Software Engineering Department
Braude College

Capstone Project Phase B 61999

Generating Multi-scale Graphs with Graph U-Net

Project code:24-1-R-9

Supervisor: Prof. Zeev Volkovich
Advisor: Dr. Renata Avros

Shahar Vachiler – shaharvac95@gmail.com
Shachar Dalal - drgwgwmd@gmail.com

Table of Contents

Abstract

This research investigates advanced techniques in hierarchical representation learning based on graphs, with the aim of addressing the challenges associated with capturing hierarchical relationships within graph structures. By employing the Graph U-Net (GU-Net) model and integrating a diffusion model into a Misc-GAN architecture, the project endeavors to automate the generation of multi-scale graphs to enhance the representation of complex relationships across diverse datasets.

The incorporation of GU-Net and diffusion models offers a unique approach to capturing intricate patterns within graph structures, enabling more effective representation learning. Through thorough testing and validation, the project seeks to establish the proposed model as a state-of-the-art solution for multi scale graph generation, with the objective of achieving greater efficiency and accuracy compared to previous methods.

Keywords: Graph U-Net, Graph Generation, Multi-scale Graphs, Graph neural network, GAN

Github link: https://github.com/shaharvac95/Capstone-project-2024-generating-multi-scale-graphs

1. Introduction and Background

Deep learning, a powerful artificial intelligence methodology, enables computers to derive insights from data similar to the human brain functioning [7] . This technique equips machines to perform intricate tasks and provide accurate predictions, finding widespread applications in domains such as image and speech recognition, autonomous vehicles, and medical diagnostics.

One area where deep learning holds significant potential is in graph-based hierarchical representation learning [2] . Traditionally, generating multi-scale graphs has been challenging due to the inherent complexity of capturing hierarchical relationships within graph structures. Existing methods may lack efficiency and struggle to provide a comprehensive representation of multi-scale features [1].

This project aims to address these limitations by leveraging the capabilities of the Graph U-Net (GU-Net) model and a diffusion model to automate the generation of multi-scale graphs, offering a superior solution for understanding complex relationships across diverse datasets [1][2] .

Graph-based representation learning, which is closely tied to deep learning techniques, poses challenges due to the complexity of hierarchical relationships within graph structures. By applying deep learning methodologies, such as the GU-Net model and diffusion model, to the domain of graph-based hierarchical representation learning, this project seeks to overcome these challenges and provide a more efficient and

comprehensive approach to capturing multi-scale features and hierarchical relationships within graph structures [2] .

This advancement could potentially enhance the performance of various applications that rely on graph-based representations, ultimately benefiting from the powerful insights derived through deep learning techniques [7] .

2. Related Work

Existing research in graph-based representation learning has explored various techniques, but challenges persist in capturing hierarchical relationships effectively [2] [5] . Methods such as GAN [6] and Graph U-Net [2] separately have shown promise but still lack efficiency and comprehensive representation of multi-scale features [1]. The proposed project aims to build upon this foundation by integrating the Graph U-Net model and a diffusion model within a Misc-GAN framework, offering a novel solution for automated multi-scale graph generation [1][2] .

2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are specialized architectures meticulously designed for processing grid-like data, particularly images [7] . By leveraging convolutional layers, CNNs adeptly capture hierarchical features and intricate spatial relationships within images, facilitating their superior performance across various computer vision tasks. From precise image classification to nuanced object detection and seamless segmentation, CNNs have become indispensable tools in modern AI applications. Their innate capability to autonomously discern and learn intricate features directly from raw data has catalyzed revolutionary breakthroughs in fields such as healthcare, robotics, and autonomous vehicles [5] .

2.1.1 Max-pooling

Max-pooling is a technique utilized in CNNs to downsample feature maps, reducing their spatial dimensions while retaining important information [5] . It operates by partitioning the input feature map into disjoint windows and selecting the maximum value from each window to form the output feature map. This process effectively reduces computational complexity and memory requirements while preserving the most relevant features for subsequent layers. Max-pooling plays a crucial role in hierarchical feature extraction, enabling CNNs to capture and emphasize significant spatial patterns across multiple scales efficiently.

2.1.2 Activation Function

In neural networks, the activation function determines whether the neuron is activated by performing calculations on a weighted sum with an attached bias [9] . Its purpose is to introduce nonlinearity into the neuron's output, enabling the network to handle complex tasks effectively.

The identity activation function is a straightforward linear function that outputs the same value as its input [9] . It does not introduce nonlinearity into the network but can be useful in scenarios where linear transformations are desired. While it does not add additional complexity, it still plays a role in the overall behavior of the network.

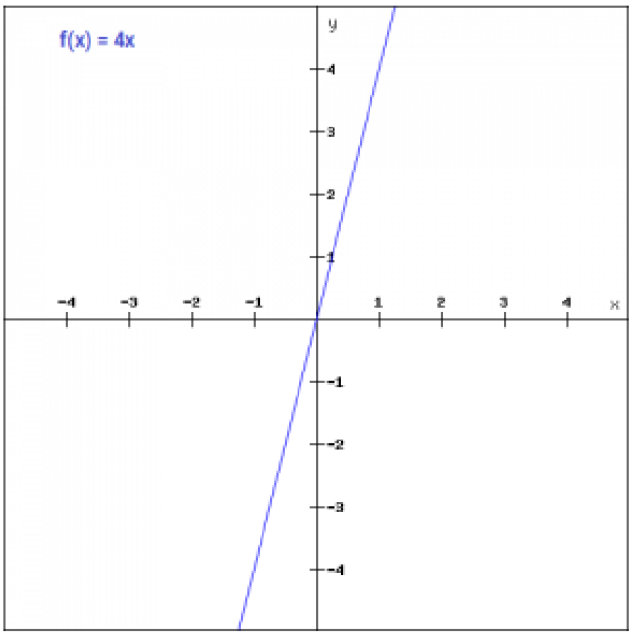


Figure 1: Identity (sometime also called Linear) Activation function [9] .

The softmax function takes a vector of real values, positive, negative, or zero, and transforms them into a vector of real values between 0 and 1 [9] . This transformation allows them to be interpreted as probabilities, with the sum of vector values being 1. Small or negative inputs are turned into low probabilities, while large inputs become high probabilities.

The sigmoid function takes a real value as input and yields an output value in the range between 0 and 1 [9] . It maps small values closer to zero and large values nearer to 1. Commonly used in the output layer of binary classification tasks, where classes are represented by zero and one. Data is classified as 1 if the sigmoid output is greater than 0.5 and as 0 otherwise.

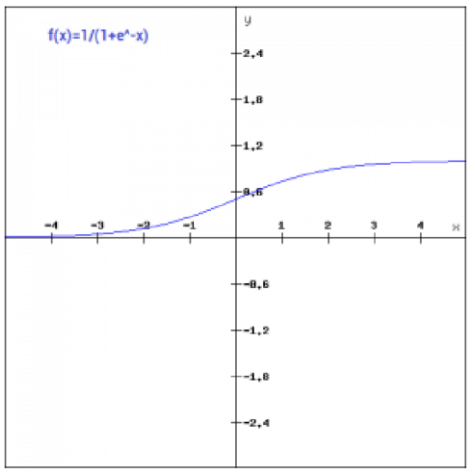


Figure 2: Sigmoid Activation function [9] .

The choice between using softmax and sigmoid in the final layer of a neural network depends on the nature of the problem and task requirements [9] . Softmax is well-suited for scenarios with multiple classes, transforming logits into a probability distribution over the classes. It allows the network to make definitive choices among classes, assuming each input belongs to only one class.

Softmax also encourages global information sharing among classes, capturing complex relationships. On the other hand, sigmoid is commonly used in binary classification tasks, providing independent probabilities for each class. It is preferred in tasks where an input can belong to multiple classes simultaneously, offering flexibility with decision thresholds. Softmax is chosen because it fits the model needs for multi-class classification.

2.2 Graph Convolutional Networks (GCNs)

Graph Convolutional Networks (GCNs) are neural networks designed for graph data [4] . They capture relationships between nodes in a graph, learning features directly from the graph's structure. GCNs use convolutional layers adapted for graphs, enabling them to handle tasks like node classification, link prediction, and graph clustering. They are widely used in social network analysis, bioinformatics, and recommendation systems.

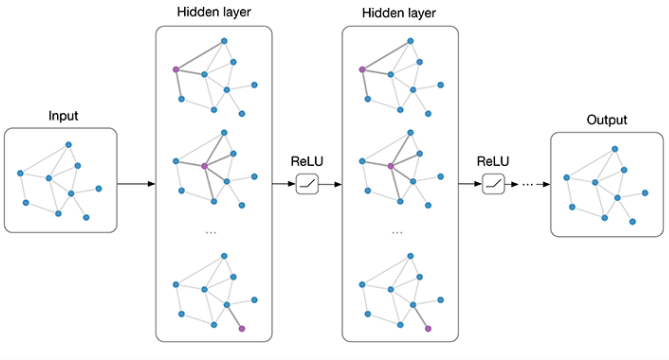


Figure 3: Multi-layer Graph Convolutional Network (GCN) with first-order filters [4] .

2.5 U-Nets

U-Net is a type of CNN designed for fast and precise segmentation of images. It has an encoder-decoder structure:

Encoder: This component captures the context in the image. It consists of a stack of convolutional layers, each followed by a max pooling layer. As we move deeper into the network, the spatial dimensions (width and height) decrease while the depth (number of feature maps) increases [5] .

Decoder: Utilizing the high-level contextual information captured by the encoder, the decoder reconstructs the input image detail. It comprises a stack of transposed convolution layers (also known as deconvolution), each followed by a concatenation with the correspondingly cropped feature map from the encoder and further convolutions. Additionally, there are skip connections between the encoder and decoder. The objective of these skip connections in the U-Net model is to help recover the fine-grained details lost during the encoding (downsampling) process [5] .

During the encoding stage, the spatial resolution of the input decreases while the semantic complexity increases. However, this process may lead to the loss of some local, detailed information. When the decoder upsamples the low-resolution encoded features, it may not fully recover the original details. Skip connections address this issue by directly forwarding the feature maps from the encoder to the corresponding decoder. This provides local, detailed information to the decoder, aiding it in better reconstructing the original input, especially in tasks like image segmentation where pixel-level detail is important [5] .

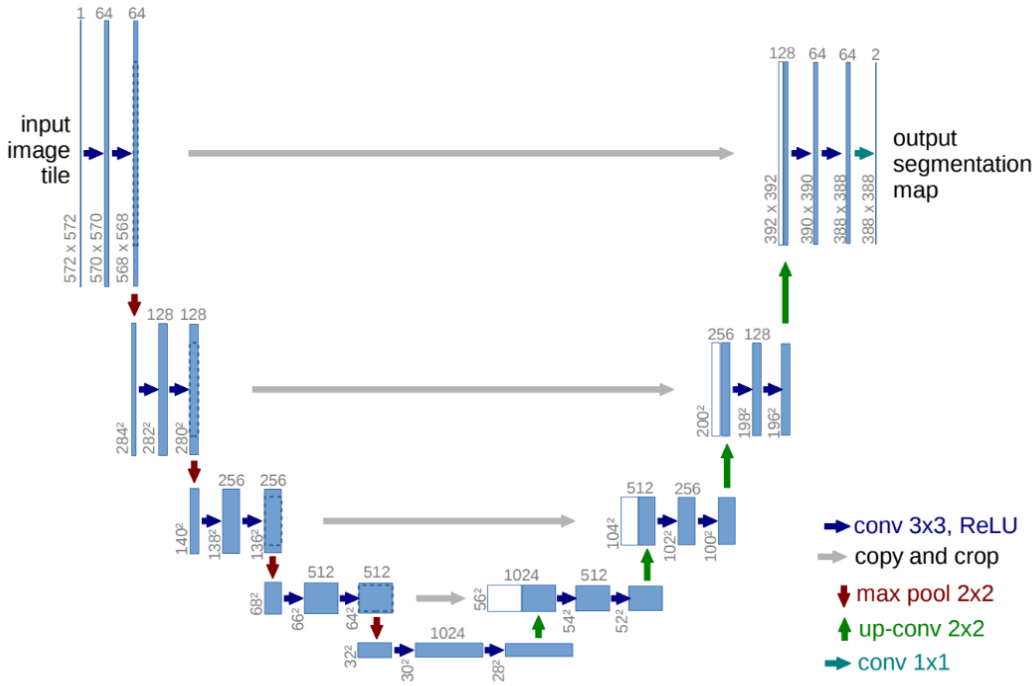


Figure 4: U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations [5] .

2.6 Graph U-Nets

Graph U-Nets represent a specialized architecture tailored for efficient and accurate graph-based hierarchical representation learning [2] . Similar to U-Nets designed for image segmentation, Graph U-Nets adopt an encoder-decoder structure to capture intricate graph structures and reconstruct the original graph detail.

Two key operations that enable the model to learn hierarchical representations of graphs and reconstruct the original graph structure are Graph Pooling (gPool) and Unpooling (gUnpool) [2] . These operations, similar to the encoder-decoder structure of U-Nets, facilitate learning hierarchical representations of graphs and reconstructing the original graph structure.

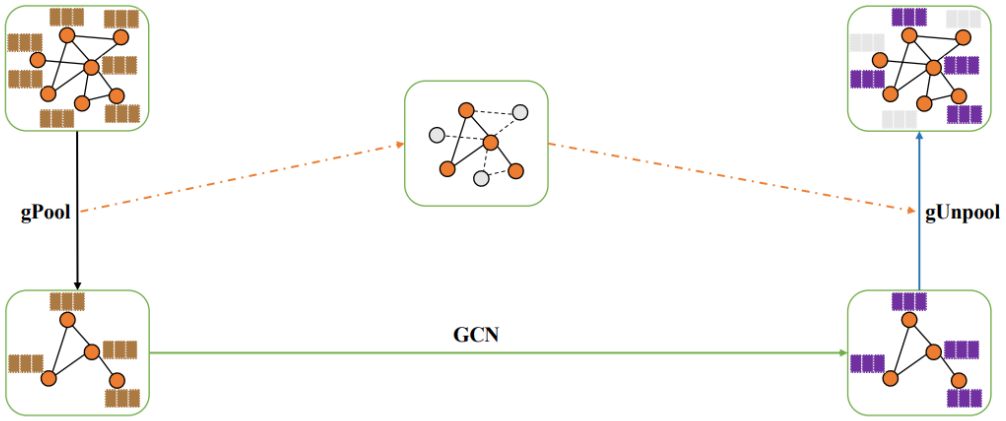


Figure 5: The architecture of gPool and gUnpool in the Graph U-net model [2] .

gPool operation is introduced for down-sampling graph data [2] . In each convolutional layer, a subset of nodes is adaptively selected to form a smaller graph. This selection process is guided by a trainable projection vector, p. By projecting all node features onto a 1D space, k-max pooling is performed for node selection. The selection of nodes is determined by the scalar projection values on p, ensuring consistency in connectivity across nodes in the new graph. Mathematically, the projection formula is represented as:

$$y_i = \frac{x_i * p}{\|p\|}$$

Where i represents a given node, x_i denotes the node feature vector, p signifies the projection vector and $\boldsymbol{y_i}$ measures how much information of node i can be retained when projected onto the direction of p. The new graph, represented by matrices A^{φ} (represents the adjacency matrix) and X^{φ} (represents the feature matrix), is obtained by selecting the k nodes with the largest scalar projection values on p.

Further, a gate operation controls the information flow by applying a sigmoid function to each element in the scalar projection vector. The element-wise matrix product of X' and the gate vector Y' governs the information of the selected nodes, resulting in the new graph with the most significant features while minimizing information loss.

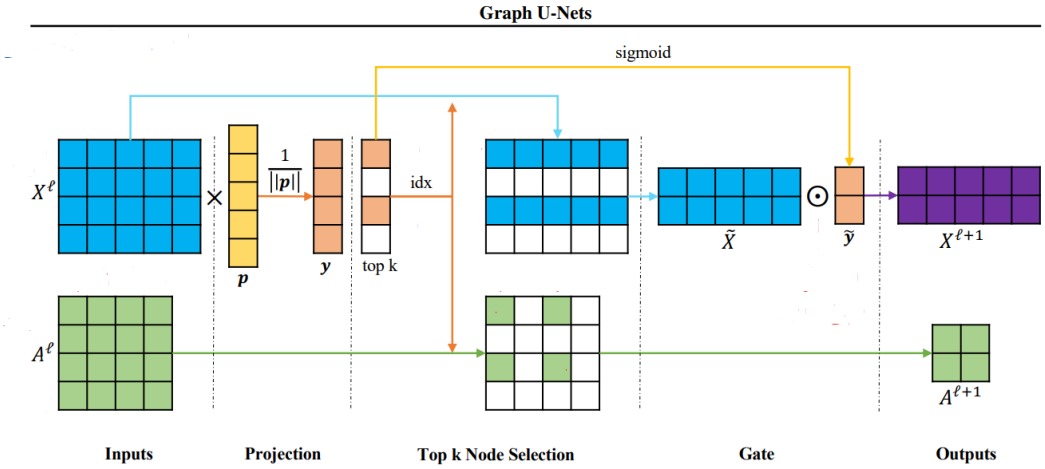


Figure 6: The proposed graph pooling layer operates on a graph with 4 nodes, each having 5 features. It starts by creating an adjacency matrix Aℓ∈R $^{4\times4}$ and an input feature matrix Xℓ∈R $^{4\times5}$ for layer ℓ. In the projection stage, a trainable projection vector p∈R5 is used to estimate scalar projection values for each node, resulting in a score vector y. The top 2 nodes (k=2) with the highest scores are selected, and their indices are used to form a new graph with a pooled feature map X~ℓ and a new adjacency matrix Aℓ+1. In the gate stage, element-wise multiplication is performed between X~ℓ and the selected node scores vector y~, resulting in Xℓ+1. The graph pooling layer outputs Aℓ+1 and Xℓ+1 [2] .

In contrast to the gPool layer, the gUnpooling layer serves to restore the graph to its original structure, enabling up-sampling operations on graph data [2] . This layer operates in tandem with the gPool layer, utilizing the information recorded during node selection in the gPool layer to reconstruct the original graph. The layer-wise propagation rule of the gUnpool is expressed as:

$$X^{l+1} = distribute(0_{NXC}, X^l, idx)$$

Where X^{l+1} denotes the feature matrix of the restored graph X^l represents the feature matrix of the current graph 0_{NXC} is the initially empty feature matrix for the new graph, with dimensions identical to the original graph but filled with zeros and idx contains the indices of the selected nodes in the corresponding gPool layer, which reduced the graph size from N nodes to K nodes.

The distribution process involves filling the initially empty feature matrix 0_{NXC} by placing the feature vectors from the current graph X^l into their corresponding positions based on the indices stored in idx. In the restored graph X^{l+1}, the row vectors corresponding to the selected nodes are updated with the corresponding row vectors from X^l, while the row vectors corresponding to nodes not selected remain zero.

The Graph U-Nets architecture begins with the application of a graph embedding layer, aiming to convert nodes into low-dimensional representations. This step is particularly crucial since some datasets utilize high-dimensional feature vectors. Following this, the encoder is constructed by stacking multiple encoding blocks, with each block comprising a GCN layer and a gPool layer.

The gPool layer plays a pivotal role in reducing the size of the graph to encode higher-order features, while the GCN layers are responsible for aggregating information from each nodes first-order neighbors. On the other hand, the decoder is designed with the

same number of decoding blocks as the encoder. Each decoding block consists of a GCN layer and a gUnpooling layer.

The gUnpooling layer serves to restore the graph to its higher-resolution structure, while the GCN layer continues to aggregate information from the neighborhood.

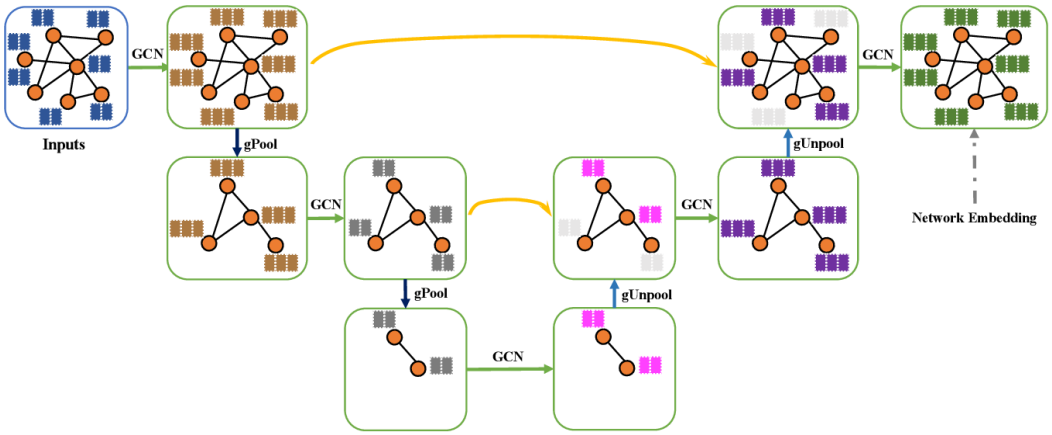


Figure 7: Architecture of G-U-net [2] .

Additionally, the architecture similar to U-nets Incorporating skip connections between the encoder and decoder, Graph U-Nets aim to mitigate information loss during the encoding process [2] . These connections facilitate the direct transmission of feature maps from the encoder to the corresponding decoder layers, providing local, detailed information essential for accurate graph reconstruction.

By preserving fine-grained details that may be lost during downsampling, skip connections enhance the decoder ability to reconstruct the original input graph, particularly in tasks requiring pixel-level accuracy. Finally, a GCN layer is employed for final predictions before applying the soft max function.

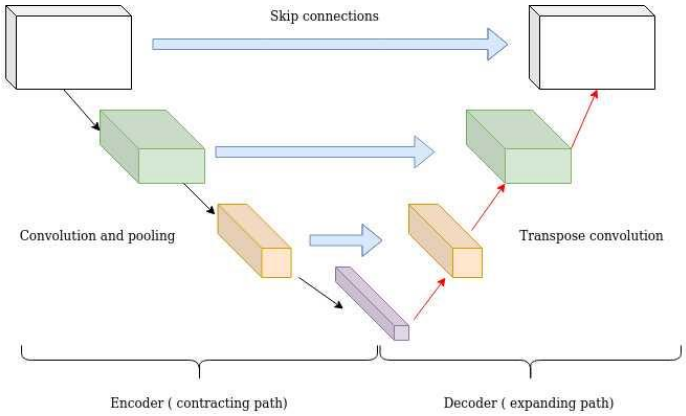


Figure 8: Macro view on the Graph U-net architecture focusing on the skip connection [11].

2.7 Generative adversarial network (GANs)

Generative Adversarial Networks (GANs) are powerful frameworks used for generating synthetic data samples. Comprising a generator responsible for creating synthetic data and a discriminator trained to differentiate between real and generated data, GANs engage in a competitive training process known as adversarial learning. This methodology has propelled their application in diverse fields such as image generation, style transfer, and data augmentation, where they exhibit exceptional performance.

The proficiency of GANs in producing realistic and diverse data has profound implications for content creation, data privacy preservation, and simulation tasks, thereby revolutionizing various industries. Additionally, GANs demonstrate a remarkable capacity to learn intricate data distributions and generate novel instances, offering significant potential for advancing machine learning tasks and addressing real-world challenges effectively. The continuous evolution and refinement of GANs underscore their pivotal role in propelling innovation and expanding the horizons of generative modeling techniques.

A GAN consists of a generator and a discriminator, where each one of the mentioned is a neural network [6] . This network is usually trained using adversarial training. The generator G creates synthetic data samples. It aims to generate data that is indistinguishable from real data by the discriminator. The generator function $G(z)$ takes random noise z as input and outputs synthetic data samples. Mathematically, the generator can be represented as: Generated Data = $G(z)$. The discriminator D is trained to differentiate between real and generated data. It provides feedback to the generator by indicating how convincing its generated samples are.

The discriminator function $D(x)$ takes data samples x as input and outputs the probability that the sample is real. Mathematically, the discriminator can be represented as: Discriminator Output = $D(x)$ Where x is a data sample, and $D(x)$ represents the probability that x is real [6] . The discriminator D aims to maximize the probability of assigning the correct label to both the real samples and the faked samples generated by the generator G, while the generator G aims to minimize the probability that the discriminator D successfully distinguishes the faked samples from the real samples. The objective of this min-max game [6] is written as:

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim P_{data}(x)}[log\, D(x)] + \mathbb{E}_{z\sim P_z(z)}[log(1 - D(G(z)))].$$

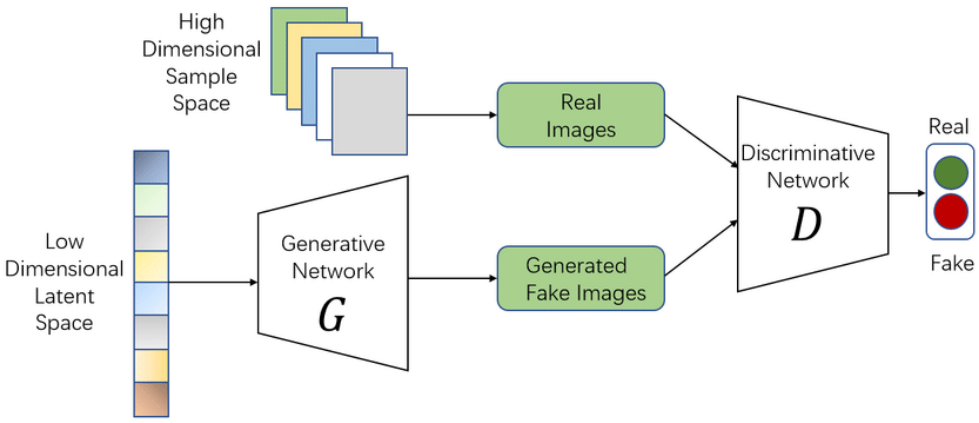


Figure 9: The architecture of a basic GAN [6] .

2.8 Misc-GAN

Misc-GAN framework comprises three stages: the Multi-Scale Graph Representation Module, the Graph Generation Module, and the Graph Reconstruction Module [1]. The Multi-Scale Graph Representation Module, a pivotal element of the Misc-GAN framework, explores hierarchical cluster-within-cluster structures to characterize input graphs. Employing methodologies like hierarchical clustering and algebraic multigrid, this module constructs coarse graphs at various granularity levels to capture the complex organization of graphs [1]. The construction of the coarser graph $G_t^{(1)}$ at the first layer is defined by the following Equation:

$$G_t^{(1)} = P^{(1)'} G_t P^{(1)}$$

Here, $P^{(1)}$ represents the coarsening operator, responsible for generating the coarse graph at the initial layer. This equation outlines the process of generating the coarse graph at the initial layer. For the recursive construction of a multi-scale hierarchy of increasingly coarser graphs at the $l-$th layer, where $l = 1, \dots, L$ the following Equation is employed:

$$G_t^{(l)} = P^{(l-1)'} \dots P^{(1)'} G_t P^{(1)} \dots P^{(l-1)}$$

These equations explain how coarse graphs are systematically generated at different granularity levels. Moving on to the Graph Generation Module, it serves as another critical component of the Misc-GAN framework. This module is tasked with generating new graphs while upholding the hierarchical structure distribution observed in the target graph. By leveraging deep models and generative adversarial networks, it acquires characteristic topological features to effectively model the complex joint probability of nodes and edges.

The objective is to produce new graphs that mirror the hierarchical structures present in the input graphs [1]. In the Graph Reconstruction Module, which constitutes the final stage of the framework, the graph is reconstructed while preserving significant local structures captured in the Multi-Scale Graph Representation Module. The reconstruction of the fine graph $\tilde{G}_t^{(1)}$ at the first layer is accomplished through the following Equation:

$$\tilde{G}_t^{(1)} = R^{(1)'} G_t^{(2)} R^{(1)}$$

Here, $R^{(1)}$ represents the reconstruction operator responsible for mapping the coarser graph back to the fine graph at the initial layer. Similarly, the recursive construction of a multi-scale hierarchy of increasingly refined graphs at the l -th layer is delineated by the following Equation:

$$\tilde{G}_t^{(l)} = R^{(1)'} \dots R^{(l-1)'} G_t^{(l)} R^{(l-1)} \dots R^{(1)}$$

These modules collectively empower the Misc-GAN framework to model the underlying distribution of graph structures at varying granularity levels and generate new graphs that faithfully maintain these structures.

2.9 Diffusion models

Diffusion probabilistic models, also known as score-based generative models, are a subset of latent variable generative models extensively employed in the realm of machine learning [8] . These models leverage a unique methodology involving the introduction of noise to the original training data, a process named the diffusion process. Subsequently, the reverse diffusion process is harnessed to reconstruct the initial data from its perturbed state.

By mastering this reverse process, these models acquire the ability to generate novel data points. This iterative learning mechanism empowers the model to effectively eliminate noise, unveiling the underlying structure of a dataset by simulating the diffusion of data points across their latent space [8] . With broad applicability, diffusion probabilistic models serve diverse purposes including image denoising, inpainting, super-resolution, and image synthesis tasks.

Rooted in stochastic processes, this modeling paradigm integrates randomness through noise scheduling to enhance the fidelity of the generated data and capture intricate patterns within the dataset.

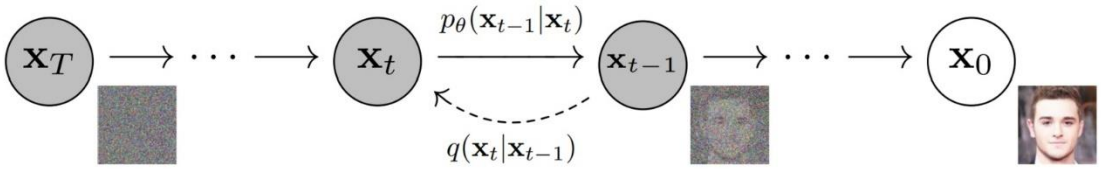


Figure 10: The main idea of diffusion model. By adding noise in the diffusion process the image became more cleaner and by imply reverse diffusion process the image return to the origin [10].

2.9.1 Diffusion models for graph generation

Diffusion models for graph generation represent a class of generative models designed to comprehend the underlying distribution of a given set of graphs and subsequently generate new graphs that adhere to the same distribution [8] . At their core, these models leverage the concept of the diffusion process.

A diffusion process in graph generation is a methodological framework where noise, in the form of nodes and edges, is systematically introduced in a controlled and incremental manner during a forward process. This process starts with a basic graph, such as an empty one or a graph with a single node, representing the initial simplicity. The controlled addition of noise at each step mimics the stochastic nature of diffusion, gradually transforming the graph into a more complex structure.

This intentional introduction of noise reflects the essence of a diffusion process, capturing the step-by-step evolution of the graph through a forward process. The diffusion process has a reverse diffusion process aims to reverse this process and generate the original data [8] . The reverse diffusion process involves training a denoising network to recursively remove the noise that has been previously added by the forward process.

Instead of removing all noise in a single timestep, a denoising network is trained to iteratively remove the noise between two consecutive timesteps. This process moves backwards on the multi-step chain as the timestep decreases from T (the total number of timesteps) to 0.

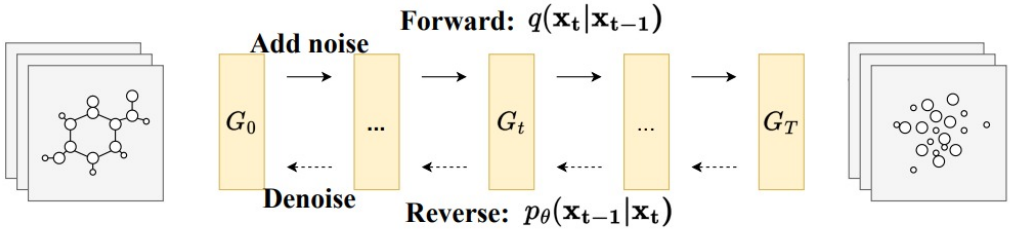


Figure 11: Diffusion model for graph generation [8] .

2.9.2 Autoregressive diffusion model for graph generation

An autoregressive diffusion model for graph generation, or in short GraphARM, operates directly in the discrete graph space and generates graphs sequentially by progressively adding nodes and their connections. In the forward diffusion process, nodes are absorbed or masked one by one along with their edges until the graph is completely simplified. The reverse process then reconstructs the graph in the opposite order using a denoising network that predicts the node type and its edges based on the already reconstructed nodes. This method benefits from learning a data-dependent node ordering that captures the graph structural regularities, leading to more efficient and accurate graph generation compared to traditional diffusion-based models that often struggle with efficiency and incorporating constraints. The autoregressive approach allows the model to easily integrate constraints during generation, providing greater flexibility and control over the structure of the generated graph.

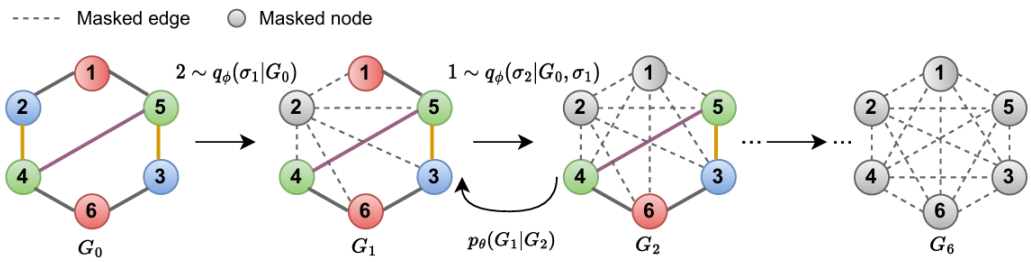


Figure 12: GraphArm forward (absorbing) diffusion chain. In the forward diffusion chain, nodes are absorbed one by one with masked features and edges, guided by the diffusion ordering network's absorption probabilities [12].

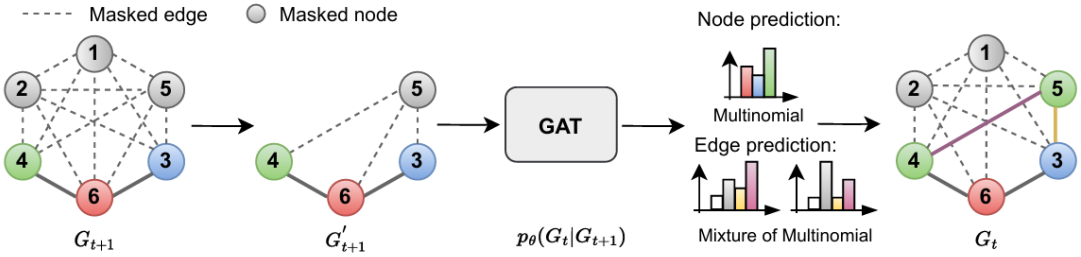


Figure 13: GraphArm reverse diffusion chain**.** In the reverse diffusion chain, the model sequentially samples nodes based on their absorption probabilities and features from the forward diffusion chain [12].

3. Process

The project model is based on the Misc-GAN framework architecture [1]. Originally, Misc-GAN comprises three components: the Multi-Scale Graph Representation Module, the Graph Generation Module, and the Graph Reconstruction Module.

The first and third component are based on the Graph U-NET [2] . The second component will be implemented using GraphARM [12]. The model input is a graph with hierarchical structures, after pre-processing operations in the Multi-Scale Graph Representation Module the model coarsens the input into representation of different granularity levels, then for each level of granularity in the Graph Generation Module the model generates a corresponding coarsen graph.

Then the Graph Reconstruction Module reconstructs the generated graphs into an output target graph that retains the same composition of the original input graph. The following sections detail the adaptations made for integration into this model.

3.1 Pre-processing

The data gathered from various sources may not be sufficiently clean, necessitating the need for a pre-processing step to obtain the most relevant data possible. The model consistent pre-processing of noise removal, handle missing values outliers and inconsistencies, and normalize features.

The noise removal involves filtering out edges or nodes that do not contribute meaningfully to the overall structure. Handle missing values outliers and inconsistencies can also be achieved by removing unwanted data. Normalize features ensures that all features contribute equally to the analysis and prevents features with larger scales from dominating the learning process.

3.2 Multi-Scale Graph Representation Module

The Graph Representation Module is employed by a graph U-net model [1] that start after dataset partitioning, aiming to refine the input graphs by the encoder process through coarsening stages. This process adapts dynamically to varying graph complexities, facilitating the extraction of hierarchical features. Ultimately, the output of the encoder process is a presentation of the input graph coarsen by various granularity levels this output act as the input to the graph generation module.

3.2.1 Graph U-net input

The data obtained after the pre-processing step is divided into three distinct parts: Training, Validation, and Testing. Each part of the dataset comprises of graphs characterized by hierarchical structures. Every graph is outlined by two matrices: the adjacency matrix $A^{\varphi} \in \mathbb{R}^{NXN}$ and the feature matrix $X^{\varphi} \in \mathbb{R}^{NXC}$.

These matrices serve as the input for the Graph U-net incorporated into the model architecture. The Graph U-net is fed with two portions of the dataset: Training and Validation, while the Testing segment is reserved for subsequent analysis.

3.2.2 The Graph U-net process

Upon completion of the pre-processing steps and subsequent partitioning of the dataset into Training, Validation, and Testing sets, the Graph U-net process begins [2] . The model accepts a graph with hierarchical structures as its input, where individual nodes signify specific entities, and edges outline relationships between said entities. The complexity of these graphs may vary, necessitating adaptive processing by the model. The initial phase of the Graph U-net encoder process entails the coarsening of the input graph into representations of varying granularity levels. It achieved by the gPool operation.

This coarsening procedure aims to streamline the graph while retaining its fundamental structural attributes. It enables the model to function across multiple abstraction levels, thereby facilitating the extraction of hierarchical features. At each granularity level, the Graph U-net generates a corresponding coarsened graph, contributing to holding the key features of the input graph without damaging or losing crucial data.

3.3 Graph Generation Module

The next step in the process is to transfer the coarsened graph representations from the Graph U-net output to the Graph Generation Module [1]. The Module consist of 3 steps where at the last step GraphARM is applied.

This module facilitates the generation of target graphs from each coarsened graph at different granularity levels. As a result, the module generates various target graphs which are the base of the input of the Graph Reconstruction Module.

3.3.1 Generate Coarsen Graphs

The Graph Generation Module applies three steps for each coarsened graph representation. "The first step involves partitioning the graph into multiple non-overlapping subgraphs using state-of-the-art graph clustering methods. Then, based on the detected communities, it generates a set of block diagonal matrices by shuffling community blocks over the diagonals, which are used to characterize the community-level graph structures" [1].

In the last step GraphARM is applied to generate the target graphs. This model adds noise in a controlled manner to the coarsened graphs, transforming them into a simple noise distribution through a series of steps. It then applies a reverse diffusion process, starting from the simple noise distribution and gradually removing the added noise. This process uses a learned distribution to guide the noise removal, resulting in graphs that closely resemble the original input graphs. The new generated graphs now pass to the Graph Reconstruction Module.

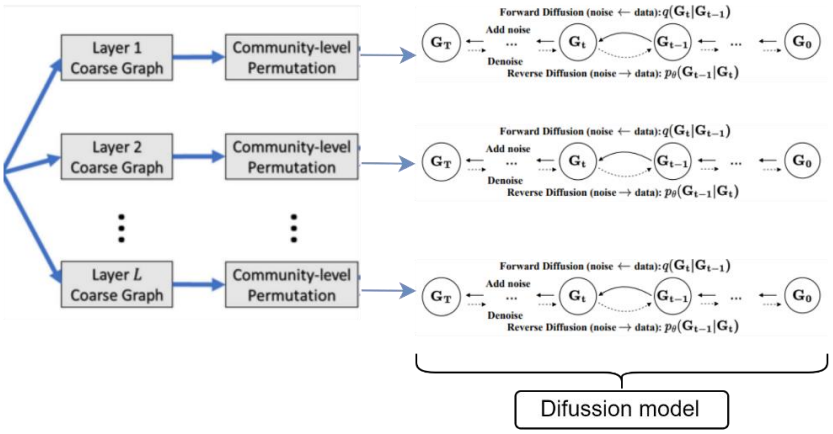


Figure 14: The 3 steps of the Graph Generation Module for each layer.

3.4 Graph Reconstruction Module

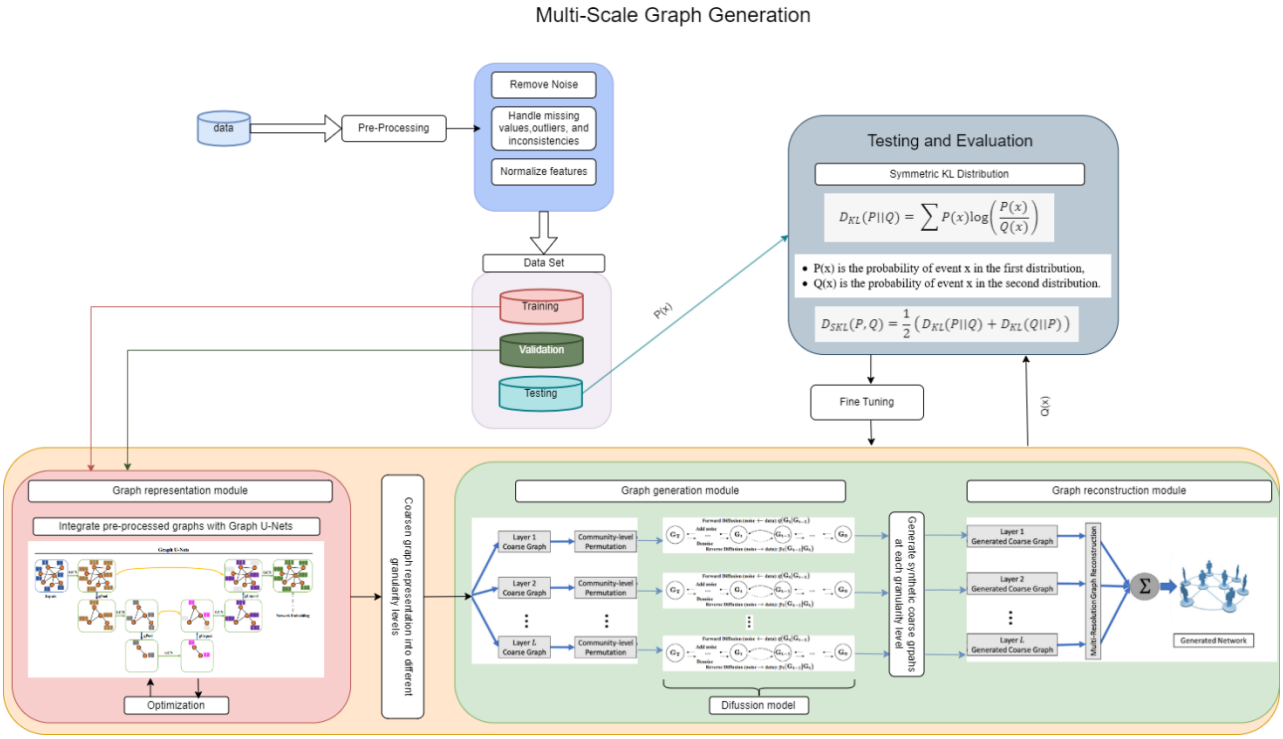
The Graph Reconstruction Module is employed by a GU-net model [[1],[2] The process starts after the coarsened graphs generation operation in the generation module. The primary objective of the Graph Reconstruction Module is to reconstruct a graph close to the original input graph from its coarsened representations, effectively reversing the coarsening operations performed during the encoding phase. This is achieved through a combination of two key operations: gUnpool and skip connections.

The gUnpool operation is responsible for repositioning nodes within the graph structure, leveraging the positional information accumulated during the corresponding gPool layer in the encoding phase. By tracking the locations of nodes selected for pooling, the gUnpool layer can effectively recover these nodes back to their original positions within the graph, preserving the overall structure and topology.

Complementing the gUnpool operation, skip connections play a vital role in retaining valuable features from the original input graph throughout the reconstruction process. These connections facilitate the direct transfer of information from the encoding phase to the decoding phase, bypassing the coarsening operations. The layer-wise propagation rule of gUnpool layers is defined in the previous section. This process specifies the precise manner in which nodes are repositioned within the graph structure, ensuring an accurate and consistent reconstruction process across multiple layers of the GU-Net model.

4. Product

The product is an interface that will implement the research model, using GU-net and diffusion model for the generation of multi-scale graphs. Below is the flow chart of the research model.



5. The Solution

The solution builds upon three IPYNB files that run in Google Colab notebooks, each serving a specific function:

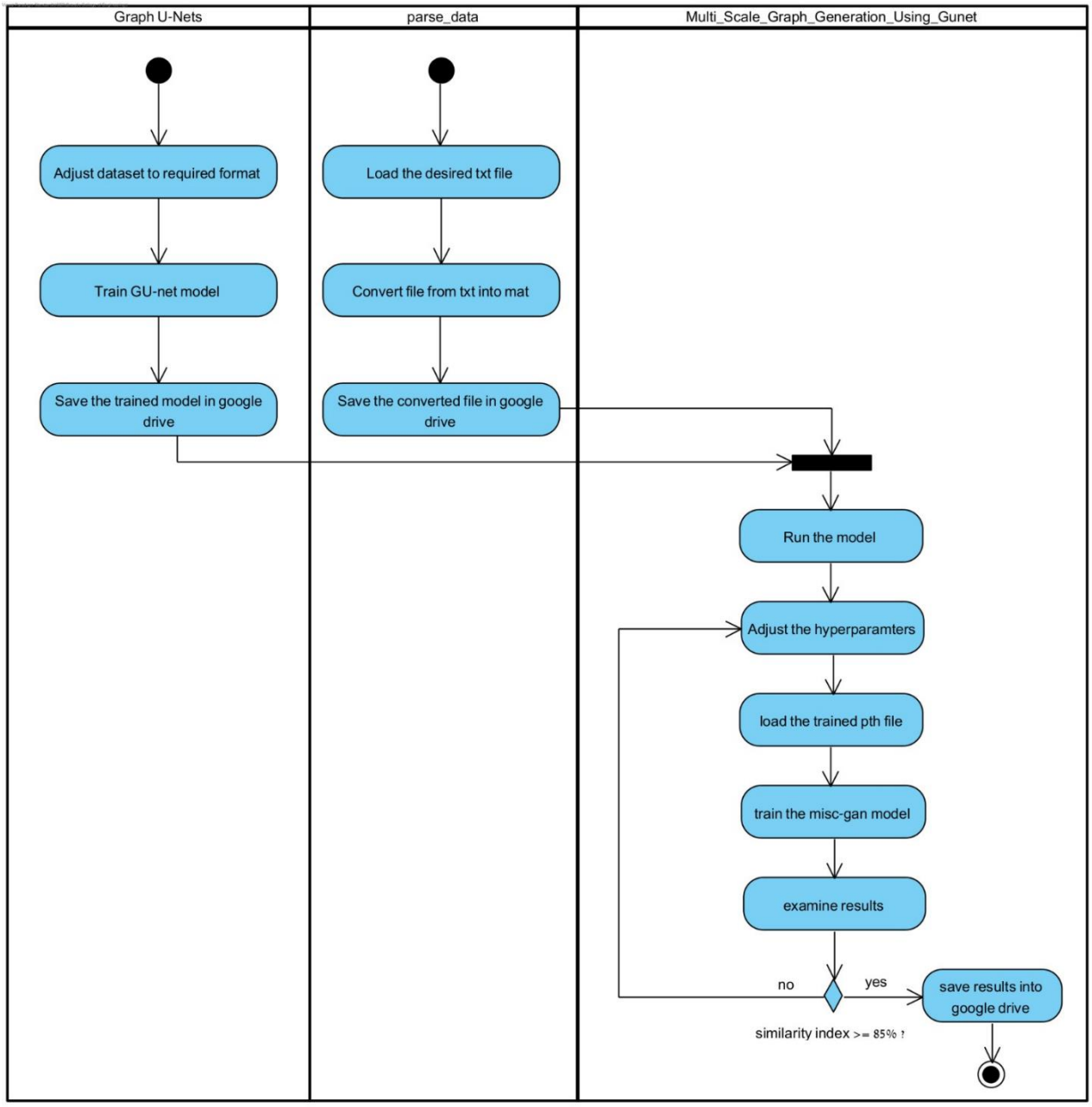
Graph U-Nets: This notebook implements the training process of the GU-net used in the MISC-GAN architecture, producing a PTH file for the user. The training is conducted on a dataset that must be pre-saved in a designated folder within the user's Google Drive account. This notebook manages the setup, configuration, and execution of the training procedure, ensuring the effective training of the GU-net model on the provided dataset.

parse_data: This notebook converts a given text file dataset into Matlab format (.mat) for later use. The dataset must also be pre-saved in a designated folder within the user's Google Drive account. This notebook reads the text file, processes the data, and outputs a .mat file, making the data compatible with subsequent workflow steps. The file performs necessary data transformations and ensures that the dataset adheres to the required format, facilitating its use in the subsequent stages of the project.

Multi_Scale_Graph_Generation_Using_Gunet: This notebook integrates the GU-net into the MISC-GAN architecture, replacing CycleGAN with GraphARM. It uses the files created by the previous two notebooks (.mat and .pth). Thus, it is crucial for these files to be saved in specific folders within the user's Google Drive account. This notebook represents the culmination of the research project and includes practical and numerical results. It executes the end-to-end graph generation process, evaluates the generated graphs, and compares them with the original dataset to measure performance and accuracy.

All output files required for operating the codes are saved in the user's private Google Drive. This ensures that the generated models and transformed datasets are easily accessible for future use and further analysis.

Below is the activity diagram of our solution:



5.1 The Development Process

The development process begins by examining the existing implementations of the GU-net model and the MISC-GAN architecture as proposed by the authors of the respective articles. The primary objective of this examination is to understand the input and output objects of the programs and to identify the requirements for their proper functioning.

Upon achieving a comprehensive understanding of the models' implementations, the work is divided into three parts. First, the GU-net model is trained on a suitable dataset. This involves adjusting the dataset according to the format specified by the original authors of the GU-net code. The input dataset for the Graph U-Nets model follows a structured format: The first line specifies "N", the number of graphs. Each subsequent block describes an individual graph, starting with a line containing "n" (the number of nodes) and "l" (the graph label). Each of the next "n" lines provides information for a node, beginning with "t" (the node's tag) and "m" (the number of neighbors), followed by "m" indices representing the neighbors. If any continuous node features (attributes) are present, they follow the neighbor indices. This structure ensures that the dataset encapsulates all necessary graph details in a consistent and readable format. For the second part, the GU-net's gPool and gUnpool operations are integrated into the MISC-GAN code, replacing the original steps as outlined in the first part of the project (as mentioned above).

Following the successful integration and training of the GU-net model, a suitable diffusion model for graph generation is sought. The chosen model must be simple, easy to implement, effective, quick, and suitable for working with graphs. The Autoregressive diffusion model for graph generation is found to meet these requirements, and it replaces the CycleGAN in the MISC-GAN architecture.

The final part involves training the MISC-GAN model. The original code requires training with a dataset in Matlab format. To accomplish this, the original code from the article is converted from Matlab to Python. Once the dataset is in the correct format, the MISC-GAN model is trained, and the results are examined.

5.2 Constraints and Problems

During the development of the model, we encountered several challenges. Integrating pre-existing code written by others required a deep understanding, which demanded significant time and effort. By meticulously examining each implementation, we were able to comprehend and utilize the models as intended.

A major challenge was identifying a suitable diffusion model to replace CycleGAN and implementing it effectively. Given the large variety of diffusion models for graph generation, we had to evaluate numerous options. After brief research, we identified the autoregressive diffusion model for graph generation, which ultimately proved to be the perfect fit. After considerable effort, we successfully integrated it into our model.

Working with datasets that were not initially designed for our model required adjustments in both the code and the datasets to ensure proper functionality. Additionally, using Google Colab posed challenges due to limitations in GPU and RAM memory. The training and testing of our models demanded extensive computational memory, and the free version of Colab offered limited resources. Consequently, we upgraded to a paid account, which also quickly exhausted its running units.

Ultimately, the most significant challenge was integrating all the different components into one functioning model, which did not naturally cooperate. This integration process involved ensuring compatibility between various modules, synchronizing data formats, and resolving conflicts arising from different codebases. Each component had its own set of dependencies and requirements, making the integration process complex and time-consuming.

5.3 Testing And Evaluation

Below are the tests subjects of our model, the expected results and the actual results:

Test	Test Subject	Expected Result	Actual results
1	Graph Generation	The model will generate multi-scale graphs accurately based on the input.	The model generated multi-scale graph that only resembles the input graph.
2	Graph Processing	The model will correctly process the generated graphs, capturing hierarchical relationships effectively.	The model did capture the hierarchical relationships between the graph layers and process them correctly.
3	Graph Output Validation	The model will produce valid output graphs that accurately represent the hierarchical relationships in the data.	The model produces valid output plots showing the hierarchical structure and relationships in each graph layer
4	Runtime Efficiency	The model will perform the graph generation and processing within an acceptable time frame.	The model performing in range of 1.5 – 2 Hrs. per run.
5	Input Size/Graph Size	The model will be able to handle larger graphs without a significant decrease in performance.	The model can handle Graph in any size
6	Empty Graphs	If the input graph is empty, the model will return a specific error message indicating that the input graph cannot be empty. The model will not proceed with further processing until a valid graph is provided.	The model can handle empty graph inputs. If an empty graph is inserted it will return an error "Input graph is empty or not a valid dictionary of numpy arrays"
7	Incomplete Graphs	If the input graph is incomplete (missing nodes or edges), the model will either return an error message or fill in the missing information based on predefined rules or assumptions.	If the input is incomplete the model will fill the missing information by padding the objects to align together.
8	Incorrect Graphs	If the input graph is incorrect (contains invalid nodes or edges), the model will return an error message indicating the nature of the error. The model will not proceed with further processing until a valid graph is provided.	If the input graph is incorrect, the model will not procced with the generation and will ask for the user to correct the input graph.
9	Incompatible Graphs	If the input graph is incompatible with the model (for example, the graph is too large or the structure is not supported), the model will return an error message indicating the incompatibility. The model will not proceed with further processing until a compatible graph is provided.	In the process of the generation any type of object (tensor, dictionary, graph, etc.) will be padded beforehand Making the model dynamic for any type of graphs.
10	Generation Similarity Index	Similarity Index > 85%.	The similarity Index is: Similarity Index < 20%

Our model is evaluated by creating four output plots: the first plot depicts the original graph, the second plot illustrates the generated graph without edges, the third plot shows the original graph with edges, and the final plot displays the generated graph with edges. These four plots are generated for each layer in the process of creating the output graph.

During the plotting process, the similarity percentage between the original and generated graphs is measured. To strengthen the research, a series of calculations is conducted on two other models, GEA and Music-GAN, in addition to our model. The calculations include degree, largest connected component (LCC), wedge count, claw count, triangle count, power-law alpha, Gini coefficient, edge distribution entropy, assortative coefficient, connected components, and cluster properties. Each calculation for every model is then compared to the original graph. The comparisons are conducted using KL-divergence between the original graph and the generated graphs from our model and the two other models.

Different approaches are tested by varying the hyperparameters input into our model. The number of layers, epochs, batch sizes, matrix dimensions, and more are adjusted in an attempt to identify the optimal hyperparameters for our model.

6. How To Run The Program

Copy of our dataset files and trained files from previous executions can be found in the project GitHub repo.

To run the model, first you have to upload the desired dataset for the GU-net training and another dataset for misc-GAN to convert into .mat file into your google drive account. Both datasets need to be in .txt format.
The dataset for GU-net you simply need to save in a new folder called: " Misc-Gan_Project", and for the dataset of misc-GAN you need to save the txt file on the path: myDrive/Misc-Gan_Project/data.

Run the two notebooks below:

Graph U-Nets –

https://colab.research.google.com/drive/1OWNrxSuFkL0iUCv0T-kTbiWpKvZO8EAx?usp=sharing

parse-data –

https://colab.research.google.com/drive/1Hztresdr-ft4hr_iUiovaG20WS-GR0sP?usp=sharing

The .pth file of the trained GU-net model you save in Misc-Gan_Project folder.
And the .mat file for the training of misc-GAN you save at: myDrive/Misc-Gan_Project/data.

Run the main model in the notebook below:

https://colab.research.google.com/drive/1wSMw9jA8D4-jkE210aiQOCQX3Nu5-B1v?usp=sharing

6.1 Maintenance

In order to maintain the project and run it in the future under different dataset or configuration, there are few hyperparameters that may need to change

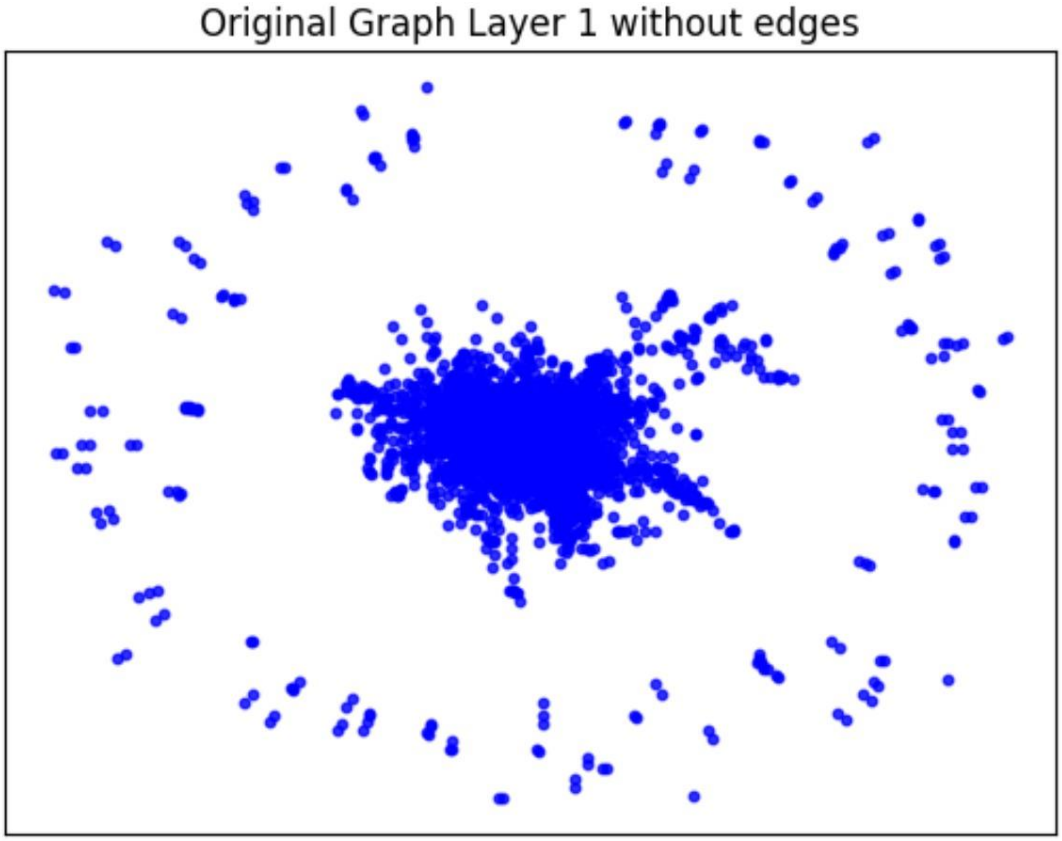
- self.dataset_A - The dataset for misc-GAN (. mat format).
- self.checkpoint – The destination path for saving checkpoints in the training process.
- self.filename - The name of the output network.
- self.output_dir - The destination path for the output file.
- self.epoch - Number of epochs (Recommended at least 50)
- self.layer – Number of granularity layers (Depend on the dataset)
- self.clusters - Number of clusters (2 or 3)
- self.starting_layer - The starting layer

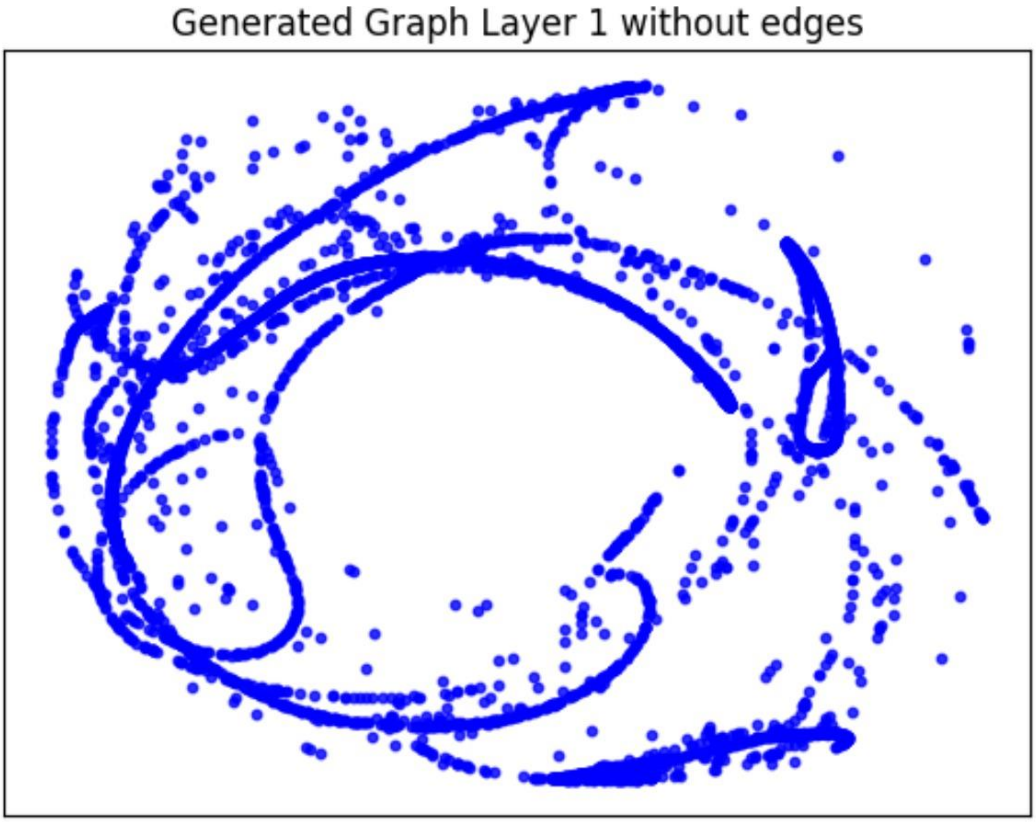
The rest of the maintenance should be straightforward.

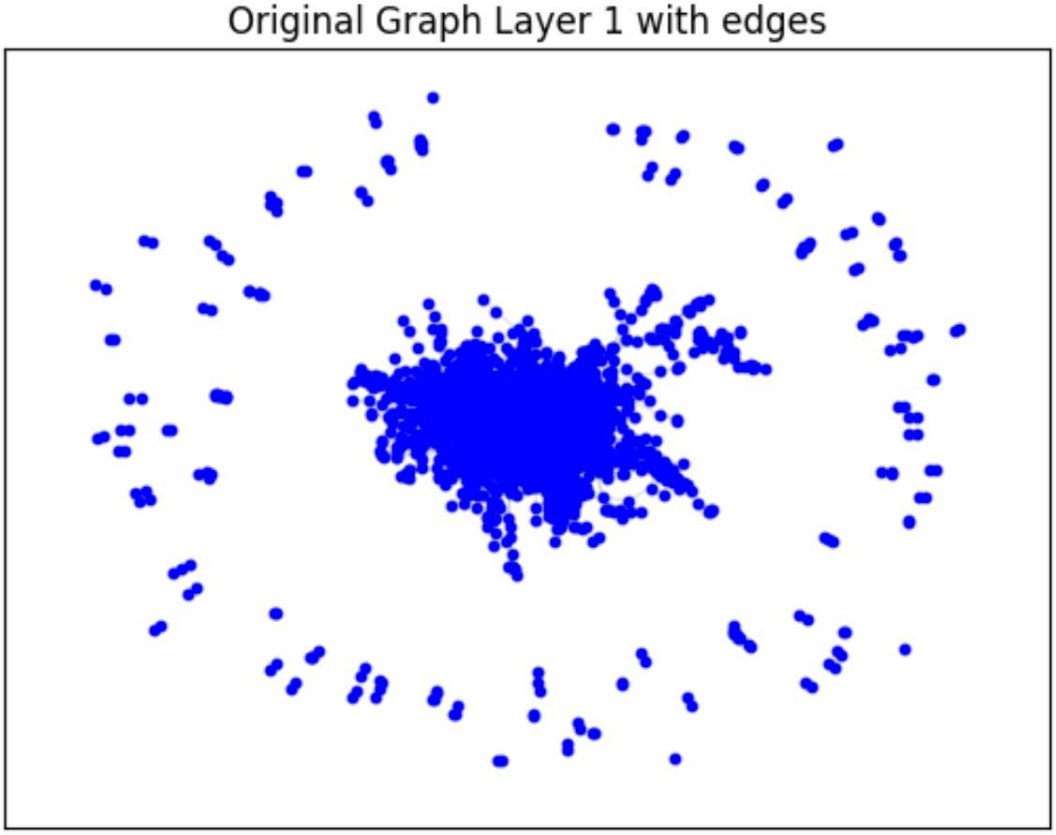
7. Results And Conclusion

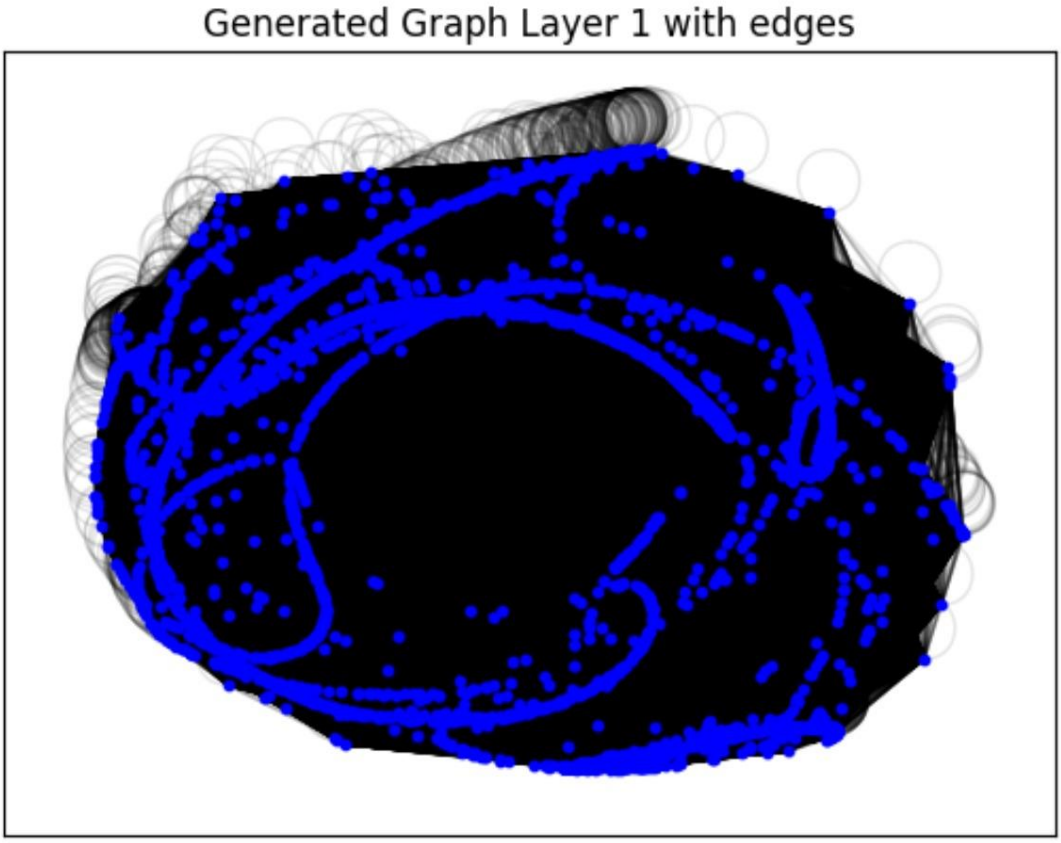
Below are the final results, after running the model on 5 layers and 100 epochs:

Layer 1:

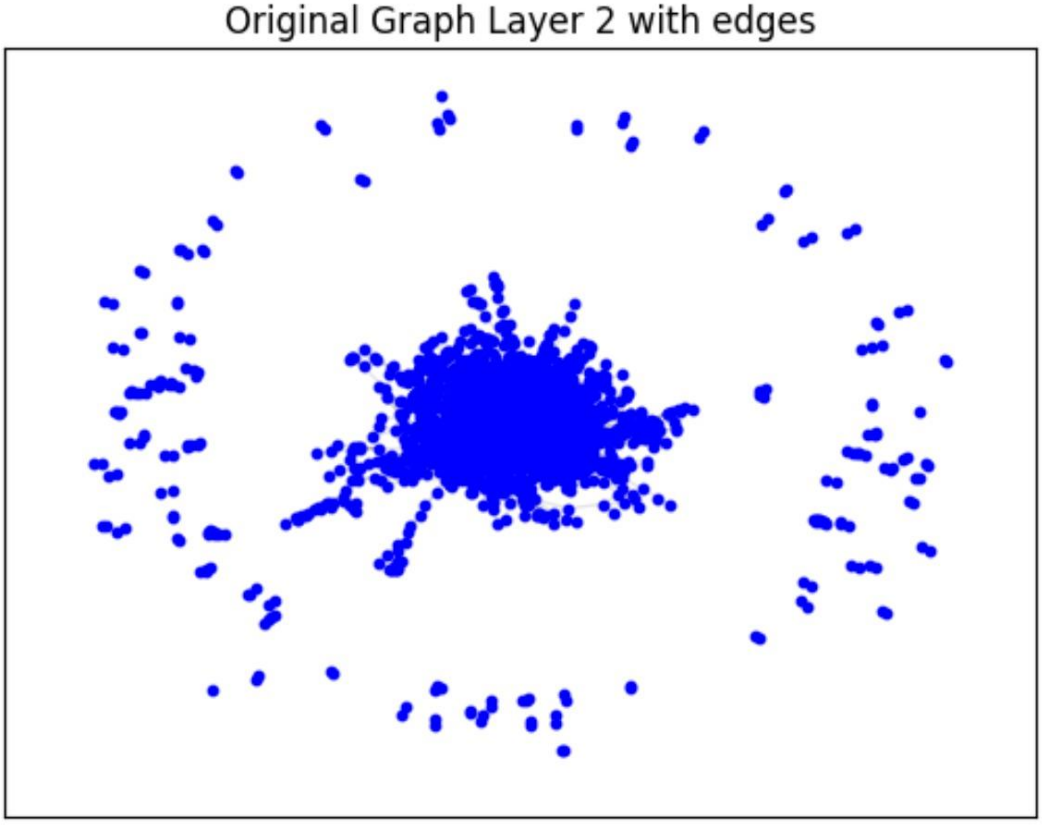



Generated Graph Layer 1 without edges

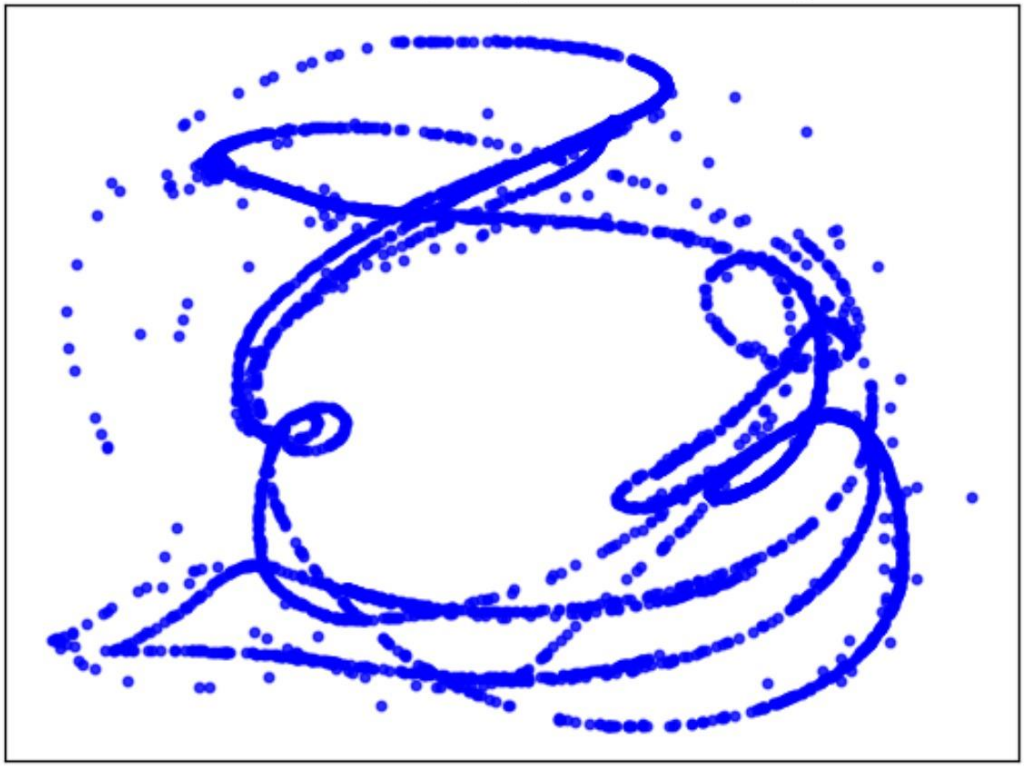

Original Graph Layer 1 with edges


Generated Graph Layer 1 with edges

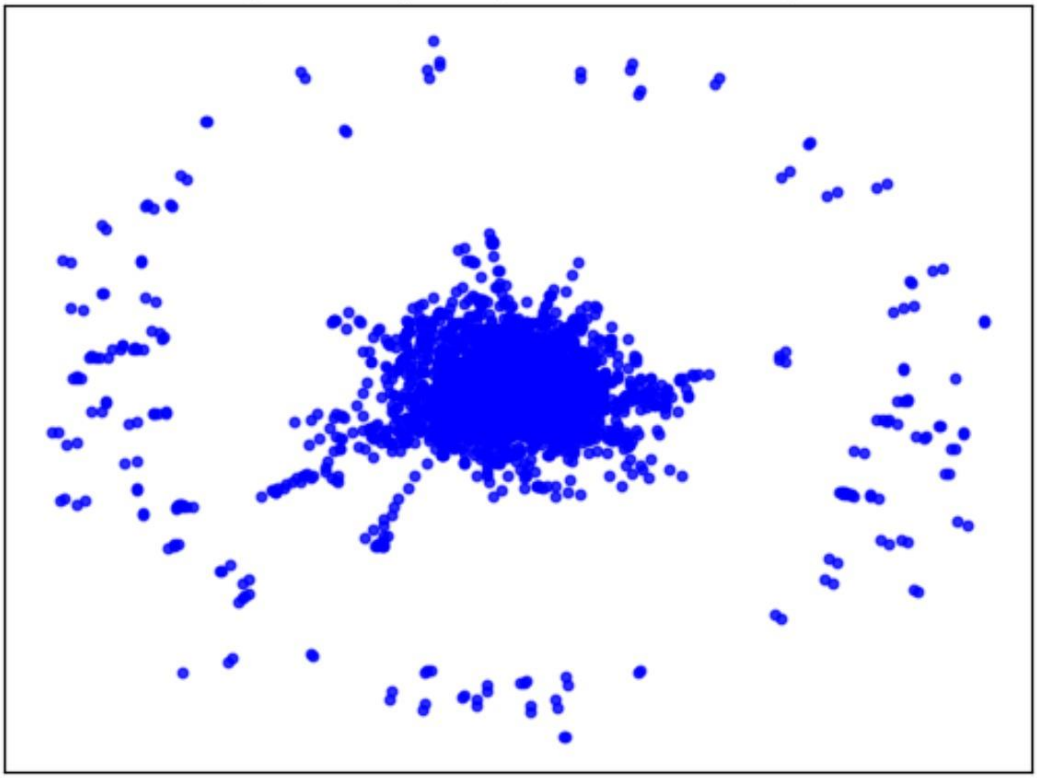
Layer 2:

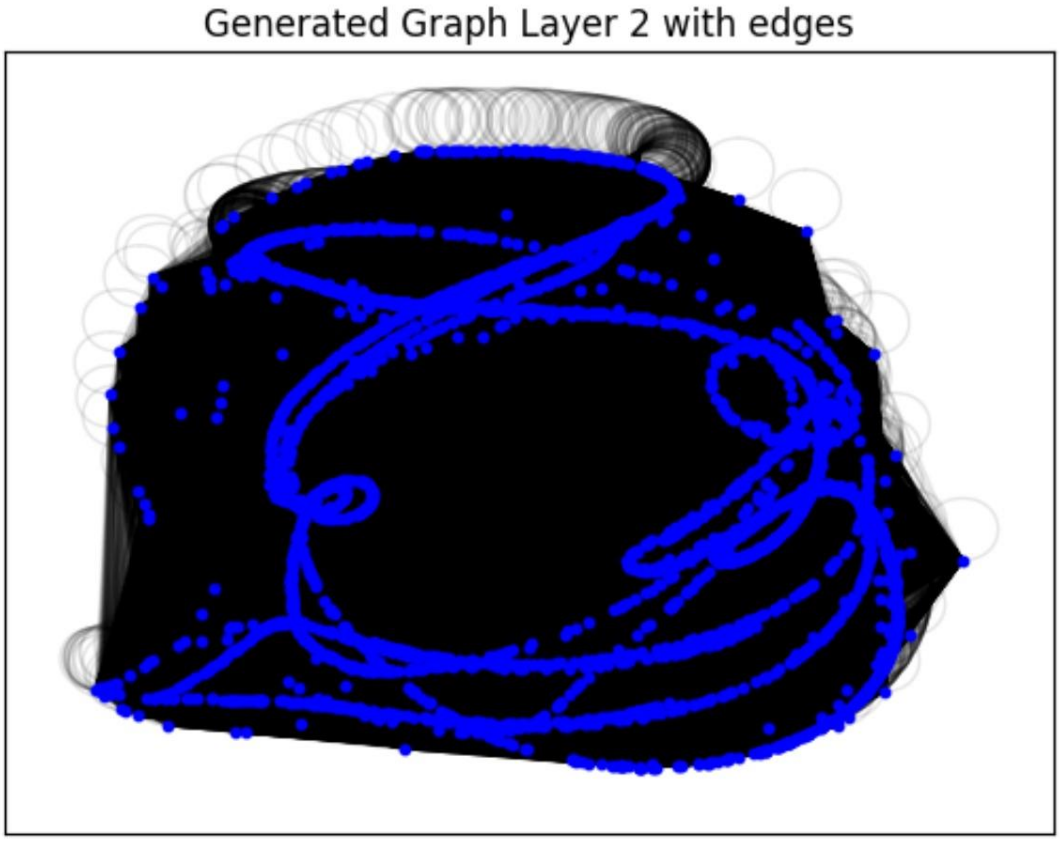

Original Graph Layer 2 with edges

Generated Graph Layer 2 without edges

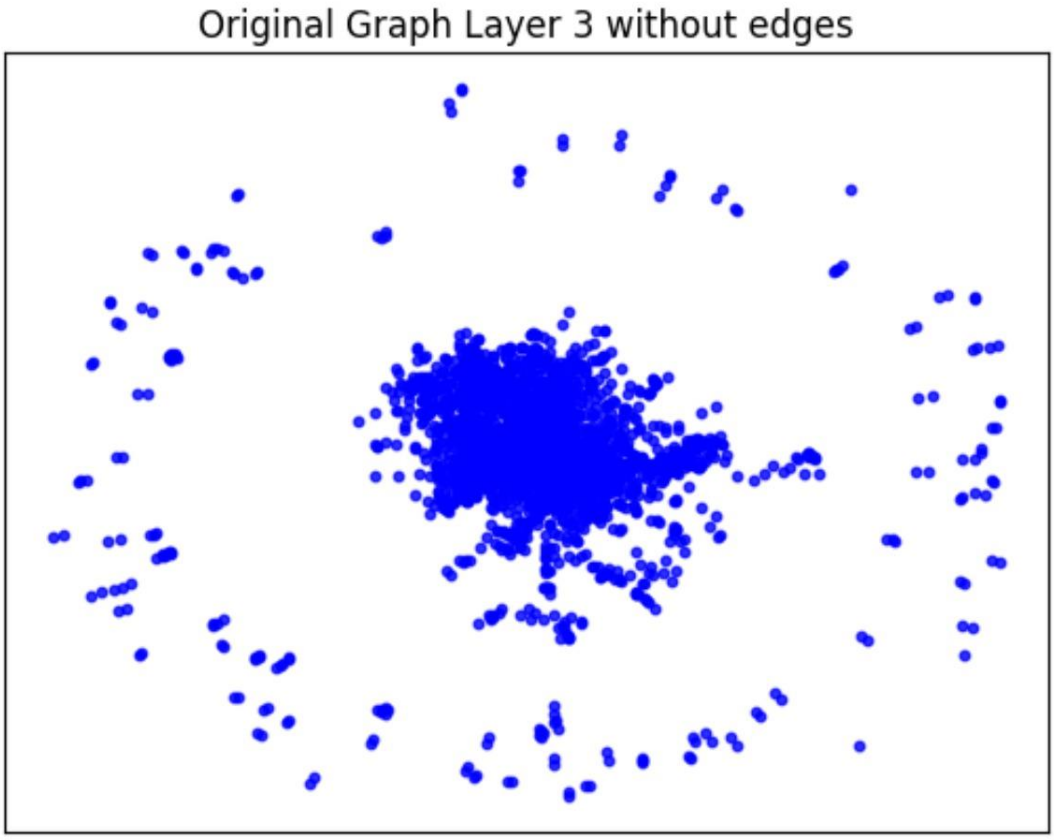


Original Graph Layer 2 without edges

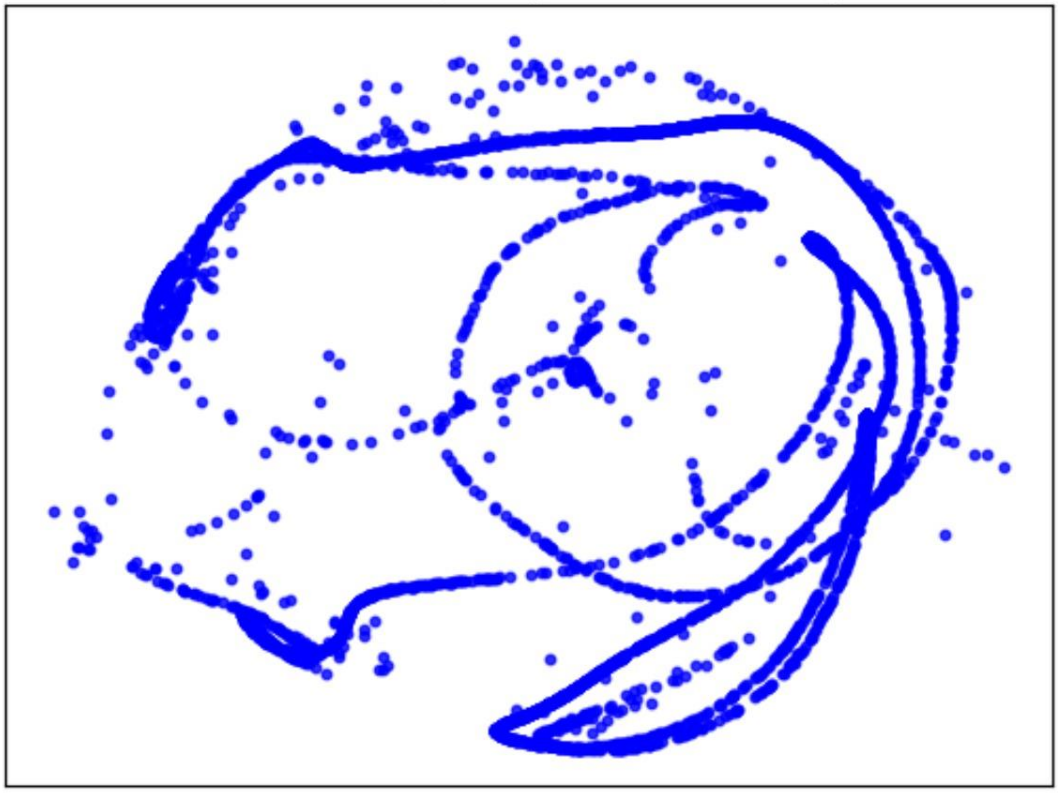



Generated Graph Layer 2 with edges

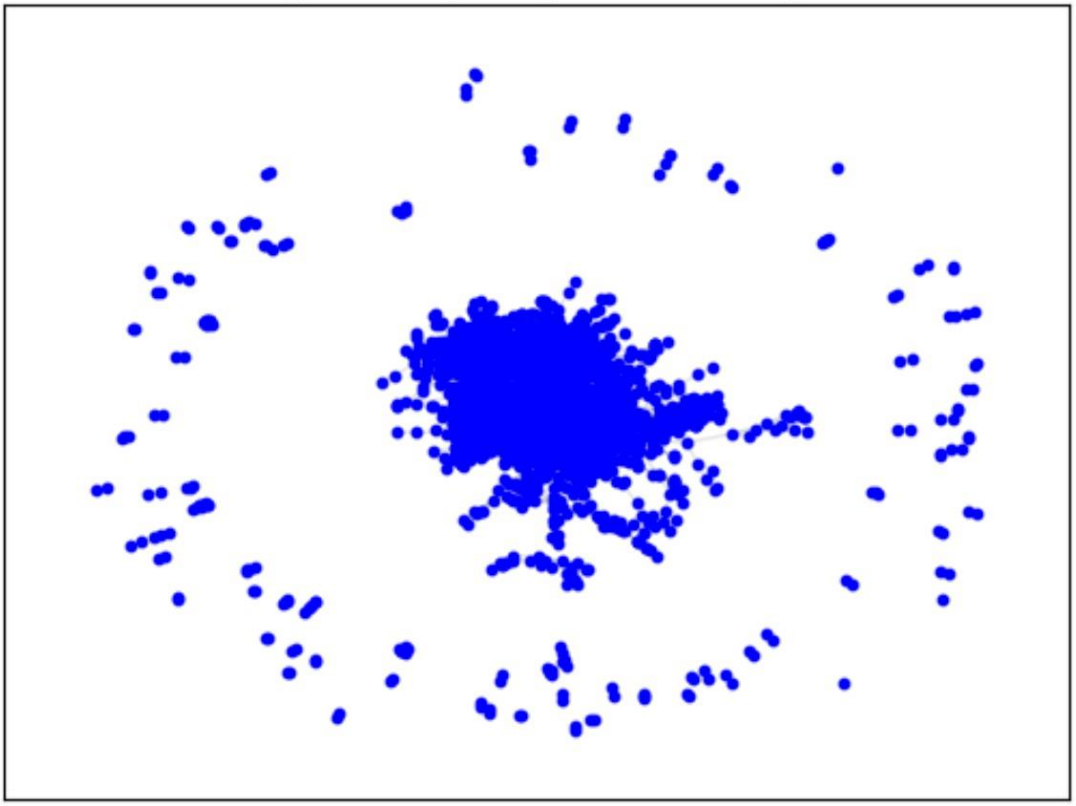
Layer 3:

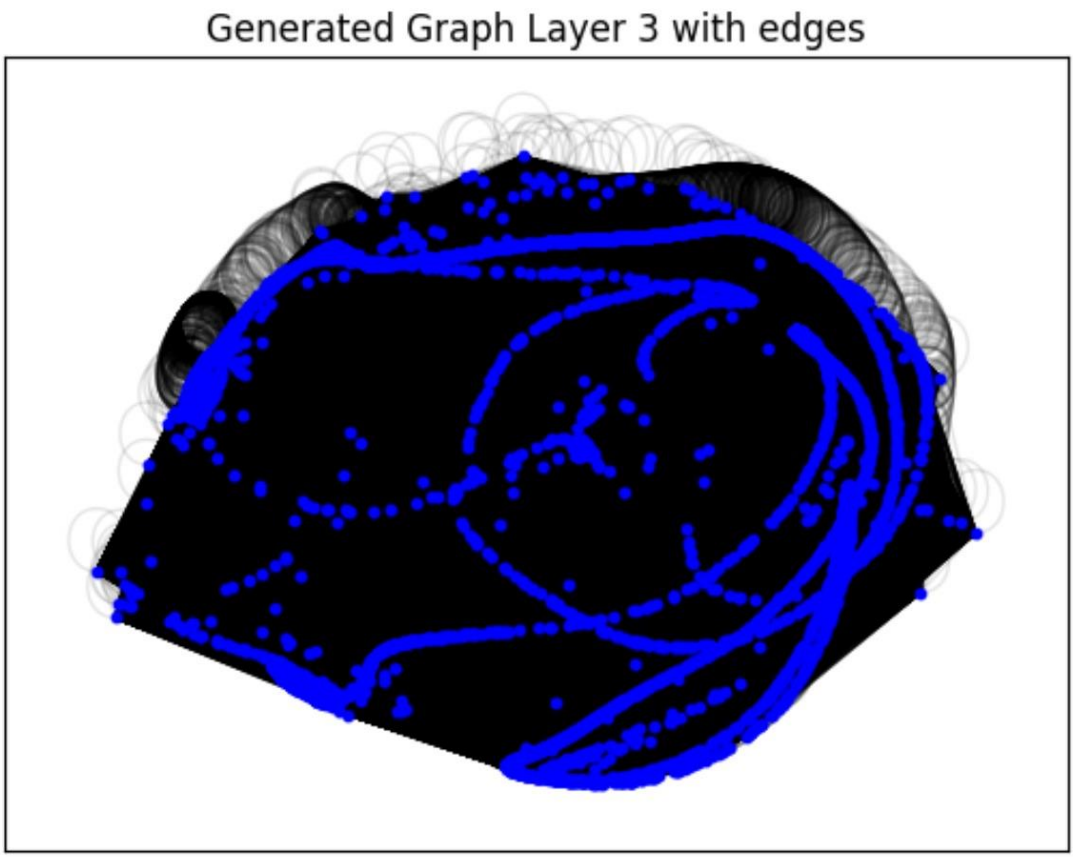

Original Graph Layer 3 without edges

Generated Graph Layer 3 without edges

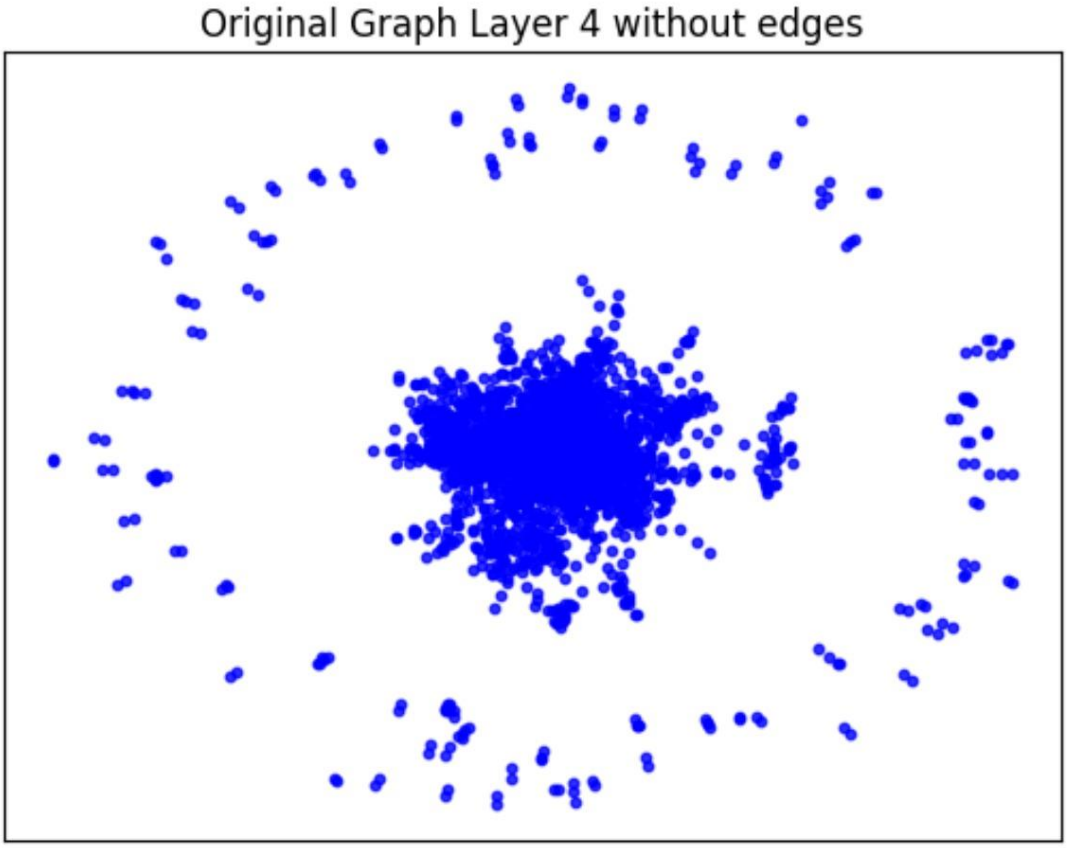


Original Graph Layer 3 with edges

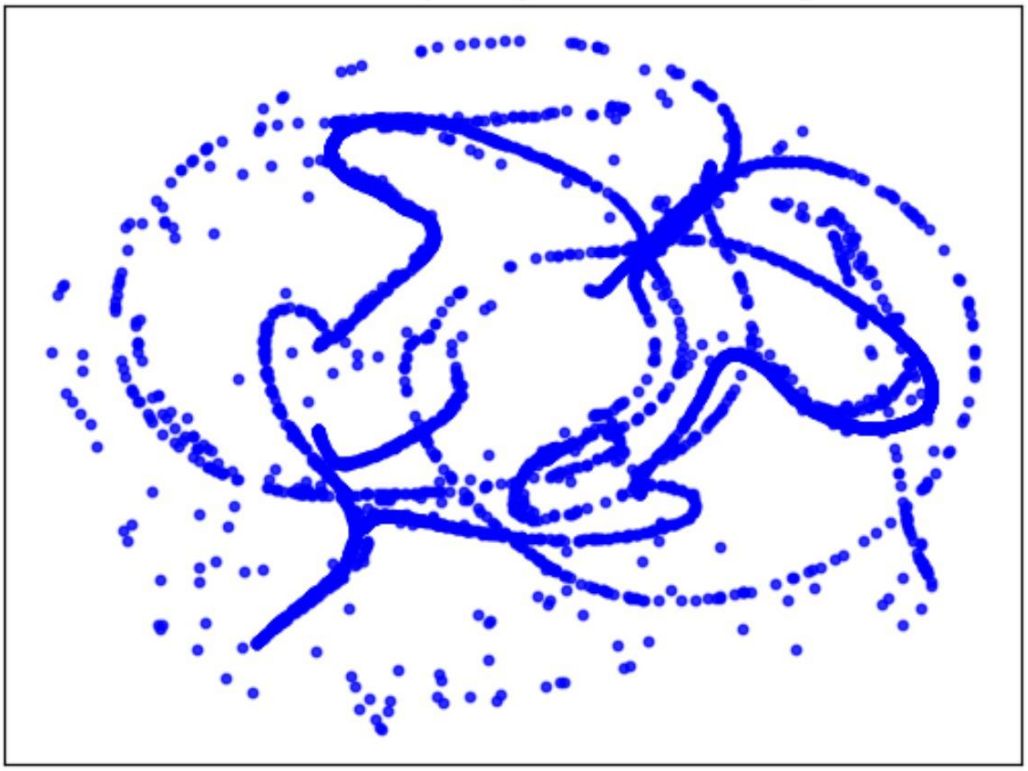



Generated Graph Layer 3 with edges

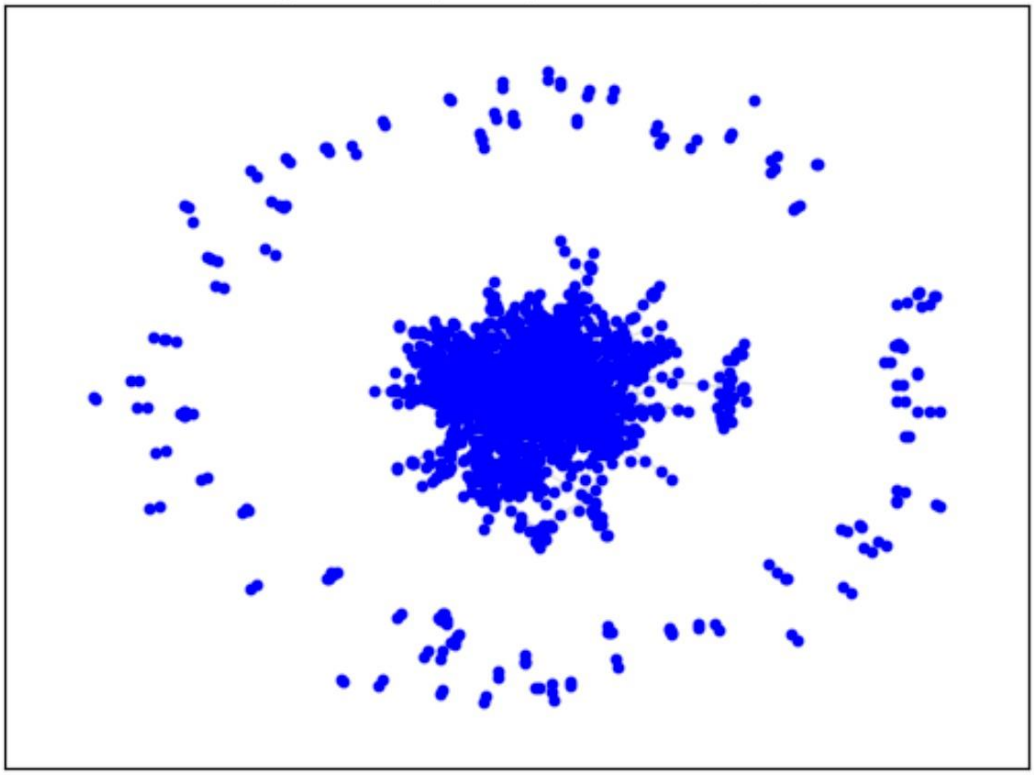
Layer 4:

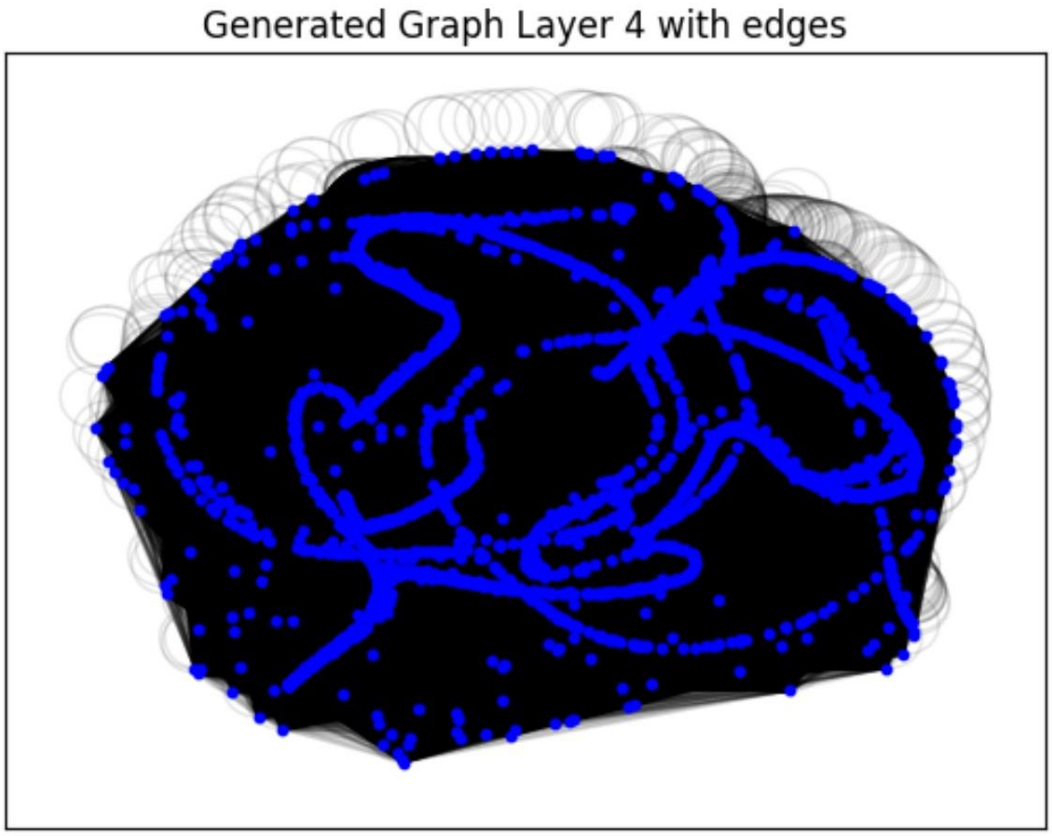

Original Graph Layer 4 without edges

Generated Graph Layer 4 without edges

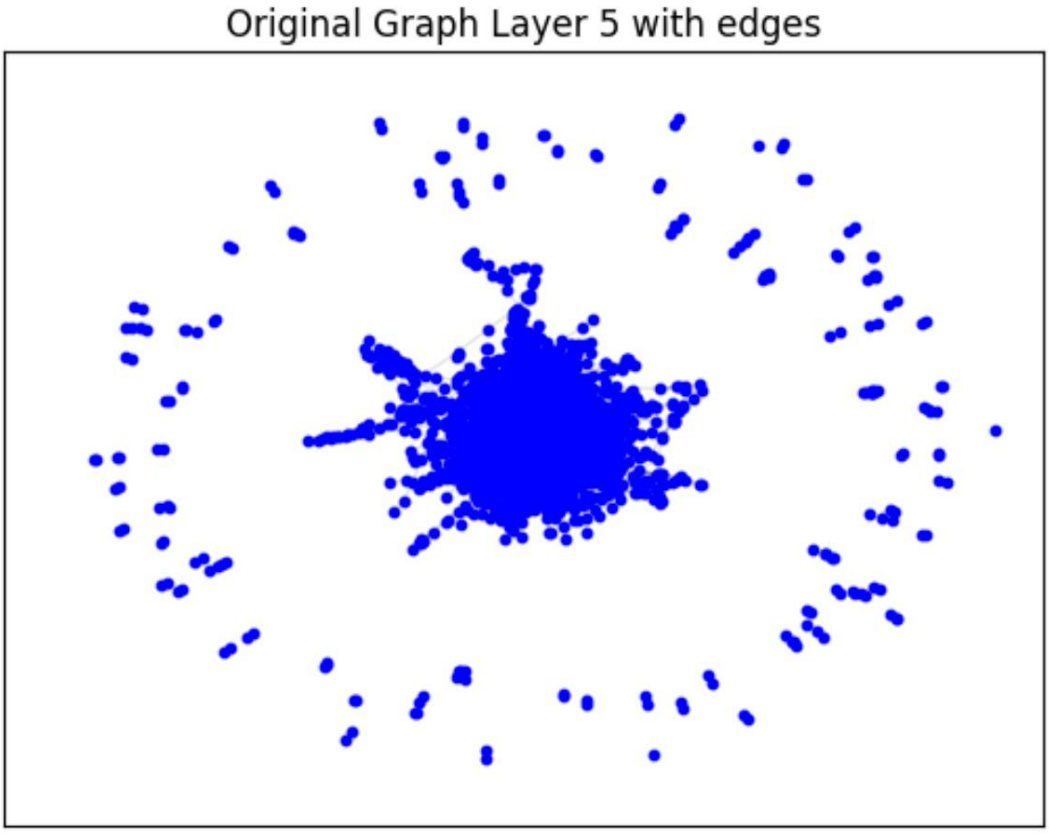


Original Graph Layer 4 with edges

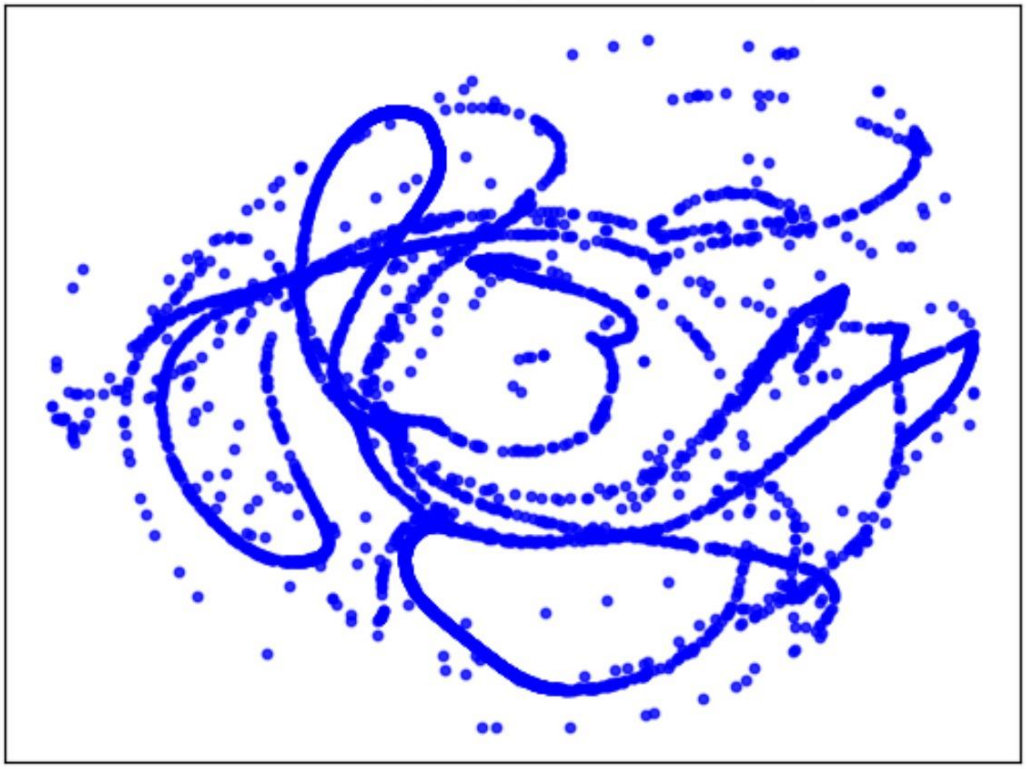



Generated Graph Layer 4 with edges

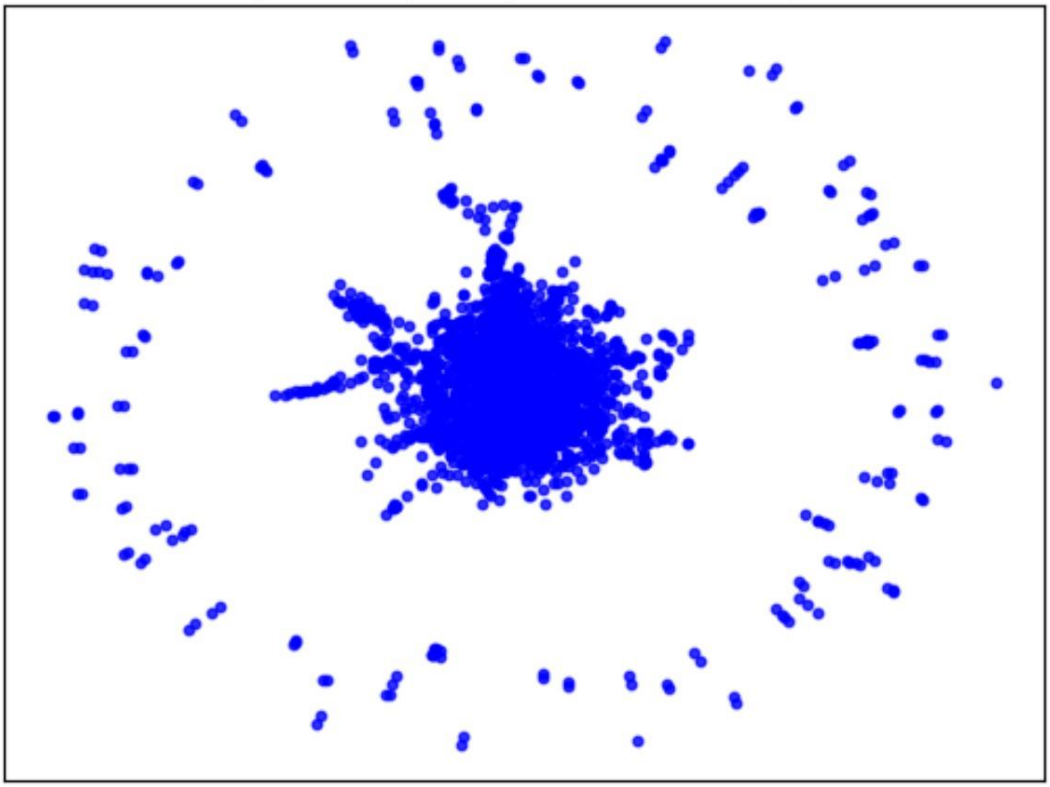
Layer 5:

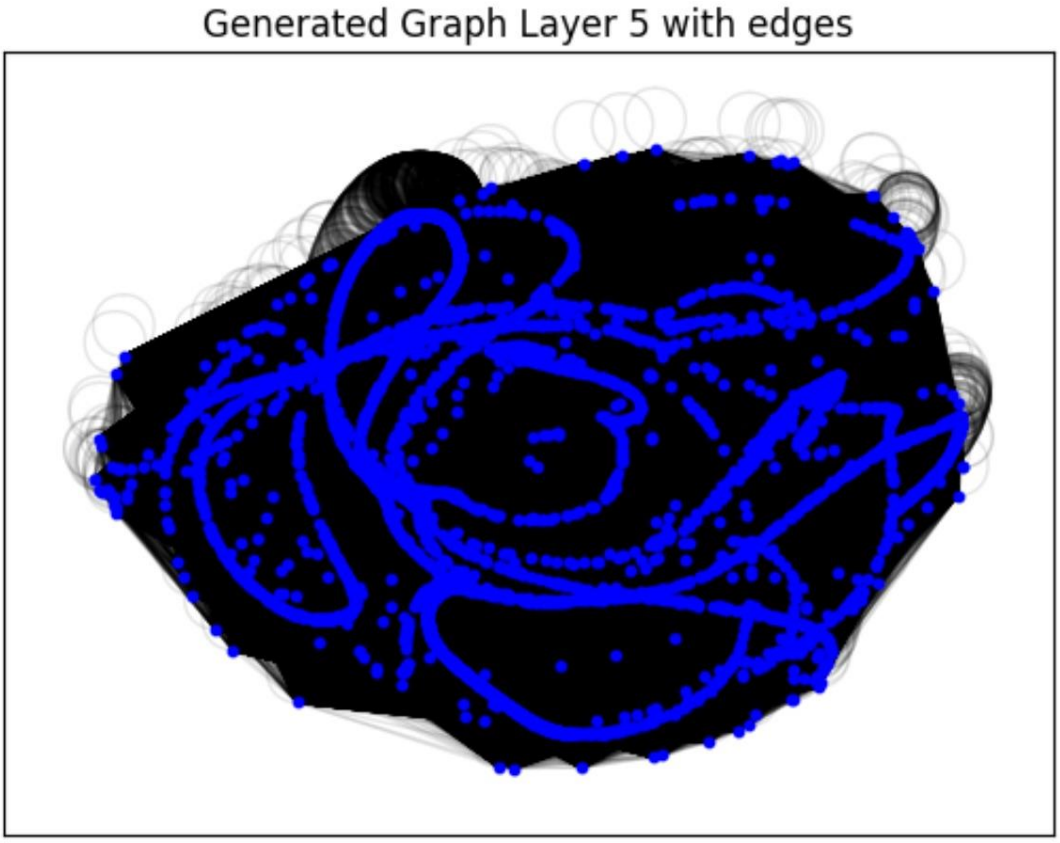

Original Graph Layer 5 with edges

Generated Graph Layer 5 without edges



Original Graph Layer 5 without edges





From the results shown above, we can conclude that the generated graphs differ significantly from the original graphs at each layer.

Original Graphs (Layers 1-5):

The original graphs show a relatively consistent pattern with nodes mostly concentrated in the center and some nodes spread around the periphery. The graphs are generally circular with a dense core.

Generated Graphs (Layers 1-5):

The generated graphs appear to have a more chaotic and spread-out distribution of nodes. There are noticeable patterns or structures that do not exist in the original graphs, such as loops and irregular node clusters. The generated graphs seem to lose the dense core seen in the original graphs and instead exhibit a more dispersed and complex structure.

Edges:

When edges are included in the generated graphs, they form intricate patterns that do not resemble the edge distribution in the original graphs. The edges in the original graphs are more uniformly distributed and follow the node distribution closely.

The generated graphs irregular and more dispersed patterns indicate that the current model might not be effectively learning or replicating the underlying structure of the original graphs. This discrepancy could stem from several factors. One possibility is that the implementation of the GraphARM diffusion model is incomplete, leading to an insufficient representation of the graph's structural properties. Alternatively, the choice of the current diffusion model may not be suitable for this specific task, and it may fail to meet the project's requirements.

In conclusion, while the generated graphs exhibit some level of complexity, they fall short of accurately replicating the original graphs' structure and connectivity. Continued efforts to enhance the model's capabilities are necessary to achieve more faithful graph generation, ultimately aligning the outputs more closely with the original graphs' inherent patterns and relationships.

References

[1] Zhou, D., Zheng, L., Xu, J., & He, J. (2019). Misc-GAN: A multi-scale generative model for graphs. Frontiers in big Data, 2, 3.

[2] Gao, H., & Ji, S. (2019, May). Graph u-nets. In international conference on machine learning (pp. 2083-2092). PMLR.

[3] Wang, Z., Zheng, H., He, P., Chen, W., & Zhou, M. (2022). Diffusion-gan: Training gans with diffusion. arXiv preprint arXiv:2206.02262.

[4] https://tkipf.github.io/graph-convolutional-networks/

[5] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III 18 (pp. 234-241). Springer International Publishing.

[6] Cai, L., Chen, Y., Cai, N., Cheng, W., & Wang, H. (2020). Utilizing amari-alpha divergence to stabilize the training of generative adversarial networks. Entropy, 22(4), 410.

[7] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT press.

[8] Liu, C., Fan, W., Liu, Y., Li, J., Li, H., Liu, H., ... & Li, Q. (2023). Generative diffusion models on graphs: Methods and applications. arXiv preprint arXiv:2302.02591.

[9] Sharma, S., Sharma, S., & Athaiya, A. (2017). Activation functions in neural networks. Towards Data Sci, 6(12), 310-316.

[10] https://learnopencv.com/denoising-diffusion-probabilistic-models/

[11] https://theaisummer.com/skip-connections/

[12] Kong, L., Cui, J., Sun, H., Zhuang, Y., Prakash, B. A., & Zhang, C. (2023, July). Autoregressive diffusion model for graph generation. In *International conference on machine learning* (pp. 17391-17408). PMLR.

ChatGPT

Link: https://chatgpt.com/
Prompts: "Explain to me the concept of graph generation and the application of graph generation in real world".
"Can you help me create a Flowchart of every step i need to take in order to collaborate Graph U-Nets, Misc-GAN and Diffusion-GAN in my project?".
"Can you assure if this next sentence is in present simple and 3rd person view, if not correct it?".