Learning Powerful Representations of Molecular Graphs With Graph Isomorphism Network

Ben Parker

Advisor: Nikolay M. Sirakov, Ph.D

Math 595 - Research Literature and Techniques

Spring 2023

TEXAS A&M UNIVERSITY-COMMERCE

Contents

Abstract

The predominant framework for representation learning on graphs is the Graph Neural Network (GNN). This model aims to condense high-level structure and feature information of graph-structured data into vector embeddings, which are suitable inputs for downstream prediction modeling. In this paper, I cover the essential processes intrinsic to all Graph Neural Networks, provide a theoretical rationale for these learning mechanisms, discuss methods for evaluating the representational power of GNNs, and describe recent developments aiming to improve representational power of graph machine learning. In the experiment portion of this paper, I implement a powerful GNN, Graph Isomorphism Network, which is used to perform a graph classification task using the MCF-7 benchmark dataset. The results from this experiment suggest that Graph Isomorphism Network is an effective tool for learning graph representations on this particular dataset. Furthermore, I observe the improvements in the model's ability to generalize when trained on resampled data.

1 Introduction

Research interest in deep learning has expanded in recent decades, thanks to the success of artificial neural networks with various machine learning tasks, such as object detection and speech recognition [15]. Due to the effectiveness of deep learning architectures in learning latent feature representations in Euclidean data, such as images and text, a natural extension of this research is the generalization of these methods to non-Euclidean data, such as graphs.

Graphs are widely used to capture relations between discrete objects in many disciplines, such as physics, chemistry, biology, sociology, and finance. Successful representation learning of graphs presents numerous opportunities for the predictive modeling of these relations. In particular, this entails graph-level classification and node-level classification. As an example of graph classification, the ability to capture the latent patterns in molecular data enables faster and more cost effective molecular property prediction of drug candidates. Moreover, several researchers have developed node classification GNNs with the purpose of detecting fraud in financial transaction networks. Regardless of the final predictive task, effective graph-based modeling obligates powerful numerical representations of graph-structured data.

Prior to the development of modern techniques, graph statistics and graph kernels were the primary means of generating representations of graphs [4]. More recently, however, GNNs have emerged as the state of the art framework for graph representation learning due to their ability to encode more complex information about the structure and feature information of graphs. Furthermore, GNNs can *learn* representations of graph structures, unlike kernel methods, which generate these representations using a deterministic mapping [4].

In general, GNNs extract higher-level features of graph nodes ("embeddings") through a neighborhood aggregation scheme. This algorithm, commonly referred to as "message passing", updates a particular node's representation by aggregating and transforming feature information about the node and its neighbors. Analogous to the 2-dimensional convolution operation used to generate feature maps for image data, a weighted combination of features within a neighborhood is taken to be the new representation of a particular node. This updated representation, or "embedding", captures the local structural and feature information about the node.

Message Passing has been used in many different GNN variants, such as the Graph Convolutional Network [10], with significant results. However, others [16] have noted that message passing GNNs have a hard limit in their ability to discriminate between non-isomorphic substructures. Thus, developing GNN models with even greater "representational power" is an active area of research. Newer, more powerful aggregation schemes inspired by graph isomorphism testing algorithms have shown superior results to message passing algorithms and graph kernels in some commonly used benchmark datasets [16][11]. Graph Isomorphism Network (GIN) [16] is an influential GNN framework that seeks to match the representational power of the Weisfeiler-Lehman Test for Graph Isomorphism.

2 Theoretical Foundations

2.1 Message Passing

GNNs utilize the structural and feature information about a graph to learn vector node embeddings: higher-level, low-dimensional representations of a node's local structure and features. These node embeddings are generated by means of an aggregation function. We may define an aggregation function as any mapping from a multiset of feature vectors h taken from the neighborhood of node u, $\mathcal{N}(u)$, to a single output vector or "message" [16]:

$$\{h_v, \forall v \in \mathcal{N}(u)\} \xrightarrow{aggregate} m_{\mathcal{N}(u)} \tag{1}$$

In each iteration of message passing, the current embedding for a given node u, $h_u^{(k)}$, is updated by combination with the message vector $m_{\mathcal{N}(u)}$, resulting in a new node embedding $h_u^{(k+1)}$ [16]. Formally, [4]

$$h_u^{(k+1)} = update^{(k)}\big(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)}\big), \tag{2}$$

where

$$m_{\mathcal{N}(u)}^{(k)} = aggregate^{(k)}\big(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}\big), \tag{3}$$

and *update* and *aggreagate* are differentiable functions. Note that the definition of *aggregate* as a function of an unordered multiset of vectors ensures invariance to node ordering [4].

After the first iteration, the embedding $h_u^{(1)}$ contains information about u's immediate neighbors. After k iterations, $h_u^{(k)}$ contains information about nodes within its k-hop neighborhood. Upon multiple iterations of message passing, the resulting embeddings can be conceptualized as an encoded "rooted subtree" around the center node [16](Fig 1). After 1 iteration, the parent node has labeled child nodes that represent each of its neighbors; after 2 iterations, each sibling node has child nodes representing each of their respective neighbors, and so on.

2.2 Graph Neural Networks

In general, a rudimentary GNN can be thought of as stacked iterations of message passing, where the combination of node features is parameterized by trainable weights. Let V be the set of nodes in a graph where each node $u \in V$ is associated with a feature vector $\mathbf{h}_u^{(0)}$. If we choose *update* and *aggregate* to be simple sums, we may define a generic GNN message passing scheme as [4]

$$\mathbf{h}_u^{(k)} = \sigma\Bigg(\mathbf{W}_{self}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{neighbors}^{(k)}\sum_{v\in N(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\Bigg), \tag{4}$$

where σ is a non-linear activation function, $\mathbf{b}$ is a bias term, $\mathbf{W}$ are trainable weight matrices with dimensions $d^{(k)} \times d^{(k-1)}$, $\mathbf{h}_u, \mathbf{h}_v \in \mathbb{R}^d$, and d is the number of hidden neurons at layer k. Since computations at the graph level using sparse matrix representations are less computationally expensive, it is useful to redefine (4) at the graph level [4]:

$$\mathbf{H}^{(t)} = \sigma\Bigg(\mathbf{A}\mathbf{H}^{(t-1)}\mathbf{W}_{neighbors}^{(t)} + \mathbf{H}^{(t-1)}\mathbf{W}_{self}^{(t)} + \mathbf{B}^{(t)}\Bigg), \tag{5}$$

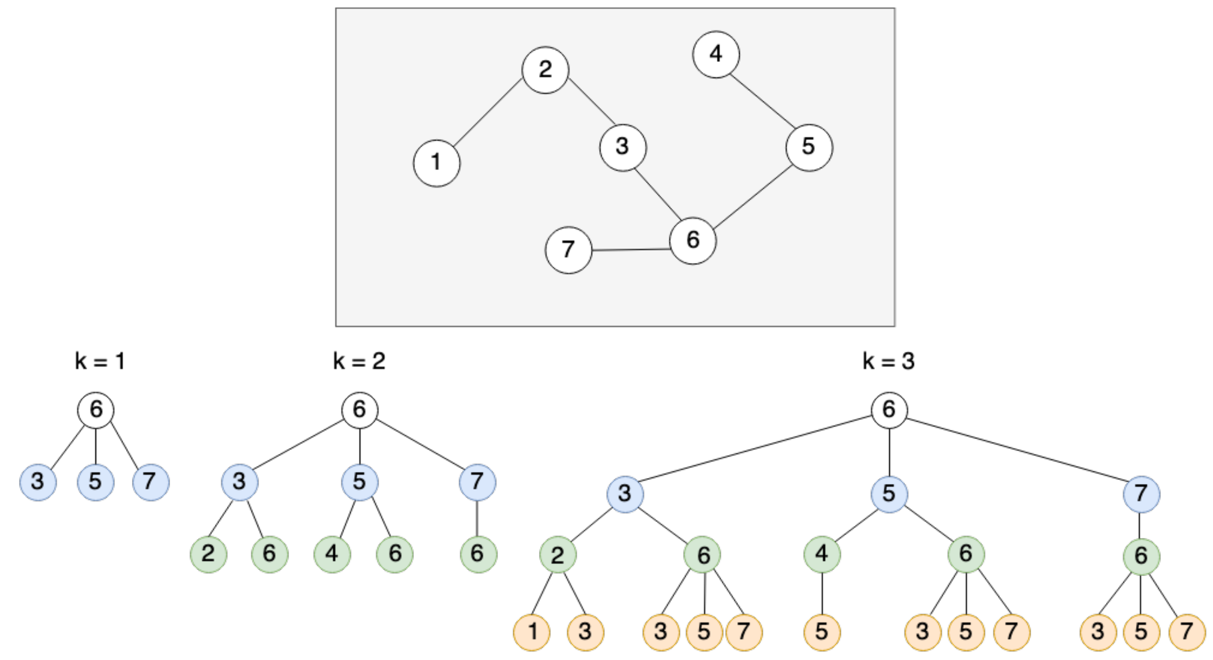

Figure 1: Message passing generates a feature vector embedding that encodes a rooted subtree around the node. This figure depicts 3 iterations of message passing for node 6 in the upper graph. In the rooted trees, each child node is a neighbor of its parent node.

where t is the layer in the neural network, $\mathbf{A}$ is the adjacency matrix of the graph, and $\mathbf{H}$ is the matrix of node features with dimension $|V| \times d$.

Commonly, message passing GNNs utilize "self-loops" in their computations. With self-loops, *update* and *aggregate* are combined into one step by aggregating feature vectors over the set $\mathcal{N}(u) \cup \{u\}$. Although incorporating these "self-loops" into the aggregation tends to reduce overfitting of the model to a particular dataset, it eliminates the opportunity for the model to distinguish between the central node and its neighbors [4]. Practically, implementing self-loops amounts to adding an identity matrix to the adjacency matrix in equation (5) and combining $\mathbf{W}_{self}$ and $\mathbf{W}_{neighbors}$ into one trainable weight matrix [4]:

$$\mathbf{H}^{(t)} = \sigma\Big((\mathbf{A} + \mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}\Big). \tag{6}$$

Learned embeddings may be used as inputs to a downstream predictive model, such as multilayer perceptron. Common prediction tasks include node-, edge-, and graph classification. For node classification tasks, the final node embeddings are simply used as input to a downstream classifier, such as a fully-connected neural network. With some adjustments, predictions can be made at the edge-, and graph-level. In the latter case, the final node embeddings for a particular graph are further pooled into a final graph-level representation before the final classification [4].

2.3 Spectral Graph Convolutions

Observe from eq. 6 that the matrix product $(A + I)H$ performs a single iteration of message passing, whereby each node's features are combined with those in its 1-hop neighborhood. The matrix $(A + I)$ can be thought of as a convolutional filter of the node feature matrix H. In fact, message passing is a specific example of a spectral convolution, a concept that inspired some popular message passing GNNs. Through the perspective of spectral convolutions, we can develop a more general understanding of information propagation through graphs.

The observation that the eigenfunctions of the Laplace operator are equivalent to the Fourier modes in the frequency domain allows us to define graph convolution in terms of the eigenvectors of the graph Laplacian [4]. Let $L = D - A$ be the graph Laplacian, where D is the degree matrix consisting of the degree of each node v in the graph along the diagonal:

$$D_{i,j} = \begin{cases} Deg(v_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Furthermore, let $L = U\Lambda U^T$ be the eigendecomposition of the graph Laplacian, where the matrix U consists of the eigenvectors of L and Λ contains the eigenvalues along the diagonal. Now, consider a vector signal $f \in \mathbb{R}^{|V|}$ (a scalar for each node in some arbitrary graph) and a filter h. Utilizing the discrete Fourier transform on graphs, the graph convolution can be expressed as [4]

$$f \star h = U(U^T f \circ U^T h) = U(U^T f \circ \theta) = (U\text{diag}(\theta)U^T)f = \tag{8}$$

$$= (Up_k(\Lambda)U^T)f = p_k(L)f, \tag{9}$$

where $\circ$ indicates an element-wise vector product and p_k denotes a degree-k polynomial. The equality in eq. 9 follows from the observation that graph Fourier coefficients θ can be approximated by polynomials of the eigenvalues of the Laplacian $p_k(\Lambda)$ [5][10]. Furthermore, this approximation ensures that convolution on a graph signal is meaningfully local [4].

From eq. 9, we note that a graph convolution consists of the matrix-vector product between a polynomial of the Laplacian matrix and a graph signal. Furthermore, the degree of the polynomial corresponds to the size of the convolutional "kernel". In other words, convolution by a degree k polynomial aggregates information from the k-hop neighborhood of a node [10]. In practice, this information aggregation is performed by stacking multiple convolutions of degree 1 polynomials of the Laplacian (or adjacency matrix) layer-wise, thereby implicitly aggregating information from multi-hop neighborhoods (as in message passing).

The notion of first-order approximation of spectral convolutions motivated the development of Graph Convolutional Network (GCN) [10]. Aggregation with GCN involves the application of "symmetric normalization" as a remedy for numerical instability and vanishing/exploding gradient associated with variability in node degree [4]. The GCN layer is defined as [10]

$$\mathbf{H}^{(t+1)} = \sigma\left(\widetilde{\mathbf{D}}^{-1/2}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-1/2}\mathbf{H}^{(t)}\mathbf{W}^{(t)}\right), \tag{10}$$

where $\widetilde{A} = A + I$ is the adjacency matrix with self-loops and $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$. From the node perspective, we can observe that the GCN propagation rule represents a form of message passing where each feature vector is normalized based on the number of nodes in the 1-hop neighborhood [4]:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_{(v)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{11}$$

2.4 Representational Power of GNNs

Learned embeddings with sub optimal representational power result in some degree of underfitting when used to train a classifier. Thus, in order to improve these predictive models, it is pertinent to evaluate the representational power of these embeddings. That is, to what extent do unique embeddings represent unique graph substructures?

The theory of graph isomorphism testing is an important tool for formally analyzing the representational power of GNNs. The objective of isomorphism testing is to determine whether two graphs are isomorphic. We define two graphs to be isomorphic if and only if there exists a permutation matrix $\mathbf{P}$ such that [4]

$$\mathbf{P}\mathbf{A}_1\mathbf{P}^T = \mathbf{A}_2, \tag{12}$$

and

$$\mathbf{P}\mathbf{X}_1 = \mathbf{X}_2, \tag{13}$$

where $\mathbf{A}_1$ and $\mathbf{A}_2$ are the adjacency matrices of the two graphs and $\mathbf{X}_1$ and $\mathbf{X}_2$ represent their node feature matrices. Here, we can see that graph isomorphism implies that the two graphs have the same structure, yet differ in a permutation of their nodes. In the analysis of GNNs, a hypothetical "most powerful" GNN learns two identical embeddings for two input graphs if and only if they are isomorphic [4].

In practice, testing for graph isomorphism is a computationally challenging problem since it requires testing the conditions in eq. 12 and 13 over all possible permutation matrices [4]. No polynomial-time algorithm is known for this problem [16]. However, an efficient approximation for distinguishing non-isomorphic graphs is the Weisfeiler-Lehman Test (WL) [14]. The test, similar to message passing, iteratively aggregates the labels of a node and its neighbors, and hashes aggregated labels into unique new labels. If at any iteration, the labels differ between two graphs, the WL test determines that they are non-isomorphic [16].

Formally, let G_1 and G_2 be two graphs with labeled nodes n_l. For iteration $t > 0$, we perform the following bijective mapping for each node [12]:

$$n_l^{(t)}(u) = \text{hash}\Bigg(\big(n_l^{(t-1)}(u), \{\{n_l^{(t-1)}(v) \forall v \in \mathcal{N}(u)\}\}\big)\Bigg). \tag{14}$$

At iteration t, if G_1 and G_2 have a different number of nodes with a given label, the algorithm determines that G_1 and G_2 non-isomorphic. Otherwise, the iterations continue until there is no change in node labels from one iteration to the next, at which point G_1 and G_2 are deemed isomorphic [12].

Analogous to message passing, the WL test embeds separate graph structures by encoding a rooted tree structure about a central node, which summarizes the central node's k-hop neighborhood. However, unlike message passing, the updated node labels generated by the WL test represent a (near) injective mapping from the neighborhood multiset to the embedding space [4] [16]. As a consequence, message passing GNNs such as GCN have representational power *at most* as great as the WL test [16].

2.5 Graph Isomorphism Network

Current research into developing a "most powerful" GNN are heavily inspired by the WL test. As shown by Xu, et al. [16], a GNN with expressive power comparable to the WL test exists and can be implemented by means of an injective aggregation algorithm. Graph Isomorphism Network (GIN) [16] is a GNN architecture that was developed with this purpose. The authors claim that due to the recursive nature of rooted subtree structures, it is sufficient to use an injective aggregation mapping from each neighborhood multiset to each central node embedding. With GIN, instead of explicitly defining this injective function, it is approximated at each iteration using a multilayer perception. This approximation follows from the theorem that multilayer perceptrons are universal function approximators [7].

Each iteration of the algorithm is defined as [16]

$$\mathbf{h}_u^{(k)} = \phi^{(k)}\left(\big(1+\epsilon^{(k)}\big)\mathbf{h}_u^{(k-1)} + \sum_{v\in\mathcal{N}(u)} \mathbf{h}_u^{(k-1)}\right), \tag{15}$$

where ϕ is a multilayer perceptron and ϵ is an optional, trainable parameter. From eq. 15, we can see that when $\epsilon = 0$, each GIN layer generates an embedding by first taking a sum of feature vectors within a 1-hop neighborhood and second, transforming this aggregated vector using a fully-connected neural network. From a convolutional perspective, eq. 15 can be written as [2]

$$\mathbf{H}^{(k)} = \phi^{(k)}\left(\big(\mathbf{A} + (1+\epsilon)\mathbf{I}\big)\mathbf{H}^{(k-1)}\right). \tag{16}$$

3 Model Design

3.1 Architecture

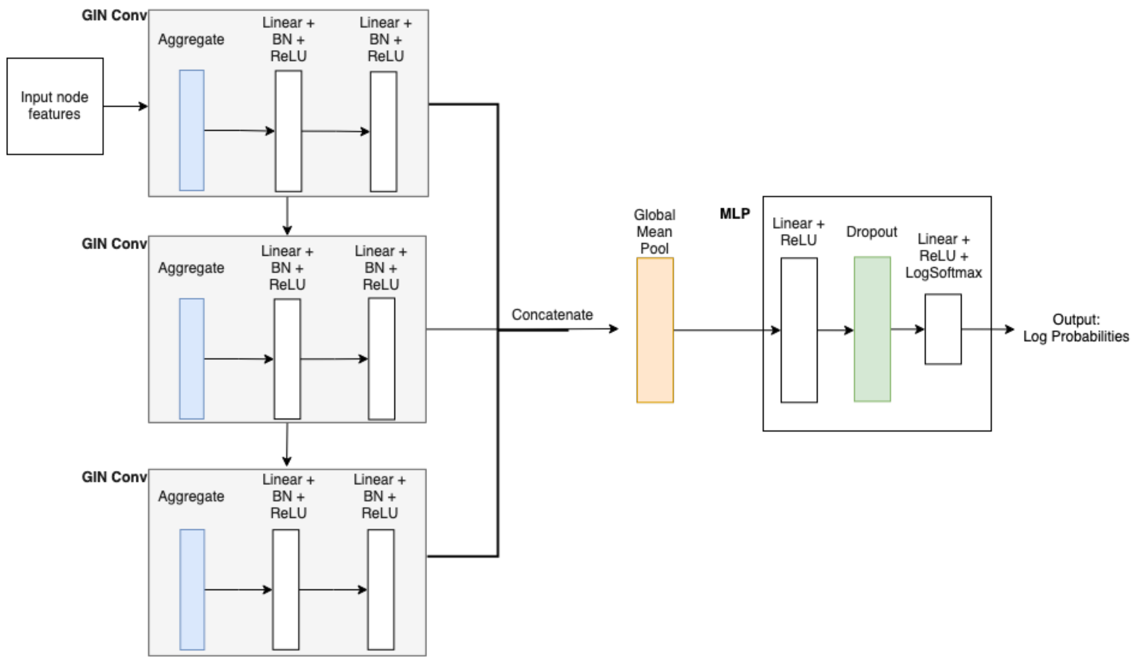

Figure 2: GIN Architecture

The model architecture begins with a number of iterations of node feature aggregation, followed by classification by a multilayer perceptron (MLP). Figure 2 shows the detailed architecture of the final model using 3 GIN convolutional layers. The GIN Conv blocks represent iterations of eq. 16 with $\epsilon = 0$. (Note that the decision to fix $\epsilon = 0$ was motivated by empirical results questioning the advantage of implementing a trainable ϵ [16].)

In the GIN Conv block, each node u's features are summed with those in $\mathcal{N}(u)$. Following [11] and [17], these aggregated feature vectors are then inputted into a MLP with 2 fully-connected (linear) layers. Following [17], I use Batch Normalization (eq. 17) at each layer prior to the ReLU activation function (eq. 18). Following [17], the output from this iteration is then passed to the next iteration and concatenated column-wise (along dimension 1) with the outputs from each of the other iterations (Fig 2).

To generate a graph-level embedding for each graph, I apply a global mean pool function (eq. 19) to the concatenated embeddings. Following [11], these embeddings are then classified by another 2-layer MLP with a dropout layer with $p = 0.5$ between the two layers. To the final output, I apply a log softmax function (eq. 20) to generate log probabilities. For training, I use Adam optimization (Algorithm 1) with negative log likelihood (eq. 21) to compute the loss. To ameliorate overfitting, I include weight decay (eq. 22) and learning rate decay by a factor of 0.5 with a patience parameter of 5 epochs and a minimum learning rate of 10^{-10}.

3.2 Hyperparameters

To determine a final model, I searched the following hyperparameters using validation set accuracy after 100 epochs: aggregation method {GCN, GIN}, number of convolutional layers {2, 3}, hidden units {32, 64}, learning rate {0.01, 0.001, 0.0001}, batch size {32, 64, 128}, and weight decay factor (L2 regularization) {0.01, 0.001}. The final model used to evaluate the test set uses 3 GIN layers, with 32 hidden units in each linear layer, batch size 64, starting learning rate 0.0001, and weight decay factor 0.01.

3.3 Definitions

Batch Normalization: Reduces internal covariate shift, allowing for some regularization and faster convergence [8][13]:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta, \tag{17}$$

where x is the input tensor, y is the output tensor, $\epsilon = 10^{-5}$, and γ and β are learnable parameter vectors of the same size as x.

ReLU: Rectified linear unit activation function.

$$ReLU(x) = max(0, x) \tag{18}$$

Global Mean Pool: Used to generate graph-level embeddings by averaging node features. For a single graph, this is equivalent to the following equation, where N is the number of nodes in the graph and x_n represents the vector embedding for node n [2].

$$y = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{19}$$

Dropout: During training, the dropout layer randomly (with probability p) zeroes elements from the input and scales the outputs by a factor of $\frac{1}{1-p}$. During evaluation, the dropout layer does nothing [13]. This has been shown to regularize the network parameters [6].

Log Softmax: Converts output logits to log probabilities [13]. Note that x_i represents the i^{th} element in the input tensor x.

$$LogSoftmax(x_i) = log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right) \tag{20}$$

Adam: [9] an adaptive learning rate optimization algorithm. In Adam, momentum is incorporated as an estimate of the first-order moment [3]. Moreover, Adam computes corrected estimates for the first- and second-order moments [3] (Algorithm 1).

Negative Log Likelihood: [13]

$$loss(x, y) = \sum_{n=1}^{N} -x_{n,y_n}, \tag{21}$$

where N is the batch size and x_{n,y_n} is the log probability computed by the log softmax function for the true label y_n of data point n.

Weight Decay / L2 Regularization: Adds a term to the loss function that expresses a preference for the weights to have smaller squared L^2 norm:[3]

$$J(\mathbf{w}) = Loss_{train} + \lambda \mathbf{w}^T \mathbf{w} \tag{22}$$

Learning Rate Decay Function: Reduces learning rate by a factor of 0.5 if no improvement in loss is observed after 5 epochs. Initialized with a minimum learning rate of 10^{-10}.

Algorithm 1 Adam optimization algorithm[9][13]

```
Require: $\alpha$ (learning rate), $\beta_1$, $\beta_2$ (exponential decay rates for the 1st and 2nd moment estimates), $f(\theta)$ (objective function), $\theta_0$ (initial parameter vector), $\lambda$ (weight decay factor)

  $m_0 \leftarrow 0$ (first moment)
  $v_0 \leftarrow 0$ (second moment)
  for $t$=1 to ... do
      $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
      if $\lambda \neq 0$ then
          $g_t \leftarrow g_t + \lambda\theta_{t-1}$
      end if
      $m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)g_t$
      $v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)g_t^2$
      $m_t^{corr} \leftarrow m_t/(1-\beta_1^t)$
      $v_t^{corr} \leftarrow v_t/(1-\beta_2^t)$
      $\theta_t \leftarrow \theta_{t-1} - \alpha m_t^{corr}/(\sqrt{v_t^{corr}} + \epsilon)$
  end for
      return $\theta_t$
```

4 Experiment

The model described above was trained and evaluated on a graph classification task using the MCF-7 benchmark dataset. MCF-7 is a benchmark dataset for graph machine learning published by Morris, et al [11]. The data was originally compiled from PubChem, which publishes information on the biological activities of molecules. MCF-7 contains structural and feature information of 27770 small molecules.

Each molecule in the dataset is associated with an adjacency matrix, representing the graph structure, and a feature matrix of size $|V| \times 46$, where $|V|$ is the number of nodes in a graph. Thus, the node features are described by a vector of length 46. Furthermore, the dataset consists of two classes: each data point is labelled 1 (active) or 0 (inactive) according to the results of bioassay records for anti-cancer screen tests for a particular cancer cell line. Note that the data is unbalanced with approximately 8 percent of the graphs labeled 1 (active) and 92 percent labeled 0 (inactive).

The dataset was split with a 80/10/10 ratio into training, validation, and testing sets. (The total number of graphs in each set is 22216, 2777, and 2777, respectively).

To address class imbalance, I use re-sampling of the training and validation data. The data was subject to random undersampling and oversampling, such that each batch represented the two classes with equal proportions: Oversampling entails randomly sampling data points from the minority class, with replacement, and adding them to the batch. Undersampling randomly removes data points from the majority class. This sampling technique is roughly the equivalent of weighting the classes differently in the loss computation [1]. Note that the test set was not resampled and the final evaluation metrics were computed using the test set containing 2545 samples with label 0 and 232 samples with label 1. All models were implemented using PyTorch [13] and PyTorch Geometric [2].

5 Results

The model described above was trained for 200 epochs and evaluated on the test set. Below, Fig 3 and Fig 4 depict the loss and accuracy, respectively, per epoch during training. In Table 1, I provide a comparison of my results with those published by [11] for their model trained on the same dataset. Note that accuracy from [11] was averaged over 10-fold cross validation, thus the standard deviation of the accuracy over 10 folds is provided. The confusion matrix (Fig 5) gives the True Negatives (TN), False Positives (FP), False Negatives (FN), and True Positives (TP) for each data point in the test set. Note that ”negative” indicates that the data point is labeled 0 (inactive) and ”positive” indicates that the data point is labeled 1 (active). These values were used to compute the metrics in Table 1 as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{23}$$

$$Sensitivity = \frac{TP}{TP + FN} \tag{24}$$

$$Specificity = \frac{TN}{TN + FP} \tag{25}$$

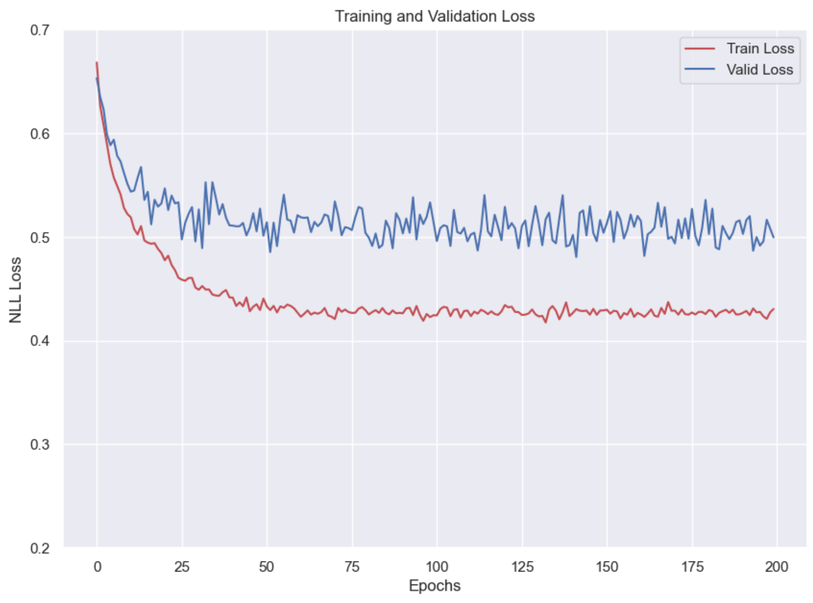

Figure 3: Training and Validation Loss

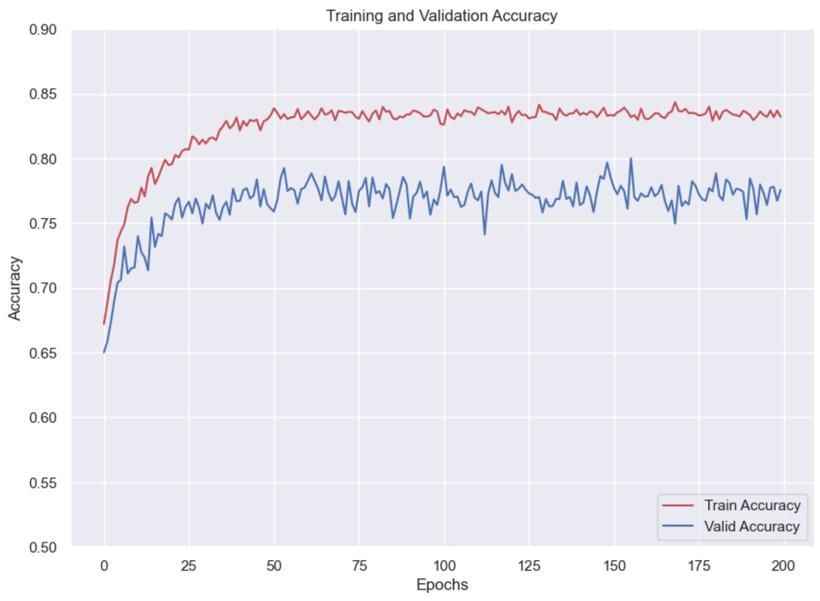

Figure 4: Training and Validation Accuracy

Model	Accuracy (%)	Sensitivity (%)	Specificity (%)
GIN (this model)	78.68	72.41	79.25
GIN (from [11])	$92.0 \pm 0.03 \pm 0.6$	N/a	N/a

Table 1: Comparison of evaluation metrics with a previous result.

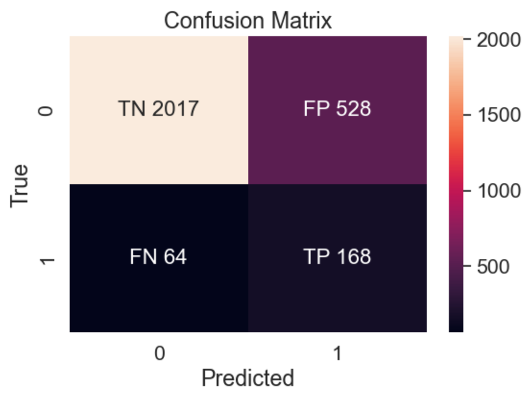

Figure 5: Confusion Matrix

6 Conclusion

In this paper, I investigated the theoretical basis of embedding learning algorithms for graph prediction models and discussed contemporary methods to improve the expressiveness of these embeddings. Furthermore, I selected a simple, yet powerful aggregation scheme, GIN, which was used to learn embeddings for a downstream graph classification task. To improve the model fit, the input data was re-sampled with a combination of over- and undersampling to amend imbalance in the class distribution. The use of training data sampling is a meaningful difference between this model and the one used by [11].

Despite this technique resulting in a lower accuracy (Table 1), the advantage is elucidated by the balanced confusion matrix (Fig 5), indicating that the model has adequately low bias and consequently, superior ability to generalize compared to a model that was trained on the original, imbalanced data. However, re-sampling techniques are associated with certain challenges, as well. Undersampling removes valuable information from the training data and oversampling is likely to cause overfitting, leading to worse classifier performance [1]. Unfortunately, many techniques for correcting class imbalance in machine learning, such as data augmentation and synthetic oversampling, are not feasible for graph-structured data.

Concerning the representational capacity of the model in this paper, note that the model achieves a maximum training accuracy of 84.36% (Fig 4). This result suggests that the GIN model has the capacity to distinguish between most of the graph structures in the training set. Despite this, there remains room for improvement in the expressiveness of the model. Current research aims to develop GNNs that surpass the power of the 1-dimensional Weisfeiler-Lehman Test (1-WL) and, by extension, the Graph Isomorphism Network framework.

To this end, one interesting direction is the introduction of GNNs based on higher-dimensional variants of 1-WL. Notably, Morris, et. al [12] propose a GNN based on the k-dimensional Weisfeiler-Lehman Test, which performs message passing not only between individual nodes, but also between subgraphs, allowing the model to characterize structures that are not captured by 1-WL alone. The authors of this paper further propose "1-k-GNN", a hierarchical variant constructed by pooling message passing schemes of various orders, thereby capturing structural information from multiple "perspectives". Furthermore, Zhang and Li [17] propose to resolve the limited expressiveness of rooted subtrees by "nesting" message passing algorithms. In this way, node embeddings encode a "subgraph" about the node, as opposed to a subtree. Both of these recently developed models have showed superior expressiveness to 1-WL and message passing GNNs.

Acknowledgements

This paper would not have been possible without the invaluable guidance of Dr. Nikolay Sirakov throughout this semester. Also, I want to thank the committee members, Dr. Mehmet Celik and Dr. Padmapani Seneviratne, for volunteering their time and expertise to improve this project.

References

[1] Paula Branco, Luís Torgo, and Rita P Ribeiro. A survey of predictive modeling on imbalanced domains. *ACM computing surveys (CSUR)*, 49(2):1–50, 2016.

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

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[4] William L Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[5] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

[6] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[8] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

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

[10] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[11] Christopher Morris, Nils M Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*, 2020.

[12] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019.

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

[14] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

[15] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[16] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[17] Muhan Zhang and Pan Li. Nested graph neural networks. *Advances in Neural Information Processing Systems*, 34:15734–15747, 2021.