

---

# Geometric Random Walk Graph Neural Networks via Implicit Layers

---

**Giannis Nikolentzos**

LIX, École Polytechnique  
Institute Polytechnique de Paris, France

**Michalis Vazirgiannis**

LIX, École Polytechnique  
Institute Polytechnique de Paris, France

## Abstract

Graph neural networks have recently attracted a lot of attention and have been applied with great success to several important graph problems. The Random Walk Graph Neural Network model was recently proposed as a more intuitive alternative to the well-studied family of message passing neural networks. This model compares each input graph against a set of latent “hidden graphs” using a kernel that counts common random walks up to some length. In this paper, we propose a new architecture, called Geometric Random Walk Graph Neural Network (GRWNN), that generalizes the above model such that it can count common walks of infinite length in two graphs. The proposed model retains the transparency of Random Walk Graph Neural Networks since its first layer also consists of a number of trainable “hidden graphs” which are compared against the input graphs using the geometric random walk kernel. To compute the kernel, we employ a fixed-point iteration approach involving implicitly defined operations. Then, we capitalize on implicit differentiation to derive an efficient training scheme which requires only constant memory, regardless of the number of fixed-point iterations. Experiments on graph classification datasets demonstrate the effectiveness of the proposed approach in comparison with state-of-the-art methods.

## 1 Introduction

Recent years have witnessed an enormous growth in the amount of data represented as graphs. Indeed, graphs emerge naturally in several domains, including social networks, bioinformatics, and neuroscience, just to name a

few. Besides the increase in the amount of graph-structured data, there is also a growing interest in applying machine learning techniques to data modeled as graphs. Among others, the graph classification and graph regression tasks have attracted a great deal of attention in the past years. These tasks have served as the fundamental building block within applications that deal with problems ranging from drug design (Kearnes et al., 2016) to protein function prediction (Gligorijević et al., 2021).

Graph Neural Networks (GNNs) provide a powerful tool for machine learning on graphs. So far, the field of GNNs has been largely dominated by message passing architectures. Indeed, most of them share the same basic idea, and can be reformulated into a single common framework, so-called message passing neural networks (MPNNs) (Gilmer et al., 2017). These models employ a message passing procedure to aggregate local information of vertices. For graph-related tasks, MPNNs usually apply some permutation invariant readout function to the vertex representations to produce a representation for the entire graph. The family of MPNNs has been heavily studied in the past few years, and there are now available very expressive models which have achieved state-of-the-art results in several tasks (Xu et al., 2019; Morris et al., 2019). Although the family of MPNNs is perhaps the most successful story in the field of graph representation learning, there exist models that follow different design paradigms and do not fall into this family. An example of such a model is the recently proposed Random Walk Graph Neural Network (RWNN) (Nikolentzos and Vazirgiannis, 2020). This model contains a number of trainable “hidden graphs”, and it compares the input graphs against these graphs using a random walk kernel which counts the number of common walks in two graphs. The emerging kernel values are fed into a fully-connected neural network which acts as the classifier or regressor. The employed random walk kernel is differentiable, and thus RWNN is end-to-end trainable. However, this kernel considers only random walks of a small length. Such local patterns may fail to capture the overall large-scale shape of the graphs, while several interesting properties of graphs depend on the graph’s global structure. Furthermore, increasing the length of the walks has a direct impact on the model’s computational complexity.

In this paper, we propose a novel approach to tackle these challenges. Specifically, we propose a new architecture, called Geometric Random Walk Graph Neural Network (GRWNN), that generalizes the RWNN model such that it can count common walks of infinite length in two graphs. The model contains a number of trainable “hidden graphs”, and it compares the input graphs against these graphs using the geometric random walk kernel. Thus, instead of walks of small length, the proposed model considers walks of infinite length. To compute the kernel, GRWNN uses a fixed-point iteration approach. The kernel values are then passed on to a fully-connected neural network which produces the output. The proposed neural network is end-to-end trainable since we can directly differentiate through the fixed-point equations via implicit differentiation, which leads to a very efficient implementation in terms of memory requirements. Hence, we can still update the “hidden graphs” during training with backpropagation. We compare the performance of the proposed model to recently-proposed GNN architectures on several graph classification datasets. Results show that in most cases, the GRWNN model matches or outperforms competing methods. Our main contributions are summarized as follows:

- We propose a novel neural network model, Geometric Random Walk Graph Neural Network, which employs the geometric random walk kernel to produce graph representations. The model counts common walks of infinite length in the input graph and a set of randomly initialized “hidden graphs”.
- We employ an efficient scheme to compute the random walk graph kernel using fixed-point iterations. We show that we can directly differentiate through the fixed-point equations via implicit differentiation, which leads to an efficient implementation.
- We evaluate the model’s performance on several standard graph classification datasets and show that it achieves results similar and in some cases superior to those obtained by recent GNNs.

The rest of this paper is organized as follows. Section 2 provides an overview of the related work. Section 3 introduces some preliminary concepts. Section 4 provides a detailed description of the proposed model. Section 5 evaluates the proposed model in graph classification tasks. Finally, Section 6 concludes.

## 2 Related Work

**Graph kernels.** Graph kernels have a long history in the field of graph representation learning (Kriege et al., 2020; Nikolentzos et al., 2021). A graph kernel is a kernel function between graphs, i. e., a symmetric positive semidefinite function defined on the space of graphs. These meth-

ods generate implicitly (or explicitly) graph representations and enable the application of kernel methods such as the SVM classifier to graphs. Most graph kernels are instances of the  $R$ -convolution framework (Haussler, 1999), and they compare substructures extracted from the graphs to each other. Such substructures include shortest paths (Borgwardt and Kriegel, 2005), random walks (Gärtner et al., 2003; Kashima et al., 2003), small subgraphs (Sheravashidze et al., 2009), and others. Our work is related to random walk kernels, i. e., kernels that compare random walks to each other. The first such kernels were proposed by Gärtner et al. (2003) and by Kashima et al. (2003). The work of Kashima *et al.* was later refined by Mahé et al. (2004). Vishwanathan et al. (2010) and Kang et al. (2012) proposed new algorithms for efficiently computing random walk kernels. These algorithms improve the time complexity of kernel computation. Sugiyama and Borgwardt (2015) studied the problem of halting (i. e., longer walks are downweighted so much that the kernel value is completely dominated by the comparison of walks of length 1) that occurs in random walk kernels, and showed that its extent depends on properties of the graphs being compared. Zhang et al. (2018b) defined a different kernel which does not compare random walks to each other, but instead, compares the return probabilities of random walks. Finally, Kalofolias et al. (2021) proposed a variant of the random walk kernel where structurally dissimilar vertices are not just down-weighted, but are not allowed to be visited during the simultaneous walk.

**GNNs.** Although the first GNNs were proposed several years ago (Sperduti and Starita, 1997; Scarselli et al., 2009; Micheli, 2009), until recently, these models had attracted limited attention. In recent years, with the rise of deep learning, a lot of models started to emerge (Bruna et al., 2014; Li et al., 2015; Duvenaud et al., 2015; Atwood and Towsley, 2016; Defferrard et al., 2016; Lei et al., 2017). Most models update the representation of each vertex by aggregating the feature vectors of its neighbors. This update procedure can be viewed as a form of message passing algorithm and thus, these models are known as message passing neural networks (MPNNs) (Gilmer et al., 2017). To compute a feature vector for the entire graph, MPNNs apply some permutation invariant readout function to all the vertices of the graph. The family of MPNNs has been heavily studied in the past few years, and there are now available several sophisticated models which can produce expressive graph representations (Xu et al., 2019; Morris et al., 2019; Dehmamy et al., 2019; Morris et al., 2020). Despite the general recent focus on MPNNs, some works have proposed architectures that are not variants of this family of models (Niepert et al., 2016; Maron et al., 2019b,a; Nikolentzos and Vazirgiannis, 2020; Nikolentzos et al., 2022). The work closest to ours is the one that proposes the Random Walk Graph Neural Network (RWNN) model (Nikolentzos and Vazirgiannis, 2020). In fact, in this paper, we

generalize the RWNN model to compare random walks of infinite length in two graphs. Recently, the integration of graph kernels into the message passing process of MPNNs has attracted some research interest (Cosmo et al., 2021; Feng et al., 2022). There also exist models that use random walks to extract features from the graph. These features are then processed by standard architectures such as convolutional neural networks (Toenshoff et al., 2021).

**Implicit models.** Our work is also related to implicit models which have been applied successfully to many problems (de Avila Belbute-Peres et al., 2018; Chen et al., 2018; Amos et al., 2018; Bai et al., 2019). The outputs of these models are determined implicitly by a solution of some underlying sub-problem. Implicit models have also been defined in the context of graph representation learning. In fact, one of the earliest GNN models, the model of Scarselli et al. (2009), employs the recurrent backpropagation algorithm (Almeida, 1987; Pineda, 1987), and can be seen as an early form of an implicit model. Recently, the recurrent backpropagation algorithm was further improved and was integrated in a GNN for node classification (Liao et al., 2018). More recently, Gu et al. (2020) proposed IGNN, a model that seeks the fixed-point of some equation which is equivalent to running an infinite number of message passing iterations. Thus, the final representation potentially contains information from all neighbors in the graph capturing long-range dependencies. Gallicchio and Micheli (2020) proposed a similar model which generates graph representations based on the fixed point of a recursive/dynamical system, but is actually only partially trained. In contrast to these approaches whose objective is to apply a large (or infinite) number of message passing layers implicitly, in our setting, we employ a fixed-point iteration approach to compute the random walk kernel and then we directly differentiate through the fixed point equations via implicit differentiation.

### 3 Preliminaries

#### 3.1 Notation

Let  $[n] = \{1, \dots, n\} \subset \mathbb{N}$  for  $n \geq 1$ . Let  $G = (V, E)$  be an undirected graph, where  $V$  is the vertex set and  $E$  is the edge set. We will denote by  $n$  the number of vertices and by  $m$  the number of edges. The adjacency matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$  of a graph  $G$  is a symmetric (typically sparse) matrix used to encode edge information in the graph. The element of the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column is equal to the weight of the edge between vertices  $v_i$  and  $v_j$  if such an edge exists, and 0 otherwise. The degree  $d(v)$  of a vertex  $v$  is equal to the sum of the weights of the edges that are adjacent to the vertex. For vertex-attributed graphs, every vertex in the graph is associated with a feature vector. We use  $\mathbf{X} \in \mathbb{R}^{n \times d}$  to denote the vertex features where  $d$  is the feature dimension size. The feature of a given vertex  $v_i$  corresponds to the  $i^{\text{th}}$

row of  $\mathbf{X}$ .

The direct (tensor) product  $G_{\times} = (V_{\times}, E_{\times})$  of two graphs  $G = (V, E)$  and  $G' = (V', E')$  is defined as follows:

$$\begin{aligned} V_{\times} &= \{(v, v') \in V \times V'\} \\ E_{\times} &= \{((v, v'), (u, u')) \in V_{\times} \times V_{\times} \mid (v, u) \in E, \\ &\quad \text{and } (v', u') \in E'\} \end{aligned}$$

For weighted graphs, the weight of an edge  $((v, v'), (u, u')) \in V_{\times} \times V_{\times}$  is equal to the product of the weights of the edges from which it emerged. We denote by  $\mathbf{A}_{\times}$  the adjacency matrix of  $G_{\times}$ , and denote by  $\Delta_{\times}$  and  $\bar{d}_{\times}$  the maximum and average of the vertex degrees of  $G_{\times}$ , respectively. Thus,  $\bar{d}_{\times} = 1/n \sum_{v \in V_{\times}} d(v)$ . A walk in a graph is a sequence of vertices such that consecutive vertices are linked by an edge. Performing a random walk on the direct product  $G_{\times}$  of two graphs  $G$  and  $G'$  is equivalent to performing a simultaneous random walk on the two graphs  $G$  and  $G'$ .

We use  $\otimes$  to represent the Kronecker product, and use  $\odot$  to represent elementwise multiplication between two matrices or vectors of the same dimension. For a  $p \times q$  matrix  $\mathbf{V}$ ,  $\text{vec}(\mathbf{V}) \in \mathbb{R}^{pq}$  represents the vectorized form of  $\mathbf{V}$ , obtained by stacking its columns. Let also  $\text{vec}^{-1}$  denote the inverse vectorization operator which transforms a vector into a matrix, i. e., for a  $pq$  vector  $\mathbf{v}$ ,  $\mathbf{V} = \text{vec}^{-1}(\mathbf{v})$  where  $\mathbf{V} \in \mathbb{R}^{p \times q}$  (see Appendix A for the exact definition of the  $\text{vec}$  and  $\text{vec}^{-1}$  operators).

#### 3.2 Random Walk Kernel

Given two graphs  $G$  and  $G'$ , the random walk kernel counts all pairs of matching walks on  $G$  and  $G'$  (Gärtner et al., 2003). There are different variants of the kernel. For instance, the  $p$ -step random walk kernel (where  $p \in \mathbb{N}$ ) counts all pairs of matching walks up to length  $p$  on two graphs. The number of matching walks can be obtained through the adjacency matrix  $\mathbf{A}_{\times}$  of the product graph  $G_{\times}$  (Vishwanathan et al., 2010) since a random walk on  $G_{\times}$  is equivalent to a simultaneous random walk on the two graphs. Assuming a uniform distribution for the starting and stopping probabilities over the vertices of two graphs, the  $p$ -step random walk kernel is defined as:

$$\kappa^p(G, G') = \sum_{i=1}^{|V_{\times}|} \sum_{j=0}^{|V_{\times}|} \left[ \sum_{l=0}^p \lambda_l \mathbf{A}_{\times}^l \right]_{ij}$$

where  $\lambda_0, \lambda_1, \lambda_2, \dots, \lambda_p$  are positive, real-valued weights, and  $\mathbf{A}_{\times}^0$  is the identity matrix, i. e.,  $\mathbf{A}_{\times}^0 = \mathbf{I}$ . For  $p \rightarrow \infty$ , we obtain  $\kappa^{\infty}(G, G')$  which is known as the random walk kernel.

It turns out that if the sequence of weights  $\lambda_0, \lambda_1, \lambda_2, \dots$  corresponds to the geometric sequence defined as  $\lambda_l = \lambda^l$ ,

then the limit  $\kappa^\infty(G, G')$  can be computed analytically as follows:

$$\begin{aligned} k^\infty(G, G') &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} \left[ \sum_{l=0}^{\infty} \lambda^l \mathbf{A}_\times^l \right]_{ij} \\ &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} [(\mathbf{I} - \lambda \mathbf{A}_\times)^{-1}]_{ij} \\ &= \mathbf{1}^\top (\mathbf{I} - \lambda \mathbf{A}_\times)^{-1} \mathbf{1} \end{aligned}$$

It is well-known that the geometric series of matrices  $\mathbf{I} + \lambda \mathbf{A}_\times + (\lambda \mathbf{A}_\times)^2 + \dots$  converges only if the largest-magnitude eigenvalue of  $\mathbf{A}_\times$  (which is also the maximum eigenvalue if  $G_\times$  is a graph with non-negative edge weights), denoted by  $\mu_\times^{max}$ , is strictly smaller than  $1/\lambda$ . Therefore, the geometric random walk kernel  $k^\infty$  is well-defined only if  $\lambda < 1/\mu_\times^{max}$ . Interestingly, the maximum eigenvalue of  $\mathbf{A}_\times$  is sandwiched between the average and the maximum of the vertex degrees of  $G_\times$  (Brouwer and Haemers, 2011). We thus have that  $\bar{d}_\times \leq \mu_\times^{max} \leq \Delta_\times$ , and by setting  $\lambda < 1/\Delta_\times$ , the geometric series of matrices is guaranteed to converge.

By defining initial and stopping probability distributions over the vertices of  $G$  and  $G'$ , we can obtain a probabilistic variant of the geometric random walk kernel. Let  $\mathbf{p}$  and  $\mathbf{p}'$  be two vectors that represent the initial probability distributions over the vertices of  $G$  and  $G'$ . Likewise, let  $\mathbf{q}$  and  $\mathbf{q}'$  denote stopping probability distributions over the vertices of  $G$  and  $G'$ . For uniform distributions for the initial and stopping probabilities over the vertices of the two graphs, we have  $\mathbf{p}_i = \mathbf{q}_i = 1/|V|$  and  $\mathbf{p}'_i = \mathbf{q}'_i = 1/|V'|$ . Then,  $\mathbf{p}_\times = \mathbf{p} \mathbf{p}'^\top$  and  $\mathbf{q}_\times = \mathbf{q} \mathbf{q}'^\top$ , and the variant of the geometric random walk kernel can be computed as  $k^\infty(G, G') = \text{vec}(\mathbf{q}_\times)^\top (\mathbf{I} - \lambda \mathbf{A}_\times)^{-1} \text{vec}(\mathbf{p}_\times)$ .

## 4 Geometric Random Walk Graph Neural Networks

The proposed GRWNN model maps input graphs to vectors by comparing them against a number of “hidden graphs”, i.e., graphs whose adjacency and attribute matrices are trainable. The function that we employ to compare the input graphs against the “hidden graphs” is the geometric random walk graph kernel, one of the most well-studied kernels between graphs (Gärtner et al., 2003; Mahé et al., 2004; Vishwanathan et al., 2010). The proposed GRWNN model contains  $N$  “hidden graphs” in total. The graphs may differ from each other in terms of size (i.e., number of vertices). Furthermore, the vertices and/or edges of those graphs can be annotated with continuous multi-dimensional features. As mentioned above, both the structure and the vertex attributes (if any) of these “hidden graphs” are trainable. Thus, the adjacency matrix of a “hidden graph”  $G_i$  of size  $n$  is described by a trainable matrix

$\mathbf{W}_i \in \mathbb{R}^{n \times n}$ , while the vertex attributes are contained in the rows of another trainable matrix  $\mathbf{Q}_i \in \mathbb{R}^{n \times d}$ . Note that the “hidden graphs” correspond to weighted graphs, which can be directed or undirected graphs with or without self-loops. In our implementation, we constraint them to be undirected graphs without self-loops ( $n(n-1)/2$  trainable parameters in total).

To compare an input graph  $G$  against a “hidden graph”  $G_i$ , the model uses the geometric random walk kernel that was introduced in the previous section:

$$\begin{aligned} k^\infty(G, G_i) &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} \left[ \sum_{l=0}^{\infty} \lambda^l \mathbf{A}_\times^l \right]_{ij} \\ &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} [(\mathbf{I} - \lambda \mathbf{A}_\times)^{-1}]_{ij} \\ &= \mathbf{1}^\top (\mathbf{I} - \lambda \mathbf{A}_\times)^{-1} \mathbf{1} \end{aligned} \quad (1)$$

where  $\mathbf{A}_\times = \mathbf{A} \otimes \mathbf{A}_i$  and  $\mathbf{A}_i$  is the adjacency matrix of “hidden graph”  $G_i$  obtained as  $\mathbf{A}_i = f(\mathbf{W}_i)$ . Here,  $f(\cdot)$  is a function whose output is non-negative and potentially bounded, i.e.,  $f(\mathbf{W}_i) = \text{ReLU}(\mathbf{W}_i)$  or  $f(\mathbf{W}_i) = \sigma(\mathbf{W}_i)$  where  $\sigma(\cdot)$  denotes the sigmoid activation function. Then, given the set  $\mathcal{G}_h = \{G_1, G_2, \dots, G_N\}$  where  $G_1, G_2, \dots, G_N$  denote the  $N$  “hidden graphs”, we can compute  $N$  kernel values in total. These kernel values can be thought of as features of the input graph, and can be concatenated to form a vector representation of the input graph. This vector can then be fed into a fully-connected neural network to produce the output.

Following Vishwanathan et al. (2010), to compute the geometric random walk graph kernel shown in Equation (1) above, we employ a two-step approach. We first set  $\mathbf{z}$  equal to:

$$\mathbf{z} = (\mathbf{I} - \lambda \mathbf{A}_\times)^{-1} \mathbf{1} \Leftrightarrow (\mathbf{I} - \lambda \mathbf{A}_\times) \mathbf{z} = \mathbf{1}$$

Then, given  $\mathbf{z}$ , we can compute the kernel value as  $k^\infty(G, G_i) = \mathbf{1}^\top \mathbf{z} = \mathbf{1}^\top (\mathbf{I} - \lambda \mathbf{A}_\times)^{-1} \mathbf{1}$ . Thus, we first need to solve the above linear system for  $\mathbf{z}$ . We first rewrite the above system as:

$$\mathbf{z} = \mathbf{1} + \lambda \mathbf{A}_\times \mathbf{z} \quad (2)$$

Now, solving for  $\mathbf{z}$  is equivalent to finding a fixed point of Equation (2) (Nocedal and Wright, 2006). Such a fixed point can be obtained by simply iterating the first part of the forward pass. Letting  $\mathbf{z}^{(t)}$  denote the value of  $\mathbf{z}$  at iteration  $t$ , we set  $\mathbf{z}^{(0)} = \mathbf{1}$ , and then compute the following:

$$\mathbf{z}^{(t+1)} = \mathbf{1} + \lambda \mathbf{A}_\times \mathbf{z}^{(t)}$$

repeatedly until  $\|\mathbf{z}^{(t+1)} - \mathbf{z}^{(t)}\| < \epsilon$ , where  $\|\cdot\|$  denotes the Euclidean norm and  $\epsilon$  some predefined tolerance or until a specific number of iterations has been reached. As mentioned in the previous section, the above problem is

guaranteed to converge if the maximum eigenvalue of  $\mathbf{A}_\times$  is strictly smaller than  $1/\lambda$ , thus if all the eigenvalues of  $\lambda\mathbf{A}_\times$  lie inside the unit disk. If the values of the elements of  $\mathbf{A}_i$  are bounded, we can compute an upper bound on the maximum degree of  $G_\times$  and set the parameter  $\lambda$  to some value smaller than the inverse of the upper bound.

Once we have computed the kernel  $k^\infty(G, G_i)$  for all  $i \in [N]$ , the representation of the input graph  $G$  is defined as  $\mathbf{h}_G = (k^\infty(G, G_1), \dots, k^\infty(G, G_N))^\top \in \mathbb{R}^N$ . This  $N$ -dimensional vector can be used as features for different machine learning tasks such as graph regression or graph classification.

**Efficient implementation.** If the input graph  $G$  consists of  $n$  vertices and a “hidden graph”  $G_i$  consists of  $c$  vertices, then  $\mathbf{A}_\times$  is an  $nc \times nc$  matrix. Thus, multiplying  $\mathbf{A}_\times$  by some vector inside the fixed-point algorithm requires  $\mathcal{O}(n^2c^2)$  operations in total. Fortunately, to compute the kernel, it is not necessary to explicitly compute matrix  $\mathbf{A}_\times$ . Specifically, the Kronecker product and vec operator are linked by the well-known property (Bernstein, 2009):

$$\text{vec}(\mathbf{A}\mathbf{B}\mathbf{C}) = (\mathbf{C}^\top \otimes \mathbf{A})\text{vec}(\mathbf{B}) \quad (3)$$

Then, let  $\mathbf{Z} \in \mathbb{R}^{c \times n}$  be a matrix such that  $\mathbf{Z} = \text{vec}^{-1}(\mathbf{z})$ . Recall also that  $\mathbf{A}_\times = \mathbf{A} \otimes \mathbf{A}_i$ . Based on the above and on Equation (3), we can write:

$$\begin{aligned} \mathbf{A}_\times \mathbf{z} &= (\mathbf{A} \otimes \mathbf{A}_i)\text{vec}(\mathbf{Z}) \\ &= \text{vec}(\mathbf{A}_i \mathbf{Z} \mathbf{A}^\top) \\ &= \text{vec}(\mathbf{A}_i \text{vec}^{-1}(\mathbf{z}) \mathbf{A}^\top) \end{aligned}$$

The above matrix-vector product can be computed in  $\mathcal{O}(n^2c)$  time in case  $n > c$ . If  $\mathbf{A}$  is sparse, then it can be computed yet more efficiently. Furthermore, we do not need to compute and store matrix  $\mathbf{A}_\times$  which might not be feasible due to high memory requirements. Then, instead of solving the system of Equation (2), we solve the following equivalent system:

$$\mathbf{z} = \mathbf{1} + \lambda \text{vec}(\mathbf{A}_i \text{vec}^{-1}(\mathbf{z}) \mathbf{A}^\top)$$

**Node attributes.** In many real-world problems, vertices of the input graphs are annotated with real-valued multi-dimensional vertex attributes. We next generalize the proposed model to graphs that contain such vertex attributes. Let  $\mathbf{X} \in \mathbb{R}^{n \times d}$  denote the matrix that contains the vertex attributes of the input graph  $G$ . As already mentioned, we also associate a trainable matrix  $\mathbf{Q}_i \in \mathbb{R}^{c \times d}$  to each “hidden graph”  $G_i$ , where  $c$  is the number of vertices of  $G_i$ . Then, let  $\mathbf{S} = \sigma(\mathbf{X} \mathbf{Q}_i^\top) \in \mathbb{R}^{n \times c}$  where  $\sigma(\cdot)$  denotes the sigmoid function. The  $(j, k)^{\text{th}}$  element of matrix  $\mathbf{S}$  is equal to the inner product (followed by a sigmoid) between the attributes of the  $j^{\text{th}}$  vertex of the input graph  $G$  and the  $k^{\text{th}}$  vertex of the “hidden graph”  $G_i$ . Roughly speaking,

this matrix encodes the similarity between the attributes of the vertices of the two graphs. Note that instead of directly using matrix  $\mathbf{X}$ , we can first transform it into matrix  $\tilde{\mathbf{X}}$  using a single- or a multi-layer perceptron. Let  $\mathbf{s} = \text{vec}(\mathbf{S})$  where  $\mathbf{s} \in \mathbb{R}^{nc}$ . Each element of  $\mathbf{s}$  corresponds to a vertex of  $G_\times$  and quantifies the similarity between the attributes of the pair of vertices (i. e., one from  $G$  and one from  $G_i$ ) it represents. Then, we can compute the geometric random walk kernel as follows:

$$\begin{aligned} k^\infty(G, G') &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} \left[ \sum_{l=0}^{\infty} \lambda^l ((\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_\times)^l \right]_{ij} \\ &= \sum_{i=1}^{|V_\times|} \sum_{j=0}^{|V_\times|} \left[ (\mathbf{I} - \lambda(\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_\times)^{-1} \right]_{ij} \\ &= \mathbf{1}^\top (\mathbf{I} - \lambda(\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_\times)^{-1} \mathbf{1} \end{aligned}$$

Note that since the elements of  $\mathbf{s}$  take values between 0 and 1, the same applies to the elements of the output of the outer product  $\mathbf{s} \mathbf{s}^\top$ . Therefore, the maximum degree of the vertices of the graph derived from the matrix  $\mathbf{s} \mathbf{s}^\top \odot \mathbf{A}_\times$  is not greater than that of the graph derived from matrix  $\mathbf{A}_\times$ , and we do not thus need to set  $\lambda$  to a new value. Then, to compute the kernel, we first need to solve the following system:

$$\mathbf{z} = \mathbf{1} + \lambda(\mathbf{s} \mathbf{s}^\top \odot \mathbf{A}_\times) \mathbf{z} \quad (4)$$

Again, naively computing the right part of the above Equation is expensive and requires  $\mathcal{O}(n^2c^2)$  operations in total. The following result shows that in fact we can compute the above in a more time and memory efficient manner.

**Proposition 4.1.** *Let  $\mathbf{A}_1 \in \mathbb{R}^{n \times n}$  and  $\mathbf{A}_2 \in \mathbb{R}^{m \times m}$  be two real matrices. Let also  $\mathbf{s}, \mathbf{y} \in \mathbb{R}^{nm}$  be two real vectors. Then, we have that:*

$$(\mathbf{s} \mathbf{s}^\top \odot (\mathbf{A}_1 \otimes \mathbf{A}_2)) \mathbf{y} = \mathbf{s} \odot \text{vec}(\mathbf{A}_2 \text{vec}^{-1}(\mathbf{y} \odot \mathbf{s}) \mathbf{A}_1^\top)$$

Based on the above result (the proof is left to the Appendix), the system that needs to be solved is:

$$\mathbf{z} = \mathbf{1} + \lambda(\mathbf{s} \odot \text{vec}(\mathbf{A}_i \text{vec}^{-1}(\mathbf{z} \odot \mathbf{s}) \mathbf{A}^\top))$$

Since we store matrix  $\mathbf{A}$  as a sparse matrix, if there are  $\mathcal{O}(m)$  non-zero entries in  $\mathbf{A}$ , then computing one iteration of the above equation for all  $N$  “hidden graphs” takes  $\mathcal{O}(Nc(n(d+c)+m))$  computational time where  $d$  is the size of the vertex attributes.

**Infinite length walks vs. fixed length walks.** Standard GNNs might fail to capture long-range dependencies between nodes in graphs. This motivated the development of graph Transformers which inherently model long-range dependencies (Wu et al., 2021). The proposed GRWNN model can also capture such dependencies since it counts walks of infinite length. As we show next, this property gives the model more expressive power than that of the  $p$ -step RWNN for bounded number of steps.

**Proposition 4.2.** *There are infinitely many pairs of non-isomorphic node-attributed graphs which can be distinguished by the proposed GRWNN model, but not by the  $p$ -step RWNN for bounded number of steps.*

**Convergence.** The geometric series converges if  $\lambda < 1/\mu_{\times}^{max}$  where  $\mu_{\times}^{max}$  denotes the largest-magnitude eigenvalue of  $\mathbf{A}_{\times}$ . We know that  $\mu_{\times}^{max} \leq \Delta_{\times}$  where  $\Delta_{\times}$  is the maximum degree of  $G_{\times}$ . Thus, the series still converges if  $\lambda < 1/\Delta_{\times}$ . Let  $\Delta_{train}$  denote the maximum degree of the vertices of all graphs of the training set and  $\Delta_{hidden}$  denote the maximum degree of the vertices of all “hidden graphs”. Then,  $\Delta_{\times} \leq \Delta_{train} \Delta_{hidden}$ . Thus, the series is well-defined if  $\lambda < 1/(\Delta_{train} \Delta_{hidden})$ . Since we apply the sigmoid function to the adjacency matrix of each “hidden graph”,  $\Delta_{hidden}$  is bounded by  $c - 1$  (where  $c$  is the number of vertices of the “hidden graph”). Then,  $\Delta_{\times} \leq \Delta(c - 1)$  holds. Thus, by setting  $\lambda < 1/(\Delta_{train}(c - 1))$ , the series will converge for all training examples. At inference time, if we assume that the unseen graphs that the model receives are similar to those of the training set, the series are still likely to converge since the maximum degrees of the “hidden graphs” is usually much smaller than the upper bound (i. e.,  $c - 1$ ). In the extreme case where the maximum degree of some unseen graph is much larger than  $\Delta_{train}$ , the series might not converge and in such a case, we need to retrain the model.

**Implicit differentiation.** Clearly, iteratively computing Equation (2) or Equation (4) to find the fixed point corresponds to a differentiable module. However, to train the model, we need to backpropagate the error through the fixed point solver in the backward pass. That would require storing all the intermediate terms, which could be prohibitive in practice. Fortunately, thanks to recent advances in implicit layers and equilibrium models (Bai et al., 2019), this can be performed in a simple and efficient manner which requires constant memory, and assumes no knowledge of the fixed point solver. Specifically, based on ideas from Bai et al. (2019), we derive the form of implicit backpropagation specific to the employed fixed point iteration layer.

**Theorem 4.3.** *Let  $f_{\theta}$  be the system of Equation (2) or Equation (4), and  $\mathbf{z}^* \in \mathbb{R}^{nc}$  be a solution to that linear system. Let also  $g_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) = f_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) - \mathbf{z}^*$ . Since  $\mathbf{z}^*$  is a fixed point, we have that  $g_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) \rightarrow 0$  and  $\mathbf{z}^*$  is thus the root of  $g_{\theta}$ . Let  $y \in \mathbb{R}$  denote the ground-truth target of the input sample,  $h : \mathbb{R} \rightarrow \mathbb{R}$  be any differentiable function and let  $\mathcal{L} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  be a loss function that computes:*

$$\ell = \mathcal{L}(h(\mathbf{1}^{\top} \mathbf{z}^*), y) = \mathcal{L}(h(\mathbf{1}^{\top} \text{FindRoot}(g_{\theta}; \mathbf{A}, \mathbf{X})), y)$$

Then, the gradient of the loss w.r.t.  $(\cdot)$  (e. g.,  $\theta$ ,  $\mathbf{A}$  or  $\mathbf{X}$ ) is:

$$\begin{aligned} \frac{\partial \ell}{\partial (\cdot)} &= -\frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_{\theta}}^{-1} |_{\mathbf{z}^*}) \frac{\partial f_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial (\cdot)} \\ &= -\frac{\partial \ell}{\partial h} \frac{\partial h}{\partial \mathbf{z}^*} (J_{g_{\theta}}^{-1} |_{\mathbf{z}^*}) \frac{\partial f_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial (\cdot)} \end{aligned}$$

where  $J_{g_{\theta}}^{-1} |_{\mathbf{z}^*}$  is the inverse Jacobian of  $g_{\theta}$  evaluated at  $\mathbf{z}^*$ .

The above formula gives a form for the necessary Jacobian without needing to backpropagate through the method used to obtain the fixed point. Thus, as mentioned above, we only need to find the fixed point, and we can compute the necessary Jacobians at this specific point using the above analytical form. No intermediate terms of the iterative method used to compute the fixed point need to be stored in memory, while there is also no need to unroll the forward computations within an automatic differentiation layer. Still, to compute the analytical backward gradient at the solution of the fixed point equation, it is necessary to first compute the exact inverse Jacobian  $J_{g_{\theta}}^{-1}$  which has a cubic cost. As shown in the work of Bai et al. (2019), we can instead compute the  $-\frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_{\theta}}^{-1} |_{\mathbf{z}^*})$  term by solving the following linear system:

$$\mathbf{x} = \left( \frac{\partial f_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)^{\top} \mathbf{x} + \left( \frac{\partial \ell}{\partial \mathbf{z}^*} \right)^{\top}$$

which in fact is also a fixed point equation and can be solved via some iterative procedure. Note that the first term of the above Equation is a vector-Jacobian product which can be efficiently computed via autograd packages (e. g., PyTorch (Paszke et al., 2017)) for any  $\mathbf{x}$ , without explicitly writing out the Jacobian matrix. Finally, we can compute  $\frac{\partial \ell}{\partial (\cdot)}$  as follows:

$$\frac{\partial \ell}{\partial (\cdot)} = \left( \frac{\partial f_{\theta}(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial (\cdot)} \right)^{\top} \mathbf{x}$$

where again this product is itself a vector-Jacobian product, computable via normal automatic differentiation packages.

## 5 Experimental Evaluation

We next evaluate the proposed GRWNN model on standard graph classification datasets.

**Datasets.** We evaluate the proposed model on 10 publicly available graph classification datasets including 5 bio/chemo-informatics datasets: MUTAG, D&D, NCI1, PROTEINS, ENZYMES, and 5 social interaction datasets: IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI-5K, COLLAB (Kersting et al., 2016). To show that the proposed model also scales to larger datasets, we additionally use two Open Graph Benchmark (OGB) datasets (Hu et al., 2020). Specifically, we use a molecular property prediction dataset:

ogbg-molhiv, and a code summarization dataset: ogbg-code2. More details about the datasets are given in Appendix E.

**Experimental Setup.** In the case of the 10 standard benchmark datasets, we compare the proposed model against the following six neural network models: (1) DGCNN (Zhang et al., 2018a), (2) DiffPool (Ying et al., 2018), (3) ECC (Simonovsky and Komodakis, 2017), (4) GIN (Xu et al., 2019), (5) GraphSAGE (Hamilton et al., 2017), and (6) RWNN (Nikolentzos and Vazirgiannis, 2020). We also compare the proposed model against GRWNN-fixed, a variant of the model whose “hidden graphs” are randomly initialized and kept fixed during training. To evaluate the proposed model, we employ the experimental protocol proposed by Errica et al. (2020). Therefore, we perform 10-fold cross-validation to obtain an estimate of the generalization performance of each method, while within each fold a model is selected based on a 90%/10% split of the training set. We use exactly the same splits as Errica et al. (2020) and Nikolentzos and Vazirgiannis (2020), hence, for the different datasets, we use the results reported in these two papers.

For all datasets, we set the batch size to 64 and the number of epochs to 300. We use the Adam optimizer with initial learning rate 0.001 and applied an adaptive learning rate decay based on validation results. We use a 1-layer perceptron to transform the vertex attributes. We apply layer normalization (Ba et al., 2016) on the generated graph representations (i. e., vector consisting of kernel values). The hyper-parameters we tune for each dataset are: (1) the number of “hidden graphs”  $\in \{32, 64\}$ , (2) the number of vertices of the “hidden graphs”  $\in \{5, 10\}$ , (3) the hidden-dimension size of the vertex features  $\in \{32, 64\}$  for the bio/chemo-informatics datasets and  $\in \{8, 16\}$  for the social interaction datasets, and (4) the dropout ratio  $\in \{0, 0.1\}$ .

For both OGB datasets, we used the available predefined splits. We compare the proposed model against the following neural network models: GCN (Kipf and Welling, 2017), GIN (Xu et al., 2019), GCN-FLAG (Kong et al., 2020), GIN-FLAG (Kong et al., 2020), PNA (Corso et al., 2020), GSN (Bouritsas et al., 2020), HIMP (Fey et al., 2020), and DGN (Beaini et al., 2020). For all models, we use the results that are reported in the respective papers. For ogbg-code2, we did not add the inverse edges to the graphs. All reported results are averaged over 10 runs.

For both OGB datasets, we set the batch size to 128. For the ogbg-molhiv dataset, we set the number of epochs to 300, the number of “hidden graphs” to 200, the number of vertices of the “hidden graphs” to 5, the hidden-dimension size of the vertex features to 128 and the dropout ratio to 0.1. Furthermore, we employ the probabilistic variant of the geometric random walk kernel and use uniform distributions for the initial and stopping probabilities over the

vertices of the two compared graphs. For the ogbg-code2 dataset, we set the number of epochs to 100, the number of “hidden graphs” to 200, the number of vertices of the “hidden graphs” to 5, the hidden-dimension size of the vertex features to 128 and the dropout ratio to 0.2. For both datasets, we apply layer normalization (Ba et al., 2016) on the generated graph representations.

**Implementation Details.** To set the value of parameter  $\lambda$ , we assume a transductive setting, where we are given a collection of graphs beforehand. Therefore, we can find the vertex of highest degree across all graphs and set the value of  $\lambda$  accordingly. We can explicitly compute the upper bound of  $\lambda$ , which is  $(\max_{G, G' \in \mathcal{G}} \mu_{\times, max})^{-1}$  with the maximum of the maximum eigenvalues over all pairs of graphs  $G, G' \in \mathcal{G}$ . In the inductive learning setting, since we do not know a priori target graphs that the model may receive in the future,  $\lambda$  should be small enough so that  $\lambda < 1/\mu_{\times}^{max}$  for any pair of an unseen graph and a “hidden graph”. This is a limitation of the proposed model since in case the model receives at test time a graph whose largest eigenvalue is higher than expected, we need to set  $\lambda$  to a smaller value and retrain the model.

To compute the fixed point of Equation (2) or (4), we followed the naive approach where we simply performed multiple times the forward iteration. In practice, there are more efficient fixed point iteration methods, such as Anderson Acceleration (Walker and Ni, 2011), that converge faster than the naive forward iteration at the cost of some additional memory complexity. However, as shown next, we found that in our setting, the naive forward iteration converges in a small number of steps, while the additional cost introduced by more efficient methods associated with the generation and manipulation of new tensors made them overall slower than the naive forward iteration even though they required fewer iterations to converge.

The model was implemented with PyTorch (Paszke et al., 2019), and all experiments were run on a single machine equipped with an NVidia Titan Xp GPU card. Our code is available at <https://github.com/giannisnik/grwnn>.

**Results.** Table 1 illustrates average prediction accuracies and standard deviations for the 10 standard graph classification datasets. We observe that the proposed GRWNN model is the most successful method since it outperforms all baseline models on 4 out of the 10 datasets, while it provides the second best and third best accuracy on 1 and 2 out of the remaining 6 datasets, respectively. On the remaining 6 datasets, GIN is the best-performing model on half of them, and RWNN on the other half. On the ENZYMES and IMDB-BINARY datasets, our model offers respective absolute improvements of 3.1%, and 1.6% in accuracy over GIN and even greatest improvements over RWNN. On the NCI1 dataset, all models that count random walks achieve

Table 1: Classification accuracy ( $\pm$  standard deviation) of the proposed model and the baselines on the 5 chemo/bio-informatics and on the 5 social interaction benchmark datasets. OOR means Out of Resources, either time ( $> 72$  hours for a single training) or GPU memory. Best performance per dataset in **bold**.

	MUTAG	D&D	NCII	PROTEINS	ENZYMES
DGCNN	84.0 ( $\pm 6.7$ )	76.6 ( $\pm 4.3$ )	76.4 ( $\pm 1.7$ )	72.9 ( $\pm 3.5$ )	38.9 ( $\pm 5.7$ )
DiffPool	79.8 ( $\pm 7.1$ )	75.0 ( $\pm 3.5$ )	76.9 ( $\pm 1.9$ )	73.7 ( $\pm 3.5$ )	59.5 ( $\pm 5.6$ )
ECC	75.4 ( $\pm 6.2$ )	72.6 ( $\pm 4.1$ )	76.2 ( $\pm 1.4$ )	72.3 ( $\pm 3.4$ )	29.5 ( $\pm 8.2$ )
GIN	84.7 ( $\pm 6.7$ )	75.3 ( $\pm 2.9$ )	<b>80.0</b> ( $\pm 1.4$ )	73.3 ( $\pm 4.0$ )	59.6 ( $\pm 4.5$ )
GraphSAGE	83.6 ( $\pm 9.6$ )	72.9 ( $\pm 2.0$ )	76.0 ( $\pm 1.8$ )	73.0 ( $\pm 4.5$ )	58.2 ( $\pm 6.0$ )
1-step RWNN	<b>89.2</b> ( $\pm 4.3$ )	<b>77.6</b> ( $\pm 4.7$ )	71.4 ( $\pm 1.8$ )	74.7 ( $\pm 3.3$ )	56.7 ( $\pm 5.2$ )
2-step RWNN	88.1 ( $\pm 4.8$ )	76.9 ( $\pm 4.6$ )	73.0 ( $\pm 2.0$ )	74.1 ( $\pm 2.8$ )	57.4 ( $\pm 4.9$ )
GRWNN-fixed	81.9 ( $\pm 6.4$ )	73.2 ( $\pm 3.5$ )	66.9 ( $\pm 2.4$ )	74.6 ( $\pm 4.0$ )	56.8 ( $\pm 3.7$ )
GRWNN	83.4 ( $\pm 5.6$ )	75.6 ( $\pm 4.6$ )	67.7 ( $\pm 2.2$ )	<b>74.9</b> ( $\pm 3.5$ )	<b>62.7</b> ( $\pm 5.2$ )

	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
DGCNN	69.2 ( $\pm 3.0$ )	45.6 ( $\pm 3.4$ )	87.8 ( $\pm 2.5$ )	49.2 ( $\pm 1.2$ )	71.2 ( $\pm 1.9$ )
DiffPool	68.4 ( $\pm 3.3$ )	45.6 ( $\pm 3.4$ )	89.1 ( $\pm 1.6$ )	53.8 ( $\pm 1.4$ )	68.9 ( $\pm 2.0$ )
ECC	67.7 ( $\pm 2.8$ )	43.5 ( $\pm 3.1$ )	OOO	OOO	OOO
GIN	71.2 ( $\pm 3.9$ )	48.5 ( $\pm 3.3$ )	89.9 ( $\pm 1.9$ )	<b>56.1</b> ( $\pm 1.7$ )	<b>75.6</b> ( $\pm 2.3$ )
GraphSAGE	68.8 ( $\pm 4.5$ )	47.6 ( $\pm 3.5$ )	84.3 ( $\pm 1.9$ )	50.0 ( $\pm 1.3$ )	73.9 ( $\pm 1.7$ )
1-step RWNN	70.8 ( $\pm 4.8$ )	47.8 ( $\pm 3.8$ )	<b>90.4</b> ( $\pm 1.9$ )	51.7 ( $\pm 1.5$ )	71.7 ( $\pm 2.1$ )
2-step RWNN	70.6 ( $\pm 4.4$ )	48.8 ( $\pm 2.9$ )	90.3 ( $\pm 1.8$ )	51.7 ( $\pm 1.4$ )	71.3 ( $\pm 2.1$ )
GRWNN-fixed	72.1 ( $\pm 4.1$ )	48.1 ( $\pm 3.6$ )	82.2 ( $\pm 2.4$ )	53.1 ( $\pm 1.8$ )	71.3 ( $\pm 1.9$ )
GRWNN	<b>72.8</b> ( $\pm 4.2$ )	<b>49.0</b> ( $\pm 2.9$ )	90.0 ( $\pm 1.8$ )	54.4 ( $\pm 1.7$ )	72.1 ( $\pm 1.9$ )

low levels of accuracy. We believe that this is due to some underlying properties of the dataset that need to be investigated. Overall, the model exhibits highly competitive performance on the graph classification datasets, while the achieved accuracies follow different patterns from all the baseline methods. Furthermore, the proposed model outperforms GRWNN-fixed on all datasets, demonstrating that the set of trainable “hidden graphs” is an indispensable component of the model. With regards to the comparison between GRWNN and RWNN, we should mention that the former is sensitive to the large scale structure of graphs, while the latter computes a function of two graphs that is purely local. We hypothesize that on some datasets, local patterns are better indicators of class membership than the overall structure of the graph, and therefore, 1- and 2-step RWNNs outperform the proposed model.

The Table shown in Figure 1a illustrates the performance on the two OGB datasets. Note that the proposed model does not utilize the edge features that are provided for the different datasets. Still, we can see that it managed to outperform several of the baselines on the `ogbg-molhiv` dataset, where it achieved the fourth best ROC-AUC. On the `ogbg-code2` dataset, GRWNN outperformed GIN, while it achieved an F1-score similar to that of GCN. However, all these three models achieved a much smaller F1-

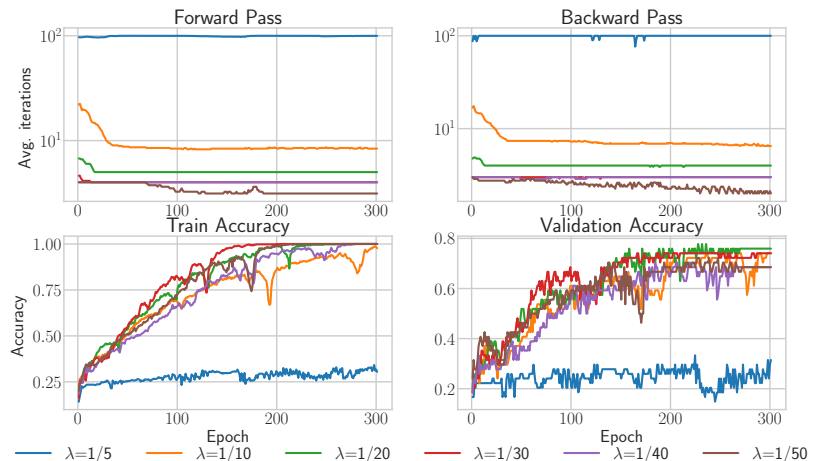
score than the one achieved by PNA which is the best-performing model.

As already discussed, the running time of the model depends on the number of fixed point iterations that need to be performed until convergence. Figure 1b (top) illustrates the average number of iterations (across all batches) for the forward and backward pass for different values of  $\lambda$  and for each epoch. The model was trained on a single split of the ENZYMES dataset. The maximum eigenvalue of all graphs of the dataset is equal to 5.47, while the highest degree is equal to 9. The number of nodes of the “hidden graphs” was set to 5. If the elements of the adjacency matrices of the “hidden graphs” take values no greater than one, then no vertex of  $G_x$  can have a degree greater than  $9 * 4 = 36$ . Thus, setting  $\lambda < 1/36$  guarantees convergence. In practice, as shown in the Figure, we found that even if  $\lambda$  takes larger values, we only need a small number of iterations. For  $\lambda = 1/5$ , we can see that the fixed point equation fails to converge since the average number of iterations is close to 100 (which is the upper limit we have set). For  $\lambda = 1/10$  and for smaller values of  $\lambda$ , the system converges in a small number of iterations. In terms of performance, as shown in Figure 1b (bottom), the model achieves the highest levels of validation accuracy for  $\lambda = 1/20$  and  $\lambda = 1/30$ , while for  $\lambda = 1/5$ , the model yields much worse



Method	Dataset	
	ogbg-molhiv	ogbg-code2
GCN	76.06 ± 0.97	15.07 ± 0.18
GIN	75.58 ± 1.40	14.95 ± 0.23
GCN+FLAG	76.83 ± 1.02	–
GIN+FLAG	76.54 ± 1.14	–
GSN	77.99 ± 1.00	–
HIMP	78.80 ± 0.82	–
PNA	79.05 ± 1.32	15.70 ± 0.32
DGN	79.70 ± 0.97	–
GRWNN	78.38 ± 0.99	15.03 ± 0.21

(a) Performance of the proposed model and the baselines on the OGB datasets. Reported values correspond to ROC-AUC scores for `ogbg-molhiv` and F1-scores for `ogbg-code2`.



(b) Number of fixed point iterations during the forward and backward pass (top), and training and validation accuracy (bottom) on the ENZYMES dataset for different values of  $\lambda$ .

Figure 1: Performance on the OGB datasets and impact of the value of parameter  $\lambda$  on running time and performance of the model.

performance compared to the other values of  $\lambda$ . Similar behavior was observed on the other datasets.

## 6 Conclusion

In this paper, we introduced the GRWNN model, a new architecture which generates graph representations by comparing the input graphs against “hidden graphs” using the geometric random walk kernel. To compute the kernel, the proposed model uses a fixed point iteration algorithm, and to update the “hidden graphs” during backpropagation, the model capitalizes on implicit differentiation and employs an efficient training scheme which requires only constant memory, regardless of the number of fixed-point iterations. The model was evaluated on several graph classification datasets where it achieved competitive performance.

### Acknowledgements

G.N. is supported by the French National research agency via the AML-HELAS (ANR-19-CHIA-0020) project.

### References

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Proceedings of the 1st IEEE International Conference on Neural Networks*, pages 609–618.

Amos, B., Rodriguez, I. D. J., Sacks, J., Boots, B., and Kolter, J. Z. (2018). Differentiable MPC for End-to-end Planning and Control. In *Advances in Neural Information Processing Systems*, volume 31, pages 8299–8310.

Atwood, J. and Towsley, D. (2016). Diffusion-Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 29, pages 1993–2001.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bai, S., Kolter, J. Z., and Koltun, V. (2019). Deep Equilibrium Models. In *Advances in Neural Information Processing Systems*, volume 32, pages 690–701.

Beaini, D., Passaro, S., Létourneau, V., Hamilton, W. L., Corso, G., and Liò, P. (2020). Directional Graph Networks. *arXiv preprint arXiv:2010.02863*.

Bernstein, D. S. (2009). *Matrix mathematics: theory, facts, and formulas*. Princeton University Press.

Borgwardt, K., Ong, C., Schönauer, S., Vishwanathan, S., Smola, A., and Kriegel, H. (2005). Protein function prediction via graph kernels. *Bioinformatics*, 21(Suppl. 1):i47–i56.

Borgwardt, K. M. and Kriegel, H. (2005). Shortest-path kernels on graphs. In *Proceedings of the 5th International Conference on Data Mining*, pages 74–81.

Bouritsas, G., Frasca, F., Zafeiriou, S., and Bronstein, M. M. (2020). Improving graph neural network expressivity via subgraph isomorphism counting. *arXiv preprint arXiv:2006.09252*.

Brouwer, A. E. and Haemers, W. H. (2011). *Spectra of graphs*. Springer Science & Business Media.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2014). Spectral Networks and Locally connected networks on

- Graphs. In *2nd International Conference on Learning Representations*.
- Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. (2018). Neural Ordinary Differential Equations. In *Advances in Neural Information Processing Systems*, volume 31, pages 6572–6583.
- Corso, G., Cavalleri, L., Beaini, D., Liò, P., and Veličković, P. (2020). Principal Neighbourhood Aggregation for Graph Nets. In *Advances in Neural Information Processing Systems*, volume 33.
- Cosmo, L., Minello, G., Bronstein, M., Rodolà, E., Rossi, L., and Torsello, A. (2021). Graph Kernel Neural Networks. *arXiv preprint arXiv:2112.07436*.
- de Avila Belbute-Peres, F., Smith, K., Allen, K., Tenenbaum, J., and Kolter, J. Z. (2018). End-to-End Differentiable Physics for Learning and Control. *Advances in neural information processing systems*, 31:7178–7189.
- Debnath, A., Lopez de Compadre, R., Debnath, G., Shusterman, A., and Hansch, C. (1991). Structure-Activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro Compounds. Correlation with Molecular Orbital Energies and Hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797.
- Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems*, volume 29, pages 3837–3845.
- Dehmamy, N., Barabasi, A.-L., and Yu, R. (2019). Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology. In *Advances in Neural Information Processing Systems*, volume 32.
- Dobson, P. and Doig, A. (2003). Distinguishing Enzyme Structures from Non-enzymes Without Alignments. *Journal of Molecular Biology*, 330(4):771–783.
- Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Advances in Neural Information Processing Systems*, volume 28.
- Errica, F., Podda, M., Bacciu, D., and Micheli, A. (2020). A Fair Comparison of Graph Neural Networks for Graph Classification. In *Proceedings of the International Conference on Learning Representations*.
- Feng, A., You, C., Wang, S., and Tassiulas, L. (2022). KerGNNs: Interpretable Graph Neural Networks with Graph Kernels. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, pages 6614–6622.
- Fey, M., Yuen, J.-G., and Weichert, F. (2020). Hierarchical Inter-Message Passing for Learning on Molecular Graphs. *arXiv preprint arXiv:2006.12179*.
- Gallicchio, C. and Micheli, A. (2020). Fast and Deep Graph Neural Networks. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, pages 3898–3905.
- Gärtner, T., Flach, P., and Wrobel, S. (2003). On Graph Kernels: Hardness Results and Efficient Alternatives. In *Learning Theory and Kernel Machines*, pages 129–143.
- Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1263–1272.
- Gligorijević, V., Renfrew, P. D., Kosciulek, T., Leman, J. K., Berenberg, D., Vatanen, T., Chandler, C., Taylor, B. C., Fisk, I. M., Vlamakis, H., et al. (2021). Structure-based protein function prediction using graph convolutional networks. *Nature communications*, 12(1):3168.
- Gu, F., Chang, H., Zhu, W., Sojoudi, S., and El Ghaoui, L. (2020). Implicit Graph Neural Networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 11984–11995.
- Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034.
- Hausler, D. (1999). Convolution kernels on discrete structures. Technical report, Technical report, Department of Computer Science, University of California at Santa Cruz.
- Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. (2020). Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687*.
- Kalofolias, J., Welke, P., and Vreeken, J. (2021). SUSAN: The Structural Similarity Random Walk Kernel. In *Proceedings of the 2021 SIAM International Conference on Data Mining*.
- Kang, U., Tong, H., and Sun, J. (2012). Fast Random Walk Graph Kernel. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 828–838.
- Kashima, H., Tsuda, K., and Inokuchi, A. (2003). Marginalized Kernels Between Labeled Graphs. In *Proceedings of the 20th International Conference on Machine Learning*, pages 321–328.
- Kearnes, S., McCloskey, K., Berndl, M., Pande, V., and Riley, P. (2016). Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, 30(8):595–608.
- Kersting, K., Kriege, N. M., Morris, C., Mutzel, P., and Neumann, M. (2016). Benchmark data sets for graph kernels. <http://graphkernels.cs.tu-dortmund.de>.

- Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *In 5th International Conference on Learning Representations*.
- Kong, K., Li, G., Ding, M., Wu, Z., Zhu, C., Ghanem, B., Taylor, G., and Goldstein, T. (2020). Flag: Adversarial Data Augmentation for Graph Neural Networks. *arXiv preprint arXiv:2010.09891*.
- Kriege, N. M., Johansson, F. D., and Morris, C. (2020). A survey on graph kernels. *Applied Network Science*, 5(1):1–42.
- Lei, T., Jin, W., Barzilay, R., and Jaakkola, T. (2017). Deriving Neural Architectures from Sequence and Graph Kernels. In *Proceedings of the 34th International Conference on Machine Learning*, pages 2024–2033.
- Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2015). Gated Graph Sequence Neural Networks. In *3rd International Conference on Learning Representations*.
- Liao, R., Xiong, Y., Fetaya, E., Zhang, L., Yoon, K., Pitkow, X., Urtasun, R., and Zemel, R. (2018). Reviving and Improving Recurrent Back-Propagation. In *Proceedings of the 35th International Conference on Machine Learning*, pages 3082–3091.
- Mahé, P., Ueda, N., Akutsu, T., Perret, J.-L., and Vert, J.-P. (2004). Extensions of Marginalized Graph Kernels. In *Proceedings of the 21st International Conference on Machine Learning*.
- Maron, H., Ben-Hamu, H., Serviansky, H., and Lipman, Y. (2019a). Provably Powerful Graph Networks. In *Advances in Neural Information Processing Systems*.
- Maron, H., Ben-Hamu, H., Shamir, N., and Lipman, Y. (2019b). Invariant and Equivariant Graph Networks. In *7th International Conference on Learning Representations*.
- Micheli, A. (2009). Neural Network for Graphs: A Contextual Constructive Approach. *IEEE Transactions on Neural Networks*, 20(3):498–511.
- Morris, C., Rattan, G., and Mutzel, P. (2020). Weisfeiler and Leman go sparse: Towards scalable higher-order graph embeddings. In *Advances in Neural Information Processing Systems*, volume 33, pages 21824–21840.
- Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. (2019). Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 4602–4609.
- Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning Convolutional Neural Networks for Graphs. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 2014–2023.
- Nikolentzos, G., Dasoulas, G., and Vazirgiannis, M. (2022). Permute Me Softly: Learning Soft Permutations for Graph Representations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- Nikolentzos, G., Siglidis, G., and Vazirgiannis, M. (2021). Graph kernels: A survey. *Journal of Artificial Intelligence Research*, 72:943–1027.
- Nikolentzos, G. and Vazirgiannis, M. (2020). Random Walk Graph Neural Networks. *Advances in Neural Information Processing Systems*, 33:16211–16222.
- Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer Science & Business Media.
- Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037.
- Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229.
- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80.
- Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., and Borgwardt, K. M. (2009). Efficient graphlet kernels for large graph comparison. In *The 12th International Conference on Artificial Intelligence and Statistics*, pages 488–495.
- Simonovsky, M. and Komodakis, N. (2017). Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3693–3702.
- Sperduti, A. and Starita, A. (1997). Supervised Neural Networks for the Classification of Structures. *IEEE Transactions on Neural Networks*, 8(3):714–735.
- Sugiyama, M. and Borgwardt, K. (2015). Halting in Random Walk Kernels. *Advances in Neural Information Processing Systems*, 28:1639–1647.
- Toenshoff, J., Ritzert, M., Wolf, H., and Grohe, M. (2021). Graph learning with 1d convolutions on random walks. *arXiv preprint arXiv:2102.08786*.
- Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph Kernels. *Journal of Machine Learning Research*, 11:1201–1242.
- Wale, N., Watson, I., and Karypis, G. (2008). Comparison of descriptor spaces for chemical compound retrieval and classification. *Knowledge and Information Systems*, 14(3):347–375.

- Walker, H. F. and Ni, P. (2011). Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49(4):1715–1735.
- Wu, Z., Jain, P., Wright, M., Mirhoseini, A., Gonzalez, J. E., and Stoica, I. (2021). Representing Long-Range Context for Graph Neural Networks with Global Attention. In *Advances in Neural Information Processing Systems*, pages 13266–13279.
- Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., and Pande, V. (2018). Moleculenet: a benchmark for molecular machine learning. *Chemical Science*, 9(2):513–530.
- Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How Powerful are Graph Neural Networks? In *7th International Conference on Learning Representations*.
- Yanardag, P. and Vishwanathan, S. (2015). Deep Graph Kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374.
- Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. (2018). Hierarchical Graph Representation Learning with Differentiable Pooling. In *Advances in Neural Information Processing Systems*, pages 4801–4811.
- Zhang, M., Cui, Z., Neumann, M., and Chen, Y. (2018a). An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 4438–4445.
- Zhang, Z., Wang, M., Xiang, Y., Huang, Y., and Nehorai, A. (2018b). RetGK: Graph Kernels based on Return Probabilities of Random Walks. *Advances in Neural Information Processing Systems*, 31:3964–3974.

The Appendix is organized as follows. In section A, we define some basic concepts from linear algebra. In sections B and C, we prove Propositions 4.1 and 4.2, respectively. In section D, we give more details about the direct differentiation through the fixed point, while section E provides a detailed description of the graph classification datasets. Section F gives details about parameter  $\lambda$ , and section G about the running time of the model. Finally, in section H, we present the results of a sensitivity analysis.

## A Linear Algebra Concepts

In this section, we provide definitions for concepts of linear algebra, namely the vectorization operator, the inverse vectorization operator and the Kronecker product, which we use heavily in the main paper.

**Definition A.1.** Given a real matrix  $\mathbf{A} \in \mathbb{R}^{m \times n}$ , the vectorization operator  $\text{vec} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{mn}$  is defined as:

$$\text{vec}(\mathbf{A}) = \begin{bmatrix} \mathbf{A}_{:1} \\ \mathbf{A}_{:2} \\ \vdots \\ \mathbf{A}_{:n} \end{bmatrix}$$

where  $\mathbf{A}_{:i}$  is the  $i^{\text{th}}$  column of  $\mathbf{A}$ .

**Definition A.2.** Given a real vector  $\mathbf{b} \in \mathbb{R}^{nm}$ , the inverse vectorization operator  $\text{vec}^{-1} : \mathbb{R}^{nm} \rightarrow \mathbb{R}^{n \times m}$  is defined as:

$$\text{vec}^{-1}(\mathbf{b}) = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_{n+1} & \dots & \mathbf{b}_{n(m-1)+1} \\ \mathbf{b}_2 & \mathbf{b}_{n+2} & \dots & \mathbf{b}_{n(m-1)+2} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{b}_n & \mathbf{b}_{2n} & \dots & \mathbf{b}_{nm} \end{bmatrix}$$

**Definition A.3.** Given real matrices  $\mathbf{A} \in \mathbb{R}^{n \times m}$  and  $\mathbf{B} \in \mathbb{R}^{p \times q}$ , the Kronecker product  $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{np \times mq}$  defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}_{11} \mathbf{B} & \mathbf{A}_{12} \mathbf{B} & \dots & \mathbf{A}_{1m} \mathbf{B} \\ \mathbf{A}_{21} \mathbf{B} & \mathbf{A}_{22} \mathbf{B} & \dots & \mathbf{A}_{2m} \mathbf{B} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{A}_{n1} \mathbf{B} & \mathbf{A}_{n2} \mathbf{B} & \dots & \mathbf{A}_{nm} \mathbf{B} \end{bmatrix}$$

## B Proof of Proposition 4.1

For convenience, we restate the Proposition below.

**Proposition 4.1.** Let  $\mathbf{A}_1 \in \mathbb{R}^{n \times n}$  and  $\mathbf{A}_2 \in \mathbb{R}^{m \times m}$  be two real matrices. Let also  $\mathbf{s}, \mathbf{y} \in \mathbb{R}^{nm}$  be two real vectors. Then, we have that:

$$(\mathbf{s} \mathbf{s}^\top \odot (\mathbf{A}_1 \otimes \mathbf{A}_2)) \mathbf{y} = \mathbf{s} \odot \text{vec}(\mathbf{A}_2 \text{vec}^{-1}(\mathbf{s} \odot \mathbf{y}) \mathbf{A}_1^\top)$$

*Proof.* Let  $\mathbf{D}_s$  denote a diagonal matrix with the vector  $\mathbf{s}$  as its main diagonal. Then, we have:

$$(\mathbf{s} \mathbf{s}^\top \odot (\mathbf{A}_1 \otimes \mathbf{A}_2)) \mathbf{y} = (\mathbf{D}_s (\mathbf{A}_1 \otimes \mathbf{A}_2) \mathbf{D}_s) \mathbf{y}$$

The Hadamard product of two vectors  $\mathbf{s}$  and  $\mathbf{y}$  is the same as matrix multiplication of vector  $\mathbf{y}$  by the corresponding diagonal matrix  $\mathbf{D}_s$  of vector  $\mathbf{s}$ , i. e.,  $\mathbf{D}_s \mathbf{y} = \mathbf{s} \odot \mathbf{y}$ . Thus, it follows that:

$$(\mathbf{D}_s (\mathbf{A}_1 \otimes \mathbf{A}_2) \mathbf{D}_s) \mathbf{y} = \mathbf{D}_s (\mathbf{A}_1 \otimes \mathbf{A}_2) (\mathbf{s} \odot \mathbf{y})$$

Note that the Kronecker product and vec operator are linked by the well-known property (Bernstein, 2009)(Proposition 7.1.9):

$$\text{vec}(\mathbf{A} \mathbf{B} \mathbf{C}) = (\mathbf{C}^\top \otimes \mathbf{A}) \text{vec}(\mathbf{B})$$

Therefore, we have that:

$$(\mathbf{D}_s (\mathbf{A}_1 \otimes \mathbf{A}_2) (\mathbf{s} \odot \mathbf{y})) = \mathbf{D}_s \text{vec}(\mathbf{A}_2 \text{vec}^{-1}(\mathbf{s} \odot \mathbf{y}) \mathbf{A}_1^\top) = \mathbf{s} \odot \text{vec}(\mathbf{A}_2 \text{vec}^{-1}(\mathbf{s} \odot \mathbf{y}) \mathbf{A}_1^\top)$$

which concludes the proof.  $\square$

## C Proof of Proposition 4.2

For convenience, we restate the Proposition below.

**Proposition 4.2.** *There are infinitely many pairs of non-isomorphic node-attributed graphs which can be distinguished by the proposed GRWNN model, but not by the  $p$ -step RWNN for bounded number of steps.*

*Proof.* Let  $C_n$  denote an  $n$ -cycle, i. e., a graph on  $n$  nodes containing a single cycle through all nodes. Let also  $C_n^2$  be the union of two copies of  $n$ -cycle with disjoint vertex sets  $V_1 = \{1, \dots, n\}$  and  $V_2 = \{n+1, \dots, 2n\}$  and edge sets  $E_1$  and  $E_2$ , i. e.,  $C_n^2 = (V, E)$  with  $V = V_1 \cup V_2$  and  $E = E_1 \cup E_2$ . Let  $n \in \{n \mid n \in \mathbb{N}, n \geq 3\}$ . We annotate vertex  $\lceil \frac{n}{2} \rceil$  of the first component of  $C_n^2$  with a feature equal to  $x_1$  and vertex  $\lceil \frac{n}{2} \rceil + n$  of the second component of  $C_n^2$  with a feature equal to  $x_2$  with  $x_1 \neq x_2$ . We also annotate vertices  $\lceil \frac{n}{2} \rceil$  and  $\lceil \frac{n}{2} \rceil + n$  of  $C_{2n}$  (with shortest path distance  $n$ ) with a feature equal to  $x_1$  and  $x_2$ , respectively. The rest of the vertices of both graphs (i. e.,  $C_n^2$  and  $C_{2n}$ ) are annotated with a feature equal to  $x \neq 0$  where  $x \neq x_1$  and  $x \neq x_2$ . Since the two graphs share identical node features, these can be stored into a matrix  $\mathbf{X} \in \mathbb{R}^{2n \times 1}$ .

The  $p$ -step RWNN model computes features for the different values of  $p$  (where  $p \geq 0$ ) as follows (Nikolentzos and Vazirgiannis, 2020):

$$k^{(p)}(G, G_h) = \sum_{i=1}^c \sum_{j=1}^n \left[ \mathbf{Q} \mathbf{X}^\top \odot (\text{RELU}(\mathbf{W})^p \mathbf{Q} \mathbf{X}^\top \mathbf{A}^p) \right]_{ij}$$

where  $G$  is the input graph and  $G_h$  is a ‘‘hidden graph’’ consisting of  $c$  vertices whose attributes are stored in matrix  $\mathbf{Q}$ , while  $\text{RELU}(\mathbf{W})$  is the adjacency matrix of  $G_h$ . Then, note that for  $p < \lceil \frac{n}{2} \rceil$ , corresponding vertices of  $C_n^2$  and  $C_{2n}$  have isomorphic  $p$ -hop neighborhoods (also considering node features). Since these neighborhoods are isomorphic, it turns out that  $\mathbf{X}^\top \mathbf{A}_{C_n^2}^p = \mathbf{X}^\top \mathbf{A}_{C_{2n}}^p$  where  $\mathbf{A}_{C_n^2}$  and  $\mathbf{A}_{C_{2n}}$  denote the adjacency matrices of  $C_n^2$  and  $C_{2n}$ , respectively. Therefore,  $k^{(p)}(C_n^2, G_h) = k^{(p)}(C_{2n}, G_h)$  holds for any ‘‘hidden graph’’  $G_h$  and for any  $p < \lceil \frac{n}{2} \rceil$ . We then have that the  $p$ -step RWNN maps the two graphs (i. e.,  $C_n^2$  and  $C_{2n}$ ) to the same vector.

We will next show that there exists weights such that the proposed GRWNN model can distinguish between graphs  $C_n^2$  and  $C_{2n}$ . Without loss of generality, assume that the model contains a single ‘‘hidden graph’’ which consists of two vertices. Let the weights that represent the adjacency matrix of the ‘‘hidden graph’’  $\text{RELU}(\mathbf{W}) \in \mathbb{R}^{2 \times 2}$  and the corresponding vertex features  $\mathbf{Q} \in \mathbb{R}^{2 \times 1}$  be defined as follows:

$$\text{RELU}(\mathbf{W}) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

The kernel between the ‘‘hidden graph’’ and the input graph is computed as follows:

$$\begin{aligned} k^\infty(G, G') &= \sum_{i=1}^{|V_x|} \sum_{j=0}^{|V_x|} \left[ \sum_{l=0}^{\infty} \lambda^l ((\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_x)^l \right]_{ij} \\ &= |V_x| + \lambda \mathbf{1}^\top ((\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_x) \mathbf{1} + \lambda^2 \mathbf{1}^\top ((\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_x)^2 \mathbf{1} + \dots \end{aligned}$$

where  $\mathbf{A}_x = \text{RELU}(\mathbf{W}) \otimes \mathbf{A}$ ,  $\mathbf{s} = \text{vec}(\mathbf{S})$  where  $\mathbf{S} = \sigma(\mathbf{X} \mathbf{Q}^\top)$ ,  $\sigma(\cdot)$  denotes the sigmoid function, and  $\lambda$  is a hyperparameter such that the above series converges for both  $C_n^2$  and  $C_{2n}$ . Then, if  $\mathbf{A}$  and  $\mathbf{X}$  denote the adjacency matrix of the input graph and the matrix of features, respectively, we have:

$$\mathbf{A}_x = \begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A} & 0 \end{bmatrix} \quad \mathbf{s} = [\sigma(\mathbf{Z}) \quad \sigma(\mathbf{Z})] \quad \mathbf{s} \mathbf{s}^\top = \begin{bmatrix} \sigma(\mathbf{Z}) & \sigma(\mathbf{Z}) \\ \sigma(\mathbf{Z}) & \sigma(\mathbf{Z}) \end{bmatrix}$$

where  $\mathbf{Z} = \mathbf{X} \mathbf{X}^\top$ . Then, we have:

$$(\mathbf{s} \mathbf{s}^\top) \odot \mathbf{A}_x = \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A} \\ \sigma(\mathbf{Z}) \odot \mathbf{A} & 0 \end{bmatrix}$$

Note that  $\sigma(\mathbf{Z}) \odot \mathbf{A}$  reweights the edges of the input graph, i. e., the weight of an edge is set equal to the sigmoid of the product of the features of the two endpoints. The kernel between the ‘‘hidden graph’’ and the input graph can then be

computed as follows:

$$\begin{aligned} k^\infty(G, G') &= |V_\times| + \lambda \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A} \\ \sigma(\mathbf{Z}) \odot \mathbf{A} & 0 \end{bmatrix} \mathbf{1} + \lambda^2 \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A} \\ \sigma(\mathbf{Z}) \odot \mathbf{A} & 0 \end{bmatrix}^2 \mathbf{1} + \dots \\ &= |V_\times| + \lambda \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A} \\ \sigma(\mathbf{Z}) \odot \mathbf{A} & 0 \end{bmatrix} \mathbf{1} + \lambda^2 \mathbf{1}^\top \begin{bmatrix} (\sigma(\mathbf{Z}) \odot \mathbf{A})^2 & 0 \\ 0 & (\sigma(\mathbf{Z}) \odot \mathbf{A})^2 \end{bmatrix} \mathbf{1} + \dots \end{aligned}$$

Given the adjacency matrices  $\mathbf{A}_{C_n^2}$  and  $\mathbf{A}_{C_{2n}}$  of graphs where  $C_n^2$  and  $C_{2n}$ , we have that:

$$\lambda^p \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2} \\ \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2} & 0 \end{bmatrix}^p \mathbf{1} = \lambda^p \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_{2n}} \\ \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_{2n}} & 0 \end{bmatrix}^p \mathbf{1}$$

for  $p \in [n-1] \cup \{0\}$ . This holds since the sets of weighted paths of length  $p$  in the two graphs are identical to each other. Thus, the set of weighted walks of length  $p$  are also identical. Then, it suffices to show that for any  $p \geq n$ , the following holds:

$$\lambda^p \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2} \\ \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2} & 0 \end{bmatrix}^p \mathbf{1} > \lambda^p \mathbf{1}^\top \begin{bmatrix} 0 & \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_{2n}} \\ \sigma(\mathbf{Z}) \odot \mathbf{A}_{C_{2n}} & 0 \end{bmatrix}^p \mathbf{1} \quad (5)$$

This is true if  $(\sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2})^p > (\sigma(\mathbf{Z}) \odot \mathbf{A}_{C_{2n}})^p$  holds. Note that the entry at position  $(i, j)$  of  $(\sigma(\mathbf{Z}) \odot \mathbf{A}_{C_n^2})^p$  is the sum of the products of all weights of all walks from node  $i$  to node  $j$  of length exactly  $p$ . For any  $p$ , the number of walks of length exactly  $p$  in graph  $C_n^2$  is equal to the number of walks of length exactly  $p$  in graph  $C_{2n}$ . However, for  $p \geq n$ , in the case of  $C_{2n}$ , there exist walks that visit both vertices  $\lceil \frac{n}{2} \rceil$  and  $\lceil \frac{n}{2} \rceil + n$ , while in  $C_n^2$ , there are no such walks, but instead vertices  $\lceil \frac{n}{2} \rceil$  and  $\lceil \frac{n}{2} \rceil + n$  are visited multiple times. Thus, in the case of  $C_{2n}$ , there are entries that correspond to sums that contain products of the form  $\sigma(x \cdot x_1) \cdot \sigma(x \cdot x_2)$ , while in the case of  $C_n^2$ , the sum emerges among others from products of the form  $\sigma(x \cdot x_1)^2$  and  $\sigma(x \cdot x_2)^2$ . Since  $\sigma(\cdot)$  is the sigmoid function,  $0 < \sigma(x \cdot x_1), \sigma(x \cdot x_2) < 1$  holds. Furthermore, since we have assumed that  $x_1 \neq x_2$ , we have that  $\sigma(x \cdot x_1) \neq \sigma(x \cdot x_2)$  and therefore  $\sigma(x \cdot x_1)^2 + \sigma(x \cdot x_2)^2 > 2\sigma(x \cdot x_1) \cdot \sigma(x \cdot x_2)$ . Thus, Equation (5) holds.  $\square$

## D Implicit Differentiation

Clearly, iteratively computing Equation (2) or Equation (4) to find the fixed point corresponds to a differentiable module. However, to train the model, we need to backpropagate the error through the fixed point solver in the backward pass. That would require storing all the intermediate terms, which could be prohibitive in practice. Fortunately, thanks to recent advances in implicit layers and equilibrium models (Bai et al., 2019), this can be performed in a simple and efficient manner which requires constant memory, and assumes no knowledge of the fixed point solver. Specifically, based on ideas from (Bai et al., 2019), we derive the form of implicit backpropagation specific to the employed fixed point iteration layer.

**Theorem D.1.** *Let  $f_\theta$  be the system of Equation (2) or Equation (4), and  $\mathbf{z}^* \in \mathbb{R}^{nc}$  be a solution to that linear system. Let also  $g_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) = f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) - \mathbf{z}^*$ . Since  $\mathbf{z}^*$  is a fixed point, we have that  $g_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) \rightarrow 0$  and  $\mathbf{z}^*$  is thus the root of  $g_\theta$ . Let  $y \in \mathbb{R}$  denote the ground-truth target of the input sample,  $h: \mathbb{R} \rightarrow \mathbb{R}$  be any differentiable function and let  $\mathcal{L}: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  be a loss function that computes:*

$$\ell = \mathcal{L}(h(\mathbf{1}^\top \mathbf{z}^*), y) = \mathcal{L}(h(\mathbf{1}^\top \text{FindRoot}(g_\theta; \mathbf{A}, \mathbf{X})), y)$$

Then, the gradient of the loss w.r.t.  $(\cdot)$  (e. g.,  $\theta$ ,  $\mathbf{A}$  or  $\mathbf{X}$ ) is:

$$\frac{\partial \ell}{\partial (\cdot)} = - \frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_\theta}^{-1} |_{\mathbf{z}^*}) \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial (\cdot)} = - \frac{\partial \ell}{\partial h} \frac{\partial h}{\partial \mathbf{z}^*} (J_{g_\theta}^{-1} |_{\mathbf{z}^*}) \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial (\cdot)}$$

where  $J_{g_\theta}^{-1} |_{\mathbf{z}^*}$  is the inverse Jacobian of  $g_\theta$  evaluated at  $\mathbf{z}^*$ .

The above Theorem gives a form for the necessary Jacobian without needing to backpropagate through the method used to obtain the fixed point. We can thus treat the fixed point algorithm as a black box, and we do not need to store intermediate terms associated with the fixed point algorithm into memory. We only need to apply some algorithm that will produce a solution to the system (i. e., it will compute the fixed point).

Following (Bai et al., 2019), we differentiate the two sides of the fixed point equation  $\mathbf{z}^* = f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})$  with respect to  $(\cdot)$ :

$$\begin{aligned} \frac{d\mathbf{z}^*}{d(\cdot)} &= \frac{df_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{d(\cdot)} = \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial(\cdot)} + \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \frac{d\mathbf{z}^*}{d(\cdot)} \\ \implies \left( \mathbf{I} - \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right) \frac{d\mathbf{z}^*}{d(\cdot)} &= \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial(\cdot)} \end{aligned}$$

Since  $g_\theta(\mathbf{z}^*) = f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X}) - \mathbf{z}^*$ , we have:

$$J_{g_\theta}|_{\mathbf{z}^*} = - \left( \mathbf{I} - \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)$$

which implies the following:

$$\frac{\partial \ell}{\partial(\cdot)} = \frac{\partial \ell}{\partial \mathbf{z}^*} \frac{d\mathbf{z}^*}{d(\cdot)} = - \frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_\theta}^{-1}|_{\mathbf{z}^*}) \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial(\cdot)}.$$

Unfortunately, computing the exact inverse Jacobian  $J_{g_\theta}^{-1}$  has a cubic cost. As shown in (Bai et al., 2019), we can instead compute the  $-\frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_\theta}^{-1}|_{\mathbf{z}^*})$  term of the gradient (which contains the Jacobian) by solving the following linear system:

$$\begin{aligned} \mathbf{x}^\top &= - \frac{\partial \ell}{\partial \mathbf{z}^*} (J_{g_\theta}^{-1}|_{\mathbf{z}^*}) = \frac{\partial \ell}{\partial \mathbf{z}^*} \left( \mathbf{I} - \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)^{-1} \\ \implies \mathbf{x} &= \left( \left( \mathbf{I} - \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)^\top \right)^{-1} \left( \frac{\partial \ell}{\partial \mathbf{z}^*} \right)^\top \\ \implies \left( \mathbf{I} - \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)^\top \mathbf{x} &= \left( \frac{\partial \ell}{\partial \mathbf{z}^*} \right)^\top \\ \implies \mathbf{x} &= \left( \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial \mathbf{z}^*} \right)^\top \mathbf{x} + \left( \frac{\partial \ell}{\partial \mathbf{z}^*} \right)^\top \end{aligned}$$

which in fact is also a fixed point equation and can be solved via some iterative procedure. Note that the first term of the above Equation is a vector-Jacobian product which can be efficiently computed via autograd packages (e. g., PyTorch (Paszke et al., 2017)) for any  $\mathbf{x}$ , without explicitly writing out the Jacobian matrix. Finally, we can compute  $\frac{\partial \ell}{\partial(\cdot)}$  as follows:

$$\frac{\partial \ell}{\partial(\cdot)} = \left( \frac{\partial f_\theta(\mathbf{z}^*; \mathbf{A}, \mathbf{X})}{\partial(\cdot)} \right)^\top \mathbf{x}$$

where again this product is itself a vector-Jacobian product, computable via normal automatic differentiation packages.

## E Datasets

We evaluated the proposed model on 10 publicly available graph classification datasets including 5 bio/chemo-informatics datasets: MUTAG, D&D, NCI1, PROTEINS and ENZYMES, as well as 5 social interaction datasets: IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY, REDDIT-MULTI-5K and COLLAB (Kersting et al., 2016). A summary of the 10 datasets is given in Table 2. MUTAG consists of 188 mutagenic aromatic and heteroaromatic nitro compounds. The task is to predict whether or not each chemical compound has mutagenic effect on the Gram-negative bacterium *Salmonella typhimurium* (Debnath et al., 1991). ENZYMES contains 600 protein tertiary structures represented as graphs obtained from the BRENDA enzyme database. Each enzyme is a member of one of the Enzyme Commission top level enzyme classes (EC classes) and the task is to correctly assign the enzymes to their classes (Borgwardt et al., 2005). NCI1 contains more than four thousand chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines (Wale et al., 2008). PROTEINS contains proteins represented as graphs where vertices are secondary structure elements and there is an edge between two vertices if they are neighbors in the amino-acid sequence or in 3D space. The task is to classify proteins into enzymes and non-enzymes (Borgwardt et al., 2005). D&D contains over a thousand protein structures. Each protein is a graph whose nodes correspond to amino acids and a pair of amino acids are connected by an edge if they are less than 6 Ångstroms apart. The task is to predict if a protein is an enzyme or not (Dobson and



Table 2: Summary of the 10 datasets that were used in our experiments.

Dataset	MUTAG	D&D	NCII	PROTEINS	ENZYMES	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
Max # vertices	28	5,748	111	620	126	136	89	3,782	3,648	492
Min # vertices	10	30	3	4	2	12	7	6	22	32
Average # vertices	17.93	284.32	29.87	39.05	32.63	19.77	13.00	429.61	508.50	74.49
Max # edges	33	14,267	119	1,049	149	1,249	1,467	4,071	4,783	40,119
Min # edges	10	63	2	5	1	26	12	4	21	60
Average # edges	19.79	715.66	32.30	72.81	62.14	96.53	65.93	497.75	594.87	2,457.34
# labels	7	82	37	3	–	–	–	–	–	–
# attributes	–	–	–	–	18	–	–	–	–	–
# graphs	188	1,178	4,110	1,113	600	1,000	1,500	2,000	4,999	5,000
# classes	2	2	2	2	6	2	3	2	5	3

Table 3: Statistics of the 2 OGB datasets that we used in our experiments.

Dataset	ogbg-molhiv	ogbg-code2
Average # vertices	25.5	125.2
Average # edges	27.5	124.2
Node features	✓	✓
Edge features	✓	✓
Directed	–	✓
# graphs	41,127	452,741
# tasks	1	1
Split scheme	Scaffold	Project
Split ratio	80/10/10	90/5/5
Task type	Binary class.	Sub-token prediction
Metric	ROC-AUC	F1-score

Doig, 2003). IMDB-BINARY and IMDB-MULTI were created from IMDb, an online database of information related to movies and television programs. The graphs contained in the two datasets correspond to movie collaborations. The vertices of each graph represent actors/actresses and two vertices are connected by an edge if the corresponding actors/actresses appear in the same movie. Each graph is the ego-network of an actor/actress, and the task is to predict which genre an ego-network belongs to (Yanardag and Vishwanathan, 2015). REDDIT-BINARY and REDDIT-MULTI-5K contain graphs that model the social interactions between users of Reddit. Each graph represents an online discussion thread. Specifically, each vertex corresponds to a user, and two users are connected by an edge if one of them responded to at least one of the other’s comments. The task is to classify graphs into either communities or subreddits (Yanardag and Vishwanathan, 2015). COLLAB is a scientific collaboration dataset that consists of the ego-networks of several researchers from three subfields of Physics (High Energy Physics, Condensed Matter Physics and Astro Physics). The task is to determine the subfield of Physics to which the ego-network of each researcher belongs (Yanardag and Vishwanathan, 2015).

We also evaluated the proposed model on two datasets from the Open Graph Benchmark (OGB) (Hu et al., 2020), a collection of large-scale and diverse benchmark datasets for machine learning on graphs. A summary of the two datasets is given in Table 3.

The `ogbg-molhiv` dataset is a molecular property prediction dataset that is adopted from the MoleculeNet (Wu et al., 2018). The dataset consists of 41,127 molecules and corresponds to a binary classification dataset where the task is to predict whether a molecule inhibits HIV virus replication or not. The molecules in the training, validation and test sets are divided using a scaffold splitting procedure that splits the molecules based on their two-dimensional structural frameworks. The scaffold splitting attempts to separate structurally different molecules into different subsets. The `ogbg-code2` dataset contains a large number of Abstract Syntax Trees (ASTs) that are extracted from approximately 450,000 Python method definitions. For each method, the AST edges, the AST nodes, and the tokenized method name are retrieved. Given the body of a method represented by the AST and its node features, the task (which is known as code summarization) is to

Table 4: Values of  $\lambda$  that we used in our experiments.

	MUTAG	D&D	NCI1	PROTEINS	ENZYMES	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
$\lambda$	1/5	1/20	1/20	1/30	1/20	1/200	1/300	1/500	1/400	1/2000

Table 5: Average running time per epoch (in seconds) of the proposed model and 3 baselines on the 10 graph classification datasets.

	MUTAG	D&D	NCI1	PROTEINS	ENZYMES	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
GIN	0.03	0.34	0.50	0.14	0.07	0.13	0.19	0.81	2.43	0.98
2-RWNN	0.03	0.19	0.57	0.16	0.08	0.14	0.20	0.43	1.13	0.89
3-RWNN	0.04	0.23	0.76	0.21	0.11	0.18	0.28	0.55	1.42	1.04
GRWNN	0.07	0.77	0.94	0.32	0.17	0.24	0.34	2.93	6.19	2.69

predict the sub-tokens forming the name of the method. The ASTs for the training set are obtained from GitHub projects that do not appear in the validation and test sets. We refer the reader to (Hu et al., 2020) for more details about the OGB datasets.

## F Parameter $\lambda$

Given an input graph  $G$  and a “hidden graph”  $G_i$ , since the “hidden graph” is trainable, the maximum vertex degree of the product graph  $G_\times$  is not known beforehand. However, in case the weights of the edges of the “hidden graph” are bounded, we can compute an upper bound to that. Let  $\Delta$  denote the maximum vertex degree of  $G$ ,  $c$  denote the number of vertices of the “hidden graph”  $G_i$ , and  $b$  the maximum edge weight of the “hidden graph”. Then, we have that  $\Delta_\times \leq \Delta(c-1)b$ , and therefore, to guarantee convergence, we need to set  $\lambda \leq 1/\Delta(c-1)b$ . In practice, we empirically found that even if  $\lambda$  takes higher values, the geometric series converges within a small number of iterations. Table 4 shows the value of  $\lambda$  that we used for each dataset.

## G Runtime Analysis

The proposed model is indeed computationally more expensive than the RWNN model due to the fixed point iteration which is not parallelizable. However, as already discussed, we empirically observed that the forward iteration converges in a small number of steps, thus incurring a relatively small overhead in the model’s running time. We have computed the average running time per epoch of the proposed model, and 3 of the baselines (2-RWNN, 3-RWNN and GIN) on the 10 graph classification datasets. We use the same values for the common hyperparameters (e. g., number and size of hidden graphs for GRWNN and RWNN, and hidden dimension size, batch size, etc for all 3 models). The results are shown in Table 5 (in seconds). As we can see, the proposed model is not much more expensive than the baselines. In fact, in most cases, its average running time per epoch is 1 – 3 times higher than that of the baselines, which is by no means prohibitive for real-world scenarios.

## H Sensitivity Analysis

The proposed GRWNN model involves two main parameters: (1) the number of “hidden graphs”, and (2) the number of vertices of “hidden graphs”. We next investigate how these two parameters influence the performance of the GRWNN model. Specifically, in Figure 2, we examine how the different values of these parameters affect the performance of GRWNN on the ENZYMES dataset. We observe that the accuracy on the test set increases as the number of “hidden graphs” increases. The number of “hidden graphs” seems to have a significant impact on the performance of the model. When the number of graphs is set equal to 5, the model achieves an accuracy smaller than 50%, while when the number of

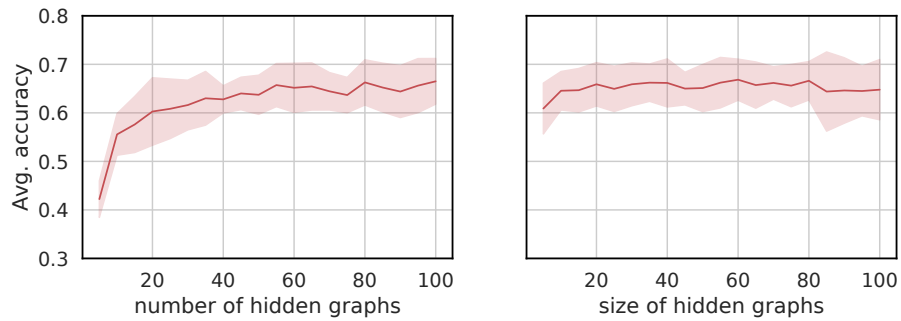


Figure 2: Performance on the ENZYMES dataset as a function of the number of “hidden graphs” (left) and the number of vertices of the “hidden graphs” (right).

graphs is set equal to 100, it yields an accuracy greater than 65%. On the other hand, the number of vertices of the “hidden graphs” does not affect that much the performance of the model.