Adversarial Attack on Graph Structured Data

Hanjun Dai [1] **Hui Li** [2] **Tian Tian** [3] **Xin Huang** [2] **Lin Wang** [2] **Jun Zhu** [3] **Le Song** [1 2]

Abstract

Deep learning on graph structures has shown exciting results in various applications. However, few attentions have been paid to the robustness of such models, in contrast to numerous research work for image or text adversarial attack and defense. In this paper, we focus on the adversarial attacks that fool deep learning models by modifying the combinatorial structure of data. We first propose a reinforcement learning based attack method that learns the generalizable attack policy, while only requiring prediction labels from the target classifier. We further propose attack methods based on genetic algorithms and gradient descent in the scenario where additional prediction confidence or gradients are available. We use both synthetic and real-world data to show that, a family of Graph Neural Network models are vulnerable to these attacks, in both graph-level and node-level classification tasks. We also show such attacks can be used to diagnose the learned classifiers.

1. Introduction

Graph structure plays an important role in many real-world applications. Representation learning on the structured data with deep learning methods has shown promising results in various applications, including drug screening (Duvenaud et al., 2015), protein analysis (Hamilton et al., 2017), knowledge graph completion (Trivedi et al., 2017), *etc.*.

Despite the success of deep graph networks, the lack of interpretability and robustness of these models make it risky for some financial or security related applications. As analyzed in Akoglu et al. (2015), the graph information is proven to be important in the area of risk management. A graph sensitive evaluation model will typically take the user-user relationship into consideration: a user who connects with many high-credit users may also have high credit. Such heuristics learned by the deep graph methods would often yield good predictions, but could also put the model in a risk. A criminal could try to disguise himself by connecting other people using Facebook or Linkedin. Such ‘attack’ to the credit prediction model is quite cheap, but the consequence could be severe. Due to the large number of transactions happening every day, even if only one-millionth of the transactions are fraudulent, fraudsters can still obtain a huge benefit. However, few attentions have been put on domains involving graph structures, despite the recent advances in adversarial attacks and defenses for other domains like images (Goodfellow et al., 2014) and text (Jia & Liang, 2017).

So in this paper, we focus on the graph adversarial attack for a set of graph neural network(GNN) (Scarselli et al., 2009) models. These are a family of supervised (Dai et al., 2016) models that have achieved state-of-the-art results in many transductive tasks (Kipf & Welling, 2016) and inductive tasks (Hamilton et al., 2017). Through experiments in both node classification and graph classification problems, we will show that the adversarial samples do exist for such models. And the GNN models can be quite vulnerable to such attacks.

However, effectively attacking graph structures is a non-trivial problem. Different from images where the data is continuous, the graphs are discrete. Also the combinatorial nature of the graph structures makes it much more difficult than text. Inspired by the recent advances in combinatorial optimization (Bello et al., 2016; Dai et al., 2017), we propose a reinforcement learning based attack method that learns to modify the graph structure with only the prediction feedback from the target classifier. The modification is done by sequentially add or drop edges from the graph. A hierarchical method is also used to decompose the quadratic action space, in order to make the training feasible. Figure 1 illustrates this approach. We show that such learned agent can also propose adversarial attacks for new instances without access to the classifier.

Several different adversarial attack settings are considered in our paper. When more information from the target classifier is accessible, a variant of the gradient based method and a genetic algorithm based method are also presented. Here we mainly focus on the following three settings:

- **white box attack (WBA)**: in this case, the attacker is allowed to access any information of the target classifier, including the prediction, gradient information, *etc.*.
- **practical black box attack (PBA)**: in this case, only the prediction of the target classifier is available. When the prediction confidence is accessible, we denote this setting

[1]Georgia Institute of Technology [2]Ant Financial [3]Tsinghua University. Correspondence to: Hanjun Dai <hanjundai@gatech.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

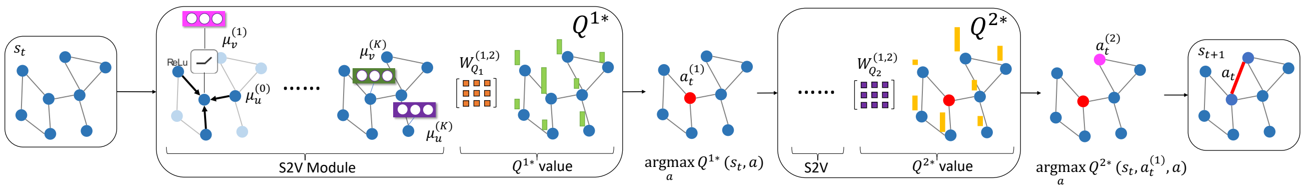

Figure 1. Illustration of applying hierarchical Q-function to propose adversarial attack solutions. Here adding a single edge a_t is decomposed into two decision steps $a_t^{(1)}$ and $a_t^{(2)}$, with two Q-functions Q^{1*} and Q^{2*}, respectively.

as PBA-C; if only the discrete prediction label is allowed, we denote the setting as PBA-D.

- **restrict black box attack (RBA)**: this setting is one step further than PBA. In this case, we can only do black-box queries on some of the samples, and the attacker is asked to create adversarial modifications to other samples.

As we can see, regarding the amount of information the attacker can obtain from the target classifier, we can sort the above settings as WBA > PBA-C > PBA-D > RBA. For simplicity, we focus on the non-targeted attack, though it is easy to extend to the targeted attack scenario.

In Sec 2, we first present the background about GNNs and two supervised learning tasks. Then in Sec 3 we formally define the graph adversarial attack problem. Sec 3.1 presents the attack method *RL-S2V* that learns the generalizable attack policy over the graph structure. We also propose other attack methods with different levels of access to the target classifier in Sec 3.2. We experimentally show the vulnerability of GNN models in Sec 4, and also present a way of doing defense against such attacks.

2. Background

A set of graphs is denoted by $\mathcal{G} = \{G_i\}_{i=1}^N$, where $|\mathcal{G}| = N$. Each graph $G_i = (V_i, E_i)$ is represented by the set of nodes $V_i = \{v_j^{(i)}\}_{j=1}^{|V_i|}$ and edges $E_i = \{\mathbf{e}_j^{(i)}\}_{j=1}^{|E_i|}$. Here the tuple $\mathbf{e}_j^{(i)} = (\mathbf{e}_{j,1}^{(i)}, \mathbf{e}_{j,2}^{(i)}) \in V_i \times V_i$ represents the edge between node $\mathbf{e}_{j,1}^{(i)}$ and $\mathbf{e}_{j,2}^{(i)}$. In this paper, we focus on undirected graphs, but it is straightforward to extend to directed ones. Optionally, the nodes or edges can have associated features. We denote them as $x(v_j^{(i)}) \in \mathbb{R}^{D_{node}}$ and $w(\mathbf{e}_j^{(i)}) = w(\mathbf{e}_{j,1}^{(i)}, \mathbf{e}_{j,2}^{(i)}) \in \mathbb{R}^{D_{edge}}$, respectively.

This paper works on attacking the graph supervised classification algorithms. Here two different supervised learning settings are considered:

Inductive Graph Classification: We associate each graph G_i with a label $y_i \in \mathcal{Y} = \{1, 2, \ldots, Y\}$, where Y is the number of categories. The dataset $\mathcal{D}^{(ind)} = \{(G_i, y_i)\}_{i=1}^N$ is represented by pairs of graph instances and graph labels. This setting is *inductive* since the test instances will never be seen during training. Examples of such task including classifying the drug molecule graphs according to their functionality. In this case, the classifier $f^{(ind)} \in \mathcal{F}^{(ind)} : \mathcal{G} \mapsto \mathcal{Y}$ is optimized to minimize the following loss:

$$\mathcal{L}^{(ind)} = \frac{1}{N} \sum_{i=1}^{N} L(f^{(ind)}(G_i), y_i) \tag{1}$$

where $L(\cdot,\cdot)$ is the cross entropy by default.

Transductive Node Classification: In node classification setting, a target node $c_i \in V_i$ of graph G_i is associated with a corresponding node label $y_i \in \mathcal{Y}$. The classification is on the nodes, instead of the entire graph. We here focus on the transductive setting, where only a single graph $G_0 = (V_0, E_0)$ is considered in the entire dataset. That is to say, $G_i = G_0, \forall G_i \in \mathcal{G}$. It is transductive since test nodes (but not their labels) are also observed during training. Examples in this case include classifying papers in a citation database like Citeseer, or entities in a social network like Facebook. Here the dataset is represented as $\mathcal{D}^{(tra)} = \{(G_0, c_i, y_i)\}_{i=1}^N$, and the classifier $f^{(tra)}(\cdot; G_0) \in \mathcal{F}^{(tra)} : V_0 \mapsto \mathcal{Y}$ minimizes the following loss:

$$\mathcal{L}^{(tra)} = \frac{1}{N} \sum_{i=1}^{N} L(f^{(tra)}(c_i; G_0), y_i) \tag{2}$$

When not causing confusion, we will overload the notations $\mathcal{D} = \{(G_i, c_i, y_i)\}_{i=1}^N$ to represent the dataset, and $f \in \mathcal{F}$ in either settings. In this case, c_i is implicitly omitted in inductive graph classification setting; While in transductive node classification setting, G_i always refers to G_0 implicitly.

GNN family models

The Graph Neural Networks (GNNs) define a general architecture for neural network on graph $G = (V, E)$. This architecture obtains the vector representation of nodes through an iterative process:

$$\begin{aligned} \mu_v^{(k)} = \quad & h^{(k)}\big(\{w(u,v), x(u), \mu_u^{(k-1)}\}_{u \in \mathcal{N}(v)}, \\ & x(v), \mu_v^{(k-1)}\big), k \in \{1, 2, \ldots, K\} \end{aligned} \tag{3}$$

where $\mathcal{N}(v)$ specifies the neighborhood of node $v \in V$. The initial node embedding $\mu_v^{(0)} \in \mathbb{R}^d$ is set to zero. For simplicity, we denote the outcome node embedding as $\mu_v = \mu_v^{(K)}$. To obtain the graph-level embedding from node embeddings, a global pooling is applied over the node embeddings.

The vanilla GNN model runs the above iteration until convergence. But recently, people find a fixed number of propagation steps T with various different parameterizations (Li et al., 2015; Dai et al., 2016; Gilmer et al., 2017; Lei et al., 2017) work quite well in various applications.

3. Graph adversarial attack

Given a learned classifier f and an instance from the dataset $(G,c,y) \in \mathcal{D}$, the graph adversarial attacker $g(\cdot,\cdot): \mathcal{G} \times \mathcal{D} \mapsto \mathcal{G}$ asks to modify the graph $G=(V,E)$ into $\tilde{G}=(\tilde{V},\tilde{E})$, such that

$$\begin{aligned} \max_{\tilde{G}} \quad & \mathbb{I}(f(\tilde{G},c) \neq y) \\ s.t. \quad & \tilde{G} = g(f,(G,c,y)) \\ & \mathcal{I}(G,\tilde{G},c) = 1. \end{aligned} \tag{4}$$

Here $\mathcal{I}(\cdot,\cdot,\cdot): \mathcal{G} \times \mathcal{G} \times V \mapsto \{0,1\}$ is an equivalency indicator that tells whether two graphs G and $\tilde{G}$ are equivalent under the classification semantics.

In this paper, we focus on the modifications to the discrete structures. The attacker g is allowed to add or delete edges from G to construct the new graph. Such type of actions are rich enough, since adding or deleting nodes can be performed by a series of modifications to the edges. Also modifying the edges is harder than modifying the nodes, since choosing a node only requires $O(|V|)$ complexity, while naively choosing an edge requires $O(|V|^2)$.

Since the attacker is aimed at fooling the classifier f, instead of actually changing the true label of the instance, the equivalency indicator should be defined first to restrict the modifications an attacker can perform. We use two ways to define the equivalency indicator:

1) Explicit semantics. In this case, a gold standard classifier f^* is assumed to be accessible. Thus the equivalency indicator $\mathcal{I}(\cdot,\cdot,\cdot)$ is defined as:

$$\mathcal{I}(G,\tilde{G},c) = \mathbb{I}(f^*(G,c) = f^*(\tilde{G},c)), \tag{5}$$

where $\mathbb{I}(\cdot) \in \{0,1\}$ is an indicator function.

2) Small modifications. In many cases when explicit semantics is unknown, we will ask the attacker to make as few modifications as possible within a neighborhood graph:

$$\begin{aligned} \mathcal{I}(G,\tilde{G},c) = & \mathbb{I}(|(E-\tilde{E}) \cup (\tilde{E}-E)| < m) \\ & \cdot \mathbb{I}(\tilde{E} \subseteq \mathcal{N}(G,b))). \end{aligned} \tag{6}$$

In the above equation, m is the maximum number of edges that allowed to modify, and $\mathcal{N}(G,b) = \{(u,v): u,v \in V, d^{(G)}(u,v) <= b\}$ defines the b-hop neighborhood graph, where $d^{(G)}(u,v) \in \{1,2,...\}$ is the distance between two nodes in graph G.

Take an example in friendship networks, a suspicious behavior would be adding or deleting many friends in a short period, or creating the friendship with someone who doesn't share any common friend. The "small modification" constraint eliminates the possibility of above two possibilities, so as to regulate the behavior of g. With either of the two realizations of robust classifier r, it is easy to enforce the attacker. Each time when an invalid modification proposed, the classifier can simply ignore such move.

Below we first introduce our main algorithm, *RL-S2V*, for learning attacker g in Section 3.1. Then in Section 3.2, we present other possible attack methods under different scenarios.

3.1. Attacking as hierarchical reinforcement learning

Given an instance (G,c,y) and a target classifier f, we model the attack procedure as a Finite Horizon Markov Decision Process $\mathcal{M}^{(m)}(f,G,c,y)$. The definition of such MDP is as follows:

- **Action** As we mentioned in Sec 3, the attacker is allowed to add or delete edges in the graph. So a single action at time step t is $a_t \in \mathcal{A} \subseteq V \times V$. However, simply performing actions in $O(|V|^2)$ space is too expensive. We will shortly show how to use hierarchical action to decompose this action space.
- **State** The state s_t at time t is represented by the tuple $(\hat{G}_t,c)$, where $\hat{G}_t$ is a partially modified graph with some of the edges added/deleted from G.
- **Reward** The purpose of the attacker is to fool the target classifier. So the non-zero reward is only received at the end of the MDP, with reward being

$$r\big((\tilde{G},c)\big) = \begin{cases} 1: f(\tilde{G},c) \neq y \\ -1: f(\tilde{G},c) = y \end{cases} \tag{7}$$

  In the intermediate steps of modification, no reward will be received. That is to say, $r(s_t,a_t) = 0, \forall t = 1,2,...,m-1$. In PBA-C setting where the prediction confidence of the target classifier is accessible, we can also use $r\big((\tilde{G},c)\big) = \mathcal{L}(f(\tilde{G},c),y)$ as the reward.
- **Terminal** Once the agent modifies m edges, the process stops. For simplicity, we focus on the MDP with fixed length. In the case when fewer modification is enough, we can simply let the agent to modify the dummy edges.

Given the above settings, a sample trajectory from this MDP will be: $(s_1,a_1,r_1,...,s_m,a_m,r_m,s_{m+1})$, where $s_1 = (G,c)$, $s_t = (\hat{G}_t,c), \forall t \in \{2,...,m\}$ and $s_{m+1} = (\tilde{G},c)$. The last step will have reward $r_m = r(s_m,a_m) = r\big((\tilde{G},c)\big)$ and all other intermediate rewards are zero: $r_t = 0, \forall t \in \{1,2,...,m-1\}$. Since this is a discrete optimization problem with a finite horizon, we use Q-learning to learn the MDPs. In our preliminary experiments we also tried with policy optimization methods like Advantage Actor Critic, but found Q-learning works more stable. So below we focus on the modeling with Q-learning.

Q-learning is an off-policy optimization where it fits the

Bellman optimality equation directly as below:

$$Q^*(s_t,a_t) = r(s_t,a_t) + \gamma \max_{a'} Q^*(s_{t+1},a'). \tag{8}$$

This implicitly suggests a greedy policy:

$$\pi(a_t|s_t;Q^*) = \arg\max_{a_t} Q^*(s_t,a_t). \tag{9}$$

In our finite horizon case, γ is fixed to 1. Note that directly operating the actions in $O(|V|^2)$ space is too expensive for large graphs. Thus we propose to decompose the action $a_t \in V \times V$ into $a_t = (a_t^{(1)}, a_t^{(2)})$, where $a_t^{(1)}, a_t^{(2)} \in V$. Thus a single edge action a_t is decomposed into two ends of this edge. The hierarchical Q-function is then modeled as below:

$$\begin{aligned} Q^{1*}(s_t,a_t^{(1)}) = & \max_{a_t^{(2)}} Q^{2*}(s_t,a_t^{(1)},a_t^{(2)}) \\ Q^{2*}(s_t,a_t^{(1)},a_t^{(2)}) = & \; r\big(s_t,a_t = (a_t^{(1)},a_t^{(2)})\big) + \\ & \max_{a_{t+1}^{(1)}} Q^{1*}(s_t,a_{t+1}^{(1)}). \end{aligned} \tag{10}$$

In the above formulation, Q^{1*} and Q^{2*} are two functions that implement the original Q^*. An action is considered as completed only when a pair of $(a_t^{(1)}, a_t^{(2)})$ is chosen. Thus the reward will only be valid after $a_t^{(2)}$ is made. It is easy to see that such decomposition has the same optimality structure as in Eq (8), but making an action would only require $O(2 \times |V|) = O(|V|)$ complexity. Figure 1 illustrates this process.

Take a further look at Eq (10), since only the reward in last time step is non-zero, and also the budget of modification m is given, we can explicitly unroll the Bellman equations as:

$$\begin{aligned} Q^*_{1,1}(s_1,a_1^{(1)}) = & \max_{a_1^{(2)}} Q^*_{1,2}(s_1,a_1^{(1)},a_1^{(2)}) \\ Q^*_{1,2}(s_1,a_1^{(1)},a_1^{(2)}) = & \max_{a_2^{(1)}} Q^*_{2,1}(s_2,a_2^{(1)}) \\ & \ldots \\ Q^*_{m,1}(s_m,a_m^{(1)}) = & \max_{a_m^{(2)}} Q^*_{m,2}(s_m,a_m^{(1)},a_m^{(2)}) \\ Q^*_{m,2}(s_m,a_m^{(1)},a_m^{(2)}) = & \; r(\tilde{G},c) \end{aligned} \tag{11}$$

To make notations compact, we still use $Q^* = \{Q^*_{t,1|2}\}_{t=1}^m$ to denote the Q-function. Since each sample in the dataset defines an MDP, it is possible to learn a separate Q function for each MDP $M_i^{(m)}(f, G_i, c_i, y_i), i = 1, \ldots, N$. However, we here focus on a more practical and challenging setting, where only one Q^* is learned. The learned Q-function is thus asked to generalize or transfer over all the MDPs:

$$\max_{\boldsymbol{\theta}} \sum_{i=1}^N \mathbb{E}_{t, a = \arg\max_{a_t} Q^*(a_t|s_t;\boldsymbol{\theta})} \left[r\big((\tilde{G}_i, c_i) \big) \right], \tag{12}$$

where Q^* is parameterized by $\boldsymbol{\theta}$. Below we present the parameterization for such Q^* that generalizes over MDPs.

3.1.1. Parameterization of Q^*

From above, we can see the most flexible parameterization would be implementing $2 \times m$ time-dependent Q functions. However, we found two distinct parametrization is typically enough, *i.e.*, $Q^*_{t,1} = Q^{1*}, Q^*_{t,2} = Q^{2*}, \forall t$.

Since the Q function is scoring the nodes in the state graph, it is natural to use GNN family models for parameterization, in order to learn a generalizable attacker. Specifically, Q^{1*} is parameterized as:

$$Q^{1*}(s_t,a_t^{(1)}) = W_{Q_1}^{(1)} \sigma\big(W_{Q_1}^{(2)\top} [\mu_{a_t^{(1)}}, \mu(s_t)]\big), \tag{13}$$

where $\mu_{a_t^{(1)}}$ is the embedding of node $a_t^{(1)}$ in graph $\hat{G}_t$, obtained by structure2vec (S2V) (Dai et al., 2016):

$$\mu_v^{(k)} = \text{relu}(W_{Q_1}^{(3)} x(v) + W_{Q_1}^{(4)} \sum_{u \in \mathcal{N}(v)} \mu_u^{(k-1)}), \tag{14}$$

where $\mu_v = \mu_v^{(K)}$ and $\mu_v^{(0)} = 0$. Also $\mu(s_t) = \mu(\hat{G}_t = (\hat{V}_t, \hat{E}_t), c)$ is the representation of entire state tuple:

$$\mu(s_t) = \begin{cases} \sum_{v \in \hat{V}} \mu_v : \text{graph attack} \\ [\sum_{v \in \mathcal{N}_{\hat{G}_t}(c,b)} \mu_v, \mu_c] : \text{node attack} \end{cases} \tag{15}$$

In node attack scenario, the state embedding is taken from the b-hop neighborhood of node c, denoted as $\mathcal{N}_{\hat{G}_t}(c,b)$. The parameter set of Q^{1*} is $\theta_1 = \{W_{Q_1}^{(i)}\}_{i=1}^4$. Q^{2*} is parameterized similarly with parameter θ_2, with an extra consideration of the chosen node $a_t^{(1)}$:

$$Q^{2*}(s_t,a_t^{(1)},a_t^{(2)}) = W_{Q_2}^{(1)} \sigma\big(W_{Q_2}^{(2)\top} [\mu_{a_t^{(1)}}, \mu_{a_t^{(2)}}, \mu(s_t)]\big) \tag{16}$$

We denote this method as *RL-S2V* since it learns a Q-function parameterized by S2V to perform attack.

3.2. Other attacking methods

The *RL-S2V* is suitable for black-box attack and transfer. However, for different attack scenarios, other algorithms might be preferred. We first introduce *RandSampling* that requires least information in Sec 3.2.1; Then in Sec 3.2.2, a white-box attack *GradArgmax* is proposed; Finally the *GeneticAlg*, which is a kind of evolutionary computing, is proposed in Sec 3.2.3.

3.2.1. Random sampling

This is the simplest attack method that randomly adds or deletes edges from graph G. When an edge modification action $a_t = (u,v)$ is sampled, we will only accept it when it satisfies the semantic constraint $\mathcal{I}(\cdot,\cdot,\cdot)$. It requires the least information for attack. Despite its simplicity, sometimes it can get good attack rate.

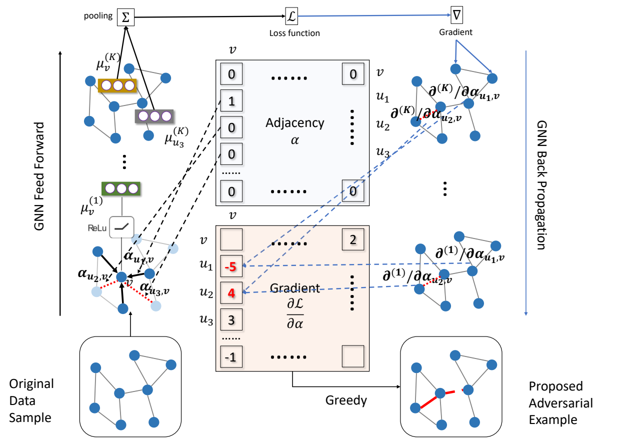

Figure 2. Illustration of graph structure gradient attack. This white-box attack adds/deletes the edges with maximum gradient (with respect to α) magnitudes.

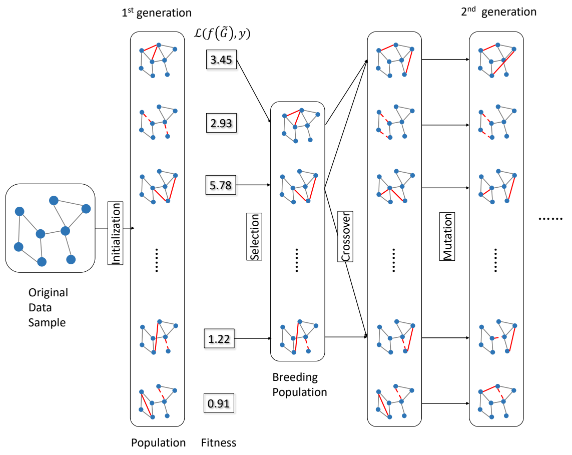

Figure 3. Illustration of attack using genetic algorithm. The population evolves with selection, crossover and mutation operations. Fitness is measured by the loss function.

3.2.2. Gradient based white box attack

Gradients have been successfully used for modifying continuous inputs, *e.g.*, images. However, taking gradient with respect to a discrete structure is non-trivial. Recall the general iterative embedding process defined in Eq (3), we associate a coefficient $\alpha_{u,v}$ for each pair of $(u,v) \in V \times V$:

$$\mu_v^{(k)} = h^{(k)}\Big(\{\alpha_{u,v}\left[w(u,v), x(u), \mu_u^{(k-1)}\right]\}_{u \in \mathcal{N}(v)} \cup \{\alpha_{u',v}\left[w(u',v), x(u'), \mu_{u'}^{(k-1)}\right]\}_{u' \notin \mathcal{N}(v)}, x(v), \mu_v^{(k-1)}\Big), k \in \{1,2,...,K\} \quad (17)$$

Let $\alpha_{u,v} = \mathbb{I}(u \in \mathcal{N}(v))$. That is to say, α itself is the binary adjacency matrix. It is easy to see that the above formulation has the same effect as in Eq (3). However, such additional coefficients give us the gradient information with respect to each edge (either existing or non-existing):

$$\frac{\partial \mathcal{L}}{\partial \alpha_{u,v}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}}{\mu_k}^{\top} \cdot \frac{\partial \mu_k}{\partial \alpha_{u,v}}. \quad (18)$$

In order to attack the model, we could perform the gradient ascent, *i.e.*, $\alpha_{u,v} \leftarrow \alpha_{u,v} + \eta \frac{\partial \mathcal{L}}{\partial \alpha_{u,v}}$. However, the attack is on a discrete structure, where only m edges are allowed to be added or deleted. So here we need to solve a combinatorial optimization problem:

$$\begin{aligned} \max_{\{u_t,v_t\}_{t=1}^m} \quad & \sum_{t=1}^{m} \left|\frac{\partial \mathcal{L}}{\partial \alpha_{u_t,v_t}}\right| \\ s.t. \quad & \tilde{G} = \text{Modify}(G, \{\alpha_{u_t,v_t}\}_{t=1}^m) \\ & \mathcal{I}(G, \tilde{G}, c) = 1. \end{aligned} \quad (19)$$

We simply use a greedy algorithm to solve the above optimization. Here the modification of G given a set of coefficients $\{\alpha_{u_t,v_t}\}_{t=1}^m$ is performed by sequentially modifying edges (u_t, v_t) of graph $\hat{G}_t$:

$$\hat{G}_{t+1} = \begin{cases} (\hat{V}_t, \hat{E}_t \setminus (u_t, v_t)) : \frac{\partial \mathcal{L}}{\partial \alpha_{u_t,v_t}} < 0 \\ (\hat{V}_t, \hat{E}_t \cup \{(u_t, v_t)\}) : \frac{\partial \mathcal{L}}{\partial \alpha_{u_t,v_t}} > 0 \end{cases} \quad (20)$$

That is to say, we modify the edges who are most likely to cause the change to the objective. Depending on the sign of the gradient, we either add or delete the edge. We name it as *GradArgmax* since it does the greedy selection based on gradient information.

The attack procedure is shown in Figure 2. Since this approach requires the gradient information, we consider it as a white-box attack method. Also, the gradient considers all pairs of nodes in a graph, the computation cost is at least $O(|V|^2)$, excluding the back-propagation of gradients in Eq (18). Without further approximation, this approach cannot scale to large graphs.

3.2.3. Genetic algorithm

Evolution computing has been successfully applied in many zero-order optimization scenarios, including neural architecture search (Real et al., 2017; Miikkulainen et al., 2017) and adversarial attack for images (Su et al., 2017). We here propose a black-box attack method that implements a type of genetic algorithms. Given an instance (G, c, y) and the target classifier f, Such algorithm involves five major components, as elaborated below:

- **Population**: the population refers to a set of candidate solutions. Here we denote it as $\mathcal{P}^{(r)} = \{\hat{G}_j^{(r)}\}_{j=1}^{|\mathcal{P}^{(r)}|}$, where each $\hat{G}_j^{(r)}$ is a valid modification solution to the original graph G. $r = 1,2,...,R$ is the index of generation and R is the maximum numbers of evolutions allowed.
- **Fitness**: each candidate solution in current population will get a score that measures the quality of the solution. We use the loss function of target model $\mathcal{L}(f(\hat{G}_j^{(r)}, c), y)$

Table 1. Application scenarios for different proposed graph attack methods. Cost is measured by the time complexity for proposing a single attack.

	WBA	PBA-C	PBA-D	RBA	Cost
RandSampling			✓	✓	$O(1)$
GradArgmax	✓				$O(\|V\|^2)$
GeneticAlg		✓			$O(\|V\|+\|E\|)$
RL-S2V		✓	✓	✓	$O(\|V\|+\|E\|)$

as the score function. A good attack solution should increase such loss. Since the fitness is a continuous score, it is not applicable in PBA-D setting, where only classification label is accessible.

- **Selection**: Given the fitness scores of current population, we can either do weighted sampling or greedy selection to select the 'breeding' population $\mathcal{P}_b^{(r)}$ for next generation.
- **Crossover**: After the selection of $\mathcal{P}_b^{(r)}$, we randomly pick two candidates $\hat{G}_1,\hat{G}_2 \in \mathcal{P}_b^{(r)}$ and do the crossover by mixing the edges from these two candidates:

$$\hat{G}' = (V,(\hat{E}_1 \cap \hat{E}_2) \cup \text{rp}(\hat{E}_1 \setminus \hat{E}_2) \cup \text{rp}(\hat{E}_2 \setminus \hat{E}_1)). \quad (21)$$

  Here rp$(\cdot)$ means randomly picking a subset.
- **Mutation**: the mutation process is also biology inspired. For a candidate solution $\hat{G} \in \mathcal{P}^{(r)}$, suppose the modified edges are $\delta E = \{(u_t,v_t)\}_{t=1}^m$. Then for each edge (u_t,v_t), we have a certain probability to change it to either (u_t,v') or (u',v_t).

The population size $|\mathcal{P}^{(r)}|$, the probability of crossover used in rp$(\cdot)$, the mutation probability and the number of evolutions R are all hyper-parameters that can be tuned. Due to the limitation of the fitness function, this method can only be used in the PBA-C setting. Also since we need to execute the target model f to get fitness scores, the computation cost of such genetic algorithm is $O(|V|+|E|)$, which is mainly made up by the computation cost of GNNs. The overall procedure is illustrated in Figure 3. We simply name it as *GeneticAlg* since it is an instantiation of general genetic algorithm framework.

4. Experiment

For *GeneticAlg*, we set the population size $|\mathcal{P}| = 100$ and the number of rounds $R = 10$. We tune the crossover rate and mutation rate in $\{0.1,...,0.5\}$. For *RL-S2V*, we tune the number of propagations of its S2V model $K = \{1,...,5\}$. There is no parameter tuning for *GradArgmax* and *RandSampling*.

We use the proposed attack methods to attack the graph classification model in Sec 4.1 and node classification model in Sec 4.2. In each scenario, we first show the attack rate when queries are allowed for target model, then we show the generalization ability of the *RL-S2V* for RBA setting.

4.1. Graph-level attack

In this set of experiments, we use synthetic data, where the gold classifier f^* is known. Thus the explicit semantics is used for the equivalency indicator $\mathcal{I}$. The dataset $\mathcal{D}^{(ind)}$ we constructed contains 15,000 graphs, generated with Erdos-Renyi random graph model. It is a three class graph classification task, where each class contains 5,000 graphs. The classifier is asked to tell how many connected components are there in the corresponding undirected graph G. The label set $\mathcal{Y} = \{1,2,3\}$. So there could be up to 3 components in a graph. See Figure 4 for illustration. The gold classifier f^* is obtained by performing a one-time traversal of the entire graph. The dataset is divided into training and two test sets. The test set I contains 1,500 graphs, while test set II contains 150 graphs. Each set contains the same number of instances from different classes.

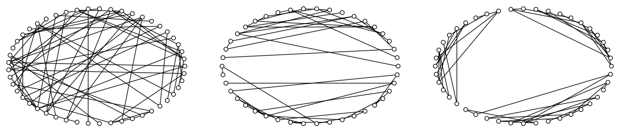
(a) # comp = 1 (b) # comp = 2 (c) # comp = 3

Figure 4. Example graphs for classification. Here we show three graphs with 1, 2, or 3 components, with 40-50 nodes.

We choose structure2vec as the target model for attack. We also tune its number of propagation parameter $K = \{2,...,5\}$. Table 2 shows the results with different settings. For test set I, we can see the structure2vec achieves very high accuracy on distinguishing the number of connected components. Also increasing K seems to improve the generalization in most cases. However, we can see under the practical black-box attack scenario, the *GeneticAlg* and *RL-S2V* can bring down the accuracy to $40\% \sim 60\%$. In attacking the graph classification algorithm, the *GradArgmax* seems not to be very effective. One reason could be the last pooling step in S2V when obtaining graph-level embedding. During back propagation, the pooling operation will dispatch the gradient to every other node embeddings, which makes the $\frac{\partial \mathcal{L}}{\partial \alpha}$ looks similar in most entries.

For restrict black-box attack on test set II (see the lower half of Table 2), the attacker is asked to propose adversarial samples without any access to the target model. Since *RL-S2V* is learned on test set I, it is able to transfer its learned policy to test set II. This suggests that the target classifier makes some form of consistent mistakes.

This experiment shows that, (1) the adversarial examples do exist for supervised graph problems; (2) a model with good generalization ability can still suffer from adversarial attacks; (3) *RL-S2V* can learn the transferrable adversarial policy to attack unseen graphs.

4.2. Node-level attack

In this experiment, we want to inspect the adversarial attack to the node classification problems. Different from Sec 4.1, here the setting is transductive, where the test samples (but not their labels) are also seen during training. Here we use four real-world datasets, namely the Citeseer, Cora, Pubmed and Finance. The first three are small-scaled citation

Table 2. Attack graph classification algorithm. We report the 3-class classification accuracy of target model on the vanilla test set I and II, as well as adversarial samples generated. The upper half of the table reports the attack results on test set I, with different levels of access to the information of target classifier. The lower half reports the results of RBA setting on test set II where only *RandSampling* and *RL-S2V* can be used. K is the number of propagation steps used in GNN family models (see Eq (3)).

attack test set I		15-20 nodes				40-50 nodes				90-100 nodes			
Settings	Methods	$K=2$	$K=3$	$K=4$	$K=5$	$K=2$	$K=3$	$K=4$	$K=5$	$K=2$	$K=3$	$K=4$	$K=5$
╲	(unattacked)	93.20%	98.20%	98.87%	99.07%	92.60%	96.20%	97.53%	97.93%	94.60%	97.47%	98.73%	98.20%
RBA	*RandSampling*	78.73%	92.27%	95.13%	97.67%	73.60%	78.60%	82.80%	85.73%	74.47%	74.13%	80.93%	82.80%
WBA	*GradArgmax*	69.47%	64.60%	95.80%	97.67%	73.93%	64.80%	70.53%	75.47%	72.00%	66.20%	67.80%	68.07%
PBA-C	*GeneticAlg*	39.87%	39.07%	65.33%	85.87%	59.53%	55.67%	53.70%	42.48%	65.47%	63.20%	61.60%	61.13%
PBA-D	*RL-S2V*	42.93%	41.93%	70.20%	91.27%	61.00%	59.20%	58.73%	49.47%	66.07%	64.07%	64.47%	64.67%
						Restricted black-box attack on test set II							
╲	(unattacked)	94.67%	97.33%	98.67%	97.33%	94.67%	97.33%	98.67%	98.67%	96.67%	98.00%	99.33%	98.00%
RBA	*RandSampling*	78.00%	91.33%	94.00%	98.67%	75.33%	84.00%	86.00%	87.33%	69.33%	73.33%	76.00%	80.00%
RBA	*RL-S2V*	44.00%	40.00%	67.33%	92.00%	58.67%	60.00%	58.00%	44.67%	62.67%	62.00%	62.67%	61.33%

Table 3. Statistics of the graphs used for node classification.

Dataset	Nodes	Edges	Classes	Train/Test I/Test II
Citeseer	3,327	4,732	6	120/1,000/500
Cora	2,708	5,429	7	140/1,000/500
Pubmed	19,717	44,338	3	60/1,000/500
Finance	2,382,980	8,101,757	2	317,041/812/800

networks commonly used for node classification, where each node is a paper with corresponding bag-of-words features. The last one is a large-scale dataset that contains transactions from an e-commerce within one day, where the node set contains buyers, sellers and credit cards. The classifier is asked to distinguish the normal transactions from abnormal ones. The statistics of each dataset is shown in Table 3. The nodes also contain features with different dimensions. For the full table please refer to Kipf & Welling (2016). We use GCN (Kipf & Welling, 2016) as the target model to attack. Here the "small modifications" is used to regulate the attacker. That is to say, given a graph G and target node c, the adversarial samples are limited to delete single edge within 2-hops of node c.

Table 4 shows the results. We can see although deleting a single edge is the minimum modification one can do to the graph, the attack rate is still about 10% on those small graphs, and 4% in the Finance dataset. We also ran an exhaustive attack as sanity check, which is the best any algorithm can do under the attack budget. The classifier accuracy will reduce to 60% or lower if two-edge modification is allowed. However, consider the average degree in the graph is not large, deleting two or more edges would violate the "small modification" constraints. We need to be careful to only create adversarial samples, instead of actually changing the true label of that sample.

In this case, the *GradArgmax* performs quite good, which is different from the case in graph-level attack. Here the gradient with respect to the adjacency matrix α is no longer averaged, which makes it easier to distinguish the useful modifications. For the restrict black-box attack on test set II, the *RL-S2V* still learns an attack policy that generalizes to unseen samples. Though we do not have gold classifier in real-world datasets, it is highly possible that the adversarial samples proposed are valid: (1) the structure modification is tiny and within 2-hop; (2) we did not modify the node features.

Table 4. Attack node classification algorithm. In the upper half of the table, we report target model accuracy before/after the attack on the test set I, with various settings and methods. In the lower half, we report accuracy on test set II with RBA setting only. In this second part, only *RandSampling* and *RL-S2V* can be used.

Method	Citeseer	Cora	Pubmed	Finance
(unattacked)	71.60%	81.00%	79.90%	88.67%
RBA, *RandSampling*	67.60%	78.50%	79.00%	87.44%
WBA, *GradArgmax*	63.00%	71.30%	72.4%	86.33%
PBA-C, *GeneticAlg*	63.70%	71.20%	72.30%	85.96%
PBA-D, *RL-S2V*	62.70%	71.20%	72.80%	85.43%
Exhaust	62.50%	70.70%	71.80%	85.22%
Restricted black-box attack on test set II				
(unattacked)	72.60%	80.20%	80.40%	91.88%
RandSampling	68.00%	78.40%	79.00%	90.75%
RL-S2V	66.00%	75.00%	74.00%	89.10%
Exhaust	62.60%	70.80%	71.00%	88.88%

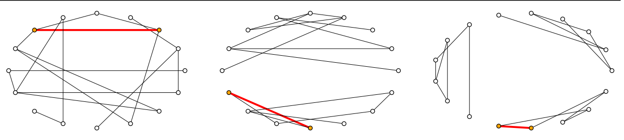
(a) pred = 2 (b) pred = 1 (c) pred = 2

Figure 5. Attack solutions proposed by *RL-S2V* on graph classification problem. Target classifier is structure2vec with $K=4$. The ground truth # components are: (a) 1 (b) 2 (c) 3.

4.3. Inspection of adversarial samples

In this section, we visualize the adversarial samples proposed by different attackers. The solutions proposed by *RL-S2V*

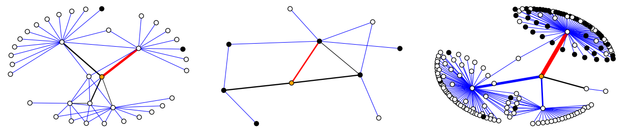

Figure 6. Attack solutions proposed by *GradArgmax* on node classification problem. Attacked node is colored orange. Nodes from the same class as the attacked node are marked black, otherwise white. Target classifier is GCN with $K = 2$.

Table 5. Results after adversarial training by random edge drop.

Method	Citeseer	Cora	Pubmed	Finance
(unattacked)	71.30%	81.70%	79.50%	88.55%
RBA, *RandSampling*	67.70%	79.20%	78.20%	87.44%
WBA, *GradArgmax*	63.90%	72.50%	72.40%	87.32%
PBA-C, *GeneticAlg*	64.60%	72.60%	72.50%	86.45%
PBA-D, *RL-S2V*	63.90%	72.80%	72.90%	85.80%

for graph-level classification problem are shown in Figure 5. The ground truth labels are 1, 2, 3, while the target classifier mistakenly predicts 2, 1, 2, respectively. In Figure 5(b) and (c), the RL agent connects two nodes who are 4 hops away from each other (before the red edge is added). This shows that, although the target classifier structure2vec is trained with $K = 4$, it didn't capture the 4-hop information efficiently. Also Figure 5(a) shows that, even connecting nodes who are just 2-hop away, the classifier makes mistake on it.

Figure 6 shows the solutions proposed by *GradArgmax*. Orange node is the target node for attack. Edges with blue color are suggested to be added by *GradArgmax*, while black ones are suggested to be deleted. Black nodes have the same node label as the orange node, while while nodes do not. The thicker the edge, the larger the magnitude of the gradient is. Figure 6(b) deletes one neighbor with the same label, but still have other black nodes connected. In this case, the GCN is over-sensitive. The mistake made in Figure 6(c) is reasonable, since although the red edge does not connect two nodes with the same label, it connects to a large community of nodes from the same class in 2-hop distance. In this case, the prediction made by GCN is reasonable.

4.4. Defense against attacks

Different from the images, here the possible number of graph structures is finite given the number of nodes. So by adding the adversarial samples back for further training, the improvement of the target model's robustness can be expected. For example, in the experiment of Sec 4.1, adding adversarial samples for training is equivalent to increasing the size of the training set, which will definitely be helpful. So here we seek to use a cheap method for adversarial training — simply doing edge drop during training for defense.

Dropping the edges during training is different from Dropout (Srivastava et al., 2014). Dropout operates on the neurons in the hidden layers, while edge drop modifies the discrete structure. It is also different from simply drop the entire hidden vector, since deleting a single edge can affect more than just one edge. For example, GCN computes the normalized graph Laplacian. So after deleting a single edge, the normalized graph Laplacian needs to be recomputed for some entries. This approach is similar to Hamilton et al. (2017), who samples a fixed number of neighborhoods during training for the efficiency. Here we drop the edges globally at random, during each training step.

The new results after adversarial training are presented in Table 5. We can see from the table that, though the accuracy of target model remains similar, the attack rate of various methods decreases about 1%. Though the scale of the improvement is not significant, it shows some effectiveness with such cheap adversarial training.

5. Related work

Adversarial attack in continuous and discrete space: In recent years, the adversarial attacks to the deep learning models have raised increasing attention from researchers. Some methods focus on the white-box adversarial attack using gradient information, like box constrained L-BFGS (Szegedy et al., 2013), Fast Gradient Sign (Goodfellow et al., 2014), deep fool (Moosavi-Dezfooli et al., 2016), *etc.*. When the full information of target model is not accessible, one can train a substitute model(Papernot et al., 2017), or use zero-order optimization method (Chen et al., 2017). There are also some works working on the attack in discrete data space. The one-pixel attack (Su et al., 2017) modifies the image by only several pixels using differential evolution; Jia & Liang (2017) attacks the text reading comprehension system with the help of rules and human efforts. Zügner et al. (2018) studies the problem of adversarial attack over graphs in parallel to our work, although with very different methods.

Combinatorial optimization: Modifying the discrete structure to fool the target classifier can be treated as a combinatorial optimization problem. Recently, there are some exciting works using reinforcement learning to learn to solve the general sequential decision problems (Bello et al., 2016) or graph combinatorial problems (Dai et al., 2017). These are closely related to *RL-S2V*. The *RL-S2V* extends the previous approach using hierarchical way to decompose the quadratic action space, in order to make the training feasible.

6. Conclusion

In this paper, we study the adversarial attack on graph structured data. To perform the efficient attack, we proposed three methods, namely *RL-S2V*, *GradArgmax* and *GeneticAlg* for three different attack settings, respectively. We show that a family of GNN models are vulnerable to such attack. By visualizing the attack samples, we can also inspect the target classifier. We also discussed about defense methods through experiments. Our future work includes developing more effective defense algorithms.

Acknowledgements

This project was supported in part by NSF IIS-1218749, NIH BIGDATA 1R01GM108341, NSF CAREER IIS-1350983, NSF IIS-1639792 EAGER, NSF CNS-1704701, ONR N00014-15-1-2340, Intel ISTC, NVIDIA and Amazon AWS. Tian Tian and Jun Zhu were supported by the National NSF of China (No. 61620106010) and Beijing Natural Science Foundation (No. L172037). We thank Bo Dai for valuable suggestions, and the anonymous reviewers who gave useful comments.

References

Akoglu, Leman, Tong, Hanghang, and Koutra, Danai. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

Bello, Irwan, Pham, Hieu, Le, Quoc V, Norouzi, Mohammad, and Bengio, Samy. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Buckman, Jacob, Roy, Aurko, Raffel, Colin, and Goodfellow, Ian. Thermometer encoding: One hot way to resist adversarial examples. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=S18Su--CW.

Chen, Pin-Yu, Zhang, Huan, Sharma, Yash, Yi, Jinfeng, and Hsieh, Cho-Jui. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp. 15–26. ACM, 2017.

Dai, Hanjun, Dai, Bo, and Song, Le. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

Dai, Hanjun, Khalil, Elias B, Zhang, Yuyu, Dilkina, Bistra, and Song, Le. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Duvenaud, David K, Maclaurin, Dougal, Iparraguirre, Jorge, Bombarell, Rafael, Hirzel, Timothy, Aspuru-Guzik, Alán, and Adams, Ryan P. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, pp. 2215–2223, 2015.

Gilmer, Justin, Schoenholz, Samuel S, Riley, Patrick F, Vinyals, Oriol, and Dahl, George E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Goodfellow, Ian J, Shlens, Jonathon, and Szegedy, Christian. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Hamilton, William L, Ying, Rex, and Leskovec, Jure. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

Jia, Robin and Liang, Percy. Adversarial examples for evaluating reading comprehension systems. *arXiv preprint arXiv:1707.07328*, 2017.

Kipf, Thomas N and Welling, Max. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Lei, Tao, Jin, Wengong, Barzilay, Regina, and Jaakkola, Tommi. Deriving neural architectures from sequence and graph kernels. *arXiv preprint arXiv:1705.09037*, 2017.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, and Zemel, Richard. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Miikkulainen, Risto, Liang, Jason, Meyerson, Elliot, Rawal, Aditya, Fink, Dan, Francon, Olivier, Raju, Bala, Navruzyan, Arshak, Duffy, Nigel, and Hodjat, Babak. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

Moosavi-Dezfooli, Seyed-Mohsen, Fawzi, Alhussein, and Frossard, Pascal. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2574–2582, 2016.

Papernot, Nicolas, McDaniel, Patrick, Goodfellow, Ian, Jha, Somesh, Celik, Z Berkay, and Swami, Ananthram. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 506–519. ACM, 2017.

Real, Esteban, Moore, Sherry, Selle, Andrew, Saxena, Saurabh, Suematsu, Yutaka Leon, Le, Quoc, and Kurakin, Alex. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

Scarselli, Franco, Gori, Marco, Tsoi, Ah Chung, Hagenbuchner, Markus, and Monfardini, Gabriele. The graph neural network model. *Neural Networks, IEEE Transactions on*, 20(1):61–80, 2009.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Su, Jiawei, Vargas, Danilo Vasconcellos, and Kouichi, Sakurai. One pixel attack for fooling deep neural networks. *arXiv preprint arXiv:1710.08864*, 2017.

Szegedy, Christian, Zaremba, Wojciech, Sutskever, Ilya, Bruna, Joan, Erhan, Dumitru, Goodfellow, Ian, and Fergus, Rob. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Trivedi, Rakshit, Dai, Hanjun, Wang, Yichen, and Song, Le. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*, 2017.

Zügner, Daniel, Akbarnejad, Amir, and Günnemann, Stephan. Adversarial attacks on neural networks for graph data. In *KDD*, 2018.