ADVANCED LEARNING FOR TEXT AND GRAPH DATA

Lab session 7: Deep Learning for Graphs (2/2)

Lecture: Prof. Michalis Vazirgiannis
Lab: Yang Zhang, Xiao Fei & Johannes Lutzeyer

Tuesday, December 02, 2025

We want to acknowledge significant contributions of Prof. Giannis Nikolentzos to the content of this lab.

This handout includes theoretical introductions, coding tasks and questions. Before the deadline, you should submit on moodle **or** here a **.zip** file named `Lab<x>_lastname_firstname.pdf` containing a `/code/` folder (itself containing your scripts with the gaps filled) and an answer sheet, following the template available here, and containing your answers to the questions. Your answers should be well constructed and well justified. They should not repeat the question or generalities in the handout. When relevant, you are welcome to include figures, equations and tables derived from your own computations, theoretical proofs or qualitative explanations. **One submission is required for each student. The deadline for this lab is December 09, 2025 11:59 PM**. No extension will be granted. Late policy is as follows: $]0, 24]$ hours late $\rightarrow$ -5 pts; $]24, 48]$ hours late $\rightarrow$ -10 pts; > 48 hours late $\rightarrow$ not graded (zero).

---

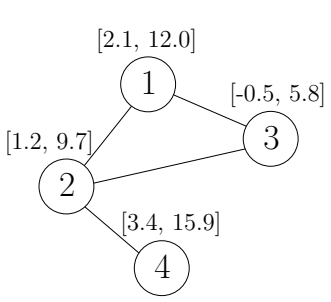
1 Setting Up the Environment

Please make use of the environment set up for Lab 4 by running the following command.

```
conda activate altegrad-lab4
```

2 Learning objective

In this lab, you will learn about neural networks that operate on graphs. These models can be employed for addressing various tasks such as node classification, graph classification and link prediction. The lab is divided into two parts. In the first part, you will implement a graph neural network which belongs to the family of message passing models. The message passing layers of the model that will be implemented employs a self-attention mechanim that allows nodes to attend to their most important neighbors. The model will be evaluated in a graph classification task. In the second part of the lab, you will learn about the graph generation problem, and you will implement a variational graph autoencoder to generate stochastic block model graphs. The two models will be implemented in Python, and we will use the following two libraries: (1) PyTorch (`https://pytorch.org/`), and (2) NetworkX (`http://networkx.github.io/`).



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} 2.1 & 12.0 \\ 1.2 & 9.7 \\ -0.5 & 5.8 \\ 3.4 & 15.9 \end{bmatrix}$$

Figure 1: An example of a graph and of its associated matrices $\mathbf{A}$ and $\mathbf{X}$.

3 Graph Attention Network for Graph-Level Tasks

In the first part of the lab, we will implement a graph neural network (GNN), and use it to classify the nodes of a small benchmark graph. Let $\mathbf{A}$ be the adjacency matrix of a graph $G = (V, E)$ and $\mathbf{X}$ its feature matrix. For attributed graphs, these features may be set equal to the attribute vectors of the nodes. For instance, in biology, proteins are represented as graphs where nodes correspond to secondary structure elements and the feature vector of each secondary structure element contains its physical properties. For graphs without node labels and node attributes, these vectors can either be random or can be initialized with a collection of local vertex features that are invariant to vertex renumbering (e.g., degree). An example of a graph and of its associated matrices $\mathbf{A}$ and $\mathbf{X}$ is given in Figure 1. A standard GNN model takes the matrices $\mathbf{A}$ and $\mathbf{X}$ as input and learns a hidden representation for each node.

Most GNN models update the representation of each node by aggregating the feature vectors of its neighbors. This update procedure can be viewed as a form of message passing algorithm. Different types of message passing layers have been proposed in the past years. Recently, a lot of interest has been devoted to layers that compute the hidden representation of each node in the graph by attending over its neighbors [3, 1]. This allows a node to selectively attend to specific neighbors while ignoring others. We will next implement such a message passing layer which follows a self-attention strategy to update the representations of the nodes.

3.1 Implementation of Graph Attention Layer

You will implement a GNN model that consists of two message passing layers that use attention, and which are followed by a fully-connected layer. As discussed above, messages from some neighbors may be more important than messages from others, thus one could apply self-attention on the nodes to capture message importance. Formally, let $\mathcal{N}(v_i)$ denote the set of neighbors of node v_i. For nodes $v_j \in \mathcal{N}(v_i)$, the attention layer that we will implement computes attention coefficients that indicate the importance of node v_j's features to node v_i as follows:

$$\alpha_{ij}^{(t+1)} = \frac{\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^{(t+1)\top}[\mathbf{W}^{(t+1)}\mathbf{z}_i^{(t)} || \mathbf{W}^{(t+1)}\mathbf{z}_j^{(t)}]\big)\Big)}{\sum_{k\in\mathcal{N}_i}\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^{(t+1)\top}[\mathbf{W}^{(t+1)}\mathbf{z}_i^{(t)} || \mathbf{W}^{(t+1)}\mathbf{z}_k^{(t)}]\big)\Big)}$$

where $[\cdot||\cdot]$ denotes concatenation of two vectors, $\mathbf{z}_i^{(t)}$ and $\mathbf{z}_j^{(t)}$ are the hidden representations of nodes v_i and v_j, and $\mathbf{W}^{(t+1)}$, $\mathbf{a}^{(t+1)}$ denote a trainable matrix and a trainable vector of the $(t+1)$-th message

passing layer. Then the representations of the nodes are updated as follows:

$$\mathbf{z}_i^{(t+1)} = \sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(t+1)} \mathbf{W}^{(t+1)} \mathbf{z}_j^{(t)}$$

In matrix form, the above is equivalent to:

$$\mathbf{Z}^{(t+1)} = \left(\mathbf{A} \odot \mathbf{T}^{(t+1)}\right) \mathbf{Z}^{(t)} \mathbf{W}^{(t+1)}$$

where $\odot$ denotes elementwise product and $\mathbf{T}^{(t)}$ is a matrix such that $\mathbf{T}_{ij}^{(t)} = \alpha_{ij}^{(t)}$.

Task 1
Implement the message passing layer presented above (fill in the body of the `forward()` function of the `GATLayer` class in the `models.py` file). More specifically, you only need to
– compute the output of LeakyReLU$\left(\mathbf{a}^\top [\mathbf{W}\mathbf{z}_i || \mathbf{W}\mathbf{z}_j]\right)$ for all pairs of nodes v_i, v_j that are connected by an edge (Hint: use the following to obtain a tensor (of dimension $2 \times m$) that contains all pairs of nodes that are connected by an edge: `adj.coalesce().indices()`. You can concatenate two tensors using the `torch.cat()` function of PyTorch).
– perform the message passing by multiplying matrix $(\mathbf{A} \odot \mathbf{T})$ by the matrix that stores the node features (Hint: you can perform a matrix-matrix multiplication using the `torch.mm()` function).

Question 1 (5 points)
Multi-Head Attention Mechanism: To make standard GAT layers more stable, we employ K independent attention mechanisms ("heads") that execute the transformation in parallel. Specifically, for each head $k \in \{1, \ldots, K\}$:

- The head receives the **complete** input feature vector $\mathbf{z}^{(t)}$ (of dimension F_{in}).
- It projects these features using a unique, learnable weight matrix $\mathbf{W}^{(k)}$ and computes attention scores using a unique attention vector $\mathbf{a}^{(k)}$.
- It produces its own output features $\mathbf{z}_i^{(t+1,k)}$ of dimension F'_{out}.

1. Write the algebraic equation for the output of a single head k, denoted as $\mathbf{z}_i^{(t+1,k)}$, in terms of its specific parameters and neighbours.
2. The final node representation $\mathbf{z}_i^{(t+1)}$ is obtained by **concatenating** the outputs of all K heads. Write the equation for $\mathbf{z}_i^{(t+1)}$ and determine its total dimensionality.
3. Derive the total number of learnable parameters (weights $\mathbf{W}$ and attention vectors $\mathbf{a}$) required for this multi-head layer. Express your answer in terms of K, F_{in}, and F'_{out}.

3.2 Implementation of Graph Neural Network

You will next implement the GNN model. Let $\mathbf{A}$ be the adjacency matrix of the graph, and $\mathbf{X}$ a matrix whose i^{th} row contains the feature vector of node v_i. The first layer of the model is a message passing layer that employs the attention mechanism presented above, and is defined as follows:

$$\mathbf{Z}^{(1)} = f\Big(\left(\mathbf{A} \odot \mathbf{T}^{(1)}\right) \mathbf{X} \mathbf{W}^{(1)}\Big)$$

where $\mathbf{W}^{(1)}$ is a matrix of trainable weights, f is an activation function (e.g., ReLU, sigmoid, tanh), and $\mathbf{T}^{(1)}$ is a matrix that contains the attention coefficients. The second layer of the model is again a message passing layer that employs the attention mechanism presented above, and is defined as follows:

$$\mathbf{Z}^{(2)} = f\Big((\mathbf{A} \odot \mathbf{T}^{(2)})\mathbf{Z}^{(1)}\mathbf{W}^{(2)}\Big)$$

where $\mathbf{W}^{(2)}$ is a second matrix of trainable weights, f is an activation function, and $\mathbf{T}^{(2)}$ is a matrix that contains the attention coefficients of the second message passing layer. The two message passing layers are followed by a fully-connected layer which makes use of the softmax function to produce a probability distribution over the class labels:

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z}^{(2)}\ \mathbf{W}^{(3)})$$

where $\mathbf{W}^{(3)}$ is a third matrix of trainable weights. Note that for clarity of presentation we have omitted biases.

Task 2
Fill in the body of the `forward()` function of the `GNN` class in the `models.py` file to implement the architecture presented above. More specifically, add the following layers:
– a message passing layer with h_1 hidden units (i.e., $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h_1}$) followed by a ReLU activation function.
– a dropout layer with with p_d ratio of dropped outputs.
– a message passing layer with h_2 hidden units (i.e., $\mathbf{W}^{(2)} \in \mathbb{R}^{h_1 \times h_2}$) followed by a ReLU activation function.
– a fully-connected layer with n_{class} units (i.e., $\mathbf{W}^{(3)} \in \mathbb{R}^{h_2 \times n_{class}}$) followed by the softmax activation function.
– (Hint: the message passing layer is already implemented, thus you only need to feed the node features and the adjacency matrix to instances of that layer).

3.3 Node Classification

We will evaluate the GNN you implemented in a node classification task. The experiments for the first part will be performed on a small dataset which is known as the *karate* network, and which has been used as a benchmark mainly for community detection algorithms. The karate dataset is a friendship social network between 34 members of a karate club at a US university in the 1970s. Due to some conflict between the club administrator and the club main instructor, the members were split into two different groups which correspond to the two classes of the dataset.

Task 3
Compute the adjacency matrix of the karate network. Initialize the features of the nodes to values drawn from the standard normal distribution (Hint: use the `randn()` function of NumPy). Then, run the `gnn_karate.py` script to train the model, make predictions, and compute the classification accuracy.

Question 2 (5 points)
Suppose that instead of informative features, the nodes are annotated with identical feature vectors (i.e., $\mathbf{x}_i = \mathbf{c}$ for all $v_i \in V$, where $\mathbf{c} \in \mathbb{R}^d$ is a constant vector).

1. Algebraically derive the value of the unnormalized attention score e_{ij} and the resulting normalized attention coefficient α_{ij}.
2. Based on your result in 1., explain what happens to the expressiveness of the GAT layer. Does it still behave as an attention mechanism? What standard Graph Neural Network propagation rule does this degenerate into?
3. Given this degeneracy, briefly explain why the model might still be able to classify nodes better than random guessing on the Karate network, despite the loss of feature information.

3.4 Visualization of Attention Scores

Finally, we will visualize the learned attention coefficients. Note that matrix $\mathbf{T}$ that stores the attention weights is not generally symetric, i.e., $\alpha_{ij} = \alpha_{ji}$ does not necessarily hold. We will visualize attention coefficients as follows. We will first create a directed graph that has the same structure as the karate network (i.e., each edge will be replaced with two edges of opposite directions). Then, we will set the width of each edge equal to the strength of the relationship between the source and the sink node. The width of an edge will thus capture how much the source node attends to the sink node.

We will re-train the model that we have already implemented, and then extract the attention scores that are produced from the second message passing layer of the model.

Task 4
Modify the code in the `models.py` file such that the model also returns the attention scores produced in the second message passing layer. Then, in the `gnn_karate.py` script, transform the Torch tensor that contains the attention scores into a NumPy vector (Hint: for a tensor `T`, you can do this using `T.detach().cpu().numpy()`).

4 Graph Generation with Variational Graph Autoencoders

In the second part of the lab, we will implement a variational graph autoencoder, and use it to generate synthetic graphs that are similar to the graphs the model was trained on. Variational autoencoders have attracted a lot of attention in the past years. Instead of embedding the input object to a vector, a variational autoencoder embedds it to a distribution. And then a random sample $\mathbf{z}$ is drawn from the distribution rather than being generated directly from encoder. The decoder is a variational approximation which takes a representation $\mathbf{z}$ and produces the output.

4.1 Dataset

We will train the model that we will implement on a dataset that contains $1,000$ stochastic block model graphs. The stochastic block model takes the following three parameters: (1) the number of nodes n; (2) a partition of the node set $\{1, 2, \ldots, n\}$ into r disjoint subsets, known as blocks; (3) a symmetric $r \times r$ matrix $\mathbf{P}$ of edge probabilities. The edge set is then sampled at random as follows: any two vertices that belong to blocks i and j are connected by an edge with probability $\mathbf{P}_{ij}$.

To generate each of the $1,000$ graphs, we sampled the number of nodes n from $\{20, 21, \ldots, 40\}$. We then, sampled the number of blocks r from $\{2, 3, 4, 5\}$. We grouped nodes into blocks of equal size (except the last block if n is not exactly divisible by r). The probability of connecting two nodes that belong to the same block by an edge was set equal to 0.8, while the probability of adding an edge between nodes that belong to different blocks was set equal to 0.05. Then, matrix $\mathbf{P}$ is defined as follows:

$$\mathbf{P}_{ij} = \begin{cases} 0.8 & \text{if } i = j \\ 0.05 & \text{otherwise} \end{cases}$$

In other words, $\mathbf{P}$ is an $r \times r$ matrix where the elements on the main diagonal are equal to 0.8, while the off-diagonal elements are set equal to 0.05.

4.2 Variational Graph Autoencoder

In the case of autoencoders for graph-structured data, the input to the model is a graph and the objective is to learn a low-dimensional representation for the graph. Let $\mathbf{A}$ be the adjacency matrix of a graph $G = (V, E)$ and $\mathbf{X}$ its feature matrix. For attributed graphs, these features may be set equal to the attribute vectors of the nodes. For instance, in biology, proteins are represented as graphs where nodes correspond to secondary structure elements and the feature vector of each secondary structure element contains its physical properties. For graphs without node labels and node attributes, these vectors can be initialized with a collection of local vertex features that are invariant to vertex renumbering (e.g., degree).

Variational graph autoencoders apply the idea of variational autoencoders on graph-structured data [2]. In our setting, the encoder produces a distribution for each graph. The distribution is usually parameterized as a multivariate Gaussian. Therefore, the encoder predicts the mean and standard deviation of the Gaussian distribution. The low-dimensional representation $\mathbf{z} \in \mathbb{R}^d$ of a graph G is then sampled from this distribution. A high-level illustration of such an autoencoder is given in Figure 2.

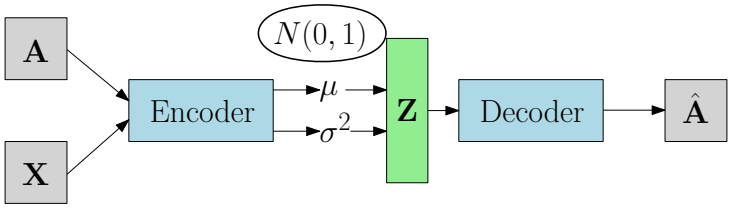


Figure 2: High-level illustration of a variational graph autoencoder.

Formally, the encoder is defined by a variational posterior $q_\phi(\mathbf{z}|G)$ and the decoder by a generative distribution $p_\theta(G|\mathbf{z})$, where ϕ and θ are learned parameters. In our setting, $q_\phi(\mathbf{z}|G) = q_\phi(\mathbf{z}|\mathbf{X}, \mathbf{A}) = N(\mathbf{z}|\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$. Both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are typically produced by graph neural networks (GNNs). Therefore, we have that $\boldsymbol{\mu} = \text{GNN}_{\boldsymbol{\mu}}(\mathbf{X}, \mathbf{A})$ and $\log \boldsymbol{\sigma} = \text{GNN}_{\boldsymbol{\sigma}}(\mathbf{X}, \mathbf{A})$. Then, the latent variable $\mathbf{z}$ can be calculated as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ where each element of $\boldsymbol{\epsilon}$ is drawn from $N(0, 1)$, and $\odot$ denotes the element-wise multiplication between two vectors.

The decoders of most graph generative models can be categorized into two groups: (1) sequential decoders which generate graphs in a set of consecutive steps [4]; and (2) one-shot decoders which generate the adjacency matrices in one single step [2]. One-shot decoders typically use a multi-layer perceptron (MLP) to reconstruct the adjacency matrix:

$$\hat{\mathbf{A}} = \text{MLP}(\mathbf{z})$$

where $\hat{\mathbf{A}}$ is the reconstructed adjacency matrix.

Question 3 (5 points)
Explain how we can introduce conditions into the Variational Graph Autoencoder?

4.3 Implementation of Variational Graph Autoencoder

Given the adjacency matrix $\mathbf{A}$ of a graph, we will first normalize it as follows:

$$\bar{\mathbf{A}} = \tilde{\mathbf{D}}^{-1}\,\tilde{\mathbf{A}}$$

where $\tilde{\mathbf{A}} = \mathbf{A}+\mathbf{I}$, and $\tilde{\mathbf{D}}$ is a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$. The above formula adds self-loops to the graph, and normalizes each row of the emerging matrix such that the sum of its elements is equal to 1. This normalization trick addresses numerical instabilities which may lead to exploding/vanishing gradients when used in a deep neural network model.

We will next implement the Variational Graph Autoencoder. As discussed above, the encoder of the model computes $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ where both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ correspond to a shared GNN followed by different projection layers. In our implementation, the GNN will consist of two message passing layers. Thus, we have:

$$\begin{aligned}
\mathbf{H}^{(1)} &= \text{MLP}_1\big(\bar{\mathbf{A}}\,\mathbf{X}\big)\\
\mathbf{H}^{(2)} &= \text{MLP}_2\big(\bar{\mathbf{A}}\,\mathbf{H}^{(1)}\big)\\
\mathbf{h} &= \text{READOUT}(\mathbf{H}^{(2)})\\
\mathbf{h}_G &= \mathbf{h}\,\mathbf{W} + \mathbf{b}
\end{aligned}$$

Task 5
Implement the encoder architecture presented above in the `models.py` file. More specifically, add the following layers:
– a message passing layer where the message passing operation is followed by an MLP (use the `torch.mm()` function to perform a matrix-matrix multiplication). Let h_2 denote the output size of this second MLP
– a second message passing layer where the message passing operation is again followed by another MLP (use the `torch.mm()` function to perform a matrix-matrix multiplication)
– a readout function that computes the sum of the node representations
– a fully-connected layer that produces $\mathbf{h}_G$

Once $\mathbf{h}_G$ is computed, we employ two fully connected layers to compute $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ as follows:

$$\begin{aligned}
\boldsymbol{\mu} &= \mathbf{h}_G\,\mathbf{W}_{\boldsymbol{\mu}} + \mathbf{b}_{\boldsymbol{\mu}}\\
\log\boldsymbol{\sigma} &= \mathbf{h}_G\,\mathbf{W}_{\boldsymbol{\sigma}} + \mathbf{b}_{\boldsymbol{\sigma}}
\end{aligned}$$

Then, $\mathbf{z}$ is calculated as $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ where each element of $\boldsymbol{\epsilon}$ is drawn from $N(0,1)$.

Question 4 (5 points)
Explain why the reparameterization trick $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ is necessary when training with back-propagation.
Explain what would happen if the model directly sampled $\mathbf{z} \sim \mathcal{N}(\mu, \sigma^2 I)$, without reparameterization.

Task 6
Add the following layers in the of the `loss_function()` function of the variational autoencoder (in the `models.py` file):
– a fully-connected layer with h_3 hidden units (i.e., $\mathbf{W}_{\boldsymbol{\mu}} \in \mathbb{R}^{h_1 \times h_2}$) that produces $\boldsymbol{\mu}$
– a fully-connected layer with h_3 hidden units (i.e., $\mathbf{W}_{\boldsymbol{\sigma}} \in \mathbb{R}^{h_1 \times h_2}$) that produces $\log \boldsymbol{\sigma}$

The decoder is implemented as follows:

$$\mathbf{T}_1 = \text{MLP}(\mathbf{z})$$
$$\mathbf{T}_2 = \mathbf{T}_1\mathbf{W} + \mathbf{b}$$
$$\hat{\mathbf{A}} = \sigma\left(\frac{\mathbf{T}_2 + \mathbf{T}_2^\top}{2}\right)$$

where $\sigma(\cdot)$ is the sigmoid function. Note that the output of the projection layer that follows the MLP is a n^2-dimensional vector and needs to be reshaped such that $\mathbf{T}_2 \in \mathbb{R}^{n \times n}$. Matrix $\mathbf{T}_2$ is not necessarily symmetric, thus we compute the sum of $\mathbf{T}_2$ and its transpose to produce a symmetric matrix.

Task 7
Implement the decoder architecture presented above in the `models.py` file. More specifically, add the following layers:
– a MLP that consists of a series of fully connected layers. Between one fully connected layer and the next, apply the ReLU activation function, layer normalization and dropout
– a fully connected layer that transforms the h-dimensional output of the MLP (for each graph) into a n^2-dimensional vector (i.e., $\mathbf{W} \in \mathbb{R}^{h \times n^2}$)
– reshape the output of the fully connected layer such that for each graph the output is an $n \times n$ matrix (use the `torch.reshape()` function)
– compute the sum of the $n \times n$ matrix and its transpose and divide by 2 (use the `torch.transpose()` function)

Note that the elements of matrix $\hat{\mathbf{A}}$ take values between 0 and 1, and ideally, we would like these values to be as close as possible to the correponding values contained in $\tilde{\mathbf{A}}$. To measure how well the model reconstructs the input data, i.e., the adjacency matrix, we will employ the binary cross entropy as follows:

$$\mathcal{L} = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{A}_{ij}\log(\hat{\mathbf{A}}_{ij}) + (1 - \mathbf{A}_{ij})\log(1 - \hat{\mathbf{A}}_{ij})$$

For very sparse adjacency matrices $\mathbf{A}$, it can be beneficial to re-weight the nonzero terms or alternatively sub-sample pairs of nodes for which $\mathbf{A}_{ij} = 0$. We choose the former strategy for our implementation.

4.4 Graph Generation

Once the model is trained, we can generate graphs that look like the ones the model was trained on by sampling vectors from the mutivariate standard normal distribution and feeding them into the decoder. We can thus use the model to generate stochastic block model graphs.

Task 8
– Create a tensor $\mathbf{Z} \in \mathbb{R}^{5 \times d}$ consisting of 5 rows where each row is a vector sampled from the multivariate normal distribution (use the `torch.randn()` function)
– Execute the `main.py` file to train the model on the dataset that contains stochastic block model graphs and to generate and visualize 5 synthetic graphs

References

[1] Shaked Brody, Uri Alon, and Eran Yahav. How Attentive are Graph Attention Networks? In *10th International Conference on Learning Representations*, 2022.

[2] Martin Simonovsky and Nikos Komodakis. Graphvae: Towards generation of small graphs using variational autoencoders. In *Proceedings of the 27th International Conference on Artificial Neural Networks*, pages 412–422, 2018.

[3] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. In *6th International Conference on Learning Representations*, 2018.

[4] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5708–5717, 2018.