Graph Adversarial Attack via Rewiring

Yao Ma
majunyao@gmail.com
New Jersey Institute of Technology

Suhang Wang
szw494@psu.edu
The Pennsylvania State University

Tyler Derr
tyler.derr@vanderbilt.edu
Vanderbilt University

Lingfei Wu
lwu@email.wm.edu
JD.COM Silicon Valley Research Center

Jiliang Tang
tangjili@msu.edu
Michigan State University

ABSTRACT

Graph Neural Networks (GNNs) have demonstrated their powerful capability in learning representations for graph-structured data. Consequently they have enhanced the performance of many graph-related tasks such as node classification and graph classification. However, it is evident from recent studies that GNNs are vulnerable to adversarial attacks. Their performance can be largely impaired by deliberately adding carefully created unnoticeable perturbations to the graph. Existing attacking methods often produce the perturbation by adding/deleting a few edges, which might be noticeable even when the number of modified edges is small. In this paper, we propose a graph rewiring operation to perform the attack. It can affect the graph in a less noticeable way compared to existing operations such as adding/deleting edges. We then utilize deep reinforcement learning to learn the strategy to effectively perform the rewiring operations. Experiments on real world graphs demonstrate the effectiveness of the proposed framework. To understand the proposed framework, we further analyze how its generated perturbation impacts the target model and the advantages of the rewiring operations. The implementation of the proposed framework is available at https://github.com/alge24/ReWatt.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Theory of computation** → *Sequential decision making*.

KEYWORDS

graph neural networks, adversarial attack, rewiring

ACM Reference Format:
Yao Ma, Suhang Wang, Tyler Derr, Lingfei Wu, and Jiliang Tang. 2021. Graph Adversarial Attack via Rewiring. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3447548.3467416

KDD '21, August 14–18, 2021, Virtual Event, Singapore

ACM ISBN 978-1-4503-8332-5/21/08…$15.00
https://doi.org/10.1145/3447548.3467416

1 INTRODUCTION

Graph structured data is ubiquitous in many real world applications. Data from different domains, such as social networks, molecular graphs and transportation networks, can be modeled as graphs. Recently, increasing efforts have been made on developing deep neural networks on graph structured data. This stream of works, which is known as Graph Neural Networks (GNN), has shown to enhance the performance in many graph related tasks such as node classification [1, 2] and graph classification [3–6]. Recent studies have shown that deep neural networks are highly vulnerable to adversarial attacks [7–10]. In computer vision, performing an adversarial attack is to add deliberately created but unnoticeable perturbations to a given image such that the deep model misclassifies the perturbed image. Unlike image data, which can be represented in the continuous space, graph structured data is discrete. Increasing attention has been paid on the robustness of graph neural networks against adversarial attacks [11]. The existing adversarial attacks on graphs can be roughly divided to white box [12, 13], gray box [14, 15] and black box attacks [16, 17] according to the level of the knowledge of the victim model that can be accessed by the attack models. The majority of these attacks have been designed for the *node classification* task by adding/deleting edges. In this work, we aim to design adversarial black box attacks for the *graph classification* task. To ensure that the difference between the attacked graph and the original graph is "unnoticeable", the number of actions (adding/deleting edges) that can be taken by the attacking algorithms is usually constrained by a limited budget. However, even when this budget is small, adding or deleting edges can still make "noticeable" changes to the graph structure [18]. For example, it is evident that many important graph properties are based on eigenvalues and eigenvectors of the Laplacian matrix of the graph [19]; while adding or deleting an edge can make remarkable changes the eigenvalues/eigenvectors [20]. Thus, in this work, we propose a new operation based on graph rewiring. A single graph rewiring operation involves three nodes $(v_{fir}, v_{sec}, v_{thi})$, where we remove the existing edge between v_{fir} and v_{sec} and add a new edge between v_{fir} and v_{thi}. Note that v_{thi} is constraint to be the 2-hop neighbor of v_{fir} in our setting. It is obvious that the proposed rewiring operation preserves some basic properties of the graph such as number of nodes and edges and total degrees of the graph, while operations like adding and deleting edges cannot. Furthermore, the proposed rewiring operation affects some of graph properties based on the Laplacian such as algebraic connectivity in a much smaller way than adding/deleting edges, which will be demonstrated theoretically and empirically. In addition, the rewiring operation is

a more natural way to modify the graph. For example, in biology, the evolution of DNA and amino acid sequences can lead to pervasive rewiring of protein–protein interactions [21]. In this paper, we aim to construct adversarial examples by performing rewiring operations for the task of graph classification. More specifically, we treat the process of conducting a series of rewiring operations to a given graph as a discrete Markov decision process (MDP) and use reinforcement learning to learn how to make these decisions.

2 RELATED WORK

In recent years, adversarial attacks on deep learning models have attracted increasing attention in the area of computer vision. Many deep models are found to be easily fooled by adversarial samples, which are generated by adding deliberately designed unnoticeable perturbation to normal images [7, 8]. Most works have focused on the computer vision domain, where the data sample can be represented in the continuous space. Few attention has been paid on the discrete data structure such as graphs. Graph Neural Networks have been shown to bring impressive advancements to many different graph related tasks such as node classification and graph classification. Recent researches show that the graph neural networks are also venerable to adversarial attacks. A greedy algorithm is proposed in [14] to perform adversarial attack to the node classification task. Their algorithm tries to change the label of a target node by modifying both the graph structure and node features. In [16], a deep reinforcement learning based attacker is proposed to attack both the node classification and the graph classification task. The metattack algorithm [15] is designed to impair the overall performance of node classification based on meta learning. In [12, 13, 22], gradient based methods are proposed to attack the graph structure by deleting/adding edges. All these aforementioned methods modify the graph structure by adding or deleting edges. Some more recent works [17, 23] on attacking node classifications proposed to modify the graph structure by injecting nodes. Some recent surveys [11, 13, 24] provide a comprehensive view on adversarial attacks on graph-structured data and some techniques to defend these attacks. In this work, we propose to modify the graph structure using rewiring, which is shown to make less noticeable changes to the graph structure.

3 BACKGROUND

In this section, we introduce notations and the target graph convolutional model to attack. We denote a graph as $G = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V} = \{v_1, \ldots, v_{|\mathcal{V}|}\}$ and $\mathcal{E} = \{e_1, \ldots, e_{|\mathcal{E}|}\}$ are the sets of nodes and edges, respectively. The edges describe the relations between nodes, which can be described by an adjacency matrix $\mathbf{A} \in \{0, 1\}^{|\mathcal{V}|\times|\mathcal{V}|}$. $\mathbf{A}_{ij} = 1$ means v_i and v_j are connected, 0 otherwise. Each node in the graph is associated with some features. These features are represented as a matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}|\times d}$, where the i-th row of $\mathbf{X}$ denotes the node features of node v_i and d is the dimension of features. Thus, an attributed graph can be represented as $G = \{\mathbf{A}, \mathbf{X}\}$.

3.1 Graph Classification

In the setting of graph classification, we are given a set of graphs $\mathcal{G} = \{G_i\}$. Each of these graphs G_i is associated with a label y_i. The task is to build a good classifier using the given set of graphs such that it can make correct predictions on unseen graphs. A graph classifier parameterized by θ can be represented as $f(G|\theta) = y^o$, where y^o denotes the label of a graph $G \in \mathcal{G}$ predicted by the classifier. The parameters θ in the classifier $f(\cdot|\theta)$ can be learned by solving the following optimization problem $\min_\theta \sum_i L(f(G_i|\theta), y_i)$, where $L(\cdot, \cdot)$ is the loss function to measure the difference between the predicted and ground truth labels. Cross entropy is a commonly adopted measurement for $L(\cdot, \cdot)$.

3.2 Graph Convolution Networks

Recently, Graph Neural Networks have been shown to be effective in graph representation learning. These models usually learn node representations by iteratively aggregating, transforming and propagating node information. In this work, we adopt the graph convolutional networks (GCN) [1]. A graph convolutional layer in the GCN framework can be represented as

$$\mathbf{F}^j = ReLU(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{F}^{j-1}\mathbf{W}^j) \tag{1}$$

where $\mathbf{F}^j \in \mathbb{R}^{N\times d_j}$ is the output of the j-th layer and $\mathbf{W}^j$ represents the parameters of this layer. A GCN model usually consists of J graph convolutional layers, with $\mathbf{F}^0 = \mathbf{X}$. The output of the GCN model is $\mathbf{F}^J$, which is denote as $\mathbf{F}$ for convenience. To obtain a graph level embedding $\mathbf{u}_G$ for the graph G to perform graph classification, we apply a global pooling over the node embeddings.

$$\mathbf{u}_G = pool(\mathbf{F}) \tag{2}$$

Different global pooling functions can be used, and we adopt the max pooling in this work. A multilayer perceptron (MLP) and softmax layer are then sequentially applied on the graph embedding to predict the label of the graph

$$y^o = \text{argmax}\ softmax(MLP(\mathbf{u}_G|\mathbf{W}_{MLP})) \tag{3}$$

where $MLP(\cdot|\mathbf{W}_{MLP})$ denotes the MLP with parameters as $\mathbf{W}_{MLP}$. A GCN-based classifier for graph classification can be described using (1), (2) and (3) as introduced above. For simplicity, we summarize it as $y^o = f_{GCN}(G|\theta_{GCN})$, where θ_{GCN} includes all the parameters in the model.

4 PROBLEM FORMULATION

In this work, we aim to build an attacker $\mathcal{T}$ that takes a graph as input and modify its structure to fool a GCN classifier. Modifying a graph structure is equivalent to modify its adjacency matrix. The function of the attacker can be represented as $\tilde{G} = \mathcal{T}(G) = \{\mathcal{T}(\mathbf{A}), \mathbf{X}\} = \{\tilde{\mathbf{A}}, \mathbf{X}\}$. Given a classifier $f(\cdot)$, the goal of the attacker is to modify the graph structure so that the classifier outputs a different label from its originally predicted one. Note here, we neglect the θ inside $f(\cdot)$, as the classifier is already trained and fixed. Mathematically, the goal of the attacker can be represented as: $f(\mathcal{T}(G)) \neq f(G)$. As described above, the attacker $\mathcal{T}$ is specifically designed for a given classifier $f(\cdot)$. To reflect this in the notation, we now denote the attacker for the classifier $f(\cdot)$ as $\mathcal{T}_f$. In our work, the attacker $\mathcal{T}_f$ has limited knowledge of the classifier. The only information the attacker can get from the classifier is the label of (modified) graphs. In other words, the classifier $f(\cdot)$ is treated as a black-box model for the attacker $\mathcal{T}_f$. An important constraint to the attacker $\mathcal{T}_f$ is that it is only allowed to make "unnoticeble" changes to the graph structure. To account for this,

we propose the *rewiring* operation, which is supposed to make more subtle changes than adding or deleting edges. We will show that the rewiring operation can better preserve a lot of important properties of the graph compared to adding or deleting edges in Section 5.1. We also empirically compare the rewiring operation with the deleting/adding edges in Section 6.3 in the supplementary file. The definition of the proposed rewiring operation is given below:

Definition 1. *A rewiring operation* $\mathbf{a}$ *involves three nodes and it can be denoted as* $\mathbf{a} = (v_{fir}, v_{sec}, v_{thi})$*, where* $v_{sec} \in N^1(v_{fir})$ *and* $v_{thi} \in N^2(v_{fir})/N^1(v_{fir})$*.* $N^k(v_{fir})$ *denotes the* k*-th hop neighbors of* v_{fir} *and the sign / stands for exclusion. The rewiring operation deletes the existing edge between nodes* v_{fir} *and* v_{sec}*, while adding an edge to connect nodes* v_{fir} *and* v_{thi}*.*

The attacker $\mathcal{T}_f$ is given a budget of K rewiring operations to modify the graph structure. A straightforward way to set K is choosing a small fixed number. However, it is likely that graphs in a given data set have various graph sizes. The same number of rewiring operations can affect the graphs of different sizes in various magnitude. Hence, a more suitable way is to allow a flexible number of rewiring operations according to the graph size. Thus, we propose to use $K = p \cdot |\mathcal{E}|$ for a given graph G, where $p \in (0, 1)$ is a ratio. With the above notations and definitions, the process and goal of the attacker on a graph G can be now denoted as $\mathcal{T}_f(G) \leftrightarrow (a_1, a_2, \ldots, a_M)[G]$ such that $f(\mathcal{T}(G)) \neq f(G)$, where the $(a_1, a_2, \ldots, a_M)[G]$ means to sequentially apply the rewiring operations $a_1, \ldots, a_M$ to the graph G, and $M \leq K$ is the number of rewiring operations.

5 REWIRING-BASED ATTACK TO GRAPH CONVOLUTIONAL NETWORKS

In this section, we first discuss the properties of the rewiring operation and then introduce the proposed attacking framework ReWatt based on reinforcement learning and rewiring.

5.1 Properties of Proposed Rewiring Operation

In this section, we describe the advantages of the proposed rewiring operation compared to simply adding or deleting edges. **More empirical discussions can be found in Section 6.3.**

Property 1. The proposed rewiring operation does not change the number of nodes, the number of edges and the total degree of a graph.

Many important graph properties are based on the eigenvalues of the Laplacian matrix of a graph [19] such as Algebraic Connectivity [25]. The algebraic connectivity of a graph G is the second-smallest eigenvalue of its Laplacian matrix [25]. The larger the algebraic connectivity is, the more difficult it is to separate the graph into components (i.e., more edges need to be removed). Next, we demonstrate that the proposed rewiring operation is likely to make smaller changes to eigenvalues, which result in unnoticeable changes under graph Laplacian based measures such as Algebraic Connectivity. For a graph G with $\mathbf{A}$ as its adjacency matrix, its Laplacian matrix $\mathbf{L}$ is defined as $\mathbf{L} = \mathbf{D} - \mathbf{A}$, where $\mathbf{D}$ is the diagonal degree matrix [26]. Let $\lambda_1, \ldots, \lambda_{|\mathcal{V}|}$ denote the eigenvalues of the Laplacian matrix arranged in the increasing order with $\mathbf{x}_1, \ldots, \mathbf{x}_{|\mathcal{V}|}$ the corresponding eigenvectors. We show how a single rewiring operation affects the eigenvalues based on the following lemma:

Lemma 1. *[27] Let* $(\alpha_i, \mathbf{h}_i)$ *be the eigen-pairs of a symmetric matrix* $\mathbf{M} \in \mathbb{R}^{N \times N}$*. Given a perturbation* $\Delta\mathbf{M}$ *to the matrix* $\mathbf{M}$*, its eigenvalues can be updated by* $\Delta\alpha_i = \mathbf{h}_i^T \Delta\mathbf{M}\mathbf{h}_i$*.*

The detailed proof can be found in [27]. According to the lemma, it is easy to verify that, for a graph G with $\mathbf{L}$ as its Laplacian matrix and $(\lambda_i, \mathbf{x}_i)$ as its eigenpairs, when we add an edge between nodes v_j and v_k, $\Delta\lambda_i = (\mathbf{x}_i[j] - \mathbf{x}_i[k])^2$; while $\Delta\lambda_i = -(\mathbf{x}_i[j] - \mathbf{x}_i[k])^2$ when we delete the edge between v_j and v_k. Using this lemma, we have the following corollary.

Corollary 1. *For a given graph* G *with Laplacian matrix* $\mathbf{L}$*, one proposed rewiring operation* $(v_{fir}, v_{sec}, v_{thi})$ *affects the eigen-value* λ_i *by* $\Delta\lambda_i$*, for* $i = 1, \ldots, |\mathcal{V}|$*, where*

$$\Delta\lambda_i = -(\mathbf{x}_i[fir] - \mathbf{x}_i[sec])^2 + (\mathbf{x}_i[fir] - \mathbf{x}_i[thi])^2 \tag{4}$$

where $\mathbf{x}_i[index]$ *denotes the index-th value of the eigenvector* $\mathbf{x}_i$*.*

Proof. Let $\Delta\mathbf{L}$ denote the change in the Laplacian matrix after applying the rewiring operation $(v_{fir}, v_{sec}, v_{thi})$ to graph G. Then we have $\Delta\mathbf{L}[fir, sec] = \Delta\mathbf{L}[sec, fir] = 1$, $\Delta\mathbf{L}[fir, thi] = \Delta\mathbf{L}[thi, fir] = -1$, $\Delta\mathbf{L}[sec, sec] = -1$, $\Delta\mathbf{L}[thi, thi] = 1$ and 0 elsewhere. Thus

$$\begin{aligned}
\Delta\lambda_i &= \mathbf{x}_i^T \Delta\mathbf{L}\mathbf{x}_i \\
&= 2\mathbf{x}_i[fir]\mathbf{x}_i[sec] - \mathbf{x}_i[sec]^2 + \mathbf{x}_i[thi]^2 - 2\mathbf{x}_i[fir]\mathbf{x}_i[thi] \\
&= \mathbf{x}_i[fir]^2 + 2\mathbf{x}_i[fir]\mathbf{x}_i[sec] - \mathbf{x}_i[sec]^2 \\
&\quad + \mathbf{x}_i[thi]^2 - 2\mathbf{x}_i[fir]\mathbf{x}_i[thi] - \mathbf{x}_i[fir]^2 \\
&= -(\mathbf{x}_i[fir] - \mathbf{x}_i[sec])^2 + (\mathbf{x}_i[fir] - \mathbf{x}_i[thi])^2
\end{aligned} \tag{5}$$

which completes the proof. □

With Corollary 1, we can obtain the following properties of the rewiring operations that are also supported by the empirical observations in Section 6.3 in the supplementary file.

Property 2. The rewiring operation is likely to make small changes to the first a few eigenvalues.

Each eigenvalue λ_i of the Laplacian matrix measures the "smoothness" of its corresponding eigenvector $\mathbf{x}_i$ [28, 29]. The "smoothness" of an eigenvector measures how different its elements are from their neighboring nodes. Thus, the first few eigenvectors with relatively small eigenvalues are rather "smooth". In the proposed rewiring operation, v_{sec} is the direct neighbor of v_{fir} and v_{thi} is the 2-hop neighbor of v_{fir}. Thus, the difference $\mathbf{x}_i[fir] - \mathbf{x}_i[thi]$ is expected to be smaller than the difference $\mathbf{x}_i[fir] - \mathbf{x}_i[can]$, where $\mathbf{x}_i[can]$ can be any other node that is further away. This means that the proposed rewiring operation (to 2-hop neighbors) tends to make smaller changes to the first a few eigenvalues than rewiring to any further away nodes or adding an edge between two nodes that are far away from each other.

Property 3. The proposed rewiring operation is less likely to change the rank of the Laplacian matrix.

Let $|\mathcal{V}|$ denote the number of nodes in the graph and $\#components$ denote the number of connected components in a given graph. Then the rank of the Laplacian matrix can be expressed as $|\mathcal{V}| -$

#components, as the multiplicity of eigenvalue 0 of Laplacian matrix is equal to *#components*. As shown in the definition of Algebraic Connectivity, the larger the second-smallest eigenvalue is, the more difficult it is to separate the graph into components. Utilizing Lemma 5.1 and following similar analysis in Corollary 1, we can validate that adding an edge will increases the second smallest eigenvalue (or the algebra connectivity) while deleting an edge will decrease it. Compared to adding/deleting edges which might delete more edges than adding ones, the rewiring operation always perform the same number of adding and deleting edges operations. The rewiring operation is less likely to disconnect the graph. As a result, the rewiring operation is also less risky to change the rank of the Laplacian matrix.

5.2 Graph Adversarial Attack with Reinforcement Learning

Given a graph G, the process of the attacker $\mathcal{T}$ is a general decision making process $M = (\mathcal{S}, \mathcal{A}, P, R)$, where $\mathcal{A} = \{a_t\}$ is the set of actions, which consists of all valid rewiring operations, $\mathcal{S} = \{s_t\}$ is the set of states that includes all possible intermediate and final graphs after rewiring, P is the transition dynamics that describes how a rewiring action a_t changes the graph structure $p(s_{t+1}|, s_t, \ldots, s_1, a_t)$. R is the reward function, which gives the reward for the action taken at a given state. Thus, the procedure of attacking a graph can be described by a trajectory $(s_1, a_1, r_1, \ldots, s_M, a_M, r_M)$, where $s_1 = G$. The key for the attacker is to learn how to make the decision of picking a suitable rewiring action at the state s_t. This can be done by learning a policy network to get the probability $p(a_t|s_t, \ldots, s_1)$ and then sampling the rewiring operation correspondingly. Following this intuition, the decision at a state s_t is dependant on all its previous states, which could be difficult due to the long-term dependency. Note that all intermediate states s_t are predicted to the same label as the original graph. Therefore, we can treat each state as a new graph to be attacked. In other words, the decision making at the state s_t can be solely dependant on the current state, $p(a_t|s_t, \ldots, s_1) = p(a_t|s_t)$. As a consequence, we model the attack process as a Markov Decision Process (MDP) [30], and we adopt reinforcement learning to learn how to make effective decisions. The key elements of the environment for the reinforcement learning are defined as follows –

- **State Space:** The state space of the environment consists of all the intermediate graphs generated after the possible rewiring operations;
- **Action Space:** The action space consists of the valid rewiring operations as defined in Definition 1. Note that the valid action space is dynamic when the state changes, as the k-th hop neighbors are different in different states;
- **State Transition Dynamics:** Given an action (rewiring operation) $a_t = \{v_{fir}, v_{sec}, v_{thi}\}$ at state s_t. The next state s_{t+1} is achieved by deleting the edge between v_{fir} and v_{sec} in the current state s_t and adding an edge to connect v_{fir} with v_{thi};
- **Reward Design:** The main goal of the attacker is to make the classifier $f(\cdot)$ predict a different label from the originally predicted one. We also want to encourage the attacker to take as a few actions as possible such that the modification to the graph structure is minimal. Thus, we assign a positive reward when the attack is successful and assign a negative reward for each action step taken. The reward $R(s_t, a_t)$ is given as

$$R(s_t, a_t) = \begin{cases} 1 & \text{if } f(s_t) \neq f(s_1); \\ n_r & \text{if } f(s_t) = f(s_1). \end{cases} \tag{6}$$

  where n_r is the negative reward to penalize each step taken. Similar to how we set a flexible rewiring budget K, we also propose to use $n_r = -\frac{1}{K} = -\frac{1}{p \cdot |\mathcal{E}|}$, which depends on the size of the graph;
- **Termination** The attack process will stop either when the number of actions reaches the budget K or the attacker successfully changes the label of the modified graph.

5.3 Policy Network

In this subsection, we introduce the policy network to learn the policy $p(a_t|s_t)$ on top of the graph representations learned by a GCN model. To choose a valid rewiring action, we decompose the rewiring action to 3 steps: 1) choosing an edge $e_t = (v_{e_1}, v_{e_2})$ from the set of edges of the intermediate graph s_t; 2) determining $v_{e_{t1}}$ or $v_{e_{t2}}$ to be v_{fir_t} and the other to be v_{sec_t}; and 3) choosing the third node v_{thi_t} from $N^2_{s_t}(v_{fir_t})/N^1_{s_t}(v_{fir_t})$. Correspondingly, we decompose $p(a_t|s_t)$ as follows

$$p(a_t|s_t) = p_{edge}(e_t|s_t) \cdot p_{fir}(v_{fir_t}|e_t, s_t) \cdot p_{thi}(v_{thi_t}|v_{fir_t}, e_t, s_t) \tag{7}$$

We design three policy networks based on GCN to estimate the three distributions in the right hand of the equation (7), which will be introduced next. To select an edge from the edge set $\mathcal{E}_{s_t}$, we generate the edge representation from the node representations $\mathbf{F}_{s_t} \in \mathbb{R}^{|\mathcal{V}_{s_t}| \times d_F}$ learned by GCN. For an edge $e = (v_{e_1}, v_{e_2})$, the edge representation can be represented as $\mathbf{e} = concat(\mathbf{u}_{s_t}, h(\mathbf{F}_{s_t}[e_1, :], \mathbf{F}_{s_t}[e_2, :]))$, where $\mathbf{u}_{s_t}$ is the graph representation of the state s_t, $h(\cdot, \cdot)$ is a function to combine the two node representations and $concat(\cdot, \cdot)$ denotes the concatenation operation. We include $\mathbf{u}_{st}$ in the representation of the edge to incorporate the graph information when making the decision. The representation of all the edges in $\mathcal{E}_{s_t}$ can be represented as a matrix $\mathbf{E}_{s_t} \in \mathbb{R}^{|\mathcal{E}_{s_t}| \times 2d_F}$, where each row represents an edge. The probability distribution over all the edges can be represented as

$$p_{edge}(\cdot|s_t) = softmax(MLP(\mathbf{E}_{s_t}|\theta_{edge})), \tag{8}$$

where we use $MLP(\cdot|\theta_{edge})$ to denote a Multilayer Perceptron that maps $\mathbf{E}_{s_t} \in \mathbb{R}^{|\mathcal{E}_{s_t}| \times 2d_F}$ to a vector in $\mathbb{R}^{|\mathcal{E}_{s_t}|}$, which, after going through the softmax layer, represents the probability of choosing each edge. Let $e_t = (v_{e_{t1}}, v_{e_{t2}})$ denote the edge sampled according to (8). To decide which node is going to be the first node, we estimate the probability distribution over these two nodes as

$$p_{fir}(\cdot|e_t, s_t) = softmax(MLP([\mathbf{v}_{e_{t1}}, \mathbf{v}_{e_{t2}}]^T|\theta_{fir})) \tag{9}$$

where $\mathbf{v}_{e_{ti}} = concat(\mathbf{e}_t, \mathbf{F}_{s_t}[e_{ti}, :]) \in \mathbb{R}^{3d_F}$ for $i = 1, 2$. The first node can be sampled from the two nodes $v_{e_{t1}}, v_{e_{t2}}$ according to (9). We then proceed to estimate the probability distribution $p(\cdot|v_{fir_t}, e_t, s_t)$. For any node $v_c \in N^2(v_{fir_t})/N^1(v_{fir_t})$, we use $\hat{\mathbf{v}}_c = concat(\mathbf{v}_{e_{t1}}, \mathbf{F}_{s_t}[c, :])$ to represent it. The representations for all the nodes in $N^2(v_{fir_t})/N^1(v_{fir_t})$

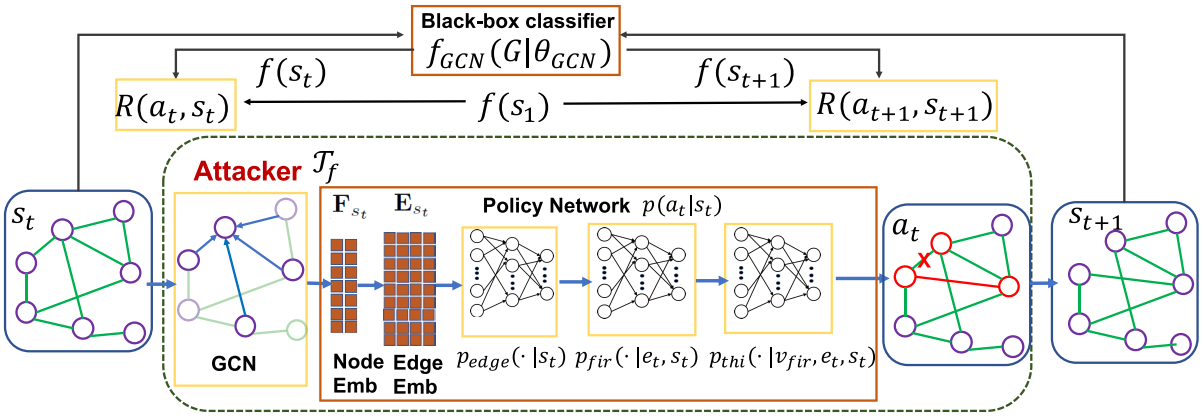


Figure 1: The overall framework of ReWatt

can be represented by a matrix $\hat{\mathbf{V}}_{s_t} \in \mathbb{R}^{|N^2(v_{fir_t})/N^1(v_{fir_t})| \times 4d_F}$ with each row representing a node. The probability distribution of choosing the third node over all the candidate nodes can be modeled as:

$$p_{thi}(\cdot|v_{fir_t}, e_t, s_t) = softmax(MLP(\hat{\mathbf{V}}_{s_t}|\theta_{thi})) \quad (10)$$

The third node v_{thi_t} can be sampled from the set of candidate nodes $N^2(v_{fir_t})/N^1(v_{fir_t})$ according to the probability distribution in (10). An action a_t can be generated by sequentially estimating and sampling from the probability distributions in (8), (9) and (10).

5.4 Proposed Framework - ReWatt

With the rewiring and the policy network defined above, our overall framework can be summarized as follows. As shown in Figure 1, with State s_t, the Attacker uses GCN to learn node and edge embeddings, which are used as input to Policy Networks to make decision about the next action. Once the new action is sampled from the policy network, rewiring is performed on s_t and we arrive in the new state s_{t+1}. We query the black-box classifier to get the prediction $f(S_{t+1})$, which is compared with $f(s_1)$ to get reward. Policy gradient [30] is adopted to learn the policies by maximizing the rewards.

6 EXPERIMENT

In this section, we conduct experiments to evaluate the performance of the proposed framework ReWatt. We also provide some empirical investigations and a case study to analyze how the trained attacker works.

6.1 Attack Performance

To demonstrate the effectiveness of ReWatt, we conduct experiments on three widely used social network data sets [31] for graph classification, i.e., REDDIT-MULTI-12K, REDDIT-MULTI-5K and IMDB-MULTI [32]. The statistics of the datasets can be found in Table 1. We use RE-12K, RE-5K and IM-M to denote the three datasets in Table 1. In this table, #nodes denotes the average number of nodes over all graphs and #edges denotes the average number of edges over all graphs. ACC denotes the mean of Average Clustering Coefficient (ACC) over all graphs. GCC denotes the mean of Global Clustering Coefficient (GCC) over all graphs.

Table 1: Statistics of the data sets

	#graphs	#labels	#nodes	#edges	ACC	GCC
RE-12K	11,929	12	391.41	456.89	0.0331	0.0087
RE-5K	4,999	5	508.52	594.87	0.0268	0.0038
IM-M	1,500	3	13	65.94	0.968	0.8955

Note that the re-wiring operation (as well as the other operations) may lead to abnormal structure of some kinds of graphs, which can make the graphs invalid, especially for chemical molecules. Thus, in this paper, We avoid chemical related datasets but only use social networks datasets. In the social domain, if the changes are subtle, it is less likely to introduce abnormal structures. Meanwhile, it is straightforward to extend our framework to datsets from the other domains if we have the domain expertise. For example, if we know what structures are abnormal, we can use such knowledge to constraint the the state space of the RL framework. We leave it as one future work. In this work, the classifier we target to attack is the GCN-based classifier as introduced in Section 3. We set the number of layers to 3. We need to train the classifier using a fraction of the data and then treat the classifier as a black box to be attacked. We then use a part of the remaining data to train the attacker and use the rest of the data to test the performance of the attacker. Thus, for each data set, we split it into three parts with the ratio of $a\% : b\% : c\%$, which are used to train the classifier, the attacker and test the performance of the attacker, respectively. For the REDDIT-MULTI-12K and REDDIT-MULTI-5K data sets, we set $a = 90$, $b = 8$ and $c = 2$. As the size of the IMDB-MULTI data set is quite small, to have enough data for test, we set $a = 50$, $b = 30$ and $c = 20$. We compare the attacking performance of the proposed framework with the RL-S2V proposed in [16], random selection method and some variants of our proposed framework. We briefly describe these baselines:

- **RL-S2V** is a reinforcement learning based attack framework [16], which allows adding and deleting edges to the graph with a fixed budget for all the graphs;

Table 2: Performance comparison in terms of the success rate

	REDDIT-MULTI-12K			REDDIT-MULTI-5K			IMDB-MULTI		
K	1	2	3	1	2	3	1	2	3
ReWatt	14.4%	21.6%	23.4%	8.99%	16.9%	18.0%	22.3%	22.3%	22.6%
RL-S2V	9.46%	18.5%	21.1%	4.49%	16.9%	18.0%	2.00%	6.00%	3.33%
p	1%	2%	3%	1%	2%	3%	1%	2%	3%
ReWatt	24.8%	33.3%	36.5%	11.2%	20.2%	27.0%	22.3%	22.3%	22.6%
ReWatt-a	26.1%	35.1%	42.8%	5.60%	21.3%	30.3%	22.0%	23.0%	23.6%
ReWatt-n	17.6%	25.7%	31.1%	5.60%	14.6%	19.1%	21.3%	21.3%	21.6%
random	10.3%	15.7%	21.6%	3.30%	12.4%	16.9%	1.33%	1.33%	1.66%
random-s	6.30%	6.70%	9.45%	5.60%	6.74%	11.0%	1.00%	1.33%	1.66%

- **Random** denotes an attacker that performs the proposed rewiring operations randomly;
- **Random-s** is also based on random rewiring. Note that ReWatt can terminate before using all the budget. We record the actual number of rewiring actions made in our method and only allow the **Random-s** to take exactly the same number of rewiring actions as ReWatt;
- **ReWatt-n** denotes a variant of the ReWatt, where the negative reward is fixed to -0.5 for all the graphs in the testing set;
- **ReWatt-a** is a variant of ReWatt, where we allow any nodes in the graph to be the third node v_{thi_t} instead of only 2-hop neighbors.

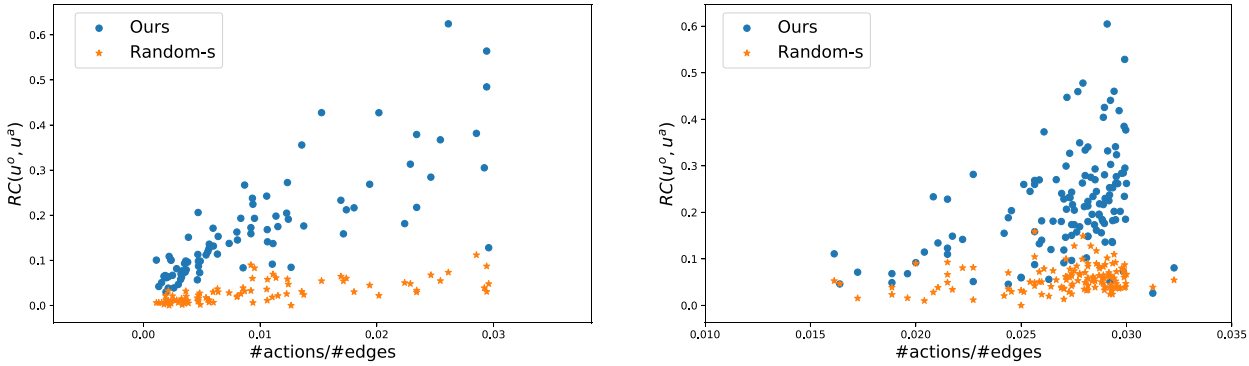
Note that there are other adversarial attack algorithms for graph neural networks, such as nettack [14] and metattack [15]. However, they have been designed to perform adversarial attacks specific to the node classification task. While in this work, we focus on attacking the graph classification task. Hence, we do not include them as baselines. As RL-S2V only allows a fixed budget for the all the graphs, when comparing to it, for ReWatt, we also fix the number of proposed rewiring operations to a fixed number K for all the graphs. A single proposed rewiring operation involves two edges. Thus, for a fair comparison, we allow RL-S2V to take $2K$ actions (adding/deleting edges). We set $K = 1, 2, 3$ in the experiments. To compare with the random selection method and the variants of ReWatt, we use flexible budget, more specially, we allow at most $p \cdot |\mathcal{E}_i|$ proposed rewiring operations for graph G_i. Here, p is a fixed percentage and we set it to $p = 1\%, 2\%, 3\%$ in our experiments. We use the success rate as measure to evaluate the performance of the attacker. A graph is said to be successfully attacked if its label is changed when it is modified within the given budget.

The results are shown in Table 2. We can make the following observations from the table.

- Compared to RL-S2V, ReWatt can perform more effective attacks. Especially, in the IMDB-MULTI data set, where ReWatt outperforms RL-S2V with a large margin;
- ReWatt outperforms the Random method as expected. Especially, ReWatt is much more effective than Random-s which performs exactly the same number of proposed rewiring operations ReWatt. This also indicates that the Random method uses more rewiring operations for successful attacking than ReWatt;
- The variant ReWatt-a outperforms ReWatt, which means if we do not constraint the rewiring operation to 2-hop neighbors, the performance of ReWatt can be further improved. However, as we discussed in earlier sections, this may lead to more "noticeable" changes of the graph structure;
- ReWatt-n performs worse than our ReWatt, which shows the advancement of using a flexible reward design.

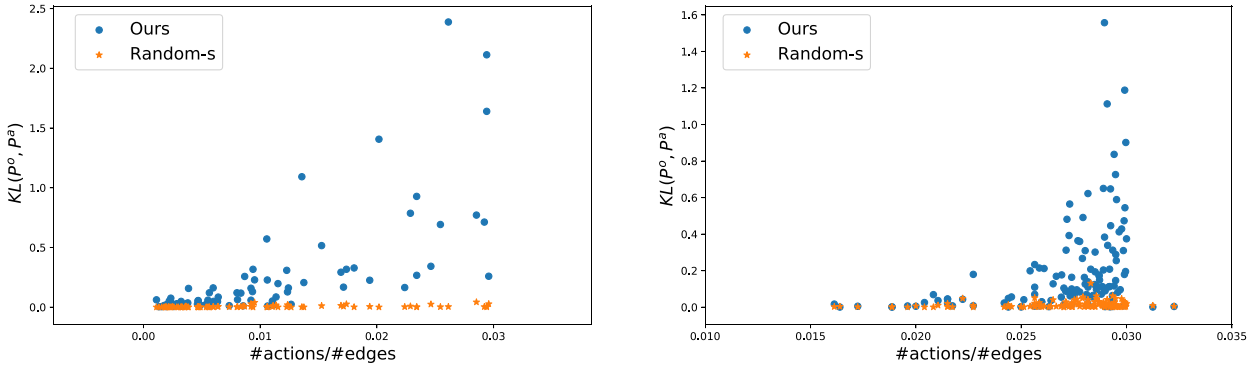
It is interesting to notice that RL-S2V has a larger search space than ReWatt, while its performance is not as good as ReWatt. With a larger action space, the optimal solution of should be as good or even better than that of ReWatt. However, both methods are not guaranteed to always achieve the optimal solution in the given action space. Next we discuss potential reasons on why ReWatt can outperform RL-S2V. More discussions are provided in the case study subsection.

- When performing an adding/deleting edge action in RL-S2V, it chooses two nodes sequentially. Then it decides to add an edge between two nodes if they are not connected, otherwise, the edge between them is removed. Since most graphs are very sparse, the RL-S2V algorithm is, by design, biased to adding an edge. However, ReWatt removes an edge and then adds another edge. The adding/deleting edge operations are more balanced.
- The reward design in ReWatt is different from RL-S2V. In RL-S2V, a non-zero reward is only given at the end of an attacking session. Specifically, at the end of an attacking session, a positive reward is given if the attack succeeded, otherwise a negative reward is given. All the intermediate steps get 0 reward. On the other hand, in ReWatt, the reward is given after each action. A positive reward is given once an action leads to a successful attack. A negative reward is penalized to take each action if it does not directly lead to a successful attack. This encourages the attacker to make as a few actions as possible. Furthermore, we also proposed an adaptive negative reward design, which determines the value of the negative reward according to the size of each graph. The advantage of this reward design has been demonstrated by the comparison between ReWatt and ReWatt-n in Table 2.



(a) Succeeded graphs (b) Failed graphs

Figure 2: The change of graph representation after attack



(a) Succeeded graphs (b) Failed graphs

Figure 3: The change of logits after attack

6.2 Attacker Analysis

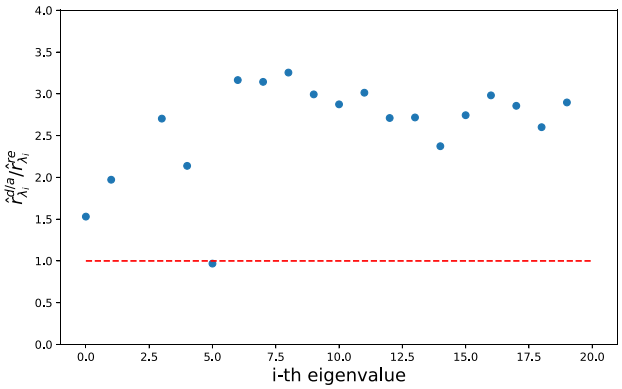


Figure 4: Comparison in the change of eigenvalues

In this subsection, we carry out experiments to analyze how ReWatt's change in graph structure affects the graph representation $\mathbf{u}$ calculated by (2) and the logits $\mathbf{P}$ (the output immediately after the softmax layer of the classifier). For convenience, we denote the original graph as G^o and the attacked graph as G^a in this subsection. Correspondingly, the graph representation and logits for the original (attacked) graph are denoted as $\mathbf{u}^o$ ($\mathbf{u}^a$) and $\mathbf{P}^o$ ($\mathbf{P}^a$), respectively. To measure the difference in graph representation, we used the relative difference in terms of 2-norm defined as $RC(\mathbf{u}^o, \mathbf{u}^a) = \frac{\|\mathbf{u}^a - \mathbf{u}^o\|_2}{\|\mathbf{u}^o\|_2}$. The logits denote the probability distribution that the given graph belongs to each of the classes. Thus, we use the KL-divergence [33] to measure the difference between the logits of the original and attacked graphs $KL(\mathbf{P}^o, \mathbf{P}^a) = \sum_{i=1}^{C} \mathbf{P}^o[i] \log\left(\frac{\mathbf{P}^o[i]}{\mathbf{P}^a[i]}\right)$, where C is the number of classes in the data set and $\mathbf{P}[i]$ denotes the logit for the i-th class. We perform the experiments on the REDDIT-MULTI-12K data set under the setting of allowing at most $3\% \cdot |\mathcal{E}|$ rewiring operations. The results for the graph representation and logits are shown in Figure 2 and Figure 3, respectively. The graphs in the test set are separated into two groups – one contains all the graphs successfully attacked by ReWatt (shown in Figure 2a and

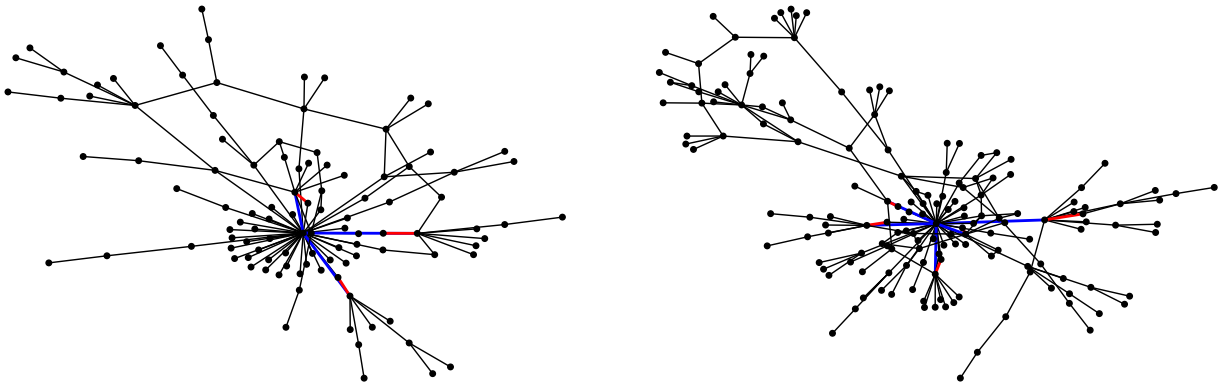


Figure 5: Case study of ReWatt attacker on two graphs sampled from REDDIT-MULTI-12K dataset.

Figure 3a), and the other contains those survived from ReWatt's attack (shown in Figure 2b and Figure 3b). Note that, for comparison, we also include the results of Random-s on these two groups of graphs. In these figures, a single point represents a test graph and the x-axis is the ratio $\frac{M}{|\mathcal{E}|}$, where M is the number of rewiring operations ReWatt used before the attacking process terminating. Note that M can be smaller than the budget as the process terminates once the attack successes. As we can observe from the figures, compared with the Random-s, ReWatt can make more changes to both the graph representation and logits, using exactly the same number of proposed rewiring operations. Comparing Figure 2a with Figure 2b, we find that the perturbation generated by ReWatt affects the graph representation a lot even when it fails to attack the graph. This means our attack is perturbing the graph structure in a right way to fool the classifier, although it fails potentially due to the limited budget. Similar observation can be made when we compare Figure 3a with Figure 3b.

6.3 Empirical Investigation of the Rewiring Operation

In this section, we conduct experiments to empirically show the advancements of the proposed rewiring operator compared with the adding/deleting edge operator. We compare them from two perspectives: 1) connectivity after the attack and 2) change in eigenvalues after the attack. The experiments are carried out on the REDDIT-MULTI-12K dataset. On each of the graph successfully attacked by ReWatt, we perform exactly the same number of deleting/adding edge operator on it. For connectivity, the average number of components in the clean graphs is 2.6, this number becomes 3.02 after the rewiring attack while it becomes 5.2 after the deleting/adding edges attack. On the other hand, only 20% of the graphs get more disconnected (having more components) after ReWatt attack than the original ones, while 87% of the graphs get more disconnected after the adding/deleting edges attack. Clearly, the rewiring operator is less likely to disconnect the graph. This is consistent with our theoretical understanding that ReWatt is likely to maintain the rank ($|\mathcal{V}| - \#components$) of the target graph.

The comparison of the change in eigenvalues is shown in Figure 4, where we compare the change in different eigenvalues of the graphs after these two attacks. Specifically, we first compute the average relative change in the i-th eigenvalue after both attacks as follows:

$$r_{\lambda_i} = \frac{|\lambda_i^{ori} - \lambda_i^{attack}|}{\lambda_i^{ori}}, \tag{11}$$

where λ_i^{ori} denotes the i-th eigenvalue of the clean graph while λ_i^{attack} denotes the i-th eigenvalue of the attacked graph. We then take the average of the above value over all the succeeded graphs, which we denoted as $\bar{r}_{\lambda_i}$. Specially, we use $\bar{r}^{re}_{\lambda_i}$ to denote the average change ratio after $ReWatt$ while using $\bar{r}^{d/a}_{\lambda_i}$ to denote the average change ratio after deleting/adding edge attack. To compare these two attacks, we calculate $\bar{r}^{d/a}_{\lambda_i}/\bar{r}^{re}_{\lambda_i}$ and the results are shown in Figure 4. The results show that in most of the cases, the deleting/adding edges attack makes much more changes to the eigenvalues as the value $\bar{r}^{d/a}_{\lambda_i}/\bar{r}^{re}_{\lambda_i}$ is way larger than 1. This observation is also aligned with the theoretical understanding.

By conducting these two experiments, we conclude that the proposed re-wiring operator makes more subtle changes to graphs than adding/deleting edges.

6.4 Case Study

We also conduct a case study to demonstrate how the attacker modifies the graph structure through rewiring operations.

Two representative graphs from REDDIT-MULTI-12K are shown in Figure 5, where the deleted edges are marked in blue while the added edges are marked in red. The rewiring operations are centered around the high degree nodes. Specifically, in both graphs shown in Figure 5, the rewiring operations all involve the nodes with the highest degree. We observe that the RL-S2V also take a similar strategy; however, it is biased to take the action of adding edges to the graph, while the proposed ReWatt performs rewiring operation, which involve both adding and deleting edges.

7 CONCLUSION

In this paper, we proposed a graph rewiring operation, which affect the graph structure in a less noticeable way than adding/deleting

edges. The rewiring operation preserves some basic graph properties such as number of nodes and number of edges. We then designed an attacker ReWatt based on the rewiring operations using reinforcement learning. Experiments in 3 real world data sets show the effectiveness of the proposed framework. Analysis on how the graph representation and logits change while the graph being attacked provide us with some insights of the attacker.

8 ACKNOWLEDGEMENTS

Yao Ma and Jiliang Tang are supported by the National Science Foundation (NSF) under grant numbers CNS1815636, IIS1928278, IIS1714741, IIS1845081, IIS1907704, IIS1955285, and Army Research Office (ARO) under grant number W911NF-21-1-0198. Suhang Wang is supported by National Science Foundation (NSF) under grant number IIS1955851 and Army Research Office (ARO) under grant number W911NF-21-1-0198.

REFERENCES

[1] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
[2] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.
[3] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.
[4] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.
[5] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning withdifferentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.
[6] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
[7] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.
[8] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
[9] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.
[10] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.
[11] Wei Jin, Yaxin Li, Han Xu, Yiqi Wang, and Jiliang Tang. Adversarial attacks and defenses on graphs: A review and empirical study. *arXiv preprint arXiv:2003.00653*, 2020.
[12] Huijun Wu, Chen Wang, Yuriy Tyshetskiy, Andrew Docherty, Kai Lu, and Liming Zhu. Adversarial examples for graph data: Deep insights into attack and defense.
[13] Han Xu, Yao Ma, Haochen Liu, Debayan Deb, Hui Liu, Jiliang Tang, and Anil Jain. Adversarial attacks and defenses in images, graphs and text: A review. *arXiv preprint arXiv:1909.08072*, 2019.
[14] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2847–2856. ACM, 2018.
[15] Daniel Zügner and Stephan Günnemann. Adversarial attacks on graph neural networks via meta learning. In *International Conference on Learning Representations*, 2019.
[16] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. Adversarial attack on graph structured data. In *International Conference on Machine Learning*, pages 1123–1132, 2018.
[17] Yiwei Sun, Suhang Wang, Xianfeng Tang, Tsung-Yu Hsieh, and Vasant Honavar. Node injection attacks on graphs via reinforcement learning. *arXiv preprint arXiv:1909.06543*, 2019.
[18] Benjamin A Miller, Mustafa Çamurcu, Alexander J Gomez, Kevin Chan, and Tina Eliassi-Rad. Improving robustness to attacks against vertex classification. 2019.
[19] Hau Chan and Leman Akoglu. Optimizing network robustness by edge rewiring: a general framework. *Data Mining and Knowledge Discovery*, 30(5):1395–1425, 2016.
[20] Arpita Ghosh and Stephen Boyd. Growing well-connected graphs. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 6605–6611. IEEE, 2006.
[21] Marinka Zitnik, Rok Sosič, Marcus W. Feldman, and Jure Leskovec. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences*, 116(10):4426–4433, 2019.
[22] Aleksandar Bojchevski and Stephan Günnemann. Adversarial attacks on node embeddings via graph poisoning. In *International Conference on Machine Learning*, pages 695–704, 2019.
[23] Xiaoyun Wang, Joe Eaton, Cho-Jui Hsieh, and Felix Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.
[24] Lichao Sun, Ji Wang, Philip S Yu, and Bo Li. Adversarial attack and defense on graph data: A survey. *arXiv preprint arXiv:1812.10528*, 2018.
[25] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.
[26] Bojan Mohar, Y Alavi, G Chartrand, and OR Oellermann. The laplacian spectrum of graphs. *Graph theory, combinatorics, and applications*, 2(871-898):12, 1991.
[27] Gilbert W Stewart. Matrix perturbation theory. 1990.
[28] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *arXiv preprint arXiv:1211.0053*, 2012.
[29] Aliaksei Sandryhaila and Jose MF Moura. Discrete signal processing on graphs: Frequency analysis. *IEEE Transactions on Signal Processing*, 62(12):3042–3054, 2014.
[30] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
[31] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.
[32] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015.
[33] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.