol{U}$ is given by $\boldsymbol{U}^{-1} = \boldsymbol{U}^\top$. We can use this to easily define the inverse of the graph Fourier transform as:

$$\mathcal{F}_G^{-1}(\boldsymbol{s}) = \boldsymbol{U}\boldsymbol{s}. \tag{2.12}$$

Where $\boldsymbol{s} \in \mathbb{R}^n$ denotes the signal frequencies in the graph Fourier domain. To simplify the notation we will use $\mathcal{F}$ to indicate the graph Fourier transform and $\mathcal{F}^{-1}$ to indicate the inverse graph Fourier transform, unless specified otherwise. With this we have defined a Fourier transform in graph-structured data, which parallels that of the Euclidean space by using the eigenbasis of the Laplacian as its frequency modes. We can now use the definition for the Fourier transform to define convolutions in graphs.

Let us remember that in Euclidean space the Fourier transform of the convolution results in element-wise multiplication of the respective Fourier transforms. We define the Fourier transform in graph-structured data in a way that preserves this property. Formally, we want our convolution operation in graph-structured data, $\star_G$, to have the following property:

$$\boldsymbol{f} \star_G \boldsymbol{h} = \mathcal{F}^{-1}\left(\mathcal{F}(\boldsymbol{f}) \odot \mathcal{F}(\boldsymbol{h})\right). \tag{2.13}$$

Where $\boldsymbol{f} \in \mathbb{R}^n$ can be a filter or signal on the graph and $\odot$ denotes the element-wise multiplication. We may also refer to $\boldsymbol{U}^\top \boldsymbol{f}$ as the filter, since it is the equivalent of filter $\boldsymbol{f}$ in the spectral domain, and is often easier to analyze. We can use the definitions in Equation 2.11 and Equation 2.12 in Equation 2.13 to define the convolutions in graph-structured data.

$$\boldsymbol{f} \star_G \boldsymbol{h} = \boldsymbol{U}\Big((\boldsymbol{U}^\top \boldsymbol{f}) \odot (\boldsymbol{U}^\top \boldsymbol{h})\Big). \tag{2.14}$$

To simplify the notation we will use $\star$ to indicate the graph convolution unless specified otherwise. With the result in Equation 2.14 we achieved the objective of defining a convolution operator in graphs.

Let us note that Equation 2.14 may also be written by replacing $\boldsymbol{U}^\top \boldsymbol{f}$ by a diagonal matrix that contains the elements of vector $\boldsymbol{U}^\top \boldsymbol{f}$. In fact, let us denote $\boldsymbol{F} = \operatorname{diag}((\boldsymbol{U}^\top \boldsymbol{f})_1, ..., (\boldsymbol{U}^\top \boldsymbol{f})_n)$. Then Equation 2.14 may be written as follows.

$$\boldsymbol{f} \star \boldsymbol{h} = \boldsymbol{U}\boldsymbol{F}\boldsymbol{U}^\top \boldsymbol{h}. \tag{2.15}$$

The equivalence between Equation 2.14 and Equation 2.15 can be verified by noting that:

$$\begin{aligned}(\boldsymbol{F}\boldsymbol{U}^\top \boldsymbol{h})_i &= (\boldsymbol{F}(\boldsymbol{U}^\top \boldsymbol{h}))_i \\ &= \boldsymbol{F}_i(\boldsymbol{U}^\top \boldsymbol{h}) \\ &= (\boldsymbol{U}^\top \boldsymbol{f})_i(\boldsymbol{U}^\top \boldsymbol{h})_i \\ &= \Big((\boldsymbol{U}^\top \boldsymbol{f}) \odot (\boldsymbol{U}^\top \boldsymbol{h})\Big)_i.\end{aligned} \tag{2.16}$$

Since Equation 2.16 shows that $\boldsymbol{F}\boldsymbol{U}^\top \boldsymbol{h} = (\boldsymbol{U}^\top \boldsymbol{f}) \odot (\boldsymbol{U}^\top \boldsymbol{h})$, we can simply multiply both sides of the equality by $\boldsymbol{U}$ to show the equivalence between Equation 2.14 and Equation 2.15. By defining the filter directly in the spectral domain, as in Equation 2.15 by using $\boldsymbol{F} \in \mathbb{R}^{n \times n}$, we can more easily design filters for specific use cases. Further, we can easily relate the filter to the spectrum of the graph by defining the filter as a function of the eigenvalues of the Laplacian $\mathbf{\Lambda} \in \mathbb{R}^{n \times n}$. This will be useful when we analyze spectral graph neural networks.

2.3. Graph Neural Networks

In the following section we introduce the basic notions of static graph neural networks, often referred to simply as graph neural networks (GNN). The section does not look to cover state-of-the-art GNN methods, nor does it aim to be a thorough survey of GNN models. Rather, this section should be thought of as a basic introduction to GNNs. At the end of the section we present a brief literature review of methods referenced in the work.

We classify GNNs as either spatial graph neural networks or spectral graph neural networks, depending on how the convolution operator is defined in the graph. It is important to note that there is overlap between spatial graph neural networks and spectral graph neural networks. Convolution operations defined in the spatial domain might have a spectral interpretation and convolutions in the spectral domain might have an equivalent spatial operation. Thus, this classification is not strict and is closely related to the motivation of each model.

2.3.1. Spatial Graph Neural Networks

Spatial GNNs are a type of GNN that define the convolution operation directly on the graph structure by taking the graph topology into account.[14] There are countless ways of defining spatial convolutions over graph-structured data. Thankfully, many of the popular spatial GNNs share characteristics that can be captured in a common framework. Arguably, the most popular spatial GNN framework is the message passing neural network (MPNN)[17]. For this description of spatial GNNs we will restrict ourselves to describing the MPNN framework and presenting examples of spatial GNNs that fit the framework. However, it is important to be aware of the existence of other spatial GNNs.

The MPNN framework considers two steps, a message passing step and an update step. In the message passing step, information in the form of messages is aggregated from the neighborhood of the node. In the update step, the nodes are updated using the messages that were previously aggregated. More concretely, if each node $v \in V$ has a feature vector $\boldsymbol{h}_v^{(k)}$, the MPNN updates them as follows.

$$\begin{aligned} \boldsymbol{m}_u^{(k+1)} &= \textsc{Aggregate}(\{M(\boldsymbol{h}_v^{(k)}, \boldsymbol{h}_u^{(k)})\}_{v \in \mathcal{N}(u)}), \\ \boldsymbol{h}_u^{(k+1)} &= \textsc{Update}(\boldsymbol{h}_u^{(k)}, \boldsymbol{m}_u^{(k+1)}). \end{aligned} \tag{2.17}$$

In Equation 2.17, $M(\cdot,\cdot)$ is the message function that takes the feature vector of node u along with that of a neighboring node $v \in \mathcal{N}(u)$ to build a message from node v to node u. The $\textsc{Aggregate}(\cdot)$ function is then used to aggregate the messages from the neighboring nodes into a single message $\boldsymbol{m}_u^{(k+1)}$. Finally, the $\textsc{Update}(\cdot,\cdot)$ function takes the aggregated message $\boldsymbol{m}_u^{(k+1)}$ and the previous feature vector $\boldsymbol{h}_u^{(k)}$ to compute the updated feature vector $\boldsymbol{h}_u^{(k+1)}$. With the description in mind, the steps of the MPNN can be understood as updating the node representation by aggregating information from neighboring nodes.

To give an example of the MPNN framework we will show how Graph Convolutional Network (GCN) can be expressed by using this framework. The basic GCN operation corresponds to updating the features in the graph, $\boldsymbol{H}^{(k)}$, through the following expression.

$$\boldsymbol{H}^{(k+1)} = \theta\left((\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})\boldsymbol{H}^{(k)}\boldsymbol{W}^{(k)}\right). \tag{2.18}$$

In Equation 2.18, θ is an activation function and $\boldsymbol{W}^{(k)}$ are learnable weights that mix the channels for each node's feature vector. To implement GCN as a MPNN we need to implement the $(\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})\boldsymbol{H}^{(k)}$ multiplication with message passing, since the remaining operations can be implemented node-wise. To understand how to implement the operation through message passing we can look at the (u, v) position of the matrix. This position the effect of the v-th node of the graph on the u-th node.

$$(\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})_{uv} = \begin{cases} 1 & u = v \\ \frac{1}{\sqrt{d_i d_j}} & v \in \mathcal{N}(u) \\ 0 & \sim . \end{cases} \tag{2.19}$$

Note that for the previous expression we are assuming that the original graph does not contain nodes that are connected to itself. The representation in Equation 2.19 shows a clear way to implement GCN as a MPNN, since any node u only interacts with its neighboring nodes. With this in mind we can define the following functions.

$$\begin{aligned} M(\boldsymbol{h}_u, \boldsymbol{h}_v) &= \frac{\boldsymbol{h}_v}{\sqrt{d_i d_j}}, \\ \textsc{Aggregate}(\{M_v\}_{v\in\mathcal{N}(u)}) &= \sum_{v\in\mathcal{N}(u)} M_v, \\ \textsc{Update}(\boldsymbol{h}_u, \boldsymbol{m}_u) &= \theta\left((\boldsymbol{h}_u + \boldsymbol{m}_u)\boldsymbol{W}\right). \end{aligned} \tag{2.20}$$

Which is the message passing implementation for GCN. Note that by defining the operations through message passing, the complexity of the method is $O(m)$ with m the number of edges in the graph, instead of $O(n^2)$ with n the number of nodes which would be the case if the matrix multiplication were done directly.

2.3.2. Spectral Graph Neural Networks

Spectral GNNs are a type of GNN that define the convolution operation on the spectral domain. This type of GNN has a inspired by Graph Spectral Theory introduced in Section 2.2.

In fact, the convolution operator we will use in the spectral domain is the one defined in Equation 2.15. Though many GNN methods may be interpreted in the spectral domain [18], we will only consider those that were originally formulated through the spectral graph convolution as spectral GNNs.

To give an example of a spectral GNN, we will analyze GCN from a spectral perspective, similar to what was done for spatial GNNs. As GCN was initially conceived in the spectral domain, it fits our description of a spectral GNN. To analyze GCN, we will use the normalized Laplacian $\boldsymbol{L}_{norm} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$ introduced in Section 2.2. We can re-write the graph operation for GCN as follows.

$$(\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2})\boldsymbol{H} = (2\boldsymbol{I} - \boldsymbol{L}_{norm})\boldsymbol{H}. \tag{2.21}$$

Let us remember that we can apply the eigendecomposition to the Laplacian to write it as $\boldsymbol{L}_{norm} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$. Further, since the matrix $\boldsymbol{U}$ is orthonormal for undirected graphs, $\boldsymbol{U}\boldsymbol{I}\boldsymbol{U}^\top = \boldsymbol{I}$. We can use this in Equation 2.21 to obtain the following expression.

$$\begin{aligned}(2\boldsymbol{I} + \boldsymbol{L}_{norm})\boldsymbol{H} &= (2\boldsymbol{U}\boldsymbol{I}\boldsymbol{U}^\top - \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top)\boldsymbol{H} \\ &= \boldsymbol{U}(2\boldsymbol{I} - \Lambda)\boldsymbol{U}^\top\boldsymbol{H}.\end{aligned} \tag{2.22}$$

We can notice that Equation 2.22 has the same structure as Equation 2.15, with $\boldsymbol{F} = 2\boldsymbol{I} - \boldsymbol{\Lambda}$. With this, the filter that GCN is applying is given by $f(\lambda) = 2 - \lambda$, with lambda the respective eigenvalue. It is also known that the eigenvalues values of the normalized Laplacian take values between 0 and 2 [19]. Taking this into account, the filter of a GCN layer can be understood as a low-pass filter from a spectral point of view.

Rather than analyzing GNN models from a spectral perspective, it is also possible to build GNNs from the spectral formulation presented in Equation 2.15. We could do this by directly learning the values for the the matrix representing the filter in the spectral domain, $\boldsymbol{F}$. Note that $\boldsymbol{U}$ is given by the graph's structure and $\boldsymbol{h}$ is the graph's features, so the filter is the only learnable portion of Equation 2.15. The problem with this non-parametric approach of learning $\boldsymbol{F}$ is that the number of parameters of the filter is tied with the number of nodes in the graph. It also does not guarantee that $\boldsymbol{F}$ leads to localized filters [20].

A way to circumvent the aforementioned problems is to approximate the filter $\boldsymbol{F}$ using a polynomial of the eigenvalue matrix $\boldsymbol{\Lambda}$. We can then represent the polynomial spectral filters with base spectrum $\boldsymbol{\Lambda}$ as follows.

$$\boldsymbol{F} = \sum_{k=0}^{K} \theta_k \boldsymbol{\Lambda}^k. \tag{2.23}$$

$\theta_0, \ldots, \theta_K$ in Equation 2.23 are the polynomial coefficients for the filter. With the parametrization in Equation 2.23, the graph convolution presented in Equation 2.15 can be re-written resulting in the following expression.

$$
\begin{aligned}
\boldsymbol{f} \star \boldsymbol{h} &= \boldsymbol{U}\boldsymbol{F}\boldsymbol{U}^\top \boldsymbol{h} \\
&= \boldsymbol{U} \sum_{k=0}^{K} \theta_k \boldsymbol{\Lambda}^k \boldsymbol{U}^\top \boldsymbol{h} \\
&= \sum_{k=0}^{K} \theta_k \boldsymbol{U}\boldsymbol{\Lambda}^k \boldsymbol{U}^\top \boldsymbol{h} \\
&= \sum_{k=0}^{K} \theta_k \boldsymbol{\Delta}^k \boldsymbol{h}.
\end{aligned}
\tag{2.24}
$$

Since the filtering operation presented in Equation 2.24 does not explicitly require the eigendecomposition of the graph Laplacian, it is often referred to spectrum-free [5]. Also note that in the last step of Equation 2.24 we used the fact that $\boldsymbol{\Delta}^k = \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^\top$. This can be shown by induction by noting that $\boldsymbol{\Delta} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top$, and that if $\boldsymbol{\Delta}^k = \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^\top$, then:

$$
\begin{aligned}
\boldsymbol{\Delta}^{k+1} &= \boldsymbol{\Delta}^k \boldsymbol{\Delta} \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^\top \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{\Lambda}\boldsymbol{U}^\top \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^{k+1}\boldsymbol{U}^\top.
\end{aligned}
\tag{2.25}
$$

The parametrization presented in Equation 2.24 can be used to build a basic polynomial spectral GNN. To achieve this we take inspiration on convolutional neural networks (CNN), where a layer consists of: i) filtering operations; ii) a summation over channels; and iii) a nonlinear activation function. With this in mind, consider a graph signal $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ with d channels, and let $\boldsymbol{H}^{(0)} := \boldsymbol{X}$. At layer ℓ, a basic polynomial spectral GNN computes:

$$
\boldsymbol{H}^{(\ell)} = \phi\left(\sum_{k=0}^{K} \boldsymbol{\Delta}^k \boldsymbol{H}^{(\ell-1)} \boldsymbol{\Theta}_k^{(\ell)}\right). \tag{2.26}
$$

In Equation 2.26, ϕ is a nonlinearity, and $\boldsymbol{\Theta}^{(\ell)} \in \mathbb{R}^{d \times d}$ are the coefficients of the spectral filters. From the formulation in Equation 2.26 we can see that it contains a filtering operation resembling that of Equation 2.24, a summation over the channels by using $\boldsymbol{\Theta}^{(\ell)}$, and a nonlineaar activation function ϕ.

Despite its intuitive nature, the model in Equation 2.26 was only hinted at by [21], but was never explored in detail. The closest model in the literature is ChebNet [21] that replaces the monomials over $\boldsymbol{\Delta}$ in Equation 2.26 by Chebyshev polynomials of a transformed Laplacian $\tilde{\boldsymbol{\Delta}} = (2/\lambda_{max})\boldsymbol{\Delta} - \boldsymbol{I}$. The idea was to exploit the orthogonality of Chebyshev polynomials to obtain more stable filters [21].

Recent works on spectral GNNs aim to improve ChebNets and GCNs by increasing model flexibility with rational filters. Nonetheless, the exact computation of these filters involves matrix inversion. CayleyNets [22] avoid this problem by solving a sequence of linear systems with the Jacobi method. [23] propose a design that approximates autoregressive moving average filters based on recursions derived by [24].

2.3.3. Literature review

In the following section we introduce GNN methods that we reference throughout the manuscript. We present a brief description of each model along with the update operation used in the respective GNN layers.

LapGCN. We refer to the basic formulation for a polynomial spectral GNN presented in Equation 2.26 as LAPGCN. Even though this specific formulation has not been explored in detail in the literature, it is an important formulation as we use it as a base for our model. A LapGCN model consists of several layers of Equation 2.26, with K being a hyperparameter of the model.

ChebNet [21]. As previously mentioned, ChebNet is a polynomial spectral GNN that replaces the formulation in Equation 2.26 by replacing the monomials over $\boldsymbol{\Delta}$ by Chebyshev polynomials of a transformed Laplacian. The formulation for ChebNet is given by the following equation.

$$\boldsymbol{H}^{(\ell)} = \phi\left(\sum_{k=0}^{K} T_k\left(\tilde{\boldsymbol{\Delta}}\right)\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}_k^{(\ell-1)}\right). \tag{2.27}$$

$\tilde{\boldsymbol{\Delta}}$ in Equation 2.27 is a transformed Laplacian given by $\tilde{\boldsymbol{\Delta}} = (2/\lambda_{max})\boldsymbol{\Delta} - \boldsymbol{I}$, with λ_{max} the largest eigenvalue of $\boldsymbol{\Delta}$. $\boldsymbol{\Theta}_k^{(\ell)}$ is a matrix of learnable weights for the ℓ-th layer of the model. $T_k(\cdot)$ is the k-th order Chebyshev polynomial, given by the following recursion.

$$T_k\left(\tilde{\boldsymbol{\Delta}}\right) = \begin{cases} \boldsymbol{I} & k = 0 \\ \tilde{\boldsymbol{\Delta}} & k = 1 \\ 2\tilde{\boldsymbol{\Delta}}T_{k-1}\left(\tilde{\boldsymbol{\Delta}}\right) - T_{k-2}\left(\tilde{\boldsymbol{\Delta}}\right) & \sim . \end{cases} \tag{2.28}$$

Using the Chebyshev polynomials instead of the monomials used in LapGCN has the objective of exploiting the orthogonality of the Chebyshev polynomials to get more stable filters.

CayleyNets [22]. The work proposes using a family of filters that is more general than that of ChebNets, known as Cayley filters. The work argues that using this more general family of filters gives the filters the capacity of focusing in specific frequency bands, along with more stability and regularity of the filters.

A single layer of CayleyNets can be represented by the following equation.

$$\boldsymbol{H}^{(\ell)} = \boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}_0^{(\ell-1)} + 2\mathrm{Re}\left(\sum_{k=1}^{K}\left(h\boldsymbol{\Delta} - j\boldsymbol{I}\right)^k\left(h\boldsymbol{\Delta} + j\boldsymbol{I}\right)^{-k}\boldsymbol{H}^{(\ell-1)}\boldsymbol{\Theta}_k^{(\ell-1)}\right). \tag{2.29}$$

In Equation 2.29, $\boldsymbol{\Theta}_k^{(\ell-1)}$ are complex matrices of learnable weights, h is a learnable scalar used as the zoom of the filter and $\mathrm{Re}(\cdot)$ is the real part of a complex value. The layer presented in Equation 2.29 is not explicitly computed due to its computational complexity. An approximation of the layer is computed by using the Jacobi method.

ARMA [23]. The work proposes the use of auto-regressive moving average (ARMA) filters in GNNs. This family of filters is characterized by the following representation.

$$h_K(\lambda) = \frac{\sum_{k=0}^{K} p_k \lambda^k}{1 + \sum_{k=1}^{K} q_k \lambda^k}. \tag{2.30}$$

The denominator contains the auto-regressive part of the filter, while the numerator contains the moving average part of the filter. Note that by setting $q_k = 0$ in Equation 2.30, the representation becomes a polynomial filter. Thus, polynomial filters are a particular case of ARMA filters. The work argues that including the auto-regressive term in the filter makes it more robust to noise, and captures longer dependencies and global structures better than polynomial filters of the same degree.

Implementing the filter in Equation 2.30 directly into a graph filter would result in the following representation.

$$\boldsymbol{H}^{(\ell)} = \left(\boldsymbol{I} + \sum_{k=1}^{K} q_k \boldsymbol{\Delta}^k\right)^{-1} \left(\sum_{k=0}^{K} p_k \boldsymbol{\Delta}^k\right) \boldsymbol{H}^{(\ell-1)}. \tag{2.31}$$

However, the matrix inversion in Equation 2.31 is slow, making the formulation impractical. The work proposes approximating an ARMA filter of degree $K = 1$ by iteratively computing $\bar{\boldsymbol{H}}^{(\ell+1)} = a\boldsymbol{M}\boldsymbol{H}^{(\ell)} + b\boldsymbol{H}^{(0)}$ until convergence, with $\boldsymbol{M} = \boldsymbol{I} - \boldsymbol{\Delta}$. This results in a filter with frequency response of $h(\lambda) = b/(1 - a\lambda)$. By combining K of these filters, we can approximate an ARMA filter of degree K, as shown in the following equation.

$$h_K(\lambda) = \sum_{k=1}^{K} \frac{b_k}{1 - a_k\lambda}. \tag{2.32}$$

Applying the same idea of Equation 2.32 to a GNN, a layer of the ARMA filter is given by the following equations.

$$\begin{aligned} \boldsymbol{H}_k^{(t+1)} &= \phi\left(\boldsymbol{M}\boldsymbol{H}_k^{(t)}\boldsymbol{\Theta}_k + \boldsymbol{H}_k^{(0)}\bar{\boldsymbol{\Theta}}_k\right), \\ \boldsymbol{H} &= \frac{1}{K}\sum_{k=1}^{K} \boldsymbol{H}_k^{(T)}. \end{aligned} \tag{2.33}$$

In Equation 2.33 we use $\boldsymbol{H}$ to denote the updated features for the graph, ϕ denotes an activation function, $\boldsymbol{\Theta}_k$ and $\bar{\boldsymbol{\Theta}}_k$ are matrices of learnable weights that are shared across each parallel filter, and T denotes a fix number of iterations that each of the parallel filters uses.

GCNII [25]. Graph Convolutional Network via Initial residual and Identity mapping

(GCNII) looks to extend GCN to simulate polynomial filters with arbitrary coefficients. To do this, the model uses initial residual connections and identity mapping. The former consists of adding a residual connection to the initial representation, while the latter adds an identity to the weight used in the feature transformation of the model. With this in mind, the GCNII model can be characterized by the following equation.

$$\boldsymbol{H}^{(\ell+1)} = \phi\left(\left((1-\alpha_\ell)\boldsymbol{\Delta}\boldsymbol{H}^{(\ell)} + \alpha_\ell \boldsymbol{H}^{(0)}\right)\left((1-\beta_\ell)\boldsymbol{I} + \beta_\ell \boldsymbol{\Theta}\right)\right). \tag{2.34}$$

In Equation 2.34, α_ℓ and β_ℓ are learnable parameters that control the effect of the initial residual connection and identity mapping respectively. $\boldsymbol{\Delta} = \boldsymbol{L}_{norm}$, ϕ is an activation function, and $\boldsymbol{\Theta}$ is a matrix of learnable weights. It can be seen from the equation that if $\alpha_\ell = 0$ and $\beta_\ell = 0$ the model simply becomes GCN. As the value of α_ℓ increases, the output representation considers more information from the input representation, which helps to deal with oversmoothing. As the value β_ℓ decreases, the effect of the learnable weights is replaced by an identity matrix, which helps to decrease the interactions between different dimensions of the features, and helps with regularization of the model.

S^{2}GC [26]. The Simple Spectral Graph Convolution (S^{2}GC) uses a modified Markov diffusion kernel to propose a model that can aggregate over large neighborhoods of a node, while limiting the effects of oversmoothing. The model consists of a single layer that is characterized by the following equation.

$$\boldsymbol{Y} = \text{softmax}\left(\frac{1}{K}\sum_{k=0}^{K}\boldsymbol{\Delta}^k \boldsymbol{H}\boldsymbol{\Theta}\right). \tag{2.35}$$

In Equation 2.35, we use $\boldsymbol{Y}$ to denote the output of the model, $\boldsymbol{\Theta}$ is a matrix of learnable weights, and $\boldsymbol{\Delta} = \boldsymbol{L}_{norm}$.

GIN [27]. The Graph Isomorphism Network (GIN) is a model built with the objective of being expressive. The model is as powerful as the Weisfeler-Lehman graph isomorphism test. A single layer of GIN is given by the following equation.

$$\boldsymbol{h}_u^{(\ell+1)} = \text{MLP}^{(\ell)}\left(\left(1+\epsilon^{(\ell)}\right)\boldsymbol{h}_u^{(\ell)} + \sum_{v\in\mathcal{N}(u)}\boldsymbol{h}_v^{(\ell)}\right). \tag{2.36}$$

The update in Equation 2.36 is applied to every node $u \in \mathcal{G}$.

GAT [28]. Graph Attention Networks (GAT) introduced self-attention to GNNs. By having layers in which the nodes can attend over their neighborhoods' features, the model can learn to weight different neighbors. A GAT layer can be characterized by the following equations.

$$
\begin{aligned}
e_{uv}^{(\ell)} &= \phi\left(\boldsymbol{a} \cdot [\boldsymbol{\Theta} \boldsymbol{h}_u^{(\ell)} || \boldsymbol{\Theta} \boldsymbol{h}_v^{(\ell)}]\right), \\
\alpha_{uv}^{(\ell)} &= \frac{\exp\left(e_{uv}^{(\ell)}\right)}{\sum_{w \in \mathcal{N}(u)} \exp\left(e_{uw}^{(\ell)}\right)}, \\
\boldsymbol{h}_u^{(\ell+1)} &= \sigma\left(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(\ell)} \boldsymbol{h}_v^{(\ell)}\right).
\end{aligned} \tag{2.37}
$$

In Equation 2.37, $e_{uv}^{(\ell)}$ is the coefficient that was computed for the edge going from node u to node v, which is later used to compute the attention weight $\alpha_{uv}^{(\ell)}$. $||$ indicates the concatenation operator. $\boldsymbol{\Theta}$ and $\boldsymbol{a}$ are a matrix and a vector of learnable weights respectively. $\phi(\cdot)$ and $\sigma(\cdot)$ are activation functions. The original work uses LeakyReLU as function ϕ.

DAGNN [29]. Deep Adaptive Graph Neural Network (DAGNN) is a model that decouples the representation transformation and the propagation steps of GNN, with the objective of dealing with oversmoothing, and incorporating information from large receptive fields. The model proposed in DAGNN is represented by the following equations.

$$
\begin{aligned}
\boldsymbol{H}_k &= \boldsymbol{\Delta}^k \text{MLP}(\boldsymbol{H}), \\
\bar{\boldsymbol{H}} &= [\boldsymbol{H}_0 ||, \ldots, || \boldsymbol{H}_K], \\
\boldsymbol{S} &= \phi\left(\boldsymbol{H} \boldsymbol{s}\right), \\
\boldsymbol{Y} &= \text{softmax}\left(\boldsymbol{S} \boldsymbol{H}\right).
\end{aligned} \tag{2.38}
$$

In Equation 2.38, $\boldsymbol{\Delta} = \boldsymbol{L}_{norm}$, ϕ is an activation function, $\boldsymbol{s}$ is a learnable projection vector and $\boldsymbol{Y}$ is the output of the model. The decoupling proposed in DAGNN can be seen in the fact that the transformation MLP($\boldsymbol{H}$) only contains node information, and the propagation step (multiplying by $\boldsymbol{\Delta}^k$) is separate from the aforementioned representation transformation.

MixHop [30]. The work proposes a model that aims to learn a general class of neighborhood mixing relationships by explicitly mixing neighborhoods of different sizes. To do this, the model concatenates representations resulting from aggregating multi-hop neighborhoods independently. A single layer of MixHop is represented by the following equation.

$$
\boldsymbol{H}^{(\ell+1)} = \Big\Vert_{k=0}^{K} \phi\left(\boldsymbol{A}^k \boldsymbol{H}^{(\ell)} \boldsymbol{\Theta}\right). \tag{2.39}
$$

In Equation 2.39 $\|$ represents the concatenation operator, where each of the outputs will be concatenated to each other, $\boldsymbol{A}$ indicates the adjacency matrix, ϕ is an activation function and $\boldsymbol{\Theta}$ is a matrix of learnable weights.

2.4. Dynamic graphs

Real-world graphs, such as social networks, often evolve over time through addition, deletion and updating of nodes and edges. To take these types of graphs into account we introduce the notion of dynamic graph, which is simply a graph that evolves over time. We will consider a particular case of dynamic graphs, known as continuous-time dynamic graphs (CTDGs) which can be represented by a sequence of timestamped events. As a simplification we consider a fixed set of nodes $\mathcal{V}$ and edge events that modify the set of edges $\mathcal{E}$.

Each edge in a CTDG is represented by a tuple $(u, v, \boldsymbol{e}, t)$, where $u, v \in \mathcal{V}$ are the nodes that are interacting in the event, $\boldsymbol{e} \in \mathbb{R}^m$ is a feature that represents the particular event and $t \in \mathbb{R}^+$ is the timestamp of the event. The set of edges is then represented by a finite sequence of events, $\mathcal{E} = ((u_i, v_i, \boldsymbol{e}^{(i)}, t_i))_{i=1}^K$, where K is the total number of events in the graph. We assume that the events are temporally ordered, which means that $t_{i-1} \leq t_i$ for all $1 < i \leq K$.

Since the number of events is finite, we can look at the dynamic graph by looking at it as a sequence of multigraphs, a generalization of graphs that considers a multiset of edges. To do this we define $\mathcal{E}(t) = \{(u_i, v_i, \boldsymbol{e}^{(i)}, t_i) : t_i \leq t\}$ the events up to a timestamp $t \in \mathbb{R}^+$. We also define $\mathcal{E}(t-) = \{(u_i, v_i, \boldsymbol{e}^{(i)}, t_i) : t_i < t\}$ the events before a timestamp $t \in \mathbb{R}^+$. The multigraph can then be defined as $\mathcal{G}(t) = (\mathcal{V}, \mathcal{E}(t))$ similarly to what was done for static graphs in Section 2.1. The reason that $\mathcal{G}(t)$ is considered a multigraph is because there might be multiple events that consider the same pair of nodes in $\mathcal{E}(t)$. With the previous definitions in mind, we can represent a CTDG as the finite sequence of multigraphs $(\mathcal{G}(t_1), \mathcal{G}(t_2), \ldots \mathcal{G}(t_K))$.

Analogous to the definition in static graphs, we denote $v \in \mathcal{V}$ to indicate the node v is in the graph and $(u, v) \in \mathcal{E}(t)$ to indicate that there is an edge between $u, v \in \mathcal{V}$ in multigraph $\mathcal{G}(t)$. The number of nodes and edges in the multigraph is denoted by $n = |\mathcal{V}|$ and $m(t) = |\mathcal{E}(t)|$. We define the concept of temporal neighborhood for a node u at a timestamp $t \in \mathbb{R}^+$ as the set of neighbors of u in multigraph $\mathcal{G}(t)$; or formally as $\mathcal{N}(u, t) = \{v : (u, v) \in \mathcal{E}(t)\}$.

2.5. Temporal Graph Neural Networks

In the following section we will introduce temporal-graph neural networks (T-GNNs). The section is not a thorough survey of T-GNN models, and is merely used to introduce basic concepts of these models along with a brief literature review of methods referenced throughout the work.

T-GNNs are neural networks built for dynamic graphs. Since the work focuses on CTDGs, we will only consider T-GNNs built for this type of dynamic graphs. Further, we will use T-GNNs to refer only to models built for CTDGs.

For an event $(u, v, \boldsymbol{e}, t)$, T-GNNs usually follow the same three steps: *i*) temporal sampling, *ii*) neighborhood aggregation, *iii*) state update. With this in mind, we propose a framework that looks to capture T-GNNs by considering these steps. Figure 2.2 illustrates the process

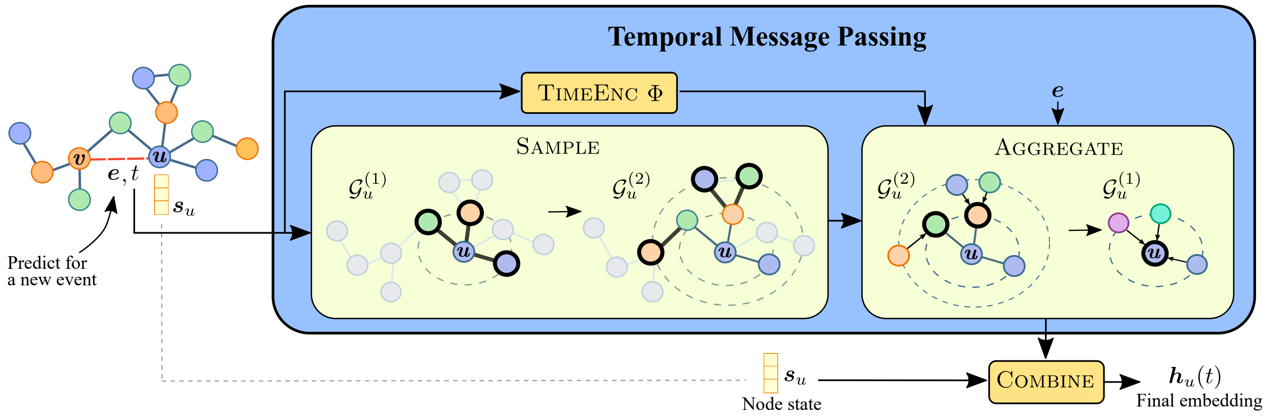


Figure 2.2: **T-GNNs: general framework to compute temporal node representations.** For clarity, we illustrate an example for node u with $L = 2$. To predict for an event between nodes u and v at time t, T-GNNs (i) iteratively sample a L-hop temporal neighborhood of u (Sample), (ii) encode the continuous timestamps (TimeEnc), (iii) iteratively aggregate the neighborhood information (Aggregate), and (iv) combine the resulting neighborhood information vector with the state vector to obtain the final embedding for node u (Combine).

to compute the embedding for a node u in the framework.

Temporal message passing. Computing the embeddings begins with temporal sampling and neighborhood aggregation, which we combine into a single step called *temporal message passing*. Temporal sampling involves iteratively selecting temporal neighbors to build a graph. Then, neighborhood aggregation takes the sampled graph and aggregates neighbors iteratively to compute an output graph. Let Φ be a time encoder used to generate time embeddings for each event and let $\mathcal{G}_u^{(0)}$ be a singleton graph, containing only node u and no edges. In general, the updates that a T-GNN with L layers does to extract an embedding for node u go as follows:

$$\mathcal{G}_u^{(l)} \leftarrow \textsc{Sample}(\mathcal{G}_u^{(l-1)}), \quad \forall l = 1, \ldots, L, \tag{2.40}$$

$$\tilde{\mathcal{G}}_u^{(0)} \leftarrow \mathcal{G}_u^{(L)}, \tag{2.41}$$

$$\tilde{\mathcal{G}}_u^{(l)} \leftarrow \textsc{Aggregate}(\tilde{\mathcal{G}}_u^{(l-1)}, \Phi, \boldsymbol{e}), \quad \forall l = 1, \ldots, L. \tag{2.42}$$

Sample$(\cdot)$ in Equation 2.40 is the sampling function and $\mathcal{G}_u^{(l)}$ is the graph obtained after l layers of sampling. Similarly, $\tilde{\mathcal{G}}_u^{(l)}$ denotes the graph after l steps of aggregation, i.e., Aggregate$(\cdot)$ in Equation 2.42. Since aggregation reduces the number of nodes, the graphs follow $\tilde{\mathcal{G}}_u^{(l)} \subseteq \tilde{\mathcal{G}}_u^{(l-1)} ... \subseteq \tilde{\mathcal{G}}_u^{(0)}$, for all $l = 1, \ldots, L$.

After aggregation, T-GNNs use a readout function to compute an embedding from graph $\tilde{\mathcal{G}}_u^{(L)}$. Combining this embedding with the node state $\boldsymbol{s}_u$, we obtain the output embedding $\boldsymbol{h}_u$ for prediction purposes. These steps are summarized as follows:

$$\boldsymbol{z}_u(t) = \textsc{Readout}(\tilde{\mathcal{G}}_u^{(L)}), \qquad \boldsymbol{h}_u(t) = \textsc{Combine}(\boldsymbol{s}_u, \boldsymbol{z}_u(t)). \tag{2.43}$$

Recursive state update. Once the output embedding is computed, the state of node u is updated. This is done to keep information about the node's previous interactions in its state vector. To capture this information the step must use the node's previous state $\boldsymbol{s}_u$, the state of the node its interacting with $\boldsymbol{s}_v$, the information about the interaction $\boldsymbol{e}$ and the timestamp of the interaction t. The state is then updates as follows:

$$\boldsymbol{s}_u \leftarrow \textsc{Update}(\boldsymbol{s}_u, \boldsymbol{s}_v, \boldsymbol{e}, t). \tag{2.44}$$

2.5.1. Literature review

In the following section we introduce T-GNN methods that we use throughout the work. We present a brief description for each method, along with a description of how it fits in the framework presented in Section 2.5.

JODIE [31]. Joint Dynamic User-Item Embedding Model (JODIE) is a model that looks to learn embedding trajectories for the nodes in the graph. Compared to learning static representation for the nodes in the graph, learning trajectories let us the model project a future embedding for an arbitrary time since its last interaction.

JODIE also assumes that the input graph is bipartite. It considers two sets of nodes denoted as "users" and "items". The embeddings for each of the set of nodes is computed using different weights.

Since JODIE learns embedding trajectories for each node it does not use the temporal message passing introduced in the T-GNN framework. Similarly since there is no temporal message passing, the readout function is not needed. The combine function can be used to compute the projection used for the prediction, which is defined as follows.

$$\boldsymbol{h}_v(t) = (1 + \boldsymbol{p}) \odot \boldsymbol{s}_v(t_v). \tag{2.45}$$

In Equation 2.45, t is the time that we are predicting at, t_v is the time of the last interaction involving node v, $\odot$ is the Hadarmard product, and $\boldsymbol{p}$ is a time-context vector that is computed as $\boldsymbol{p} = \boldsymbol{w}(t - t_v)$, with $\boldsymbol{w}$ a vector of learnable weights. The state vector is then updated by using the following equations.

$$\begin{aligned} \boldsymbol{s}_u(t) &= \phi\left(\boldsymbol{\Theta}_1^u \boldsymbol{s}_u(t^-) + \boldsymbol{\Theta}_2^u \boldsymbol{s}_i(t^-) + \boldsymbol{\Theta}_3^u \boldsymbol{e} + \boldsymbol{\Theta}_4^u (t - t_u)\right), \\ \boldsymbol{s}_i(t) &= \phi\left(\boldsymbol{\Theta}_1^i \boldsymbol{s}_i(t^-) + \boldsymbol{\Theta}_2^i \boldsymbol{s}_u(t^-) + \boldsymbol{\Theta}_3^i \boldsymbol{e} + \boldsymbol{\Theta}_4^i (t - t_i)\right). \end{aligned} \tag{2.46}$$

In Equation 2.46, u and i indicate the user and item nodes respectively, $\boldsymbol{\Theta}$ are learnable weights, with the superscript indicating whether the weights are for the user or item node updates, and t^- indicates the instant right before timestamp t.

DyRep [32]. The work proposes a model that looks to jointly capture the evolution of the graph's structure and interactions between nodes in the graph. We begin the description of the model by introducing the temporal message passing layer used in DyRep, which is denominated temporally attentive aggregation.

Temporally attentive aggregation takes the 1-hop neighborhood of a node u and aggregates the information according to the following equation.

$$\begin{aligned}\alpha_{uv} &= \frac{\exp(\boldsymbol{S}_{uv}(t^-))}{\sum_{w\in\mathcal{N}(u)}\exp(\boldsymbol{S}_{uw}(t^-))},\\ \boldsymbol{h}_v(t^-) &= \boldsymbol{\Theta}_h\boldsymbol{s}_v(t^-)+\boldsymbol{b}_h,\\ \boldsymbol{h}_u(t) &= \max\left(\{\alpha_{uv}\boldsymbol{h}_v(t^{-1}),\forall v\in\mathcal{N}(u)\}\right).\end{aligned} \tag{2.47}$$

In Equation 2.47, Θ_h is a matrix of learnable weights, $\boldsymbol{b}_h$ is a vector with a learnable bias term, $\max(\cdot)$ indicates the element-wise maximum operator, and $\boldsymbol{S}$ is a complex stochastic matrix that captures complex temporal information for the graph. The matrix $\boldsymbol{S}$ is used to attend over the neighbors of node u. Initially, the values for $\boldsymbol{S}$ are initialized as $\boldsymbol{S}_{uv}=0$ if $\boldsymbol{A}_{uv}=0$, and $\boldsymbol{S}_{uv}=1/|\mathcal{N}(u)|$ if $\boldsymbol{A}_{uv}=1$. The values are updated whenever a new edge is created between two nodes or when an interaction happens between two nodes in the graph. In both cases, the value for $\boldsymbol{S}_{uv}$ is updated proportionally to the events intensity.

With the formulation presented in Equation 2.47 we have the output of the combination step presented in the T-GNN framework. Considering this, only the update step needs to be described. The update for DyRep is given by the following equation.

$$\boldsymbol{s}_u(t)=\phi\left(\boldsymbol{\Theta}_n\boldsymbol{h}_v(t)+\boldsymbol{\Theta}_s\boldsymbol{s}_u(t_u)+\boldsymbol{\Theta}_t(t-t_u)\right). \tag{2.48}$$

In Equation 2.48, $\boldsymbol{\Theta}$ indicate learnable weights, $\boldsymbol{h}_v$ is the feature calculated in Equation 2.47 for the node that u is interacting with, and t_u is the time for the last event including node u.

TGAT [33]. Temporal Graph Attention Network (TGAT) introduces temporal graph attention layer for dynamic networks. We begin the description of the model by describing its temporal message passing.

The sampling function used in TGAT is a uniform sampling of the node's neighbors. Then, the aggregation is the temporal graph attention layer, which can be described by the following equations.

$$
\begin{aligned}
\boldsymbol{Z}_q^{(\ell)}(t) &= [\boldsymbol{h}_u^{(\ell)} || \Phi_d(0)], \\
\boldsymbol{Z}^{(\ell)}(t) &= [\boldsymbol{h}_1^{(\ell)} || \Phi_d(t - t_1), \boldsymbol{h}_2^{(\ell)}, \dots, \boldsymbol{h}_N^{(\ell)} || \Phi_d(t - t_N)], \\
\boldsymbol{Q}^{(\ell)}(t) &= \boldsymbol{Z}_q^{(\ell)}(t) \boldsymbol{\Theta}_Q, \\
\boldsymbol{K}^{(\ell)}(t) &= \boldsymbol{Z}^{(\ell)}(t) \boldsymbol{\Theta}_K, \\
\boldsymbol{V}^{(\ell)}(t) &= \boldsymbol{Z}^{(\ell)}(t) \boldsymbol{\Theta}_V, \\
\Phi_d(t) &= \sqrt{\frac{1}{d}} [\cos(\omega_1 t), \sin(\omega_1 t), \dots, \cos(\omega_d t), \sin(\omega_d t)], \\
\boldsymbol{h}_u^{(\ell+1)} &= \text{Attn}\left(\boldsymbol{Q}^{(\ell)}(t), \boldsymbol{K}^{(\ell)}(t), \boldsymbol{V}^{(\ell)}(t)\right).
\end{aligned}
\tag{2.49}
$$

In Equation 2.49, $\boldsymbol{h}_u^{(\ell)}$ is the information of the target node after ℓ layers of temporal attention, $\{\boldsymbol{h}_k^{(\ell)}\}_{k=1}^N$ are the features of the neighbors sampled from u, N is a hyperparameter indicating the number of neighbors to sample, $\boldsymbol{\Theta}$ are learnable weight matrices, ω_k are learnable parameters and Attn indicates the self-attention layer. The initial value for the features is given by $\boldsymbol{h}_v^{(0)} = \boldsymbol{s}_v$. After a fixed number L of attention layers the output feature is $\boldsymbol{h}_u = \boldsymbol{h}_u^{(L)}$. There is no update operation for TGAT since the features are static for the model.

TGN [34]. Temporal Graph Networks (TGN) introduces a generic model for T-GNNs. The work provides various alternatives for different components of the model. In this work we refer to the TGN-attn model, since it has the best results in practice.

We begin TGN's description by considering its temporal message passing. The sampling step of the model takes the last N nodes that the node u has interacted with. Then, the aggregation step uses a modified version of the temporal attention presented in Equation 2.49. This aggregation step is characterized by the following equations.

$$
\begin{aligned}
\boldsymbol{Z}_q^{(\ell)}(t) &= [\boldsymbol{h}_u^{(\ell)} || \Phi_d(0)], \\
\boldsymbol{Z}^{(\ell)}(t) &= [\boldsymbol{h}_1^{(\ell)} || \boldsymbol{e}_{u1} || \Phi_d(t - t_1), \boldsymbol{h}_2^{(\ell)}, \dots, \boldsymbol{h}_N^{(\ell)} || \boldsymbol{e}_{uN} || \Phi_d(t - t_N)], \\
\boldsymbol{Q}^{(\ell)}(t) &= \boldsymbol{Z}_q^{(\ell)}(t), \\
\boldsymbol{K}^{(\ell)}(t) &= \boldsymbol{V}^{(\ell)}(t) = \boldsymbol{Z}^{(\ell)}(t), \\
\Phi_d(t) &= \sqrt{\frac{1}{d}} [\cos(\omega_1 t), \sin(\omega_1 t), \dots, \cos(\omega_d t), \sin(\omega_d t)], \\
\tilde{\boldsymbol{h}}_u^{(\ell+1)} &= \text{MultiHeadAttn}\left(\boldsymbol{Q}^{(\ell)}(t), \boldsymbol{K}^{(\ell)}(t), \boldsymbol{V}^{(\ell)}(t)\right), \\
\boldsymbol{h}_u^{(\ell+1)} &= \text{MLP}\left(\boldsymbol{h}_u^{(\ell)} || \tilde{\boldsymbol{h}}_u^{(\ell+1)}\right).
\end{aligned}
\tag{2.50}
$$

The main modifications presented in Equation 2.50 with respect to Equation 2.49 is changing the self-attention layer for a multi head attention, including the edge features $\boldsymbol{e}_{uk}$ in the representation, and adding a residual connection with the previous layer. Similarly to TGAT, after a fixed number L of attention layers, the output feature is $\boldsymbol{h}_u = \boldsymbol{h}_u^{(L)}$.

The update operation uses a gated recurrent unit (GRU) as shown in the following equation.

$$\begin{aligned} \boldsymbol{m}_u(t) &= [\boldsymbol{s}_v(t^-)||\boldsymbol{e}_{uv}||(t - t_u)], \\ \boldsymbol{s}_u(t) &= \mathrm{GRU}(\boldsymbol{s}_u(t^-), \boldsymbol{m}_u(t)). \end{aligned} \tag{2.51}$$

In Equation 2.51, $\boldsymbol{s}_v(t^-)$ is the state of node v (the node u is interacting with) right before the interaction, $\boldsymbol{e}_{uv}$ is the feature describing the interaction, and t_u is the time of node u's last interaction.

CAW [35]. Causal Anonymous Walks (CAW) look to inductively represent dynamic graphs by using temporal random walks to characterize network dynamics. An explanation of the method is presented in Section B.1.

Chapter 3

Methods

In the following chapter we introduce two methods, Polynomial Subspace Net (PSN) for static graphs and Online Graph Nets (OGN) for dynamic graphs. We present complexity analysis and theoretical properties for both methods.

3.1. Polynomial Subspace Net (PSN)

Unlike recent spectral GNNs [22, 23] which focus on increasing filter flexibility, our work pursues a much simpler design for polynomial filters. We observe that these spectral GNNs often achieve higher performance with low-order polynomials (usually $K \leq 5$). This suggests that the success of these networks may not come from sharpening filter responses with flexible parameterizations. Rather, we believe the benefits stem from design choices that insert positive inductive biases.

In this section, we introduce *Polynomial Subspace Net* (PSN), a spectral method that builds on the simplest formulation of polynomial spectral GNNs (Equation 2.26). Our most important design choice is to share parameters across monomials in Equation 2.26, i.e., setting $\boldsymbol{\Theta}_0^{(\ell)} = \boldsymbol{\Theta}_1^{(\ell)} = \cdots = \boldsymbol{\Theta}_K^{(\ell)}$. We further compensate this drop in flexibility in a minimalistic way by adding a single scalar parameter (coefficient) to every monomial. That is, our main building block is the equation

$$\boldsymbol{H}^{(\ell)} \;=\; \phi\left(\left(\sum_{k=0}^{K} \theta_k^{(\ell)} \boldsymbol{\Delta}^k \boldsymbol{H}^{(\ell-1)}\right)\boldsymbol{W}^{(\ell)}\right). \tag{3.1}$$

where $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{d\times d}$ is a parameter matrix, and $\theta_k^{(\ell)} \in \mathbb{R}$ are scalar parameters. This formulation can be seen as the application of a basic polynomial filter $\sum_{k=0}^{K} \theta_k^{(\ell)} \boldsymbol{\Delta}^k$ to all input channels in a graph signal, and then combining the channel information with a linear transformation $\boldsymbol{W}^{(\ell)}$.

Our simple filtering operation is different from standard spectral nets which employ multiple distinct filters for each input channel. As an immediate consequence, PSN filters have significantly fewer parameters compared to, e.g., ChebNets or CayleyNets: while ChebNet and CayleyNet convolutional layers have $K \times d^2$ parameteres, a PSN layer has only $K + d^2$ parameters (see Table 3.1).

3.1.1. Further restrictions over the base filter

We introduce two additional restrictions to the scalar coefficients motivated by (i) mimicking *residual connections* that have been proved useful in graph learning with deep networks, and (ii) restricting filter coefficient to a closed interval to obtain numerical stability. For achieving this, we use a reparameterization of the simple filter $\sum_{k=0}^{K} \theta_k^{(\ell)} \boldsymbol{\Delta}^k$ as

$$\tilde{\sigma}^{(\ell)} \boldsymbol{I} + (1 - \tilde{\sigma}^{(\ell)}) \sum_{k=1}^{K} \tilde{\theta}_k^{(\ell)} \boldsymbol{\Delta}^k. \tag{3.2}$$

in which we force $\tilde{\sigma}^{(\ell)}$ to be in the interval $[0, 1]$, and $\tilde{\theta}_1^{(\ell)}, \ldots, \tilde{\theta}_K^{(\ell)}$ to be in $[-1, 1]$. In practice, we implement these restrictions by setting $\tilde{\sigma}^{(\ell)} := \text{sigmoid}(\theta_0^{(\ell)})$, and $\tilde{\theta}_k^{(\ell)} := \tanh(\theta_k^{(\ell)})$.

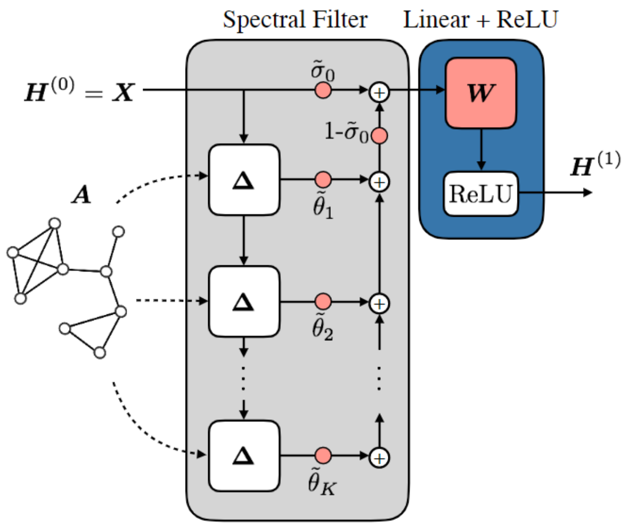


Figure 3.1: Polynomial Subspace Nets (PSNs). For clarity, we only denote the first layer ($\boldsymbol{H}^{(0)} = \boldsymbol{X}$) and omit layer indexes in the model parameters.

The reparameterization using $\tilde{\sigma}^{(\ell)}$ works as a *gate* that allows the filter to decide how much it should weight the original input signal at each layer. This effectively acts as an inner-layer residual connection [36]. Regarding the restriction for the higher-order filter coefficients to be in the interval $[-1, 1]$, we empirically show that this makes the learning process more stable (Section 4.2.2).

Figure 3.1 shows the block diagram for a PSN layer. We split the computation of the PSN convolution into two steps: (i) filtering the input signal in the spectral domain; and (ii) mixing the filtered signals using a linear layer and a non-linearity. While we can understand the linear layer as part of the spectral filter, it is useful to treat it separately since it is agnostic to the graph structure.

Implementation. To implement a PSN layer, we first compute the sequence $\boldsymbol{H}_1^{(\ell)}, \ldots, \boldsymbol{H}_K^{(\ell)}$ as $\boldsymbol{H}_k^{(\ell)} := \boldsymbol{\Delta} \boldsymbol{H}_{k-1}^{(\ell)}$, in which $\boldsymbol{H}_0^{(\ell)} = \boldsymbol{H}^{(\ell-1)}$. We can compute this sequence using either sparse-dense matrix multiplications or a message-passing architecture. We then use this sequence to carry out the spectral filtering operation:

$$\boldsymbol{S}^{(\ell)} := \text{sigmoid}(\theta_0^{(\ell)}) \boldsymbol{H}_0^{(\ell)} + \left(1 - \text{sigmoid}(\theta_0^{(\ell)})\right) \sum_{k=1}^{K} \tanh\left(\theta_k^{(\ell)}\right) \boldsymbol{H}_k^{(\ell)}, \tag{3.3}$$

and finally compute $\boldsymbol{H}^{(\ell+1)} := \text{ReLU}(\boldsymbol{S}^{(\ell)} \boldsymbol{W}^{(\ell)})$.

Restricted polynomials and expressiveness. It may seem that the restrictions that we impose over polynomial coefficients in Equation 3.2 may affect the expressiveness of our resulting filter. Nevertheless, the next result shows that we can reconstruct any possible polynomial filter.

Proposition 1 For every polynomial $P(\lambda)$ with real-valued coefficients, there exist a constant m and polynomial $R(\lambda)$ of the form $b + (1-b)c_1\lambda + (1-b)c_2\lambda^2 + \cdots + (1-b)c_K\lambda^K$with $b \in [0,1]$ and $c_i \in [-1,1]$, such that $P(\lambda) = mR(\lambda)$.

A proof of this proposition is provided in Section A.3. Since the $[0,1]$ and $[-1,1]$ intervals are fully covered by sigmoid$(\cdot)$ and tanh$(\cdot)$, respectively, and since we are allowing a linear transformation after the filter, then we know that with proper parameter values, our model will be able to represent any general filter. The proof of Proposition 1 is in the Annex (Section A.3). In Section 4.2.3, we show the flexibility of the responses that PSN filter can produce.

PSNs as low-rank spectral GNNs. We can view PSNs as a low-rank version of the polynomial spectral network described in Equation 2.26. Assume in that formulation that $\mathbf{\Theta}_0^{(\ell)} = \tilde{\sigma}^{(\ell)}\boldsymbol{W}^{(\ell)}$, and $\mathbf{\Theta}_k^{(\ell)} = (1-\tilde{\sigma}^{(\ell)})\tilde{\theta}_k^{(\ell)}\boldsymbol{W}^{(\ell)}$ for $k \geq 1$, where $\tilde{\sigma}^{(\ell)}, \tilde{\theta}_1^{(\ell)}, \ldots, \tilde{\theta}_K^{(\ell)}$ are as in Equation 3.2, and $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{d\times d}$. In fact, the 3D tensor $\mathbf{T}$ constructed by stacking the matrices $\mathbf{\Theta}_0^{(\ell)}, \ldots, \mathbf{\Theta}_K^{(\ell)}$ has rank at most rank$(\boldsymbol{W}^{(\ell)})$. This low-rank property implies regularization, i.e., PSNs are implicitly regularized spectral GNNs. This is a direct implication of our architectural design that applies the same polynomial filter to all input channels, and implements separately the channel-mixing operation. We provide additional details in the supplementary material.

Complexity and parameters. As in many other spectrally-inspired GNNs the main computation for the terms of the form $\mathbf{\Delta}^k\boldsymbol{H}^{(\ell)}$ in PSN can be done with a message-passing architecture. Assuming the number of edges in the input graph is m, each message-passing iteration can be done in time $O(md)$, with d the dimension of the node features. Thus for a filter of degree K, we obtain a complexity of $O(Kmd)$. What separates PSN from other proposals is that coefficients in the spectral filter are just scalar values, thus having a total complexity of $O(Kmd)$ for the whole filtering part. This contrasts with architectures such as ChebNet in which all the K filter terms are multiplied by a different $d \times d$ matrix, leading to the total complexity per layer of $O(Kmd + Knd^2)$ where n is the number of nodes in the input graph. In PSN, beyond the filtering part we have a linear mixer, thus having a complexity of $O(Kmd + nd^2)$ per layer. Table 3.1 shows the theoretical complexity per layer for ChebNet and PSN. We also include a GCN with K layers (K-GCN). This is because a PSN and ChebNet layer can aggregate information from a K-hop neighborhood, while a GCN layer can only reach a one-hop neighborhood. In terms of total number of parameters per layer, PSN is significantly more efficient especially when the degree of the polynomial filter increases. In practice one can also observe a significant reduction in the number of parameters (fourth column in Table 3.1).

It is also instructive to look at the minimum number of sequential operations per layer (K layers in the case of GCN) as it gives a strict lower bound on the parallelism achievable for each network. In this regard, PSN behaves similarly to ChebNet (third column in Table 3.1) implying that the difference in time gains will be more evident with less parallelism. This is confirmed by numerical analysis (last two columns in Table 3.1): the difference in time is bigger when we run the models in CPU (PSN is 2× faster than ChebNet) compared with running them in GPU in which the difference between PSN and ChebNets is minimal. The numbers are obtained over the MOLHIV dataset (see Section 4.1), for 8 layers of polynimial filters of degree 4 (8×4 layers in the case of GCN), but we observe a similar behavior in other datasets and for other configurations (see Section A.2.1 in the Annex for details).

Table 3.1: Complexity analysis and parameter count. In the table, n and m denote the number of nodes and edges in an input graph, K the degree of the polynomial filter, and d the input feature dimension.

	Theoretical complexity			8 layers, $K = 4$, $d = 128$ over MOLHIV		
Models	params.	complexity	seq. ops.	params.	cpu sec./epoch	gpu sec./epoch
K-GCN	$O(Kd^2)$	$O(K(md + nd^2))$	$O(K)$	1.5M	1,148.4	45.7
ChebNet	$O(Kd^2)$	$O(K(md + nd^2))$	$O(K)$	825K	477.6	16.0
PSN	$O(K + d^2)$	$O(Kmd + nd^2)$	$O(K)$	152K	214.2	15.6

3.2. Online Graph Nets (OGN)

A core idea behind T-GNNs is to maintain a state $\boldsymbol{s}_u$ for each node u, updating it whenever an event involving u (or its temporal neighbors) takes place. These updates require probing temporal and topological information to aggregate states from (possibly multi-hop) neighbors. Nonetheless, this aggregation step is the computational bottleneck of T-GNNs. To address this limitation, we propose summarizing each node's neighborhood into an auxiliary variable, which is incrementally updated as events unfold. Combining this idea with minimal design choices, we develop OGN (*Online Graph Nets*) — a fast and simple model for representation learning on dynamic graphs.

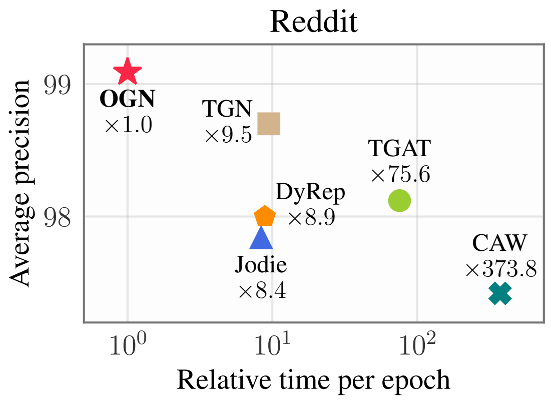


Figure 3.2: **OGN vs SOTA T-GNNs on Reddit (+500k interactions).** The horizontal axis shows the relative training time for each method as a multiple of OGN's running time. The vertical shows average precision. OGN clearly outperforms the SOTA but runs approximately 10 times faster than TGN [34] and 374 times faster than CAW [35].

In OGN every node is represented as a combination of a state and a neighborhood summary. While previous works invest most of their computation in ways to sample and aggregate neighbors, our neighborhood computation is embarrassingly simple and cheap: it is just a weighted average of all temporal neighbors states. To update the state of a node u, we use a combination of its previous state, the novel edge incident on u, and the neighborhood summary. In both computations we use the ordering of interactions between nodes as a proxy for time information, thus not considering any form of continuous time data. We also develop a scheme to efficiently and dynamically compute the node and neighborhood representations. With this, OGN does not need to store any information besides these two vectors per node, and the update can be performed in $O(1)$ time. Consequently, our method works as a fully online streaming algorithm without the need to store any previous interaction, which is especially useful for on-device predictions. Most importantly, OGN either outperforms or is competitive against state-of-the-art T-GNNs, while being much faster than previous methods (see Figure 3.2).

Basic notation. We assign a neighborhood variable $\boldsymbol{r}_u \in \mathbb{R}^d$ and a state variable $\boldsymbol{s}_u \in \mathbb{R}^d$ to each node u. We annotate these variables with superscripts to account for their evolution in time. In our formulation, we replace timestamps by an enumeration of events over time, which is equivalent to counting the events up to (and including) each interaction. For instance, $\boldsymbol{s}_u^{(n)}$ denotes the state vector for u after the n-th edge event. If the n-th added edge does not have an endpoint in u, then we set $\boldsymbol{s}_u^{(n)} = \boldsymbol{s}_u^{(n-1)}$ and $\boldsymbol{r}_u^{(n)} = \boldsymbol{r}_u^{(n-1)}$ by default. Also, we denote by $\mathcal{N}_u^{(n)}$ the set of temporal neighbors of node u prior to the n-th event in history.

Expected neighborhood state. We define the neighborhood state $\boldsymbol{r}_u^{(n)}$ as a weighted average of the states of all nodes that u interacted with, *exactly at the time of those interactions*. Thus, if node u had two distinct interactions with node i, then $\boldsymbol{r}_u^{(n)}$ considers two states of i, which need not be identical. To favor recent neighbors, we make the log-weight for $i \in \mathcal{N}_u^{(n)}$ decay linearly with the number of events $(n - m_i)$ since the interaction between i and u, which is the m_i-th event in the history. More specifically, the neighborhood state $\boldsymbol{r}_u^{(n)}$ is given by

$$\boldsymbol{r}_u^{(n)} = \sum_{i \in \mathcal{N}_u^{(n)}} w_i \boldsymbol{s}_i^{(m_i)}, \quad \text{with} \quad w_i := \frac{\exp\left(-\alpha\left(n - m_i\right)\right)}{\sum_j \exp\left(-\alpha\left(n - m_j\right)\right)}, \tag{3.4}$$

where α is an hyperparameter controlling how fast the importance of temporal neighbors decays. Note that the weight vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_{n-m_i})$ is the output of a temperature-scaled softmax. As consequence, $\boldsymbol{w}$ defines a categorical distribution over the neighborhood $\mathcal{N}_u^{(n)}$, and Equation 3.4 can be seen as the expected state of the neighbors of u. We provide more details in Section B.6.

Online computation of neighborhood states. Naively updating $\boldsymbol{r}_u^{(n)}$ requires a sweep over all previous neighbors every time a new edge event with endpoint in u occurs. Additionally, we would need to store the complete history of the states of each node. Summing up, after the n-th event we would have an overhead of $O(n)$ time and memory for each novel update. To alleviate this cost, we propose an *online* updating of $\boldsymbol{r}_u^{(n)}$. Assume that the n-th event connects nodes u and v, and let m be the number of events since the last interaction of u (with any node). We recursively compute

$$\boldsymbol{a}_u^{(n)} = \boldsymbol{s}_v^{(n)} + \exp(-\alpha m) \cdot \boldsymbol{a}_u^{(n-1)}, \tag{3.5}$$

$$b_u^{(n)} = 1 + \exp(-\alpha m) \cdot b_u^{(n-1)}, \tag{3.6}$$

with $\boldsymbol{a}_u^{(0)} = \mathbf{0}$ and $b_u^{(0)} = 0$. This recursion is particularly useful since $\boldsymbol{r}_u^{(n)} = \boldsymbol{a}_u^{(n)} / b_u^{(n)}$ — see Proposition 2, with a simple proof in Section B.7. Thus, to implement our method we simply store and update $\boldsymbol{a}_u^{(n)}$ and $b_u^{(n)}$. This scheme drops both time and memory complexity to $O(1)$ per update.

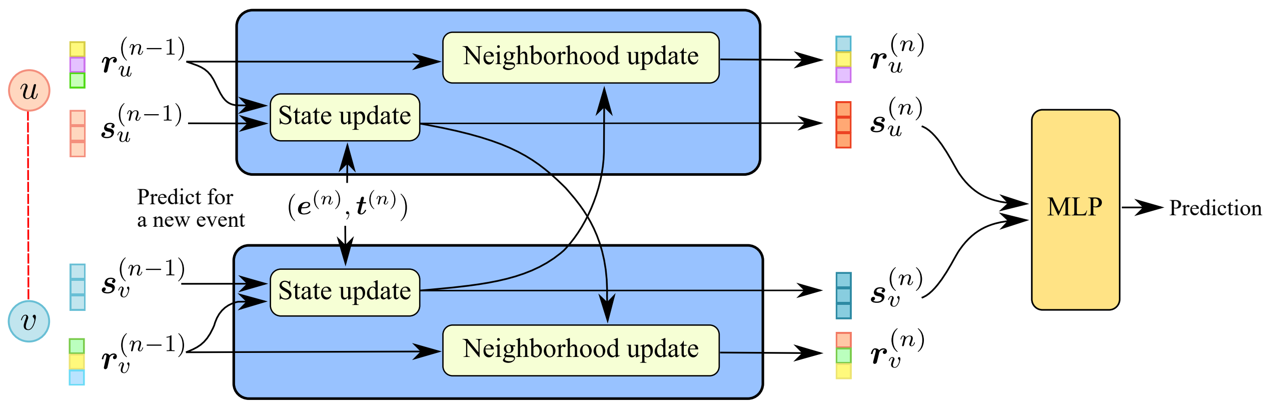


Figure 3.3: **Online Graph Nets (OGN).** OGN maintains a state vector and a neighborhood summary for each node. For a node u, to predict for a new event with node v, OGN updates the state vector of u using the neighborhood summary $r_u^{(n-1)}$ and the information of the event ($\boldsymbol{e}^{(n)}$ and $\boldsymbol{t}^{(n)}$), then updates its neighborhood information using the updated state vector of v. OGN performs this update for both nodes, and combines the resulting state vectors through an MLP for prediction purposes.

Proposition 2 For any node $u \in \mathcal{V}$, let $\boldsymbol{a}_u^{(n)}$ and $\boldsymbol{b}_u^{(n)}$ be defined according to Equation 3.5 and Equation 3.6, respectively, with $\boldsymbol{a}_u^{(0)} = \mathbf{0}$ and $\boldsymbol{b}_u^{(0)} = 0$. Then, for all $n \in \mathbb{N}$, it holds that:

$$\boldsymbol{r}_u^{(n)} = \frac{\boldsymbol{a}_u^{(n)}}{\boldsymbol{b}_u^{(n)}}.$$

Updating node states. We now describe how we update the state $\boldsymbol{s}_u^{(n)}$ when a new event involving u occurs, with edge features $\boldsymbol{e}^{(n)} \in \mathbb{R}^{\ell}$. We first compute an intermediate state $\bar{\boldsymbol{s}}_u^{(n)}$ by running the edge features, the previous node state $\boldsymbol{s}_u^{(n-1)}$, and the random Fourier features $\boldsymbol{t}^{(n)}$ of n through a linear layer followed by a non-linearity. Then, we feed $[\bar{\boldsymbol{s}}_u^{(n)} \| \boldsymbol{r}_u^{(n-1)}]$ to another single-layer net to obtain the updated state $\boldsymbol{s}_u^{(n)}$, where $\|$ denotes concatenation. The resulting node state update is:

$$\bar{\boldsymbol{s}}_u^{(n)} = \phi\Big(\boldsymbol{W}_1\big[\, \boldsymbol{e}^{(n)} \;\|\; \boldsymbol{s}_u^{(n-1)} \;\|\; \boldsymbol{t}^{(n)}\big]\Big), \tag{3.7}$$

$$\boldsymbol{s}_u^{(n)} = \phi\Big(\boldsymbol{W}_2\big[\bar{\boldsymbol{s}}_u^{(n)} \;\|\; \boldsymbol{r}_u^{(n-1)}\big]\Big), \tag{3.8}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d\times(\ell+2d)}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d\times 2d}$ are parameters, and ϕ is an element-wise non-linearity. OGN performs this update for both nodes u and v, and uses the resulting node states $\boldsymbol{s}_u^{(n)}$ and $\boldsymbol{s}_v^{(n)}$ as node representations for prediction purposes.

3.2.1. Complexity analysis

One key aspect of our formulation is that it incurs the same amount of computation for each edge added to the graph, no matter how many events have happened thus far. Notably, the computation time for each edge does not depend on any particular graph property, nor on the number of edges previously added or its distribution, nor in the degree of nodes, etc. More specifically, in terms of the graph size, our computation for each addition is $O(1)$. Moreover, whenever an edge is added to the graph and values $\boldsymbol{s}_u$, $\boldsymbol{a}_u$ and b_u are updated for the nodes incident to that edge, we no longer need that edge information in any way for future computations and we can safely drop it. In this sense, our method is a fully *online streaming method*, i.e., its computation time does not increase with the number of previous events (graph size).

This clearly differs from previous methods that need access to some form of memory about the previous events, and whose time complexity for processing new edges or making predictions also depends on the total number of previous events. For concreteness, assume a dynamic graph with E edges added so far, and with d as its maximum degree (maximum amount of events for a single node). For instance, TGAT's implementation requires a binary search over the history of previous events for each node. This implies that every node should store information of all its previous events, thus having a $\Omega(E)$ requirement for total memory and a $O(\log d)$ time overhead to process each new event [33]. A naive implementation of the path sampling in CAW would also need $\Omega(E)$ requirement for memory (potentially accessing any edge in the graph) and $O(d \times L)$ when sampling paths of length L. In the CAW paper [35], the authors state that sampling can be done with constant time and memory overhead, nevertheless our experiments using their own implementation exhibit the bigger running times of all methods we tested (Figure 4.3).

3.2.2. On-edge prediction

Dynamic graphs in real-life applications, such as social networks, may capture millions of new events every day. Thus, methods that require storing and updating the history of events for making predictions are not suitable for real-time applications. The updating must be performed in a centralized system to keep the last version of the complete graph, encompassing all the history, which becomes impractical when the graph size scales hugely. State-of-the-art T-GNNs, such as TGN, TGAT, and CAW, fall into this category. These methods rely on neighborhood sampling, thus requiring to at least update the (often multi-hop) neighborhood of the involving nodes as events unroll.

In contrast to the latest T-GNNs, predictions for OGN only depend on the nodes involved in the new event. Since OGN only relies on local information, it needs not an updated version of the graph nor even the graph structure to make a prediction. This property enables on-edge prediction. To illustrate, if nodes in the graph represent users in a social network and edges represent interactions between them, the predictions and the updates can be performed in each user's device by considering its state and neighborhood vectors and the ones from its interacting user, without requiring large computations and information exchange with a centralized system. Changing the predictions and updates from centralized to on-edge makes the models suitable for the aforementioned applications.

Chapter 4

Results and discussion

In the following chapter we assess the performance of both PSN and OGN in a variety of tasks. In particular, PSN is evaluated on a wide range of graph classification/regression and semi-supervised node classification tasks, while OGN is evaluated in temporal link prediction and node classification tasks.

4.1. Polynomial Subspace Net (PSN)

4.1.1. Graph-level prediction tasks

Datasets. [37] and [38] have recently shown that many popular benchmarks are inappropriate to assess GNNs. To avoid misleading conclusions, we consider two large-scale molecular datasets from the Open Graph Benchmark (OGB) [39]: MOLHIV and MOLPCBA. Following [37], we also consider ZINC [40], a graph regression dataset. These datasets are reproducible benchmarks stemming from real-world problems and for which GNNs perform better than structure-agnostic models.

We also evaluate PSN on five TU datasets [41]: TOX21, D&D, REDDIT-B, PROTEINS and ENZYMES. Importantly, D&D and REDDIT-B contain larger graphs compared to the other datasets we employ. We report summary statistics of the datasets in the Annex.

Baselines. We compare PSN against three spectral GNNs: GCN [42], ChebNet [21], and the general polynomial model in Equation 2.26 — henceforth referred to as LapGCN. We also consider Graph Isomorphism Networks (GINs) [27], MixHop [30] and Principal Neighborhood Aggregation networks (PNA) [43] as representative state-of-the-art GNNs based on message passing.

Experimental setup. To compare the model performance, we use the mean absolute error (MAE) for ZINC. Following the standard protocol, we use the area under the Receiver-Operating characteristic curve (ROC-AUC) for MOLHIV, and the average precision (AP) for MOLPCBA. We report mean and standard deviation of the performance metrics computed over five independent runs. We provide further implementation details in the Annex (Section A.1).

Table 4.1: Graph-level prediction on OGB datasets and ZINC. PSN reaches the best result for ZINC and MOLPCBA, and is competitive against PNA in MOLHIV. For ZINC, the lower the better. Boldface indicates the best average result in each dataset, and underline the second best.

Models	ZINC ↓	MOLHIV ↑	MOLPCBA ↑
GIN	$0.408_{\pm 0.008}$	$75.58_{\pm 1.40}$	$0.2266_{\pm 0.0028}$
MixHop	$0.442_{\pm 0.014}$	$73.73_{\pm 1.70}$	$0.1646_{\pm 0.0041}$
PNA	$0.320_{\pm 0.032}$	$\mathbf{79.05_{\pm 1.32}}$	$0.2284_{\pm 0.0085}$
GCN	$0.469_{\pm 0.002}$	$76.06_{\pm 0.97}$	$0.2020_{\pm 0.0024}$
LapGCN	$\underline{0.313_{\pm 0.009}}$	$75.01_{\pm 1.88}$	$0.0926_{\pm 0.0017}$
ChebNet	$0.360_{\pm 0.028}$	$76.31_{\pm 1.27}$	$\underline{0.2306_{\pm 0.0016}}$
PSN	$\mathbf{0.264_{\pm 0.014}}$	$\underline{78.84_{\pm 1.71}}$	$\mathbf{0.2314_{\pm 0.0062}}$

We report results for ChebNet on ZINC, MOLHIV, and MOLPCBA as provided by [44]. The results for GIN are available in [37] and in the OGB leaderboards [39]. Moreover, we take performance numbers for PNA from the original work [43], except for MOLPCBA, for which we run the model using the official released code. Since there are neither common dataset splits nor a standard evaluation protocol for TU datasets, we run all methods from scratch.

Results. Table 4.1 shows the results for ZINC, MOLHIV, and MOLPCBA datasets. PSN is the best-performing model for ZINC and achieves similar performance to PNA on MOLHIV with only 0.21% difference in ROC-AUC. In both datasets PSN clearly outperforms all other spectral graph nets. For MOLPCBA, the results of PSN, ChebNet, and PNA are very close to one another. Although PSN presents the best result, there is only a 0.3% difference in average precision among the three best models. Table 4.2 shows that PSN obtains significantly better results for all TU datasets.

Table 4.2: Graph classification on TU datasets. PSN shows better overall performance than other methods.

Models	TOX21	D&D	REDDIT	PROT.	ENZ.
GIN	$74.2_{\pm 2.8}$	$75.5_{\pm 2.8}$	$90.3_{\pm 3.0}$	$74.8_{\pm 4.1}$	$59.6_{\pm 4.5}$
MixHop	$80.6_{\pm 0.6}$	$\underline{87.8_{\pm 5.3}}$	$92.3_{\pm 1.0}$	$67.4_{\pm 4.3}$	$\underline{81.0_{\pm 4.8}}$
GCN	$77.3_{\pm 4.0}$	$61.7_{\pm 1.1}$	$89.3_{\pm 3.3}$	$75.9_{\pm 4.3}$	$58.0_{\pm 7.3}$
ChebNet	$\underline{82.3_{\pm 0.7}}$	$61.0_{\pm 1.1}$	$\underline{96.6_{\pm 0.6}}$	$\underline{74.6_{\pm 3.5}}$	$78.3_{\pm 3.4}$
PSN	$\mathbf{82.4_{\pm 0.3}}$	$\mathbf{93.6_{\pm 3.6}}$	$\mathbf{97.0}_{\pm 1.1}$	$\mathbf{79.8_{\pm 6.4}}$	$\mathbf{84.2_{\pm 0.8}}$

4.1.2. Node classification

Datasets & Baselines. We gauge the performance of PSN on three semi-supervised node classification tasks over citation networks: Arxiv, Cora, and Citeseer. The Arxiv dataset is part of OGB [39] whereas Cora and Citeseer [45] have been broadly used to assess GNNs. We

Table 4.3: Node classification on citation networks. Despite its simple formulation, PSN is competitive with recently proposed methods, such as DAGNN and GCNII.

Models	Arxiv	Cora	Citeseer
GIN	67.74 ± 0.29	75.1 ± 1.7	63.1 ± 2.0
DAGNN	72.09 ± 0.25	$\underline{84.4 \pm 0.5}$	73.3 ± 0.6
GCNII	$\underline{72.74 \pm 0.16}$	$\mathbf{85.5 \pm 0.5}$	$\underline{73.4 \pm 0.6}$
GAT	$\mathbf{73.65 \pm 0.11}$	83.0 ± 0.7	72.5 ± 0.7
MixHop	N/A	81.9 ± 0.4	71.4 ± 0.8
GCN	71.74 ± 0.29	81.6 ± 0.4	70.1 ± 0.7
ChebNet	70.78 ± 0.16	80.5 ± 1.1	70.1 ± 0.8
CayleyNet	N/A	81.9 ± 0.7	67.1 ± 2.4
ARMA	69.56 ± 0.20	83.4 ± 0.6	72.5 ± 0.4
S^{2}GC	72.01 ± 0.25	83.5 ± 0.0	73.6 ± 0.1
PSN	72.27 ± 0.21	83.1 ± 0.4	$\mathbf{73.7 \pm 0.9}$

compare PSN to five spectral GNNs: GCN [42], ChebNet [21], CayleyNet [22], ARMA [23] and S^{2}GC [26]. We also consider five message-passing representative models: GIN [27], DAGNN [29], GCNII [25], GAT [28] and MixHop [30].

Experimental setup. We evaluate all models in a transductive setting. We recover most results from existing works that use the same data splits and training setup [22, 23, 29]. Additionally, we run the ARMA model on Arxiv using the implementation available in the PyTorch Geometric framework [46]. We provide further implementation details in the Annex (Section A.1).

Results. Table 4.3 reports the results for all node-level tasks in terms of accuracy. Notably, PSN presents the best values for Citeseer. On Arxiv and Cora, PSN is competitive with message-passing GNNs. Compared to the other spectral GNNs, PSN shows the best overall results, being surpassed only by ARMA in Cora by a very small margin.

4.2. Where is the gain coming from?

In this section, we analyze the components that make PSN either surpass or be competitive with more complex spectral GNNs. We first carry out an ablation study to assess the impact of each model component. After that, we show that our restrictions on the coefficients make PSN training more stable. We then show that our simple design can learn a variety of filters depending on the task at hand. Like other high-order spectral GNNs, we show that PSNs can handle oversmoothing and benefit from deep architectures. Lastly, we discuss the limitations of PSN and potential solutions.

Table 4.4: Ablation study. For each model, we report the number of layers inside parenthesis.

Models	ZINC $\downarrow$	MOLHIV $\uparrow$	TOX21 $\uparrow$
no-tanh/res	0.311 ± 0.005 (32)	75.42 ± 0.94 (8)	81.95 ± 0.68 (1)
no-tanh	0.283 ± 0.007 (16)	76.47 ± 1.06 (4)	82.63 ± 0.75 (1)
no-res	0.316 ± 0.010 (16)	76.53 ± 1.60 (8)	81.81 ± 0.35 (1)
PSN	0.264 ± 0.014 (16)	78.84 ± 1.71 (8)	82.66 ± 0.69 (2)

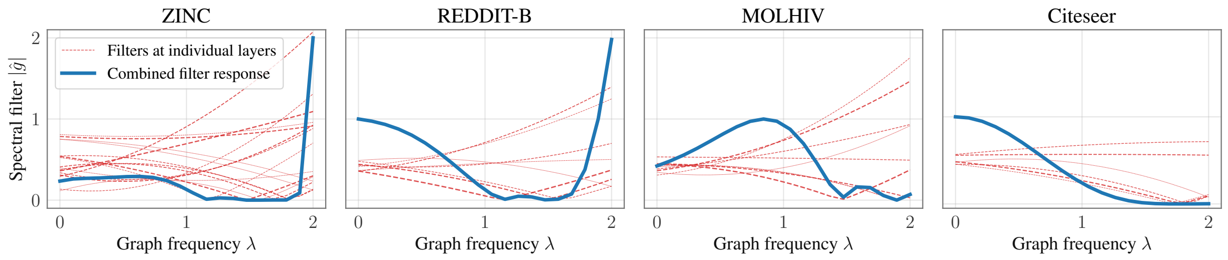


Figure 4.1: Filter responses for PSN in different datasets. We show the individual filters at each layer (red) and the combined filter response (blue). PSN layers combine to generate diverse filters: high-pass (ZINC), band-stop (REDDIT-B), and band-pass (MOLHIV). On Citesser, PSN learns a low-pass filter, meeting well-known efficient filter designs for node classification tasks [47].

4.2.1. Ablation study

Table 4.4 shows the best performance that we could obtain with each PSN-like model that does not use all the architectural components of PSN's filters. Regarding terminology, "no-res" refers to the model without the convex combination, that is, the filter is simply $\sum_{k=0}^{K} \tanh(\theta_k^{(\ell)})\boldsymbol{\Delta}^k$. Similarly, "no-tanh" refers to PSN without the application of the tanh function, i.e., the filter is given by $\text{sigmoid}(\theta_0^{(\ell)})\boldsymbol{I} + (1 - \text{sigmoid}(\theta_0^{(\ell)}))\sum_{k=1}^{K} \theta_k^{(\ell)}\boldsymbol{\Delta}^k$. Finally the "no-tanh/res" is the basic filter $\sum_{k=0}^{K} \theta_k^{(\ell)}\boldsymbol{\Delta}^k$. For the ZINC and MOLHIV datasets, we observe that each PSN component brings something to the table. In contrast, we do not see significant gains on TOX21. One reason for this is that simple shallow models obtain good performance on TOX21, even surpassing deep models. Accordingly, we have empirically observed that our design decisions make a more significant impact when models are deeper (see supplement for details).

We stress that we are comparing the best architecture we could find for each PSN variant. Thus, the selected models may not have the same number of layers. For instance, we achieve the best result for "PSN no-tanh/res" on ZINC with a 32-layer model, which is twice as deep as the best-performing PSN. Interestingly, our simplest PSN (no-tanh/res) achieves competitive performance compared to message-passing GNNs such as GIN [27].

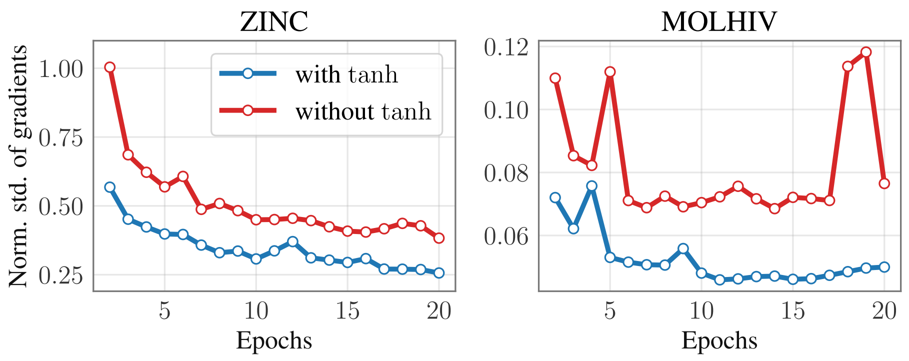


Figure 4.2: Normalized std of gradients. Using tanh leads to more stable training as gradient values fluctuate less.

4.2.2. The impact of restricting the parameters

As shown in Lemma 1, restricting the polynomial coefficients to $[-1, 1]$ does not affect the expressiveness of our convolutional layer. In addition, we hypothesize that the main gain of this design choice comes from making the learning process more stable. To test this hypothesis, we compare the behavior of gradients during training for two models: with and without coefficient constraints. At each epoch, we collect gradients for each parameter, such that we get as many values as minibatches. We then compute the standard deviation of these gradients and divide the deviation by the absolute value of the parameter at the end of the epoch. This normalization ensures comparability across different parameter scales and epochs. The resulting number measures the fluctuation level of the gradients. We do so independently for each parameter and average the results. We call this metric *normalized standard deviation of gradients*.

Figure 4.2 shows this metric for models trained on ZINC and MOLHIV for 20 epochs, before convergence. Notably, the models with constrained coefficients are more stable during training, experiencing smoother gradient variations within each epoch. We provide further implementation details and plots for additional datasets in the Annex.

4.2.3. Spectral analysis

An interesting property of PSN is that its formulation can be interpreted as a single polynomial filter followed by a linear layer, as can be seen from Equation 3.1. Other spectral GNNs such as ChebNets and CayleyNets mix the filter and linear layer, which makes studying the effect of each component more difficult. We believe that this property for PSN makes them simpler and easier to analyze. To show this, we consider the response of the whole network as the composition of the spectral filters in each layer, bypassing nonlinearity and mixing operations. This does not show the exact frequency response of the whole network, but it gives an idea of what type of filters the model learns for different datasets. Figure 4.1 demonstrates that PSN can learn different types of filters for MOLHIV, REDDIT-B, ZINC, and Citeseer. For instance, PSN achieves a high-pass filter for ZINC, a band-stop filter for REDDIT-B, and a band-pass filter for MOLHIV. On Citesser, PSN learns a low-pass filter, meeting well-known efficient filter designs for node classification tasks [47]. The figure also shows that the final filter is obtained by combining several simple filters at each layer. Given that PSN uses only a few filter coefficients in each layer, we can increase model flexibility by stacking more layers without severally increasing the model complexity in terms of parameter count.

4.2.4. Measuring oversmoothing

Oversmoothing is a phenomenon observed in several GNNs in which representations for different nodes become progressively more similar as the depth increases [48, 49]. This has been acknowledged as one of the main reasons why going deeper in GNNs does not usually lead to better performance. We assess how PSN behaves in terms of oversmoothing considering two models: i) 5 layers with polynomial degree 4; and ii) 10 layers with degree 2. To quantify oversmoothing, we compute the average distance between the final embeddings of all pairs of neighbors. Table 4.5 presents the average of this metric over all graphs in each dataset. It has been shown that residuals usually help in avoiding oversmoothing [25], thus we consider a 20-layers GCN with residual connections as a simple baseline. We report numbers for (plain) GCN, PNA, and ChebNet acting on a 20-hop neighborhood (20 layers for GCN, and PNA and 5 layers with polynomial degree 4 for ChebNet). We also tested GIN but all the numbers were less than 10^{-5} so we do not report them in the table. PSN significantly outperforms the baseline and yields the most consistent results throughout the datasets.

Table 4.5: Oversmoothing for different GNNs (higher is better). Entries represent normalized mean distance between pairs of neighbors, averaged over all graphs. Overall, PSN shows the most consistent results.

Models	ZINC	MOLHIV	MOLPCBA
Baseline	0.140 ± 0.033	0.13 ± 0.05	0.47 ± 0.12
GCN	0.138 ± 0.033	0.09 ± 0.03	0.14 ± 0.03
PNA	0.399 ± 0.177	0.89 ± 0.16	1.45 ± 0.12
ChebNet	1.076 ± 0.156	0.25 ± 0.07	0.99 ± 0.07
PSN (5×4)	1.062 ± 0.060	1.22 ± 0.12	1.23 ± 0.05
PSN (10×2)	0.901 ± 0.065	0.79 ± 0.10	1.34 ± 0.05

4.2.5. Limitations

A limitation of PSN is that its convolutions are *isotropic*, meaning that, although node representations depend on the features of their neighbors, the PSN representation is completely independent of the position of each neighbor. This prevents PSNs from showing competitive performance on graph benchmarks in which neighbors have a natural orientation in space. Typical examples are synthetic graph datasets derived from image data, such as MNIST and CIFAR10 [37]. In these cases, PSN performs poorly compared to the state-of-the-art: 86.87 ± 1.49 in MNIST and 56.67 ± 0.71 in CIFAR10. Nonetheless, PSN results are similar to those from other isotropic methods such as GCN, GIN, and GraphSAGE [37]. There are several ways of making a GNN *anisotropic*, such as adding identifiers in node features [50], or

Table 4.6: **Results in average precision (AP) on link prediction for datasets with edge features.** In all datasets, OGN is either the best performing method or is closely behind it. * Corresponds to the original standard deviation 0.04 rounded to the first decimal.

Model	**Reddit**		**Wikipedia**		**MOOC**		**Twitter**	
	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive
GAT	97.33±0.2	95.37±0.3	94.73±0.2	91.27±0.4	-	-	-	-
GraphSAGE	97.65±0.2	96.27±0.2	93.56±0.3	91.09±0.3	-	-	-	-
Jodie	97.84±0.3	93.97±1.3	95.70±0.2	93.61±0.2	81.16±1.0	78.77±1.6	98.23±0.1	96.06±0.1
DyRep	98.00±0.1	95.18±0.2	94.66±0.1	91.91±0.2	79.57±1.5	79.37±0.7	98.48±0.1	96.33±0.2
TGAT	98.12±0.2	96.62±0.3	95.34±0.1	93.99±0.3	64.36±3.3	61.74±3.2	98.70±0.1	96.33±0.1
TGN	98.70±0.1	97.55±0.1	98.46±0.1	97.81±0.1	82.10±0.4	77.70±0.3	98.00±0.1	95.76±0.1
CAW	98.39±0.1	97.81±0.1	**98.63±0.1**	**98.52±0.1**	**89.76±0.4**	**89.72±0.4**	98.72±0.1	**98.54±0.3**
OGN (ours)	**99.09±0.0***	**98.66±0.1**	97.16±0.2	**98.41±0.2**	**88.71±1.3**	85.65±1.5	**99.01±0.0***	**98.46±0.1**

using anisotropic aggregation schemes in message passing, e.g. attention-based aggregation [28]. However, these choices would deviate us from our initial motivation of sticking to a simple spectral definition. We leave anisotropic formulations of PSN for future work.

4.3. Online Graph Nets (OGN)

4.3.1. Temporal link prediction

Datasets. We assess the performance of OGN on four commonly used link prediction benchmarks: Reddit, Wikipedia, MOOC and Twitter. These datasets are attributed, i.e., they contain feature vectors for their events. The Twitter dataset is not publicly available, but we follow instructions from [34] to create a version of the dataset. Node features are absent in all datasets, thus we follow previous work [33, 34] and set them to zero. We provide datasets statistics in Section B.2. Additionally, we report results for non-attributed datasets in Section B.5.

Baselines. We compare OGN against five state-of-the-art temporal models: Jodie [31], DyRep [32], TGAT [33], TGN [34], and CAW [35]. We also show results for two static graph methods, GAT [28] and GraphSage [51], which do not use temporal information. Most of the results for these baselines are available in previous works [33, 34]. However, we re-run experiments for Jodie and DyRep due to conflicting numbers in different papers. We noticed an incorrect implementation of attention in the released code of CAW, thus we modify it and report the corrected numbers. We provide a complete description of this change in Section B.1.

Experimental setup. The goal in the link prediction task is to classify whether an interaction between two nodes happens at a given time. Since our datasets only contain positive observations, we follow previous works [33, 34] and create negative links artificially. For every edge event, we create a false event at the same time by re-assigning one of the edge endpoints to a random node. Following [33, 34], we consider a 70%-15%-15% (train-val-test) split. We evaluate all models in both transductive and inductive settings in this experiment.

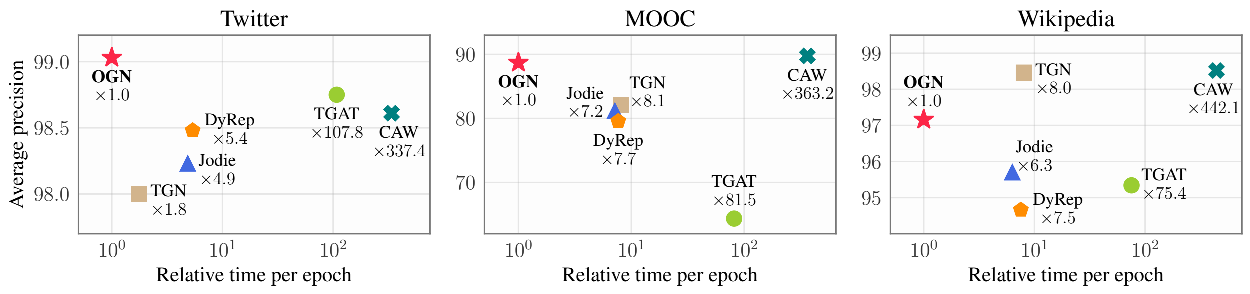


Figure 4.3: **Time/epoch vs average precision.** We normalize the execution time per epoch of all models with respect to the execution time of OGN. In all cases, OGN nearly matches or surpasses the SOTA. Also, OGN is the fastest method overall. For instance, OGN is more than two orders of magnitude faster than CAW and at least one order of magnitude faster than TGAT.

In the transductive setting, we predict interactions involving nodes seen during training. In the inductive setting, we evaluate the models on nodes never observed before. We use average precision (AP) as performance metric and repeat each experiment for ten independent runs.

Results. Table 4.6 present the results in the link prediction task for the transductive and inductive settings in attributed datasets. The results show that OGN achieves comparable performance to state-of-the-art methods. Notably, it obtains the best results in the Reddit dataset for both the transductive and inductive settings and in the Twitter dataset for the transductive setting.

Figure 4.3 shows the performance and time per epoch for different T-GNN models. We can see that OGN is competitive with state-of-the-art methods, while being orders of magnitude faster than some of the methods. In particular, OGN is always two orders of magnitude faster than CAW, which is the best performing method in Wikipedia and MOOC. Further, OGN is the fastest considered method.

4.3.2. Node classification

Datasets. We evaluate OGN on two node classification benchmarks: Reddit and Wikipedia. Reddit contains labeled links that indicate whether a user will be banned from a subreddit. Only 366 out of $672,447$ interactions result in a ban. Wikipedia contains labeled events that represents whether a user will be kept from editing a Wikipedia page. Again, only a small fraction (217 out of $157,474$) of edits lead to bans on Wikipedia [31].

Baselines. We compare OGN against four temporal models: Jodie, DyRep, TGAT and TGN. We also evaluate two static methods: GAT and GraphSage. Most of the results for our baselines are available in the literature [33, 34].

Experimental setup. Due to the imbalance between positive and negative examples in the dataset, we measure performance in terms of AUC. We report average and standard

Table 4.7: **Results for node classification (AUC)**. OGN is the best performing model on Wikipedia and obtains the second best AUC on Reddit.

Model	**Reddit**	**Wikipedia**
GAT	64.52$\pm$0.5	82.34$\pm$0.8
GraphSage	61.24$\pm$0.6	82.42$\pm$0.7
Jodie	61.83$\pm$2.7	84.84$\pm$1.2
DyRep	62.91$\pm$2.4	84.59$\pm$2.2
TGAT	65.56$\pm$0.7	83.69$\pm$0.7
TGN	**67.06$\pm$0.9**	$\underline{87.81\pm0.3}$
OGN	$\underline{65.59\pm1.0}$	**88.36$\pm$0.4**

deviation for ten repetitions of the experiments.

Results. Table 4.7 presents the results for the node classification task. For Wikipedia, OGN presents the best performance. For Reddit, OGN is the second best method in average AUC.

4.4. Ablation Studies

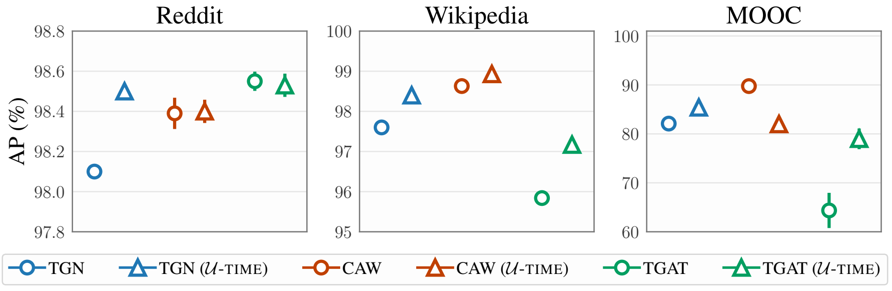
In the following section we present ablation studies to understand the effect of different components for T-GNNs and OGN.

4.4.1. Temporal GNNs

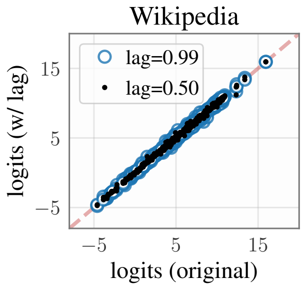
The importance of fine-grained time information. We design simple experiments to challenge the need for fine-grained time information (timestamps), and use TGAT, TGN and CAW as running examples. Intuitively, this feature is essential to capture the dynamics of real-world applications (e.g., social networks), in which events naturally occur in continuous time. To challenge this intuition, we artificially discretize the time information and re-evaluate these methods. We do so by setting the gap between successive events to a fixed value $\Delta = 0.1$ in training and testing. We refer to this approach as $\mathcal{U}$-TIME.

Figure 4.4.a reports the performance of TGN and TGAT with and without the discretization approach. Surprisingly, we find that continuous-time information generally does not improve and sometimes even hurts the performance of T-GNNs. For instance, $\mathcal{U}$-TIME leads to an increase of $\approx 3\%$ and 14% in AP for TGN and TGAT on the MOOC dataset, respectively. For all other datasets and methods, we only observe small fluctuations in performance, except for CAW on the MOOC dataset.

A possible explanation for this phenomenon is that T-GNNs are insensitive to the specific values of the timestamp, leveraging the ordering instead. To test this hypothesis, we take TGN (with full-fledged timestamps) and evaluate their predictions when each timestamp of the test set is shifted by a relative lag to approach the timestamp of the subsequent event. For example, with a relative lag of 0.5, we shift an event with timestamp 20000 to 15000 if the previous event happened at timestamp 10000. Note that this procedure preserves the original ordering of events. Figure 4.4.b compares the logits of TGN with relative lags $\{0.5, 0.99\}$ on Wikipedia. Notably, TGN produces virtually the same predictions regardless of the amount of lag we apply.



(a)



(b)

Figure 4.4: (**a**) **Effect of using regularly spaced timestamps ($\mathcal{U}$-time) on TGAT, TGN and CAW.** In general, using $\mathcal{U}$-TIME ($\triangle$ markers) does not harm performance. In fact, TGN and TGAT benefit from time discretization. Also, the models show smaller standard deviations overall. (**b**) **Predictions of TGN with shifted timestamps on test data for Wikipedia.** Even after applying a 0.99 relative lag, the logits remain rather similar to those from the original no-shifted timestamps (red line).

The importance of attention. State-of-the-art T-GNN models, such as TGN and TGAT, use attention as their aggregation layers. We look to evaluate the importance of this component in the performance of T-GNNs. For this purpose, we replace the attention module for an element-wise mean or max followed by a linear layer.

In Table 4.8 we compare the performance of the attention module to the best performance between the mean and max layers. Interestingly, we do not see a significant gain by using attention over a mean or max layer in most datasets. In fact, the largest gap in performance of 7.53 for TGAT in the MOOC dataset is actually in favor of using the mean/max layer. Using attention only outperforms using mean/max in Wikipedia for both models.

4.4.2. OGN

Time information. To evaluate the effect of discretizing timestamps, we consider two alternatives: (i) using the original timestamps, and (ii) removing the vector representation of the time. Results in Table 4.9 show that using discretized time information generally leads to higher AP values. The exception is Reddit, in which using the original timestamps yields slightly better results. Similarly, using the discretized time also outperforms OGN with no time embeddings at all. These results highlight that the only information needed from the

Table 4.8: **Results for TGN and TGAT with different attention modules in the transductive setting.** Using attention either brings marginal gains or winds up hurting performance (TGAT on MOOC). Superscripts * and ** denote a standard deviation of 0.04 and 0.03 rounded to the first decimal, respectively.

Model	Module	Reddit	Wikipedia	MOOC
TGN	Original	98.1±0.0*	**97.6±0.1**	82.1±0.4
	Mean/Max	98.1±0.0**	97.3±0.1	82.0±0.2
TGAT	Original	98.51±0.1	**95.85±0.1**	66.09±3.4
	Mean/Max	98.47±0.0*	94.96±0.2	**74.34±0.5**

Table 4.9: **Ablation study for OGN**. Neighborhood state and edge information play a crucial role on the model's performance. Discretized timestamps generally lead to better performance.

Model	**Reddit**	**Wikipedia**	**MOOC**
Original timestamps	**99.13±0.02**	96.45±0.97	81.06±2.82
No time embedding	98.96±0.03	95.63±0.29	86.33±3.12
No edge features	96.11±1.02	96.57±0.16	69.87±6.74
No neighborhood states	97.00±0.10	89.33±0.87	83.40±0.29
OGN	**99.09±0.04**	**97.16±0.21**	**88.71±1.34**

timestamps is the sequence of events.

Edge information. Since most datasets do not have node features, the edge features are the only source of information besides timestamps. To study their importance to our method, we remove them from the dataset by setting their values to zero. Results in Table 4.9 show that the performance deteriorates across all datasets when we remove the edge features, which demonstrates their importance for accurate predictions. The drop in performance without edge features is especially noticeable for the MOOC dataset, where the performance drop amounts to 18% when we remove edge features.

Neighborhood information. One of the key insights of our method consists of using a state vector $\boldsymbol{r}_u$ to summarize the neighborhood information for the nodes in the dataset. To study the impact of this component, we remove $\boldsymbol{r}_u$ from our model and only consider the node state and the random Fourier time embedding to update the node embedding. The results in Table 4.9 show a significant decrease in performance when the neighborhood state is removed. These results indicate that neighborhood states are crucial components of our method.

Limitations. As noted previously, edge features play an important role in the node state updating after each event. Since node features are absent in the datasets, the performance of OGN mainly relies on edge information. Therefore, its performance drops when learning on non-attributed temporal graphs, which suggests that OGN does not fully exploit the structural information of the graph, given its simplicity. We report results on benchmarks with non-attributed events in Section B.5.

Chapter 5

Conclusions

In this thesis we have introduced polynomial subspace net (PSN) and online graph nets (OGN), simple GNN models that pursue a simple design over flexibility of the models. We empirically show that these simple methods outperform state-of-the-art models in several benchmarks, while running asymptotically faster than other methods. Additionally, we present a comprehensive suite of experiments that validate the design choices for both models.

PSN has the added benefit of being a spectral GNN with few parameters that can easily be interpreted as a filter in the spectral domain. We present theoretical findings that prove that PSN can represent any polynomial filter. Alongside this result, we show experiments that: i) shows evidence that our constrained parameterization leads to more stable training; ii) demonstrates that, despite its simplicity, PSN can learn different expressive filters; and iii) illustrates that PSN is robust to oversmoothing compared to other popular GNNs.

An attractive feature is that OGN is an online streaming method, i.e.: i) updating the model does not require access to previous events; ii) the computational cost of OGN does not increase with history length. Since OGN does not use neighborhood sampling, its predictions are extremely scalable and suitable for use on edge devices.

By proposing PSN and OGN, simple GNN models with state-of-the-art performance, we have validated the hypothesis that was presented in Section 1. We also present a detailed study on the success of the models along with its limitations, fulfilling the general objectives presented at the beginning of the work.

The specific objectives that were stated in Section 1 are also fulfilled.

- PSN has lower computational complexity and number of parameters than other spectral GNNs, while outperforming state-of-the-art methods in graph classification tasks and staying competitive in node classification tasks.
- OGN has lower computational complexity and memory usage than other T-GNNs. It also presents significant speedup over such methods, while obtaining state-of-the-art performance in attributed datasets and competitive performance in unattributed datasets. OGN is a fully online streaming method, which enables its use in real-life data.

- We present detailed experiments for both methods that study its components to understand how they obtain competitive performance while staying simple.
- Through the experiments we identify the limitations for both methods when compared to more complex alternatives, and explain the reasons behind them.

The limitations of both methods (Section 4.2.5 and Section 4.4.2) present an interesting pathway for further investigation. For PSN, extending the method to make it anisotropic might result in significant gain in several benchmarks where nodes have a natural position in space. For OGN, adding more detailed structural information that does not incur in significant computational complexity could be the key to increase its performance in unattributed datasets. We leave the exploration of these problems as future work.

Bibliography

[1] Y. Lecun and Y. Bengio, "Convolutional networks for images, speech, and time-series," 01 1995.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 5998–6008.

[4] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, p. 61–80, 2009.

[5] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond Euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[6] W. Fan, Y. Ma, Q. Li, Y. He, Y. E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019.* ACM, 2019, pp. 417–426.

[7] C. Huang, H. Xu, Y. Xu, P. Dai, L. Xia, M. Lu, L. Bo, H. Xing, X. Lai, and Y. Ye, "Knowledge-aware coupled graph neural network for social recommendation," in *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Event, February 2-9, 2021.* AAAI Press, 2021, pp. 4115–4122.

[8] D. Cummings and M. Nassar, "Structured citation trend prediction using graph neural networks," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 3897–3901.

[9] Z. Shui and G. Karypis, "Heterogeneous molecular graph neural networks for predicting molecule properties," *CoRR*, vol. abs/2009.12710, 2020.

[10] O. Wieder, S. Kohlbacher, M. Kuenemann, A. Garon, P. Ducrot, T. Seidel, and T. Langer, "A compact review of molecular property prediction with graph neural networks," *Drug Discovery Today: Technologies*, 12 2020.

[11] Y. Wang, "Bag of tricks of semi-supervised classification with graph neural networks," *CoRR*, vol. abs/2103.13355, 2021. [Online]. Available: https://arxiv.org/abs/2103.13355

[12] M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin, "Revisiting graph neural networks for link prediction," *CoRR*, vol. abs/2010.16103, 2020. [Online]. Available:

https://arxiv.org/abs/2010.16103

[13] K. Kong, G. Li, M. Ding, Z. Wu, C. Zhu, B. Ghanem, G. Taylor, and T. Goldstein, “FLAG: adversarial data augmentation for graph neural networks,” *CoRR*, vol. abs/2010.09891, 2020. [Online]. Available: https://arxiv.org/abs/2010.09891

[14] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, “Graph neural networks: A review of methods and applications,” *AI Open*, vol. 1, pp. 57–81, 2020.

[15] M. N. Bernstein, “The graph laplacian,” Nov 2020, accessed: 2022-01-15. [Online]. Available: https://mbernste.github.io/posts/laplacian_matrix/

[16] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge University Press, 1985.

[17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” in *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, ser. Proceedings of Machine Learning Research, vol. 70. PMLR, 2017, pp. 1263–1272.

[18] M. Balcilar, G. Renton, P. Héroux, B. Gaüzère, S. Adam, and P. Honeine, “Analyzing the expressive power of graph neural networks in a spectral perspective,” in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[19] J. Li, J.-M. Guo, and W. Shiu, “Bounds on normalized laplacian eigenvalues of graphs,” *Journal of Inequalities and Applications*, vol. 2014, pp. 1–8, 2014.

[20] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, “Spectral networks and locally connected networks on graphs,” in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[21] M. Defferrard, X. Bresson, and P. Vandergheynst, “Convolutional neural networks on graphs with fast localized spectral filtering,” in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 2016, pp. 3837–3845.

[22] R. Levie, F. Monti, X. Bresson, and M. M. Bronstein, “Cayleynets: Graph convolutional neural networks with complex rational spectral filters,” *IEEE Transactions on Signal Processing*, vol. 67, no. 1, pp. 97–109, 2019.

[23] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, “Graph neural networks with convolutional ARMA filters,” *arXiv preprint: 1901.01343*, 2019.

[24] E. Isufi, A. Loukas, A. Simonetto, and G. Leus, “Autoregressive moving average graph filtering,” *IEEE Transactions on Signal Processing*, vol. 65, no. 2, pp. 274–288, 2017.

[25] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, “Simple and deep graph convolutional networks,” in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 1725–1735.

[26] H. Zhu and P. Koniusz, “Simple spectral graph convolution,” in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[27] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[28] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[29] M. Liu, H. Gao, and S. Ji, "Towards deeper graph neural networks," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020.* ACM, 2020, pp. 338–348.

[30] S. Abu-El-Haija, B. Perozzi, A. Kapoor, N. Alipourfard, K. Lerman, H. Harutyunyan, G. V. Steeg, and A. Galstyan, "Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 21–29.

[31] S. Kumar, X. Zhang, and J. Leskovec, "Predicting dynamic embedding trajectory in temporal interaction networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019.* ACM, 2019, pp. 1269–1278.

[32] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "Dyrep: Learning representations over dynamic graphs," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[33] D. Xu, C. Ruan, E. Körpeoglu, S. Kumar, and K. Achan, "Inductive representation learning on temporal graphs," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[34] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, "Temporal graph networks for deep learning on dynamic graphs," *ArXiv: 2006.10637*, 2020.

[35] Y. Wang, Y. Chang, Y. Liu, J. Leskovec, and P. Li, "Inductive representation learning in temporal networks via causal anonymous walks," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016.* IEEE Computer Society, 2016, pp. 770–778.

[37] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," *ArXiv e-print*, 2020.

[38] F. Errica, M. Podda, D. Bacciu, and A. Micheli, "A fair comparison of graph neural networks for graph classification," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[39] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint: 2005.00687*, 2020.

[40] W. Jin, R. Barzilay, and T. S. Jaakkola, "Junction tree variational autoencoder for

molecular graph generation," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 2328–2337.

[41] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, "Benchmark data sets for graph kernels," 2016. [Online]. Available: http://graphkernels.cs.tu-dortmund.de

[42] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[43] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Velickovic, "Principal neighbourhood aggregation for graph nets," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[44] A. Nilsson and X. Bresson, "An experimental study of the transferability of spectral graph networks," *arXiv preprint: 2012.10258*, 2020.

[45] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad, "Collective classification in network data articles," *AI Magazine*, vol. 29, pp. 93–106, 09 2008.

[46] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019. [Online]. Available: http://arxiv.org/abs/1903.02428

[47] F. Wu, A. H. S. Jr., T. Zhang, C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, ser. Proceedings of Machine Learning Research, vol. 97. PMLR, 2019, pp. 6861–6871.

[48] D. Chen, Y. Lin, W. Li, P. Li, J. Zhou, and X. Sun, "Measuring and relieving the over-smoothing problem for graph neural networks from the topological view," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 2020, pp. 3438–3445.

[49] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[50] R. Sato, M. Yamada, and H. Kashima, "Random features strengthen graph neural networks," *arXiv preprint: 2002.03155*, 2021.

[51] W. L. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 2017, pp. 1024–1034.

[52] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. S. Pande, "Moleculenet: A benchmark for molecular machine learning," *CoRR*, vol. abs/1703.00564, 2017.

[53] M. Balcilar, G. Renton, P. Héroux, B. Gaüzère, S. Adam, and P. Honeine, "Analyzing the expressive power of graph neural networks in a spectral perspective," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

Annex A

PSN

A.1. Implementation details

A.1.1. Datasets

Table A.1 and Table A.2 report statistics of the datasets for graph and node classification, respectively. Notably, we use three large-scale datasets from the OGB suite: MOLHIV ($\approx$ 41k graphs), MOLPCBA ($\approx$ 437k graphs), and Arxiv ($\approx$ 169k nodes). We also employ ZINC, a regression dataset that has been recently introduced as a benchmark for graph-level prediction tasks [37]. The remaining datasets are part of TUDatasets [41]. Table A.1 and Table A.2 also report split ratios (train/validation/test) for each dataset. For most datasets, there are standardized split schemes which ensure comparability with prior results. As for the TOX21, D&D and REDDIT-B datasets, we apply a random split scheme (80%/10%/10%) similarly to [38] and [52].

Table A.1: Summary of graph classification and regression datasets.

Dataset	**#Graphs**	**#Nodes**	**#Edges**	**#Tasks**	**Metric**	**Split scheme**	**Split ratio**
ZINC	12,000	9-37	-	1	MAE	Fixed	83.3/8.3/8.3
MOLHIV	41,127	25.5	27.5	1	ROC-AUC	Scaffold	80/10/10
MOLPCBA	437,929	26	28.1	128	AP	Scaffold	80/10/10
TOX21	7,831	18.6	-	12	ROC-AUC	Random	80/10/10
D&D	1,178	284.3	715.7	1	accuracy	Random	80/10/10
REDDIT-B	2,000	429.63	497,754	1	accuracy	Random	80/10/10
PROTEINS	1,113	43,471	72,82	1	accuracy	Random	80/10/10
ENZYMES	600	19,580	62,14	1	accuracy	Random	80/10/10

Table A.2: Summary of node classification datasets.

Dataset	**#Nodes**	**#Edges**	**#Classes**	**Split scheme**	**Split ratio**
Arxiv	169,343	1,166,243	40	Time	54/18/28
Cora	2,708	5,429	7	Fixed	45/18/37
Citeseer	3,327	9,228	6	Fixed	55/15/30

A.1.2. Evaluation setup

Table A.3 and Table A.4 show the best hyperparameter choice for each method and dataset. For all models, we set the embedding size to 128 and use batch-normalization before each layer. We also apply global mean pooling to obtain graph-level embeddings for graph classification tasks. On top these graph-level representation, we use a linear layer followed by a ReLU activation. We train all models with Adam using weight decay and/or dropout. We do not incorporate edge features in our implementation of PSN. We provide configuration files along with our code to guarantee reproducibility.

A.1.3. Hardware

We run experiments using a set of machines comprising heterogeneous GPU resources including Nvidia Tesla P100, Tesla V100, GTX 1080Ti, and TITAN RTX cards.

Table A.3: Models used for graph classification datasets. For the cases marked as * the result that we presented in the Experiments section was obtained by a search over 2,3,4, and 5 layers by [46].

Dataset	**Model**	**Layers**	**Degree (K)**
ZINC	LapGCN	8	2
	PSN	16	2
MOLHIV	LapGCN	5	3
	PSN	8	2
MOLPCBA	LapGCN	5	3
	PSN	16	2
	PNA	3	N/A
TOX21	GCN	1	N/A
	GIN	4	N/A
	ChebNet	2	3
	PSN	4	2
D&D	GCN	3	N/A
	GIN	9	N/A
	ChebNet	8	2
	PSN	4	3
REDDIT-B	GCN	*	N/A
	GIN	*	N/A
	ChebNet	8	2
	PSN	8	2
PROTEINS	GCN	3	N/A
	GIN	5	N/A
	ChebNet	8	2
	PSN	16	2
ENZYMES	GCN	3	N/A
	GIN	4	N/A
	ChebNet	8	2
	PSN	16	2

Table A.4: Models used for node classification datasets.

Dataset	**Model**	**Layers**	**Degree (K)**
Citeseer	PSN	6	2
Cora	PSN	10	2
Arxiv	ChebNet	4	2
	PSN	4	2
	ARMA	1	5

A.2. Additional experiments

A.2.1. Complexity

Table A.5 shows time and parameter complexity for PSN, ChebNet and GCN trained over MOLHIV and ZINC. The time per epoch is obtained as the average of running 10 epochs. The machines we used for the comparison were equipped similarly with Intel Xeon E5-2630 v4 CPUs, 32Gb RAM and Nvidia GTX 1080 Ti GPUs. We used the same machines to obtain the numbers in Table 3.1 in the body of the paper.

Table A.5: Complexity analysis and parameter count for the ZINC dataset with 16 layers, $K = 4$ and $d = 128$. We empirically measure the number of parameters and the seconds per epoch both in CPU and GPU.

	16 layers, $K = 4$, $d = 128$ over ZINC		
Models	params.	cpu sec./epoch	gpu sec./epoch
K-GCN	3.17M	704,4	27.3
ChebNet	1.61M	154,8	9.1
PSN	289K	106,8	9.0

A.2.2. Ablation study

Table A.6 presents a more detailed account of the ablation study in Section 4.2.1, with results for different number of layers. The results show that our design choices (inner-layer residual connections and restricted coefficients) are specially useful for deeper models, e.g., with 16 or 32 layers. Notably, full-fledged PSNs are the best performing models.

Table A.6: **Ablation study for a varying number of layers.** For each dataset, boldface indicates the best average result for each PSN variant. An additional underline highlights the best overall model.

Dataset	Model	Layers					
		1	2	4	8	16	32
ZINC	PSN no-tanh/res	0.56 ± 0.01	0.52 ± 0.01	0.43 ± 0.02	0.33 ± 0.01	0.33 ± 0.01	$\mathbf{0.31 \pm 0.01}$
	PSN no-tanh	0.58 ± 0.01	0.53 ± 0.01	0.45 ± 0.01	0.36 ± 0.01	$\mathbf{0.28 \pm 0.01}$	0.28 ± 0.01
	PSN no-res	0.57 ± 0.01	0.54 ± 0.02	0.42 ± 0.01	0.33 ± 0.01	$\mathbf{0.32 \pm 0.01}$	0.34 ± 0.01
	PSN	0.58 ± 0.01	0.52 ± 0.01	0.46 ± 0.01	0.38 ± 0.02	$\underline{\mathbf{0.26 \pm 0.01}}$	0.28 ± 0.01
MOLHIV	PSN no-tanh/res	73.5 ± 1.7	75.3 ± 1.1	75.3 ± 1.1	$\mathbf{75.3 \pm 3.0}$	73.9 ± 1.3	68.9 ± 1.6
	PSN no-tanh	74.4 ± 1.7	76.0 ± 0.7	74.8 ± 1.4	76.3 ± 1.8	$\mathbf{76.5 \pm 1.3}$	71.4 ± 2.3
	PSN no-res	73.7 ± 2.1	$\mathbf{76.3 \pm 0.2}$	74.8 ± 1.4	75.5 ± 0.6	75.0 ± 0.5	68.4 ± 3.6
	PSN	74.5 ± 1.1	75.8 ± 1.0	76.6 ± 1.5	$\underline{\mathbf{78.8 \pm 1.7}}$	78.1 ± 0.7	72.8 ± 1.3
TOX21	PSN no-tanh/res	$\mathbf{81.6 \pm 1.5}$	80.4 ± 1.5	78.8 ± 0.7	76.8 ± 0.9	75.7 ± 0.5	77.0 ± 1.5
	PSN no-tanh	$\mathbf{82.8 \pm 0.4}$	82.6 ± 0.5	82.4 ± 0.2	81.0 ± 0.4	79.9 ± 1.0	78.7 ± 0.8
	PSN no-res	$\mathbf{81.3 \pm 1.0}$	80.4 ± 1.0	78.7 ± 1.5	77.4 ± 1.1	75.7 ± 0.7	75.9 ± 1.1
	PSN	$\underline{\mathbf{83.1 \pm 0.4}}$	82.8 ± 0.6	82.7 ± 0.7	81.5 ± 0.3	81.4 ± 0.8	80.5 ± 0.5

A.2.3. Measuring oversmoothing

Figure A.1 shows tolerance to oversmoothing as a function of depth for ChebNet, GCN, GIN, and PSN on the ZINC and MOLHIV datasets. Similar to the Measuring oversmoothing section, we again use the mean average distance metric to measure oversmoothing. Results reflect the average of 5 runs. For PSN and ChebNet, we use polynomials of degree 2 in each layer. To ensure the same reachable neighborhood size, we compare L-layer PSN/ChebNets to $2L$-layer GCN/GINs. As expected, GCNs and GINs become increasingly similar as the number of layers grows. PSN obtains the largest mean avg. distance for deep models (8, 16 and 32 layers).

Additionally, Figure A.2 shows performance as a function of depth. Notably, PSN performs better as depth increases. Arguably, this is only possible due to the robustness of PSN to oversmoothing. On the contrary, ChebNet, GCN, and GIN's performances tend to drop with deeper architectures.

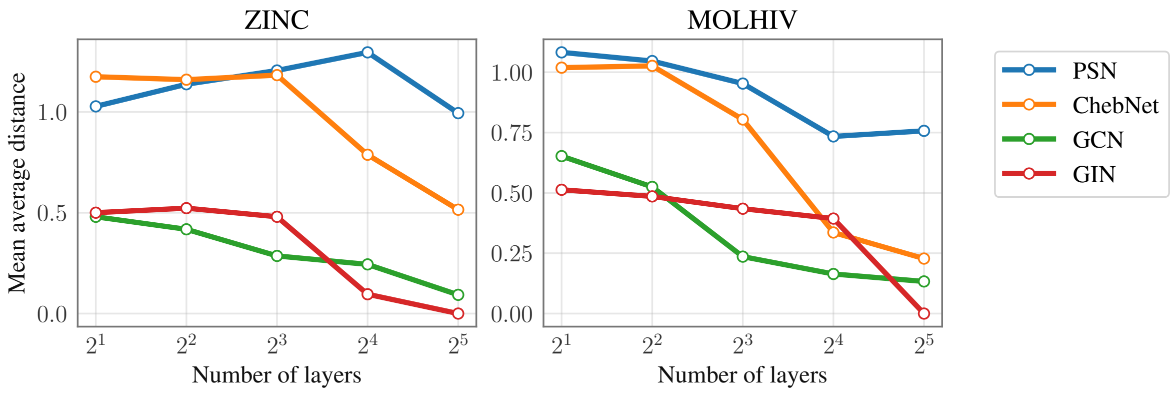


Figure A.1: **Measuring oversmoothing: mean average distance for a varying number of layers.** Among all methods, PSN is the least affected by oversmoothing when depth increases. Higher is better.

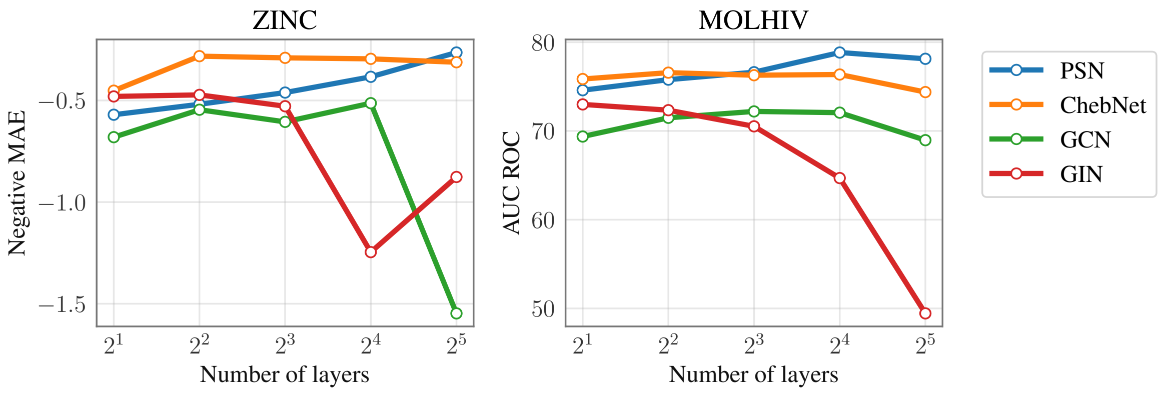


Figure A.2: **Performance as a function of network depth.** Overall, deep PSNs (32 layers) outperform the competing methods.

A.2.4. Gradients

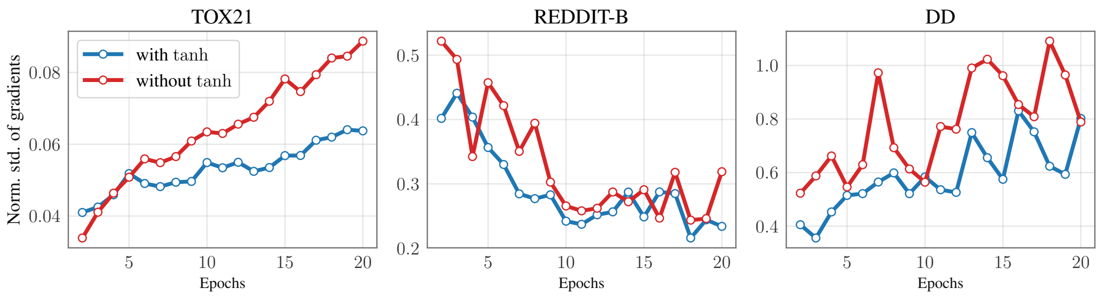


Figure A.3: **Normalized std of gradients per epoch for the TOX21, REDDIT-B, and DD datasets.** Overall, using tanh results in smaller gradient fluctuations during training.

To further support our analysis of the gradients (see Section 4.2.2), we report normalized standard deviation of the gradients during training for three additional datasets: TOX21, REDDIT-B, and D&D. Figure A.3 shows that using restricted coefficients (i.e., applying tanh) to gradients with smaller deviation, allowing for more stable training. This effect is particularly evident in TOX21 as the number of epochs increases.

A.3. Proof of Proposition 1

Proof. Let $P(x)$ be an arbitrary polynomial of degree K

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_K x^K$$

and M be the maximum absolute value of the coefficients of $P(x)$, i.e., $M = \max_{0 \le k \le K} |a_k|$. Furthermore, we define:

$$\begin{aligned} m =& 2M \operatorname{sign}(a_0), \\ b =& \frac{a_0}{m}, \\ c_k =& \frac{a_k}{(1-b)m} \quad \forall k = 1 \dots K. \end{aligned}$$

Consider the polynomial $R(x)$ defined as

$$b + (1-b)c_1 x + (1-b)c_2 x^2 + \cdots + (1-b)c_K x^K.$$

Note that $P(x) = mR(x)$ by construction. To complete the proof of Proposition 1, we only need to show that $b \in [0,1]$ and that $c_k \in [-1,1]$. For the case of b we have that

$$\begin{aligned} b &= \frac{a_0}{m} \\ &= \frac{a_0}{2M\,\mathrm{sign}(a_0)} \\ &= \frac{|a_0|}{2M} \end{aligned}$$

and since $M \geq |a_0|$, b lies in $[0, {}^1\!/\!_2]$.

For all $k = 1 \ldots K$ it follows that

$$\begin{aligned} |c_k| &= \left|\frac{a_i}{(1-b)\cdot 2M\,\mathrm{sign}(a_k)}\right| \\ &= \left|\frac{a_i}{2(1-b)M}\right| \\ &\leq \left|\frac{1}{2(1-b)}\right| \end{aligned}$$

The last equation stems from the fact that $M \geq |a_k|$. Since $b \in [0, {}^1\!/\!_2]$, we have that $1/2 \leq (1-b) \leq 1$. As a consequence, $|c_k| \in [{}^1\!/\!_2, 1]$ and therefore $c_k \in [-1,1]$ for all $k = 1 \ldots K$. □

A.4. Proof of Proposition 2

Proof. [53] show that the frequency profile for the k-th support of ChebNet is given by

$$\boldsymbol{\Phi}_k(\boldsymbol{\lambda}) = \begin{cases} \mathbf{1} & k = 1 \\ \frac{2\boldsymbol{\lambda}}{\lambda_{\max}} - \mathbf{1} & k = 2 \\ 2\boldsymbol{\Phi}_2(\lambda)\boldsymbol{\Phi}_{k-1}(\boldsymbol{\lambda}) - \boldsymbol{\Phi}_{k-2}(\boldsymbol{\lambda}) & k > 2. \end{cases}$$

Therefore, the k-th support of ChebNet has a frequency response that is a polynomial of order k with respect to $\boldsymbol{\lambda}$. Consequently, ChebNet filters of size K are combinations of polynomials up to degree K.

Note that the k-th support of PSN is $\boldsymbol{C}^{(k)} = \boldsymbol{\Delta}^k$ and that $\boldsymbol{\Delta}^k = \boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^T$, by definition. Following closely the derivations by [53], we can characterize the frequency profile for the k-th support of PSN as:

$$\begin{aligned} \boldsymbol{\Phi}_k(\boldsymbol{\lambda}) &= \mathrm{diag}^{-1}(\boldsymbol{U}^T\boldsymbol{C}^{(k)}\boldsymbol{U}) \\ &= \mathrm{diag}^{-1}(\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Lambda}^k\boldsymbol{U}^T\boldsymbol{U}) \\ &= \mathrm{diag}^{-1}(\boldsymbol{\Lambda}^k) \\ &= \boldsymbol{\lambda}^k. \end{aligned}$$

As a consequence, a PSN filter of size K can learn any polynomial filter up to degree K, similarly to ChebNet filters. Therefore, the filter in PSN is as expressive as the one in ChebNet. $\square$

A.5. Proof of Proposition 3

Proof. Let us consider two layers $F(\boldsymbol{H}) = \sum_{k=0}^{K}(\theta_k \boldsymbol{\Delta}^k \boldsymbol{H})\boldsymbol{W}$ and $G(\boldsymbol{H}) = \sum_{k=0}^{K}(\theta'_k \boldsymbol{\Delta}^k \boldsymbol{H})\boldsymbol{W}'$. We say that F and G are equivalent if $F(\boldsymbol{H}) = G(\boldsymbol{H})$ for any possible $\boldsymbol{H} \in \mathbb{R}^{n\times d}$ and $\boldsymbol{\Delta} \in \mathbb{R}^{n\times n}$. This simply means that for any input graph both layers output the same value. The equality can be written in the following way.

$$\sum_{k=0}^{K} \theta_k (\boldsymbol{\Delta}^k \boldsymbol{H})\boldsymbol{W} = \sum_{k=0}^{K} \theta'_k (\boldsymbol{\Delta}^k \boldsymbol{H})\boldsymbol{W}'.$$

The previous expression can be rewritten as

$$\sum_{k=0}^{K} \boldsymbol{\Delta}^k \boldsymbol{H}(\theta_k \boldsymbol{W} - \theta'_k \boldsymbol{W}') = \boldsymbol{0}.$$

This expression can hold for every possible $\boldsymbol{\Delta} \in \mathbb{R}^{n\times n}$ and every $\boldsymbol{H} \in \mathbb{R}^{n\times d}$ only if the following equality holds,

$$(\theta_k \boldsymbol{W} - \theta'_k \boldsymbol{W}') = \boldsymbol{0},\ \forall k.$$

With this, $\boldsymbol{W}$ and $\boldsymbol{W}'$ follow the relation

$$\boldsymbol{W} = \frac{\theta'_k}{\theta_k}\boldsymbol{W}' = \mu \boldsymbol{W}',\ \forall k.$$

Note that the previous equation implies that θ'_k/θ_k must be a constant for all values of k. Let us call y the output for the layers (since the layers are equivalent, the output is the same) and $\mathcal{L}$ any continuous loss function. The standard deviation for the gradients satisfy the following equality.

$$\begin{aligned}
\sigma\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}\right) &= \sigma\left(\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \boldsymbol{W}}\right) \\
&= \sigma\left(\frac{\partial \mathcal{L}}{\partial y}\sum_{k=0}^{K}\theta_k(\boldsymbol{\Delta}^k \boldsymbol{H})\right) \\
&= \sigma\left(\frac{\partial \mathcal{L}}{\partial y}\sum_{k=0}^{K}\frac{\theta'_k}{\mu}(\boldsymbol{\Delta}^k \boldsymbol{H})\right) \\
&= \sigma\left(\frac{1}{\mu}\frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial \boldsymbol{W}'}\right) \\
&= \frac{1}{\mu}\sigma\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}'}\right)
\end{aligned}$$

We can use the previous equality in conjunction with the relation between the layer weights to compute the relation between the normalized standard deviations as follows.

$$\begin{aligned}\frac{\sigma\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}}\right)}{\boldsymbol{W}} &= \frac{1}{\mu}\frac{\sigma\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}'}\right)}{\boldsymbol{W}} \\ &= \frac{1}{\mu^2}\frac{\sigma\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}'}\right)}{\boldsymbol{W}'}\end{aligned}$$

Which is the equality that we wanted to prove. $\square$

A.6. Proof of Proposition 4

Definition 1 (Rank-one tensor) A tensor $\mathbf{X} \in \mathbb{R}^{D_1 \times \ldots \times D_n}$ is rank one we can write it as the outer product of n vectors, i.e.,

$$\mathbf{X} = \boldsymbol{a}^{(1)} \otimes \cdots \otimes \boldsymbol{a}^{(n)},$$

where $\boldsymbol{a}^{(i)} \in \mathbb{R}^{D_i}$ for all $i = 1 \ldots n$. Equivalently, the entries in $\mathbf{X}$ are given by

$$x_{d_1 \ldots d_n} = a^{(1)}_{d_1} \ldots a^{(n)}_{d_n}.$$

Definition 2 (Rank of a tensor) The rank of a tensor $\mathbf{X}$, denoted by rank($\mathbf{X}$), is the minimum number of rank-one tensors whose sum equals $\mathbf{X}$.

Proof. Let r denote the rank of $\boldsymbol{W}^{(\ell)} \in \mathbb{R}^{d_{\ell-1} \times d_\ell}$. By definition, we have that $\boldsymbol{W}^{(\ell)}$ can be written as

$$\boldsymbol{W}^{(\ell)} = \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 + \ldots + \boldsymbol{a}_r \otimes \boldsymbol{b}_r,$$

where $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_r \in \mathbb{R}^{d_{\ell-1}}$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_r \in \mathbb{R}^{d_\ell}$.

Let $\boldsymbol{c} = [c_0, \ldots, c_K]^\top$ such that

$$c_k = \begin{cases} \sigma(\theta_0^{(\ell)}) & \text{if } k = 0, \\ \left(1 - \sigma(\theta_0^{(\ell)})\right) \tanh\left(\theta_k^{(\ell)}\right) & \text{otherwise.} \end{cases}$$

We can write $\mathbf{T}^{(\ell)}$ as a sum of r rank-one tensors

$$\begin{aligned}\mathbf{T}^{(\ell)} &= \boldsymbol{W}^{(\ell)} \otimes \boldsymbol{c}, \\ &= \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 \otimes \boldsymbol{c} + \ldots + \boldsymbol{a}_r \otimes \boldsymbol{b}_r \otimes \boldsymbol{c},\end{aligned}$$

and therefore the rank of $\mathbf{T}$ is not greater than r. $\square$

Annex B

OGN

B.1. Causal Anonymous Walk (CAW)

[35] propose Causal Anonymous Walk-Networks (CAW-N), which computes the node embeddings using sets of temporal random walks S_u and S_v, starting respectively from the endpoints u and v of an edge to be predicted at time t. The random walk is *time-aware* in the sense that the log-probability of sampling w as the next node in a walk from u, is inversely proportional to the difference between the current time and the time of the last interaction between w and u. The random-walk size is governed by an hyperparameter L.

Once S_u and S_v are computed, CAW-N anonymizes each walk by replacing every node w with a footprint vector $I_{\text{CAW}}(w) \in \mathbb{R}^{L+1}$ such that $I_{\text{CAW}}(w)_\ell$ stores how many times w was the ℓ-th node in a walk of $S_u \cup S_v$. Let $\widehat{S}_u$ and $\widehat{S}_v$ denote the anonymous versions of S_u and S_v. CAW-N then treat every anonymous walk $\widehat{W} \in \widehat{S}_u \cup \widehat{S}_v$ as a sequence of pairs $(I_{\text{CAW}}(w_i), t_i)$, with $i = 0, \ldots, L$, and produces an embedding $\text{emb}(\widehat{W})$ by first transforming each pair into a vector $[f_1(I_{\text{CAW}}(w_i)) \parallel f_2(t_{i-1} - t_i)]$, and then running a recurrent network over that sequence of vectors. In [35], function $f_1(\cdot)$ is a combination of MLPs, and $f_2(\cdot)$ is a random Fourier feature expansion. Finally, CAW-N combines all the anonymous walk embeddings using either mean-pooling or self-attention.

B.1.1. The Bug: Attention over the batch instead of the walks

In the official code by [35], we note that the self-attention used for aggregating the sampled random walks is computed over the incorrect dimension. Specifically, given the tensor that contains representations of the sampled walks $\boldsymbol{h} \in \mathbb{R}^{B,N,D}$, where B is the batch size, N the number of walks and D the embedding size, the self-attention should be computed over N, but in the official release it is instead computed over B.[1] Consequently, when predicting a link at time t_i, this implementation allows the model to get information from the other events

[1] The specific line of code that generates the problem can be found in the official released code (Line 51) https://github.com/snap-stanford/CAW/blob/master/transformer.py#L51 when using the multi-head attention implemented by the PyTorch library. That implementation expects a tensor arranged as $(\text{sequence\_length}, \text{batch}, \text{embedding\_size})$, but [35] provide a tensor arranged as $(\text{batch}, \text{sequence\_length}, \text{embedding\_size})$.

in the batch, in particular, to events at times t_j with $j > i$. This effectively allows CAW to "look to the future".

To illustrate the problem, we replicated the results of [35] in the transductive setting, and use the same training setup to evaluate the test set using batches of size 1 (we emphasize that we only change the batch size during testing). The prediction algorithm of CAW does not depend on the batch size, but the results suggest that the test performance drops noticeably using batch size 1. We further fixed the implementation error so that the self-attention is now computed over the dimension of the walks, N, and retrained CAW with the same hyperparameters. Again, there is an evident drop in performance compared to the results reported by the authors.

In addition, we note that in the officially released code of CAW, the pooling method that aggregates walks is always set as attention, even when using the flag of mean pooling[2]. Thus, we modified it to be a mean aggregation, and the performance drops w.r.t the numbers provided in [35] with mean aggregation. Table B.1 summarizes the results for the aforementioned modifications.

Table B.1: Results of CAW in transductive setting (average precision).

Model	**Reddit**	**Wikipedia**	**MOOC**	**UCI**
Original (bug in the attention)	99.75$\pm$0.12	100.0$\pm$0.0	97.55$\pm$0.45	93.56$\pm$1.33
Test batch size 1	85.29$\pm$1.08	92.94$\pm$0.52	75.57$\pm$2.52	77.10$\pm$1.31
Corrected attention	97.08$\pm$0.06	98.20$\pm$0.07	73.40$\pm$0.48	81.66$\pm$0.59
Mean	96.53$\pm$0.12	97.83$\pm$0.13	72.15$\pm$0.51	77.96$\pm$1.67

Because of the aforementioned bug we use the "corrected attention" version of CAW for the experiments in the paper.

B.2. Datasets

For temporal link prediction, we use six popular benchmarks:

- Reddit[3] is a dataset of posts made by users on subreddits over a month. Nodes correspond to either users or subreddits, and links denote posting requests from users to subreddits, annotated with timestamps.
- Wikipedia[4] is a network where links correspond to timestamped updates that users (nodes) make to wiki pages (nodes). The dataset only comprises the 1000 most edited pages, and users with at least 5 edits within a month.

[2] In the official implementation of CAW (Lines 120-124) https://github.com/snap-stanford/CAW/blob/be07783b59824fbc5ed666b3e885d4a6abc8d1a3/main.py#L120 the walk_pool argument is not fed into CAW, thus it always takes the default value, which is the attention aggregation.

[3] http://snap.stanford.edu/jodie/reddit.csv

[4] http://snap.stanford.edu/jodie/wikipedia.csv

- MOOC[5] is a dataset of students' actions on a massive open online course. Its nodes represent either students or course content units, and its temporal links represent student's access to course units.
- UCI[6] is a dataset of posts to the University of California Irvine forum. Its nodes denote either users or forums, and the links represent timestamped non-attibuted forum messages.
- Enron[7] is a dataset of email communications in Enron. Its nodes represent core employees of Enron and links represent emails between them.
- LastFM[8] records one month of who-listens-to-which song information. Its nodes correspond to either users or songs.

We also create a Twitter dataset following the work of [34]. We describe the details of the Twitter dataset in Section B.2.1 of the Annex. Table B.2 reports summary statistics for each dataset.

Table B.2: Summary statistics for temporal link prediction datasets. * Corresponds to edge features filled with zero values.

Dataset	**#Nodes**	**#Edges**	**#Edge feat.**
Reddit	10,985	672,447	172
Wikipedia	9,227	157,474	172
MOOC	7,145	411,749	4
Twitter	8,925	406,564	768
UCI	1,899	59,835	100*
Enron	184	125,235	32*
LastFM	1,980	1,293,104	2*

B.2.1. Twitter dataset

We base our Twitter dataset in the description given in the TGN paper [34]. To generate the dataset we begin with the data from the 2021 Twitter RecSys Challenge. Then we take the 10,000 nodes with the highest number of interactions in the dataset — and respectively their edge events. Note that not all of the 10,000 nodes will be left in the dataset, since some might no have interactions with other nodes in the dataset. To compute the edge features, we use Multilingual BERT on the provided text tokens.

B.3. Implementation details

[5] http://snap.stanford.edu/jodie/mooc.csv
[6] http://konect.cc/networks/opsahl-ucforum/
[7] https://www.cs.cmu.edu/~./enron/
[8] http://snap.stanford.edu/jodie/lastfm.csv

B.3.1. Evaluation setup and hyperparameters

Real-world temporal networks only comprise true edge events, i.e., *positive links* (class 1). To generate *negative links* (class 0), we follow standard methodology [33, 34, 35]: for each positive link $e_{u,v}$, we create a negative link $e_{u,v'}$ with $v' \neq v$ uniformly sampled from a set of candidate nodes, using the same feature vector and timestamp as $e_{u,v}$.

We train the models using both positive and negative links, and the binary cross-entropy loss. We use Adam with learning rate 10^{-4} during 50 epochs, with early stopping if there is no improvement greater than 10^{-5} in validation average precision for 5 epochs.

For CAW, we perform a grid search over the time decay $\alpha \in \{0.01, 0.1, 0.25, 0.5, 1.0, 2.0, 4.0, 10.0, 100.0\}$ 10^{-6}, number of walks $M \in \{1, 2, 3, 4, 5\}$ and walk length $L \in \{32, 64, 128\}$. We present the best combination of hyperparameters in Table B.3. For TGN, we follow [34] and sample twenty temporal neighbors. For TGAT, we sample twenty immediate neighbors and twenty 2-hop temporal neighbors following the guidelines in the original work [33].

Table B.3: Hyperparameters for CAW.

Dataset	**Time decay α**	**#Walks**	**Walk length**
Reddit	10^{-8}	32	3
Wikipedia	4×10^{-6}	64	4
MOOC	10^{-4}	64	3
UCI	10^{-5}	64	2
Enron	10^{-6}	64	5

B.4. Further Ablation Study

The importance of time information. We now take a step further and evaluate the performance of T-GNNs when no time information is available and only the ordering of the events is preserved. In particular, we first create the sequence of events ordered by time and then set the actual value of timestamps to zero before feeding them to TGAT and TGN. We refer to this approach as NO-TIME.

Figure B.1 shows the performance of representative T-GNNs with and without timestamps. Notably, the performance of TGN significantly decreases ($\approx 23\%$ in AP) on the UCI dataset.

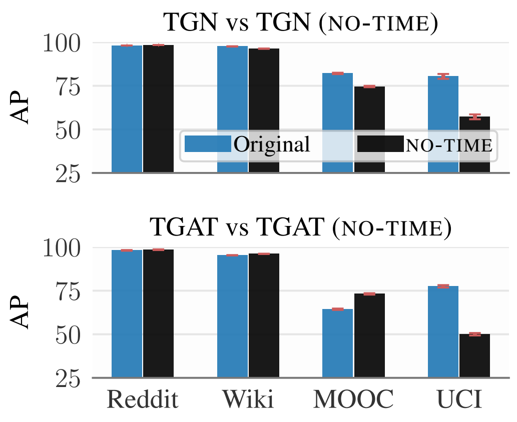


Figure B.1: Removing timestamps can hurt the performance of T-GNNs. Both TGN (NO-TIME) and TGAT (NO-TIME) experience a significant performance drop on the UCI dataset.

However, these results show that there are cases in which leveraging the ordering of events alone is not enough to learn meaningful temporal node representations. We note that the $\mathcal{U}$-TIME and NO-TIME approaches are fundamentally different. Unlike NO-TIME, $\mathcal{U}$-TIME still allows T-GNNs to count the total number of events between two interactions involving the same node.

B.4.1. Hardware

We run experiments using a set of machines comprising heterogeneous GPU resources including Nvidia Tesla P100, Tesla V100, GTX 1080Ti, and TITAN RTX cards. To ensure fairness in time comparison (Figure 3.2 and Figure 4.3), we also run all methods on the machine equipped with a consumer-grade GPU (Nvidia GTX 1080Ti).

B.5. More experiments

In this section we present results in the link prediction task for four more datasets, UCI, Enron and LastFM. All of these datasets are unattributed, which means that they do not contain edge or node features. OGN does not perform as well in these datasets. We discuss why this may be the case in Section 4.4.2 of the paper.

Table B.4: Results in average precision (AP) on link prediction for datasets that do not contain edge features.

Model	**UCI**		**Enron**		**LastFM**	
	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive
Jodie	86.73$\pm$1.0	75.26$\pm$1.7	77.31$\pm$4.2	76.48$\pm$3.5	69.32$\pm$1.0	80.32$\pm$1.4
DyRep	54.60$\pm$3.1	50.96$\pm$1.9	77.68$\pm$1.6	66.97$\pm$3.8	69.24$\pm$1.4	82.03$\pm$0.6
TGAT	77.51$\pm$0.7	70.54$\pm$0.5	68.02$\pm$0.1	63.70$\pm$0.2	54.77$\pm$0.4	56.76$\pm$0.9
TGN	80.40$\pm$1.4	74.70$\pm$0.9	79.91$\pm$1.3	<u>78.96$\pm$0.5</u>	<u>80.69$\pm$0.2</u>	<u>84.66$\pm$0.1</u>
CAW	**92.16$\pm$0.1**	**92.56$\pm$0.1**	**92.09$\pm$0.7**	**91.74$\pm$1.7**	**81.29$\pm$0.1**	**85.67$\pm$0.5**
OGN (ours)	<u>90.94$\pm$0.3</u>	<u>81.60$\pm$0.4</u>	<u>81.69$\pm$3.2</u>	77.71$\pm$5.5	71.02$\pm$0.99	83.41$\pm$1.3

B.6. Further intuition behind the weighted average

We show that the neighborhood state $\boldsymbol{r}_u^{(n)}$ in, Equation 3.4, can be seen as an expected value taken over the the neighbors $\mathcal{N}_u^{(n)}$ of node u. For this purpose, we define the probability mass function $p : \mathcal{N}_u^{(n)} \to [0,1]$ given by

$$p(i) = \frac{e^{-\alpha \Delta m_i}}{\sum_j e^{-\alpha \Delta m_j}} \quad \forall i \in \mathcal{N}_u^{(n)}, \tag{B.1}$$

where $\Delta m_i = n - m_i$ Then, it follows by definition that:

$$\boldsymbol{r}_u^{(n)} = \mathbb{E}_{i\sim p}[\boldsymbol{s}_i^{(m_i)}] = \sum_{i\in\mathcal{N}_u^{(n)}} p(i)\boldsymbol{s}_i^{(m_i)} \tag{B.2}$$

Note also that $\boldsymbol{s}_i^{(m_i)}$ also encapsulates information from the neighborhood of node i (see Equation 3.7 and Equation 3.8). Therefore, $\boldsymbol{r}_u^{(n)}$ captures multi-hop information.

B.7. Proof: tractable aggregation

We now show that the ratio between $\boldsymbol{a}_u^{(n)}$ and $\boldsymbol{b}_u^{(n)}$ equals $\boldsymbol{r}_u^{(n)}$. Recall that $\boldsymbol{r}_u^{(n)}$ is given by:

$$\boldsymbol{r}_u^{(n)} = \frac{\sum_{i\in\mathcal{N}_u^{(n)}} e^{-\alpha\Delta m_i}\boldsymbol{s}_i^{(m_i)}}{\sum_{j\in\mathcal{N}_u^{(n)}} e^{-\alpha\Delta m_j}}. \tag{B.3}$$

More specifically, we prove by induction on the number of events n that $\boldsymbol{a}_u^{(n)}$ and $\boldsymbol{b}_u^{(n)}$ equal the numerator and denominator of Equation B.3, respectively. Both proofs are straightforward and follow the same structure.

Proposition 3 For all $n \in \mathbb{N}^+ \cup \{0\}$, it holds that

$$\boldsymbol{a}_u^{(n)} = \sum_{i\in\mathcal{N}_u^{(n)}} e^{-\alpha\Delta m_i}\boldsymbol{s}_i^{(m_i)}.$$

Proof. For $n = 0$, $\boldsymbol{a}_u^{(0)} = 0$ by definition and the identity holds. Assume the above identity holds for an arbitrary $n - 1 \geq 0$, i.e.,

$$\boldsymbol{a}_u^{(n-1)} = \sum_{i\in\mathcal{N}_u^{(n-1)}} e^{-\alpha\Delta m_i}\boldsymbol{s}_i^{(m_i)} = \sum_{i\in\mathcal{N}_u^{(n-1)}} e^{-\alpha(k-m_i)}\boldsymbol{s}_i^{(m_i)},$$

where k is the latest event for node u within the $n-1$ first events. Applying the update to $\boldsymbol{a}_u^{(n-1)}$ (Equation 3.5), we get

$$\boldsymbol{a}_u^{(n)} = e^{-\alpha(n-n)}\boldsymbol{s}_v^{(n)} + e^{-\alpha(n-k)} \sum_{i\in\mathcal{N}_u^{(n-1)}} e^{-\alpha(k-m_i)}\boldsymbol{s}_i^{(m_i)} = \sum_{i\in\mathcal{N}_u^{(n)}} e^{-\alpha(n-m_i)} s_i^{(m_i)}$$

□

Proposition 4 For all $n \in \mathbb{N}^+ \cup \{0\}$, it holds that

$$\boldsymbol{b}_u^{(n)} = \sum_{j\in\mathcal{N}_u^{(n)}} e^{-\alpha\Delta m_j}.$$

Proof. For $n = 0$, $b_u^{(0)} = 0$ by definition and the identity holds. Assume the above identity holds for an arbitrary $n-1 \geq 0$, i.e.,

$$b_u^{(n-1)} = \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha \Delta m_j} = \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha(k-m_i)}$$

where k is the latest event for node u within the $n-1$ first events. Applying the update to $\boldsymbol{b}_u^{(n-1)}$ (Equation 3.6), we get

$$b_u^{(n)} = e^{-\alpha(n-n)} + e^{-\alpha(n-k)} \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha(k-m_i)} = \sum_{j \in \mathcal{N}_u^{(n)}} e^{-\alpha(n-m_j)}$$

□