Adversarial Attacks via Latent Perturbations on Graph Classification Task

Winston Yu[1], Jianming Geng[2], and Barry Xue[3]

[1,2,3]Halıcıoğlu Data Science Institute, UCSD

January 2023

Abstract

Graph neural networks (GNNs) have seen tremendous success in recent years, leading to their widespread adoption in a variety of applications. However, recent studies [1] [5] have demonstrated that GNNs can be vulnerable to adversarial attacks. Adversarial attacks allow adversaries to manipulate GNNs by introducing small, seemingly harmless perturbations to the input data, which can lead to misclassification or unexpected behavior from the GNN. Such attacks can have drastic impacts on security and privacy, particularly in safety-critical or privacy-sensitive applications. Consequently, research into adversarial attacks on GNNs is becoming increasingly important, and we thus propose a novel adversarial attack that is described in the following section.

1 Introduction

Adversarial attacks on graph-based models are attacks where an adversary deliberately introduces small perturbations to the input data to fool the target model. The perturbations can be crafted in such a way that they are not easily noticeable to human observers but can have a significant impact on the model's output. Recently, adversarial attacks have become a hot topic in the field of deep learning, particularly in the context of graph neural networks (GNNs).

Adversarial attacks on graph-based models are useful for several reasons. Firstly, they can help researchers better understand the limitations of graph-based models and identify their vulnerabilities. By understanding the types of perturbations that can be introduced to the input graph to fool the target model, researchers can develop more robust models that are less susceptible to attacks. Secondly, adversarial attacks can be used to improve the robustness of graph-based models. By training models to recognize and defend against adversarial attacks, researchers can improve the model's ability to handle noisy or incomplete data, which is a common problem in real-world applications.

Finally, adversarial attacks can be used for malicious purposes, such as to deceive a target model or to manipulate the results of a graph-based system. It is therefore important to develop robust adversarial defense mechanisms to protect against such attacks.

A lot of previous studies have been conducted on related topics. For instance, a direct perturbation on the original graph (removal or addition of edges) is feasible in order to change the topological order of the graph. In addition, changing node features to completely transform the feature matrix of the graph has also been popular. However, those manual changes are not natural enough for the model to learn. A more recent study on GraphRNN has introduced the idea of a natural generation of attacks. Given the time-series nature of the model, GraphRNN will only generate one node at a time, given the context vector from the previous iteration. Such a generation of nodes would presumably preserve the topological order of the original graph while only making trivial changes that are not perceivable to humans. This work has shed light on our approach, as we will go into more detail in the next few sections.

We thus introduce a framework that consists of three parts: Generator (GraphRNN), Inverter (Embedding + MLPs), and Discriminator (MLP). For the embedding part, we first attempted with Graph2Vec and later switched to Graph Attention Model (GAM). The main advantage and contribution of this framework is that it provides a more effective and robust method for adversarial attacks on graph-based models. By using a Graph Attention Machine as a substitute structure, the framework addresses some of the limitations of the previous attack setup, such as unnatural manipulation. The resulting model is theoretically more accurate and less prone to overfitting, which makes it more useful for real-world applications. Overall, this framework represents a significant step forward in the development of more effective adversarial attacks on graph-based models.

2 Network Architecture and Attack

The network that will generate adversarial examples has two components: a generator $\mathcal{G} : \mathcal{Z} \to \mathcal{X}$ and an inverter $\mathcal{I} : \mathcal{X} \to \mathcal{Z}$. On a high level, the network executes the attack as follows: after $\mathcal{G}$ and $\mathcal{I}$ have been trained, the inverter finds the latent space representation $z = \mathcal{I}(x)$ of some benign input $x \in \mathcal{X}$, and then we repeatedly use a search strategy to find an element $z^{'}$ of the latent space close to z such that $x \approx \mathcal{G}(z)$ and $\mathcal{G}(z')$ are also close with respect to the graph metric but the attacked model assigns to them different classes. [2]

3 Notation

Let $\mathcal{X}$ be a set of graphs - i.e. it contains graphs (V, E) such that V is a set of vertices and there exists $N \in \mathbb{Z}^+$ and $E \in \{0, 1\}^{N \times N}$. Let $\mathcal{Z} = \mathbb{R}^L$ be a latent space. $\mathbb{P}_{\mathcal{X}}$ is the data distribution on $\mathcal{X}$, and $\mathbb{P}_{\mathcal{Z}}$ is a distribution on the latent

space - most likely it will be a normal distribution, since many generative models use elements drawn from the normal distribution as inputs to their generative models.

$\mathcal{X}$ is a metric space when equipped with the Gromov-Wasserstein (GW) distance, which is defined as follows. [2] Let (X, e_X, μ_X) and (Y, e_Y, μ_Y) be metric measure networks, which are generalized notions of graphs, albeit without node features - we will use the Fused Gromov-Wasserstein distance to measure distances between graphs with node features. For the purpose of this report, e_X and e_Y may be taken to be indicator functions, although more generally they are allowed to be measurable functions. Additionally, we take μ_X and μ_Y to be uniform distributions.

Let Π be the set of couplings between μ_X and μ_Y, i.e. the set of probability distributions on $X \times Y$ whose marginals are μ_X and μ_Y. Then the first-order GW distance between metric measure networks (X, e_X, μ_X) and (Y, e_Y, μ_Y) is abbreviated as $d_{GW}(X, Y)$ and is defined as:

$$d_{GW}(X, Y) := \inf_{\pi \in \Pi} \int_{X \times Y} \int_{X \times Y} |e_X(x, x') - e_Y(y, y')| \partial\pi(x, y) \partial\pi(x', y')$$

The first-order Wasserstein distance between probability distributions will also be useful and is defined as:

$$W(\mu_X, \mu_Y) := \inf_{\pi \in \Pi} \int_{X \times Y} ||x - y||_2 \partial\pi(x, y)$$

Given a Lipschitz continuous function f, $||f||_L$ is its Lipschitz constant. $|| \cdot ||_2$ is the standard Euclidean norm; whether it applies to $\mathbb{R}$ or $\mathbb{R}^L$ will be determined by context.

4 Network Training

We use GraphRNN [6] as $\mathcal{G}_\theta$, where θ are the parameters, and train it to minimize $W(\mathbb{P}_\mathcal{X}, \mathcal{G}_\theta(\mathbb{P}_\mathcal{Z}))$, which by the Kantovorich-Rubinstein duality, equals

$$\max_{||f||_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_\mathcal{X}}[f(x)] - \mathbb{E}_{z \sim \mathbb{P}_\mathcal{Z}}[f(\mathcal{G}_\theta(z))]$$

$f : \mathcal{X} \to \mathbb{R}$ is a feedforward neural network, with some modifications. f computes various summary statistics of the input graph, such as node degree distribution and clustering coefficient distributions, and processes each statistic through separate networks with *tanh* activation functions; each network maps to $\mathbb{R}$. Then, f passes their outputs through a linear combination, where the weights are learn-able, followed by another *tanh*. Explicitly, f has form:

$$f(G) = tanh(\sum_{i=1}^{N} a_i f_i(G))$$

where $\{f_i\}$ are the separate networks mentioned previously, $\{a_i\}$ are learnable weights, and N is the number of statistics to compute.

Next, we train an off-the-shelf graph2vec model [4] $\mathcal{J}$ mapping from $\mathcal{X}$ to $\mathcal{Z}^{'}$, after which we train an inverter $\mathcal{I}_\gamma : \mathcal{Z}^{'} \to \mathcal{Z}$. From now on, the parameters of $\mathcal{J}$ are fixed. With the parameters of the inverter $\mathcal{I} = \mathcal{I}_\gamma$ as γ, we denote $\mathcal{I}_\gamma$ to be $\mathcal{I}_\gamma \circ \mathcal{J}$ for convenience. Fix some $\lambda > 0$. $\mathcal{I}_\gamma$ is trained according to the following objective:

$$\min_{\gamma} \mathbb{E}_{x \sim \mathbb{P}(\mathcal{X})} \left[d_{GW}(\mathcal{G}_\theta(\mathcal{I}_\gamma(x)), x) \right] + \lambda \mathbb{E}_{z \sim \mathbb{P}(\mathcal{Z})} \left[||\mathcal{I}_\gamma(\mathcal{G}_\theta(z)) - z||_2 \right]$$

Having trained $\mathcal{G}_\theta$ and $\mathcal{I}_\gamma$, the attack may be executed as follows [Figure 1]:

Algorithm 1 Find Latent Representations of Adversarial Examples

Require: $\mathcal{G}_\theta$ is the generator, $\mathcal{I}_\gamma$ is the inverter, $\mathbb{P}_\mathcal{X}$ is the data distribution, $n_{adv} \in \mathbb{Z}^+$ is the number of adversarial examples to generate, f is a black-box graph classifier, $\Delta r \in \mathbb{R}^+$ is an increment for the search radius, $r \in \mathbb{R}^+$ is the search radius, $B \in \mathbb{Z}^+$ is the number of iterations for the **Search** algorithm in $\mathcal{Z}$

$S \leftarrow \varnothing$
for $k \leftarrow 1$ to n_{adv} **do**
  draw $x_k \sim \mathbb{P}_\mathcal{X}$
  $y_k \leftarrow f(x_k)$
  $z_k \leftarrow \mathcal{I}_\gamma(x_k)$
  $\tilde{x}_k, \tilde{y}_k, \tilde{z}_k \leftarrow$ **Search**(x_k, y_k, z_k, r, B)
  $S \leftarrow S \cup \{(\tilde{x}_k, \tilde{y}_k, \tilde{z}_k)\}$
end for

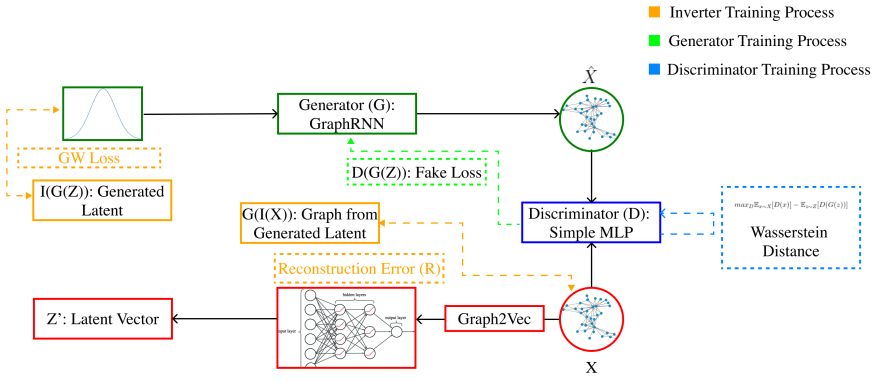

Figure 1: Training Procedure v1.

Algorithm 2 Search

Require: benign input $x \sim \mathbb{P}_{\mathcal{X}}$, label y, $z \in \mathcal{Z}$, latent space distribution $\mathbb{P}_{\mathcal{Z}}$, black-box classifier f, search radius r, number of searches done B

2: **for** $j \leftarrow 1$ to B **do**
  $\tilde{z}_j \sim \mathbb{P}_{\mathcal{Z}}$ s.t. $||z - \tilde{z}_j||_2 \leq r$
4: $\tilde{x}_j \leftarrow \mathcal{G}_\theta(\tilde{z}_j)$
  $\tilde{y}_j \leftarrow f(\tilde{x}_j)$
6: **if** $\tilde{y}_j \neq y$ **then return** $(\tilde{x}_j, \tilde{y}_j, \tilde{z}_j)$
  end if
8: **end for**
print "No adversarial examples found! Increase B or r."

5 Implementation

5.1 GraphRNN

1. It is an autoregressive model that can generate graphs of arbitrary size, providing the inverter with a diverse pool of graphs with different characteristics.

2. Unlike earlier approaches, such as GraphVAE, GraphRNN can process large graphs on limited hardware resources. Experiments with GraphVAE show that on a single GPU, the maximum number of nodes in a graph is 38. In comparison, on datasets of molecules and social networks, GraphRNN can generate graphs of hundreds of nodes.

3. In its original form, GraphRNN generates new graphs in a stochastic and autoregressive manner from an initial hidden state of a zero vector. We propose to replace this hidden state with an element of latent space. During both of our training procedures, we pass latent space elements that are either generated from a standard normal distribution or that are outputted by the inverter.

We started with the original implementation of GraphRNN and developed our training loop around it. GraphRNN's architecture contains two modified Gated Recurrent Unit (GRU) models. The modified GRU is a wrapper around the PyTorch GRU. The wrapper connects the GRUs with two 2 sets of linear dense layers. The intention is to add to the expressiveness of the model and remove the effect of variation in sequencing the adjacency vector.

The modified GRU doesn't take the original adjacency matrix of the graph as input; instead, the author intended it to accept a sequenced version of the adjacency matrix, which is smaller in size and this will lower the runtime. The sequence is done with Breadth-first search from an arbitrary starting node.

Formally, one modified GRU model generates the next node of the graph, whereas the other GRU model connects the new nodes to existing nodes. The

first GRU model is named graph-level RNN and the second model is named edge-level RNN. Theoretically, the graph-level RNN and the edge-level RNN work in a nested fashion as shown in the pseudocode. The outer loop calls the graph-level RNN to decide whether a new node will be generated or not. Then the inner loop calls the edge-level RNN to iterate through all existing nodes and update the edge connections between them. In short, the graph-level RNN updates the node list, and the edge-level RNN updates the edge list of each node. After the edge lists are generated and normalized with sigmoid, the adjacency vector is then generated from these lists using a sampling technique. From the adjacency vector, a graph adjacency matrix can then be constructed.

Algorithm 3 GraphRNN inference algorithm

Input: RNN-based transition module f_{trans} , output module f_{out} , probability distribution $\mathcal{P}_{\theta_i}$ parameterized by θ_i, start token SOS, end token EOS, empty graph state h'
Output: Graph sequence S^π
$S_1^\pi = \text{SOS}, h_1 = h', i = 1$
repeat
  $i = i + 1$
  $h_i = f_{\text{trans}}\ \left(h_{i-1}, S_{i-1}^\pi\right)$ { update graph state }
  $\theta_i = f_{\text{out}}\ (h_i)$
  $S_i^\pi \sim \mathcal{P}_{\theta_i}$ { sample node i,s edge connections }
until S_i^π is EOS
return $S^\pi = (S_1^\pi, \ldots, S_i^\pi)$

Training of the graph-level RNN and edge-level RNN is done with teacher forcing, meaning the loss is calculated with the ground truth. The procedure uses binary cross-entropy loss and Adam optimizers. The binary cross entropy loss is calculated by comparing the packed version of the generated adjacency vectors with the packed version of the ground truth. There are no nested iterations involved during training to avoid discontinuity in backward gradient graphs.

Graph generation is done differently to accommodate customized graph sizes. The number of nodes in the generated graph is specified by the user, and the graph-level RNN is run with the same number of iterations. The output of the graph-level RNN is also passed into the edge-level RNN iteratively to create the output adjacency matrix.

We tested the GraphRNN model by switching the initial hidden layer with a noisy input sampled from a normal distribution [Figure 2, Figure 3]. After 1000 iterations of training, the GraphRNN model is generating results that are comparable to the ground truth. The generated samples have a tendency to be more dense than the original graph. In the original iteration, it is recommended to train for at least 3000 iterations for the model to output realistic graphs, so this is a respectable result.

Figure 2: Original graphs from PROTEIN dataset.

Figure 3: Generated graphs after training with PROTEIN dataset.

5.2 Graph2Vec

Graph2Vec [4] is a graph embedding model. Given a graph adjacency matrix, Graph2Vec embeds it into a 1-dimensional vector with a fixed size. The implementation of Graph2Vec is based on Doc2Vec. Similar to Doc2Vec, Graph2Vec relies on a skip-gram model of the graph, where each graph is encoded by its subgraphs, similar to how each document in Doc2Vec is encoded by its context words. We are using the Graph2Vec implementation provided by the Karate Club python library.

We need to get a constant-size representation of graphs as the input to the inverter. Graph2Vec is trained every forward pass with the generated graphs. The corresponding graph embedding is then retrieved from the Graph2Vec model and serves as the input to the inverter.

We are unable to quantitatively evaluate the performance of the Graph2Vec embedding; however, the embedding output maintains a constant size and the inverter is successfully trained with the embedding as the input. We also experimented with connecting Graph2Vec with the discriminator in training procedure [Figure 1]. We encountered failing gradient backpropagation with our module, due to the poor integration between PyTorch and Karate Club, and we are unable to train the generator with this setup. For this reason, we decided to switch to other alternatives for training procedure 2 [Figure 6].

6 Results

To evaluate the output of the GraphRNN after training with Wasserstein Loss, we plotted the distribution of the generated graphs' embeddings and compared it to the original graphs' embeddings. [Figure 4] The result shown above is generated with the MUTAG dataset.

For the original graph, we colored each graph's data point according to its labels. In MUTAG, there are two labels, so each point is colored in one of two colors.

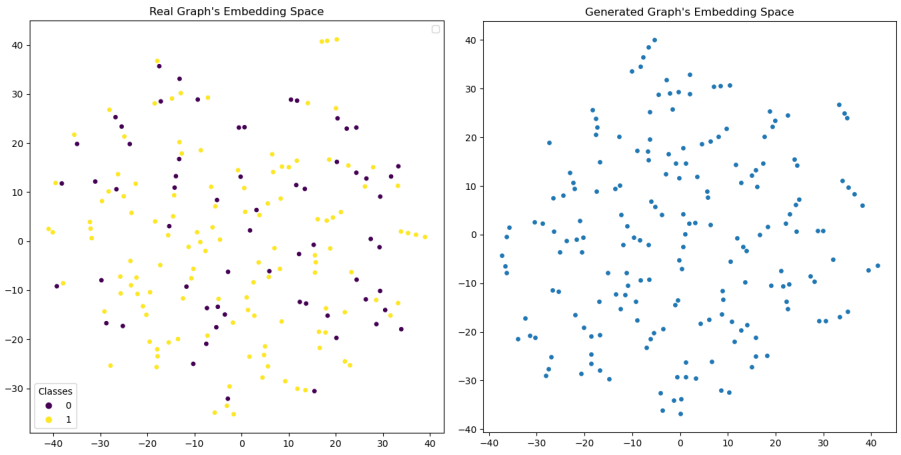

Figure 4: (Left) The original graphs' Graph2Vec embeddings after encoded by t-SNE to 2-dimensional space. (Right) The generated graphs' Graph2Vec embeddings after encoded by t-SNE to 2-dimensional space.

For the generated graph, because we don't have assigned labels to generated graphs in our training procedure, we can't apply the same coloring technique. But looking at the shape of the embedding distribution, we can see that they aren't very different.

One interesting observation is that the generated embeddings are relatively sparse at some points, and the local distributions look linear at places where the real graphs aren't. This could mean that our training isn't effective enough to capture the underlying distribution of the original dataset.

The loss plot of all three components [Figure 5] supports that the training procedure has serious underlying issues. As shown in the above plot, the loss value of all three components stagnates after the initial 10 epochs of training.

One possible explanation for this is the generating GraphRNN is not being properly trained, so the training of the inverter and the discriminator is being bottlenecked by the poor performance of the generator.

7 Conclusion

During our training, we realize there are several challenges we need to resolve. The first challenge is to ensure our models the property of permutation invariant. This is one of the major failures with the first training procedure. In our framework training, a graph can be represented by an adjacency matrix, which is a matrix that indicates which nodes in the graph are connected to each other. However, the adjacency matrix of a graph can vary depending on the node ordering used to represent the graph. This means that the graphs from a particular dataset (say, MUTAG) can have multiple adjacency matrices de-

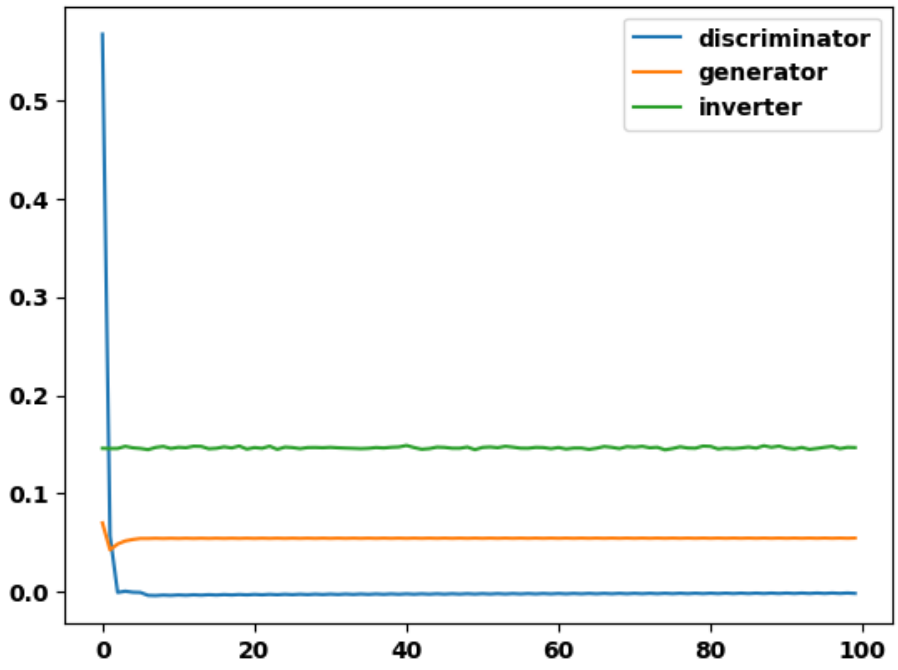

Figure 5: Loss plots of the discriminator (Multi-layer perceptron), the inverter (Graph2Vec + Multi-layer perceptron), and the generator (GraphRNN) after 100 epochs of training.

pending on the ordering of the nodes, which can make it difficult to train if our models are vulnerable to different orderings of the nodes.

Incompatible back-propagation of gradients is another challenge yet to be resolved. First, the gradients can become unstable or vanish during back-propagation when the models are highly interdependent. For example, in the case of an adversarial attack on GraphRNN, the generator model's objective is to generate perturbations that can fool the discriminator model, which is in turn trying to detect the perturbations. The gradient updates for the generator and discriminator models are highly interdependent and can lead to instability during training. An even more fatal thing is that PyTorch backpropagation is not compatible with our implementation of Graph2Vec. In other words, tensors passed into Graph2Vec will not have any gradient to train on. This is the reason we switched to GAM for our embedding technique.

In the context of an adversarial attack model, graph manipulation functions are necessary to generate perturbations to the input graph that can fool the target machine learning model. This involves adding or removing edges, changing node attributes, or altering the overall structure of the graph. However, PyTorch does not provide native support for these operations, which can make

it difficult to implement the adversarial attack model efficiently.

8 Appendix A: Training Procedure v2

After implementing the first training procedure, we observed poor training loss. The stagnated loss suggests the Wasserstein loss is insufficient to train the generator. As a result, the discriminator and the inverter stop improving after 10 epochs of training. This calls for an overhaul of the first training procedure.

We decided that one probable cause for this failure occurs in the discriminator and inverter. We experimented with two types of inputs with both models. First, input the graph adjacency matrix, and then pass in the Graph2Vec embedding. The Graph2Vec embedding approach failed due to technical incompatibility between Karateclub and PyTorch autograd. And while the vanilla adjacency matrix approach worked, it is still problematic because the adjacency matrix doesn't provide a deterministic description of graphs. The discriminator can discern the real example from a generated example abusing the properties of the adjacency matrix (e.g. artificial padding), which are not necessarily relevant to the graph. As demonstrated by the high loss value in the early epochs, GraphRNN's output graphs are unrealistic and unreliable. The Wasserstein distance is calculated with the discriminator's output to train the generator. Yet, the Wasserstein loss considers only the realness score from the discriminator; there is no description of the difference between the distribution of the generated graphs and the distribution of the sampled graphs. With this loss, the generator receives no information on how the generated graphs should look like.

To combat this issue, we switch from a WGAN style training procedure to a new autoencoder style procedure. The discriminator model is removed, and the encoder is pre-trained with other downstream tasks (e.g. graph classification). In addition, the encoder is now replaced with a Graph Attention Machine instead of Graph2Vec. [Figure 6]

9 Appendix B: Graph Attention Machine

Graph Attention Model (GAM) [3] is a neural network-based approach that can learn node embeddings by focusing on the relevant nodes in the graph using attention mechanisms. GAM can capture the local and global information of the graph and is particularly useful when the graph is large and complex. GAM is also flexible and can be easily adapted to different types of graphs. Most importantly, GAM approach is compatible with PyTorch autograd, whereas Graph2Vec from Karateclub does not.

The GAM architecture typically consists of multiple layers of graph attention modules. Each graph attention module takes the node features and adjacency matrix as input and computes the attention coefficients for each node in the graph. It is trained by optimizing a loss function that measures the discrepancy

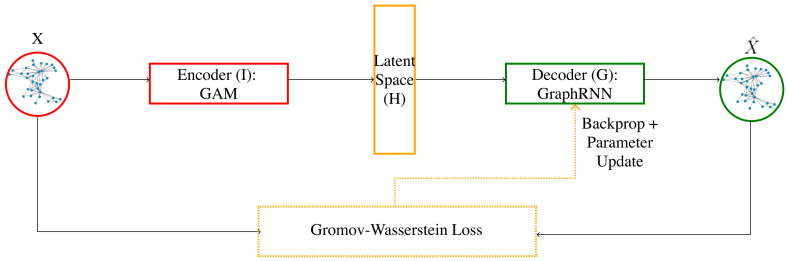

Figure 6: Training Procedure v2.

between the predicted node embeddings and the ground-truth node labels. The loss function in our case is a logmax that differentiates between predicted and ground truth labels. The loss is back propagated through the network to update the model parameters, such as the attention coefficients and node embeddings. This process is specifically done using the Adam optimizer.

References

[1] Moustafa Alzantot et al. *Generating Natural Language Adversarial Examples*. arXiv:1804.07998 [cs]. Sept. 2018. DOI: 10.48550/arXiv.1804.07998. URL: http://arxiv.org/abs/1804.07998 (visited on 02/06/2023).

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. arXiv:1701.07875 [cs, stat]. Dec. 2017. URL: http://arxiv.org/abs/1701.07875 (visited on 02/06/2023).

[3] John Boaz Lee, Ryan Rossi, and Xiangnan Kong. "Graph classification using structural attention". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018, pp. 1666–1674.

[4] Annamalai Narayanan et al. *graph2vec: Learning Distributed Representations of Graphs*. arXiv:1707.05005 [cs]. July 2017. URL: http://arxiv.org/abs/1707.05005 (visited on 02/06/2023).

[5] Xingchen Wan et al. *Adversarial Attacks on Graph Classification via Bayesian Optimisation*. arXiv:2111.02842 [cs, stat]. Nov. 2021. URL: http://arxiv.org/abs/2111.02842 (visited on 02/06/2023).

[6] Jiaxuan You et al. *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*. arXiv:1802.08773 [cs]. June 2018. URL: http://arxiv.org/abs/1802.08773 (visited on 02/06/2023).