CNN-inspired Graph Kernel Neural Networks

Adam Cahall **Avi Ruthen**

1 Abstract

In recent years, a new class of Graph Neural Network (GNN) architectures has emerged which combines ideas from graph kernel methods and the Message-Passing Neural Network (MPNN) GNN architecture. These models, which we call Graph Kernel Neural Networks, achieve state-of-the-art performance on graph classification benchmark datasets and often have an added interpretability advantage over traditional MPNNs due to frequently being parameterized by learnable hidden graphs which can be visualized. One of the most popular Graph Kernel Neural Network architectures is the KerGNN model: inspired by how Convolutional Neural Networks (CNNs) operate, KerGNN compares trainable graph filters to subgraphs of its input through a graph kernel to generate a rich graph embedding that encodes local and global structure. Given the strong similarities between KerGNN and CNNs, we take further inspiration from CNNs and incorporate pooling and attention, two popular components of modern CNN architectures, into the KerGNN architecture. Although our initial results are inconclusive, we believe that with additional time and computational resources, our extended model could potentially be shown to outperform the base KerGNN model. Our code is available at https://github.com/adamc-7/cs6850-final-project.

2 Introduction

Graph machine learning has garnered significant research interest due to the wide variety of data that can be represented by graphs, including social networks, chemical structures, and computer networks. In this paper, we focus on machine learning-based approaches for the graph classification task, in which the goal is to predict a singular overall label for an input graph. For example, one task could be to classify whether a molecule is either toxic or non-toxic.

In recent years, Graph Neural Networks (GNNs) have emerged as the leading approach for the graph classification task. Most state-of-the-art GNNs follow the Message Passing Neural Network (MPNN) framework, in which it is assumed that each node is equipped with a representation that is then updated iteratively via "message-passing" steps in which nodes share representation information with their neighbors. To generate a prediction, the final node representations are then aggregated via some sort of permutation-invariant function (e.g. mean or sum) to form a graph-level representation which is then passed through a standard multilayer perceptron (MLP).

Before the emergence of GNNs, graph kernel methods were the standard techniques used for graph classification. Graph kernels intuitively compute a measure of similarity between two graphs and can be used with traditional kernel methods like the support vector machine. Graph kernels have the advantage of implicitly operating in a high dimensional feature space (even when the input node features are low dimensional), which removes the need to explicitly engineer or learn high dimensional node representations. However, the expressivity of graph kernel methods can be considered limited since a particular kernel needs to be selected prior to the learning process, pre-determining the fixed high-dimensional feature space that will be used. In contrast, GNNs can learn their own task-dependent high dimensional representations during the training process. This advantage, along with the high computational cost of kernel methods, led to GNNs becoming the preferred choice over graph kernel methods.

Though graph kernel methods have been mostly overlooked as of late, there has been some recent work done to incorporate graph kernel functions into neural network architectures in an attempt to address limitations of the MPNN design. Two of the foundational architectures that combine GNNs and graph kernels are Random Walk Graph Neural Network (RWNN) ([11]) and KerGNN ([2]).

The authors of [11] were concerned that the structure of the input graph was being overlooked to a degree by MPNNs, due to the fact that MPNNs use permutation-invariant functions that only make use of a limited amount of structural information to update node representations and eventually construct a graph-level representation. They felt that it intuitively made more sense to apply structure-aware graph-level functions directly to the input to generate a graph-level representation. Their proposed RWNN model abandons the message-passing step entirely and uses a graph kernel, namely the random walk kernel, as this graph-level function to compare the original input graph to a series of learned "hidden graphs" to generate a graph-level representation. Note that by learning the hidden graphs used in the kernel rather than using fixed graphs, RWNN addresses the previously-mentioned expressivity concerns of kernel methods to an extent. Additionally, the RWNN model provides an interpretability advantage, as we can examine the learned hidden graphs to gain valuable insight into characteristics of the training data (as compared to traditional MPNNs, where the message-passing step is parameterized by hard-to-interpret fully-connected layers).

KerGNN, a later work, aims to improve the expressibility of GNNs by replacing the standard message-passing procedure with a subgraph-based node aggregation algorithm that also makes use of the random walk kernel. In KerGNN, each node has an associated subgraph consisting of itself and its neighbors which is compared with a set of learned hidden graphs using the random walk kernel (similar to RWNN) at each layer to update the node representation.

One particularly interesting aspect of the KerGNN model is its connection to convolutional neural networks (CNNs) used for image classification. Specifically, the authors of [2] noticed that the process of comparing local subgraphs to learned hidden graphs using a graph kernel in KerGNN seemed quite similar to the process of comparing patches of images to learned filters using the 2D convolution in CNNs. They formally analyzed the relationship between these two processes and showed that the 2D convolution can in fact be alternatively viewed as a series of kernel function applications under certain assumptions.

Inspired by this connection, we extend the KerGNN architecture by incorporating strategies and components used in modern CNN architectures. While many ideas from CNNs have already been adapted for other GNN architectures (Attention: [12], Pooling: [9]), it seems plausible that perhaps they would be particularly effective when added to KerGNN given the natural (and theoretically-verified) analogue between KerGNN and CNNs.

Specifically, as our contribution, we incorporate both pooling layers and an attention mechanism into the KerGNN architecture. For pooling, we leverage an existing graph pooling technique, TopK pooling (introduced in [4]). At a high level, TopK pooling layers iteratively reduce the size of the graph during the forward pass by repeatedly selecting the k most “important” nodes to keep (and discarding the rest). For attention, we choose to adapt the Squeeze-and-Excitation (SE) block, a popular form of channel-wise attention, for KerGNN ([5]). In our implementation, after each hidden graph layer, an SE block dynamically reweights the different features of the node representations. Intuitively, these SE blocks allows the model to emphasize important features when updating node representations.

To benchmark the effectiveness of these mechanisms, we tested our extended KerGNN model on the IMDB-BINARY, IMDB-MULTI, and PROTEINS datasets that were also used to evaluate the base KerGNN model. We examined the average accuracy of our model across multiple runs, and found that our added mechanisms were consistently within a standard deviation of the accuracy KerGNN reports in its paper. While the initial results seem to suggest that pooling and SE blocks may actually hurt performance, in our Analysis section, we offer reasonable explanations for this discrepancy and also motivate the use of these mechanisms for more complex graph classification tasks.

3 Background and Related Work

3.1 Background on Graph Kernels

Formally, a graph kernel is a symmetric positive semidefinte function $k(G, G')$ that takes as input two graphs G and G' and outputs a single real number which can be considered a similarity measure between G and G'. It is important to note that most graph kernels are non-differentiable, making them not easily usable inside of neural network architectures. Out of the limited choices for differentiable kernels, both of the architectures we focus on, RWNN and KerGNN, choose to make use of the random walk kernel.

To define the random walk kernel, we first need to introduce the direct product graph. For $G = (V, E)$ and $G' = (V', E')$, their direct product graph is $G_\times = (V_\times, E_\times)$ with $V_\times = \{(v, v') : v \in V \wedge v' \in V'\}$ and $E_\times = \{\{(u, u'), (v, v')\} : (u, v) \in E \wedge (u', v') \in E'\}$. If we denote the adjacency matrix of $G_\times$ as $A_\times$, we can define the P-step random walk kernel as:

$$k(G, G') = \sum_{i=1}^{|V_\times|} \sum_{j=1}^{|V_\times|} \left[\sum_{p=0}^{P} \lambda_p A_\times^p \right]_{ij}$$

where $\lambda_0, \ldots \lambda_P$ is a sequence of positive, real weights. Ignoring the weights, this kernel is measuring the number of common walks shared by G and G' of length at most P. To see why, first note that performing a random walk on $G_\times$ is equivalent to simultaneously performing a random walk on both G and G' since the nodes and edges in $G_\times$ represents pairs of nodes and pairs of edges in G and G'. Then, recall that the (i, j)-th entry of $A_\times^p$ (or generally any adjacency matrix raised to the power p) corresponds to the number of walks of length p that start at i and end at j.

We can also expand this definition to handle graphs that contain nodes that are labeled with a feature vector (which is often the case for graph classification tasks). Assume each node has a feature vector in $\mathbb{R}^d$, then we can define the node attribute matrix of a graph G with n nodes by stacking these feature vectors to obtain $X \in \mathbb{R}^{n \times d}$. If we have $X_1 \in \mathbb{R}^{n_1 \times d}$ and $X_2 \in \mathbb{R}^{n_2 \times d}$ for two graphs G_1 and G_2, then the product $S = X_1 X_2^T \in \mathbb{R}^{n_1 \times n_2}$ will encode pairwise similarities between the nodes of G_1 and G_2. Specifically, the (i, j)-th entry of S will contain the inner product (which can be thought of as a measure of similarity) between the feature vector of node i in G_1 and the feature vector of node j in G_2. We can use now use S to reweight the terms being summed by the kernel by similarity (note that we are only reweighting by the similarities of the first and last nodes in each simultaneous walk– this has been addressed in, [7], a later work, but turns out to have minimal effect on performance in practice). If we properly (see [11] for details) flatten S into a vector $s \in \mathbb{R}^{n_1 n_2}$, we can write our revised kernel as:

$$k(G, G') = \sum_{i=1}^{|V_\times|} \sum_{j=1}^{|V_\times|} \left[\sum_{p=0}^{P} \lambda_p s_i s_j A_\times^p \right]_{ij} = \sum_{p=0}^{P} \lambda_p s^T A_\times^p s$$

Both RWNN and KerGNN use a form similar to this version of the random walk kernel in their architecture.

3.2 Random Walk Graph Neural Network

The Random Walk Graph Neural Network (RWNN) model leverages the random walk graph kernel and utilizes trainable hidden graphs that can be visualized for interpretability. The RWNN architecture contains a single hidden graph layer, which consists of a series of hidden graphs that are parameterized by a trainable adjacency matrix and node attribute matrix (see definition in Background section), followed by a standard MLP. A forward pass through an RWNN consists of the input graph first being compared to each hidden graph via the random walk graph kernel. Each of the computed kernel outputs are then concatenated into a vector, which is then passed to the MLP to generate the predicted class for the input graph (note that the model can be adapted for a regression task as well).

3.3 KerGNN

The KerGNN architecture takes the basic idea of comparing the input graph to a series of hidden graphs from RWNN and incorporates it directly into the message-passing process of the MPNN framework.

In a typical MPNN, we start with an initial representation for each node, which can be a given node feature vector from the dataset if available, or a learned embedding if not. The MPNN then runs a series of message-passing steps, during which node representations are updated by exchanging representation information with neighbors in order to generate representations which can potentially contain both local and high-level structural information about the input graph. At each step, for a given node v, each of its neighbors u first compute a message (of same dimension as the node representations) to send to v using a learnable message function that takes into account both u and v's representations. These neighbor messages are then aggregated using a permutation-invariant function into a single combined message which is then sent to v. Finally, v's updated representation is computed by a learnable update function which takes into account both the current representation of v and the aggregated message.

In the KerGNN model, message-passing steps are replaced by neural network layers which update node representations by comparing local subgraphs with learned hidden graphs using the random walk kernel. Specifically, each KerGNN layer consists of a series of d hidden graphs which are parameterized by a trainable adjacency matrix and a trainable node attribute matrix (see background section for a definition). To compute a new d-dimensional representation for a node v, the layer compares the induced subgraph consisting of v and its neighbors with each of the d hidden graphs using the random walk kernel. In other words, each entry of the new representation corresponds to the random walk kernel output when it is applied to the induced subgraph and one of the hidden graphs.

To generate a graph-level representation in order to be able to make a graph-level classification, KerGNN functions similarly to MPNNs by using a permutation-invariant function (sum) to combine all node-level representations. Specifically, if there are L layers, KerGNN sums all node representations at each layer to create L graph-level representations, and then concatenates these layer-specific graph-level representations to get a final overall graph-level representation. As in both MPNNs and RWNN, this graph-level representation is then fed into a standard MLP to generate a final label prediction.

The KerGNN architecture has two main advantages over traditional MPNNs: interpretability and expressibility. As in RWNN, the interpretability of KerGNNs comes from being able to examine the learned hidden graphs to get insights into the data and the prediction strategies being used by the model. Looking at expressibility, observe that KerGNN takes into account the subgraph structure when computing new node representations rather than simply aggregating messages from neighboring nodes in an unordered fashion as done in MPNNs. Furthermore, KerGNN incorporates this more complex subgraph structure into its representation updates by intentionally making use of the random walk kernel, which operates in a high-dimensional space, to perform the comparisons between the subgraph and the hidden graphs. Given these apparent advantages, it seems plausible that KerGNN could be more expressive than standard MPNNs. Indeed, the authors showed that KerGNN is more powerful than the 1-WL graph isomorphism test (a common benchmark for GNN expressivity) which standard MPNNS are limited by.

3.3.1 KerGNN Connection to CNNs

In generating the novel KerGNN architecture, the authors note that they were inspired by the way convolutional neural networks (CNNs) operate on images ([2]). Many computer vision tasks involve operating on images with high resolution, and since images are two-dimensional, the number of features scales quadratically and makes processing the whole image simultaneously computationally infeasible. Consequently, CNNs take advantage of a filtering mechanism that exploits the local structure of images to cut down on runtime. For each convolutional layer, trainable filters of size $k \times k$ are applied to similar-sized sub-regions across the image, where the output pixel is computed as the convolution of the filter and a $k \times k$ subregion of the image centered around the pixel. As the image size decreases through many convolutional layers and due to pooling, the output image will eventually become so small that a $k \times k$ filter will be able to process most of the image at once; this allows the model to learn some global structure in tandem with the rich local structure it learned from convolutional layers earlier in the model architecture.

The authors of KerGNN decided to try an analogous approach for graphs, where trainable graph filters operate via the random walk graph kernel on subgraphs of the input graph (similar to how trainable image filters operate on larger images through convolution), and they clearly succeeded in generating an effective model that resembles a CNN architecture. However, we ask if we can extend the analogy between graph classification and image classification even further. Literature has shown techniques such as pooling and attention can help further reduce computational costs while simultaneously improving performance for CNNs, and we wonder if these techniques can similarly improve the KerGNN model.

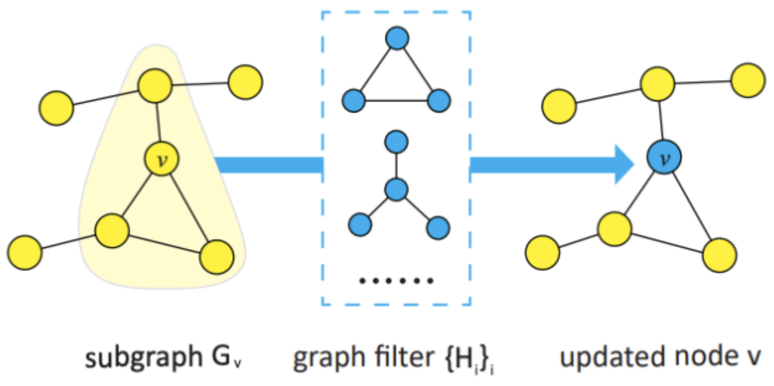

Figure 1: An abstraction of the KerGNN architecture (figure taken from [2]). Subgraphs of the input graph G are compared to graph filters through the graph kernel; this is used to compute the updated set of features for node v. The output of one layer of graph filters can be passed into subsequent layers similar to how convolutional layers work in CNNs.

4 CNN-inspired Extensions to KerGNN

Despite the success of KerGNN on a variety of graph classification benchmark datasets, we thought that extending the base KerGNN model with concepts from CNNs; namely, pooling and attention, could potentially further improve performance. In the following sections, we will motivate our decision to incorporate pooling and attention into KerGNN, and introduce gPool and the Squeeze-and-Excitation (SE) block, the two specific mechanisms we chose to implement.

4.1 Pooling Layers

4.1.1 Pooling in CNNs

Pooling layers are present in virtually every CNN. They are used to progressively downsample the image during the forward pass in order to reduce computational costs and improve model generalization ([13]). Specifically, regions of pixels (say, a 3x3 grid) are aggregated into single pixels by usually either averaging channel intensity values ("mean pooling") or taking the maximum intensity value for each channel ("max pooling").

4.1.2 Comparison of Node Clustering and TopK Pooling Approaches in GNNs

There exists GNN analogues for both mean and max pooling. One common approach is to first run a clustering algorithm (for example, Graclus [6]) to split the nodes in disjoint clusters. These clusters can be considered to correspond to the "regions of pixels" described above. In order to compute the new, coarsened graph, the nodes in each cluster are merged into a single node whose representation is computed by taking a channel-wise average or maximum of the representations of all nodes in the cluster.

Though these node clustering approaches are the GNN pooling techniques most directly related to CNN pooling techniques, we chose not to incorporate them with KerGNN due to their computational overhead. Specifically, we realized that any node clustering pooling approach would require a complete recomputation of the induced subgraphs, as each merged node no longer represents any single node from the original graph. Since we surprisingly found that the most computationally-intensive step in the authors' KerGNN implementation was actually computing the initial induced subgraphs for each node, we predicted that needing to completely recompute these subgraphs after each pooling layer would prohibitively slow down the training process.

Instead, we opted to incorporate a different, more computationally efficient style of graph pooling with KerGNN, known as "TopK pooling" or alternatively "Node drop pooling". In TopK pooling algorithms, the input graph is downsampled by selecting the top k "most important" nodes, rather than clustering and merging nodes. The methodology of determining the relative importance of nodes varies by algorithm. Though TopK pooling techniques are more lossy than node clustering techniques due to simply dropping nodes rather than merging them, they are also more efficient and often the only feasible option when dealing with large input graphs ([9]). For KerGNN specifically, we are able to construct the output of a TopK pooling layer by just discarding the induced subgraphs for dropped nodes and making slight modifications to the adjacency matrices of the induced subgraphs of remaining nodes, rather than needing to do a full recomputation of all subgraphs as for node clustering pooling algorithms.

4.1.3 gPool Layer

We specifically chose to use the gPool layer ([4]), one of the first-introduced and most widely-used TopK pooling algorithms. Each gPool layer is parameterized by a learnable projection vector $p \in \mathbb{R}^d$, where d is the dimensionality of each node representation vector. The vector p is used to define "importance" for a given gPool layer. Specifically, gPool computes importance scores y_i for each node i by calculating the scalar projection of node i's representation vector x_i onto p: $y_i = x_i p / ||p||$. Thus, nodes with larger y_i have representations that are more similar to p and so can be considered "more important". Following the scalar projection computations, a sigmoid activation function and other intermediate steps are applied to enable the gPool layer (specifically, the projection vector p) to be differentiable. As a final step, gPool retains the k nodes with the largest y_i values (where k is a hyperparameter set beforehand) and discards the rest. In our implementation, we insert a gPool layer after every hidden graph layer (following the standard practice to have a pooling layer after every message-passing step in GNNs and after every convolutional layer in CNNs).

4.2 Attention Mechanism

4.2.1 Attention in CNNs

Attention mechanisms are another CNN component which have become widely used in recent years. There are generally two forms of attention considered in CNNs: spatial attention and channel-wise attention. In spatial attention, pixel-level descriptions are first are formed by computing metrics such as average or maximum intensity across channels for each pixel in the image. These pixel-level descriptions are then fed into a shallow CNN to compute a spatial attention map (of the same height and width as the image), which each pixel's intensities in the input image are element-wise multiplied by across all channels. At a high level, spatial attention mechanisms give CNNs the capability to emphasize important regions of the image during the forward pass. Channel-wise attention is quite similar, except that it operates across channels rather than pixels. Specifically, channel-level representations are first computed, which are then passed into an MLP to compute new weights for each channel, which are then element-wise multiplied across each channel of the input image (so for a given channel, each pixel's intensity value for that channel is scaled by the same newly-computed weight). Including a channel-wise attention mechanism in a CNN architecture allows the model to alternatively emphasize important channels of the image.

4.2.2 Comparison of Spatial and Channel-wise Attention Mechanisms in GNNs

Spatial attention mechanisms have been heavily explored in the context of GNNs as well. Perhaps the most notable GNN architecture to incorporate spatial attention is the Graph Attention Network (GAT). The GAT architecture allows each node to compute attention coefficients between itself and its neighbors, which are then used to dynamically reweight incoming messages from each neighbor during the message-passing step.

On the other hand, we found that channel-wise attention in GNNs appears to be relatively less explored in the current literature. We did find one work ([3]), which as part of its contribution explores channel-wise self-attention in GNNs, but this did not seem to inspire significant future work in this area. Though at first this lack of published work seemed potentially concerning (as perhaps many people had previously tried incorporating channel-wise attention in GNNs but achieved poor results, and disregarded the idea), we decided to still choose to incorporate channel-wise attention over spatial attention in KerGNN for two major reasons.

Firstly, as the goal of our project was to take inspiration from CNNs, we ideally wanted to incorporate an attention mechanism in KerGNN that very closely resembled an analogous attention mechanism from CNNs. On this note, we felt that channel-wise attention mechanisms from CNNs were more directly adaptable to KerGNN than spatial attention mechanisms from CNNs. Since images and graphs have significantly different spatial structures, any CNN spatial attention mechanism would likely need to be significantly modified to work with KerGNN. However, the interpretation of channels in pixels is quite similar to interpretations of "channels" in node representations. In fact, we found that it seemed feasible to directly insert a CNN channel-wise attention mechanism into KerGNN, with very few necessary modifications and an essentially identical interpretation.

A second practical advantage of channel-wise attention mechanisms lies in their computational efficiency. Typically, GAT-style spatial attention mechanisms add significant overhead, as they make use of the full adjacency matrix of the graph and compute reweightings of all messages being passed to each node by its neighbors. On the other hand, channel-wise attention mechanisms do not involve the adjacency matrix, and only need to process the node representation vectors, resulting in noticeable speedups. Specifically, the authors of [3] found that using a channel-wise attention mechanism instead of a GAT resulted in memory savings and runtime speedups of 10-100x (with increasing benefits as input graph sizes increased), with only a minor performance hit (roughly 1-2 percent lower classification accuracy). As we found that training and evaluating the base KerGNN model was already quite costly, a channel-wise attention mechanism seemed like the right choice.

4.2.3 Squeeze-and-Excitation Block

After narrowing our choices to channel-wise attention mechanisms, we ended up deciding to incorporate the Squeeze-and-Excitation (SE) block into the KerGNN model ([5]). SE blocks are a popular, lightweight form of channel-wise attention used primarily in CNNs that we found could be directly inserted into the KerGNN architecture.

The first step in an SE block is the "Squeeze". Here, the collection of all node representation vectors $x_i \in \mathbb{R}^d$ is "squeezed" into a single channel descriptor vector $z \in \mathbb{R}^d$ via global average pooling, where each channel of z contains the average value of that channel across all N nodes in the input graph: $z = \frac{1}{N}\sum_{i=1}^{N} x_i$.

In the "Excitation" step, the channel descriptor vector z is fed into a shallow fully-connected network to produce an input-dependent recalibration vector $s \in \mathbb{R}^d$. Lastly, each of the node representation vectors x_i are element-wise multiplied by s to produce new, recalibrated representations.

In our implementation, we choose to insert an SE block following each hidden graph layer of KerGNN. After passing the input graph through a hidden graph layer, each channel of each node representation corresponds to the similarity between the node's induced local subgraph and one of the learned hidden graphs, as computed by the random walk kernel. Thus, each channel contains local information and is interdependent with all other channels (as each channel is computed using the same induced subgraph). The insertion of the SE block allows the model to augment these channels with global information (by using global average pooling to compute the channel description vector z), and also model these potentially nonlinear interdependencies (by passing z through an MLP to compute the recalibration vector s).

One final important note is that while we originally thought that we were the first work to directly apply the SE block to GNNs, we did eventually discover that [8] had recently made use of SE blocks in their anatomically-constrined squeeze-and-excitation graph attention network (ASEGAT) model, created specifically for performing brain cortical surface parcellation. However, their model is still significantly different from our approach, as it incorporates SE blocks with the the GAT architecture rather than the CNN-inspired KerGNN architecture.

5 Implementation Details

5.1 gPool Layer

In order to incorporate the gPool layer into the KerGNN architecture, we made use of the PyTorch Geometric (PyG) library's TopKPooling module, a convenient open-source implementation of the gPool layer. Unfortunately, since KerGNN was implemented in pure PyTorch and used a significantly

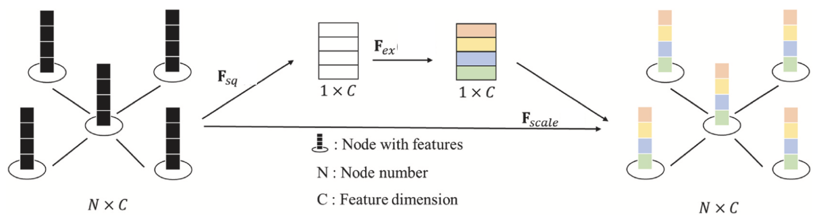

Figure 2: Design of the Squeeze-and-Excitation Block (figure taken from [8], note C here corresponds to d). $\mathbf{F}_{sq}$ represents the squeeze operation, which computes the channel descriptor vector z. $\mathbf{F}_{ex}$ represents the excite operation, which computes the recalibration vector s by passing z through an MLP. Lastly, $\mathbf{F}_{scale}$ represents the final scaling performed on each node representation vectors x_i by s.

different data format than the TopKPooling module expected, a substantial engineering effort was required to first convert the input graph into PyG's expected format, and then recover KerGNN's data format from the output of the TopKPooling module. It seems that these data conversion steps likely also added computational overhead, as we observed roughly a 2x increase in runtime over the base KerGNN after adding gPool layers.

5.2 Squeeze-and-Excitation Block

We implemented the Squeeze-and-Excitation block as described in the original paper. The exact architecture of the MLP that the channel descriptor vector y is fed into is as follows:

- Fully-connected layer with input dimension d and output dimension d/r
- ReLU activation function
- Fully-connected layer with input dimension d/r and output dimension d
- Sigmoid activation function

where r is a hyperparameter known as the reduction ratio. The two fully-connected layers form a bottleneck by first reducing the dimensionality to d/r and then increasing it back to d. The purpose of including this bottleneck is to reduce the model complexity and thus the potential for overfitting.

6 Experiments

We consider three proposed extensions to the base KerGNN model for evaluation: KerGNN with gPool layers, KerGNN with SE blocks, and KerGNN with both gPool layers and SE blocks. We train and evaluate each of these new architectures on several graph classification benchmark datasets, and also compare their performances to the results reported in the original KerGNN paper.

6.1 Datasets

IMDB-BINARY: IMDB-BINARY is a social interaction dataset featuring 1000 actors and actresses with edges drawn if two actors or actresses played roles in the same movie. This particular dataset is a binary classification task (deciding between action and romance genres), with 1000 input graphs and an average of 20 nodes per graph.

IMDB-MULTI: IMDB-MULTI is also a social interaction dataset featuring actors and actresses who played roles in the same movie, but this dataset is a multi-class setting with three potential genres (Comedy, Romance, and Sci-Fi) instead of the two used in IMDB-BIANRY. IMDB-MULTI involves 1500 input graphs, and each graph has an average of 13 nodes.

PROTEINS: PROTEINS is a binary classification dataset that attempts to classify graphical representations of proteins as enzymes or non-enzymes. Nodes are amino acids and edges are bonds, where bonds are defined as the amino acids being less than 6 Angstroms apart. The PROTEINS dataset features larger input graphs (averaging a size of 39 nodes), and there are 1113 of those graphs.

6.2 Experimental Setup

We follow an experimental setup similar to the one described in [1] as done in the original KerGNN paper. Specifically, we use a 10-fold cross-validation technique, where, for each fold, 10% of the data is first extracted to form a test set, with the remaining data being split 90/10 between training and validation, respectively (we use the same splits as [1] for fairness). For each fold, the performance of the model is evaluated on the validation set during each training epoch. The best-performing epoch model on the validation set is then evaluated on the held-out test set once training is complete to determine the test classification accuracy for the fold. After all folds are complete, the overall performance of the model is determined by the average test classification accuracy across all folds.

6.3 Hyperparameters

Due to computational limitations, we were unable to run a full hyperparameter sweep for each model. We instead chose to train and evaluate 2-3 hand-picked configurations for each model for each dataset. Since the authors of KerGNN did not release their optimal hyperparameter settings, we made use of knowledge about the datasets and common heuristics used in practice in order to select hyperparameter settings.

For instance, for the two datasets containing smaller graphs on average (IMDB-B and IMDB-M), we chose to use chose to use a shallower network and chose to retain a higher proportion of nodes after pooling layers when compared to PROTEINS, which contained relatively larger graphs on average. Additionally, for all experiments, we set the reduction ratio r to be the recommended general-use value (16) from the SE block paper ([5]). In our limited experimentation, we generally settled on the following hyperparameter settings (with some small variations depending on whether we were using pooling and/or SE blocks):

Table 1: Selection of Hyperparameter Settings

	IMDB-B + IMDB-M	PROTEINS
Number of hidden graph layers	2	3
Number of hidden graphs per layer	16, 32	16, 32, 64
Number of nodes per hidden graph	6	6
Reduction ratio r	16	16
Pooling node retention rate	0.75	0.5

6.4 Results

Table 2: Mean Test Set Accuracies and Standard Deviations

	IMDB-B	IMDB-M	PROTEINS
KerGNN	74.4 ± 4.3	51.6 ± 3.1	76.1 ± 4.1
KerGNN+Pooling	69.5 ± 4.8	46.6 ± 2.1	72.1 ± 4.9
KerGNN+SE Blocks	70.5 ± 4.4	48.6 ± 3.6	72.9 ± 5.7
KerGNN+Pooling+SE Blocks	69.0 ± 5.2	45.9 ± 2.9	71.1 ± 6.0

Unfortunately, we observed slightly lower accuracies compared to the results reported in the original KerGNN paper across all of our proposed models, though they do generally fall within one standard deviation of the original results. There are several possible explanations for this underperformance, which we discuss in the following Analysis section.

6.5 Analysis

While our initial results suggest a slight reduction in the performance of the KerGNN model, we believe that there are multiple potential explanations for the observed decrease in accuracy. To start, the authors of KerGNN performed grid search on an extremely large set of hyperparameters, including hyperparameters associated with the structure of the KerGNN model, the learning rate, the dropout rate, and the size of the subgraphs in order to find an effective hyperparameter setting for each dataset. Unfortunately, we did not have the time or resources to perform anything more than a minimal hyperparameter search, and our results are not necessarily reflective of the best accuracy we could achieve with a more optimal hyperparamter setting.

Furthermore, part of the rationale for using a CNN-inspired approach to graph classification is to have the ability to accurately process large graphs just as CNNs provide the ability to reason about large images. However, many of the datasets that KerGNN used to benchmark use graphs with an average size between 10 and 40 nodes; the corresponding adjacency matrices would be between $10^2 = 100$ and $40^2 = 1600$ entries, which is actually few enough features to be processed simultaneously (perhaps it is analogous to an image of 100 to 1600 pixels, which is a very small image for a classification problem). Consequently, losing some of the complexity of the few features via TopK pooling or adding more parameters to the model through channel-wise attention mechanisms would likely hinder performance on these smaller datasets. It is also worth mentioning that the effects of pooling in GNNs have been measured, and literature seems to suggest that pooling may not be as effective in GNNs as some may believe ([10]). Furthermore, while we wanted to test our model on larger datasets than the ones considered, training the KerGNN model proved to be too inefficient for larger datasets unless training were optimized substantially. However, we believe if we had had the opportunity to test our revised model on a larger dataset, the pooling and attention mechanisms might prove to be more useful; in fact, our results already being within one standard deviation of the original model on smaller datasets without a full hyperparameter sweep seems to support this hypothesis.

7 Future Work

Although our initial findings do not seem to suggest significant improvements in the KerGNN baseline model, there are many ways this project could be explored further. As mentioned in the analysis section of the results, much of the limitations in the results of our model came simply from a lack of time and computational resources to perform a larger hyperparameter sweep and also optimize model training and pre-processing for larger datasets (which attention and pooling may be more impactful). However, aside from gathering more results on our current model, there are also other ways we can improve it.

To start, in our reaction paper, we introduced a Sparsity-constrained Random Walk Neural Network, which incorporated L1 regularization in the parameters of the adjacency matrix for sparser hidden graphs that better mimicked the adjacency matrices of undirected graphs. The idea was to make the hidden graphs more interpretable, and our initial results (tested on a synthetic dataset where we already knew whwat identifying pattern the model should be looking for) seem to support the hypothesis that adding L1 regularization makes the hidden graphs more intereptable. We had initially hoped to incorporate L1 regularization into our revised KerGNN model and examine the effect on its interpretability, but due to differences in the design of the trainable adjacency matrix for KerGNN, we were unable to achieve the same sparsity within the hidden graphs without significant compromises in accuracy. However, we believe a more clever regularization function may be able to achieve the same results as RWNN, which could be a huge step in continuing to improve the interpetablity of these models.

In addition, we also explored the idea of using a trainable graph kernel. One major criticism of graph kernel neural networks is that the choice of kernel inhibits the expressivity of the model, but a trainable graph kernel could enhance the power of the model. In fact, KerGNN had already made an attempt to make aspects of the graph kernel trainable by setting constant terms $\lambda_0 \ldots \lambda_P$ in the random walk graph kernel to be learnable parameters, but it is worth exploring whether a fully trainable graph kernel may be more effective.

Lastly, although we managed to implement two common tools utilized in CNNs, there exist many other ideas from CNNs such as recurrent connections and positional encodings that we did not implement that could be incorporated into the architecture. While the effectiveness of these mechanisms varies with the task, it is perhaps worth exploring which other ideas from CNNs translate to the graph classification setting, and KerGNN's success certainly motivates pursuing some of these ideas further.

8 Conclusion

The results of our paper are inconclusive in determining the effects of the squeeze-and-excitation block and gPool layer on the KerGNN model, but given more time for an extensive hyperparameter sweep, it does not seem unreasonable to hypothesize that our added mechanisms could improve the model's performance, especially on more complex datasets (while also saving on computational costs). The KerGNN architecture itself has quite practical applications in industry due the interpretability of the model, which makes it (and variations of the model like ours) invaluable in illustrating to users what it is focusing on and thereby building trust in its predictions. Our analysis and future works sections motivate some additional directions for this project, including adding regularization for more interpretable results and potentially repurposing additional mechanisms from CNNs into the KerGNN architecture. We hope that with this additional work, our model may be competitive with other GNNs in performance while leveraging its interpretability for its users.

References

[1] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *ICLR*, 2020.

[2] Aosong Feng, Chenyu You, Shiqiang Wang, and Leandros Tassiulas. Kergnns: Interpretable graph neural networks with graph kernels. *AAAI 2022*.

[3] Hongyang Gao and Shuiwang Ji. Graph representation learning via hard and channel-wise attention networks. *ACM SIGKDD International Conference on Knowledge Discovery Data Mining*, 2019.

[4] Hongyang Gao and Shuiwang Ji. Graph u-nets. *International Conference on Machine Learning*, 2019.

[5] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks. *Conference on Computer Vision and Pattern Recognition*, 2018.

[6] Brian Kulis Inderjit S. Dhillon, Yuqiang Guan. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007.

[7] Meng-Chieh Lee, Lingxiao Zhao, and Leman Akoglu. Descriptive kernel convolution network with improved random walk kernel. *ACM TheWebConf*, 2024.

[8] Xinwei Li, Jia Tan, Panyu Wang, Hong Liu, Zhangyong Li, and Wei Wang. Anatomically constrained squeeze-and-excitation graph attention network for cortical surface parcellation. *Computers in Biology and Medicine*, 2022.

[9] Chuang Liu, Yibing Zhan, Jia Wu, Chang Li, Bo Du, Wenbin Hu, Tongliang Liu, and Dacheng Tao. Graph pooling for graph neural networks: Progress, challenges, and opportunities. *International Joint Conference on Artificial Intelligence*, 2023.

[10] Diego Mesquita, Amauri H. Souza, and Samuel Kaski. Rethinking pooling in graph neural networks. *NeurIPS*, 2020.

[11] Giannis Nikolentzos and Michalis Vazirgiannis. Random walk graph neural networks. *NeurIPS 2020*.

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

[13] Afia Zafar, Muhammad Aamir, Nazri Mohd Nawi, Ali Arshad, Saman Riaz, Abdulrahman Alruban, Ashit Kumar Dutta, and Sultan Almotairi. A comparison of pooling methods for convolutional neural networks. *Applied Sciences*, 2022.