Survey on Applications of Graph Neural Networks in Cyber Security

Thesis submitted in partial fulfillment of requirements
For the degree of
Master of Technology in Computer Technology
of
Computer Science and Engineering Department
of
Jadavpur University

by

Mohammed Saif
Regn. No. - 149840 of 2019-2020
Exam Roll No.-M6TCT22007

under the supervision of

Mridul Sankar Barik
Assistant Professor

Department of Computer Science and Engineering
Jadavpur University
Kolkata, West Bengal, India
2022

Certificate from the Supervisor

This is to certify that the work embodied in this thesis entitled **"Survey on Applications of Graph Neural Networks in Cyber Security"** has been satisfactorily completed by **Mohammed Saif** (Registration Number 149840 of $2019 - 20$; Class Roll No. 001910504005; Examination Roll No. $M6TCT22007$). It is a bona-fide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata for partial fulfilment of the requirements for the awarding of the **Master of Technology in Computer Technology** degree of the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, during the academic year $2021 - 22$.

Mridul Sankar Barik,
Assistant Professor,
Department of Computer Science and Engineering,
Jadavpur University.
(Supervisor)

Forwarded By:

Prof. Anupam Sinha,
Head,
Department of Computer Science and Engineering,
Jadavpur University.

Prof. Chandan Mazumdar,
Dean,
Faculty of Engineering & Technology,
Jadavpur University.

Department of Computer Science and Engineering
Faculty of Engineering And Technology
Jadavpur University, Kolkata - 700 032

Certificate of Approval

This is to certify that the thesis entitled **"Survey on Applications of Graph Neural Networks in Cyber Security"** is a bona-fide record of work carried out by **Mohammed Saif** (Registration Number 149840 of 2019 − 20; Class Roll No. 001910504005; Examination Roll No. $M6TCT22007$) in partial fulfilment of the requirements for the award of the degree of **Master of Technology in Computer Technology** in the **Department of Computer Science and Engineering, Jadavpur University**, during the period of July 2021 to June 2022. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the thesis only for the purpose of which it has been submitted.

Examiners:

(Signature of The Examiner)

(Signature of The Supervisor)

Department of Computer Science and Engineering
Faculty of Engineering And Technology
Jadavpur University, Kolkata - 700 032

Declaration of Originality and Compliance of Academic Ethics

I hereby declare that the thesis entitled **"Survey on Applications of Graph Neural Networks in Cyber Security"** contains literature survey and original research work by the undersigned candidate, as a part of his degree of **Master of Technology in Computer Technology** in the **Department of Computer Science and Engineering, Jadavpur University**. All information have been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Name: Mohammed Saif

Examination Roll No.: M6TCT22007

Registration No.: 149840 of $2019 - 20$

Thesis Title: Survey on Applications of Graph Neural Networks in Cyber Security

Signature of the Candidate:

ACKNOWLEDGEMENT

I am pleased to express my gratitude and regards towards my Project Guide **Shri Mridul Sankar Barik**, Assistant Professor, Department of Computer Science and Engineering, Jadavpur University, without whose valuable guidance, inspiration and attention towards me, pursuing my project would have been impossible.

Last but not the least, I express my regards towards my friends and family for bearing with me and for being a source of constant motivation during the entire term of the work.

Mohammed Saif
MTCT Final Year
Exam Roll No. -M6TCT22007
Regn. No. - 149840 of 2019 – 20
Department of Computer Science and Engineering,
Jadavpur University.

Contents

List of Figures

Abstract

The recent success of neural networks has boosted research on pattern recognition and data mining. Machine learning tasks like object detection, machine translation, and speech recognition, have been given new life with end-to-end deep learning paradigms like CNN, RNN etc. Deep Learning is good at capturing hidden patterns of Euclidean data (images, text, videos). But what about applications where data is generated from non-Euclidean domains, represented as graphs with complex relationships and inter-dependencies between objects? That's where Graph Neural Networks (GNN) come in. Graph Neural Networks (GNNs) are a class of deep learning methods designed to perform inference on data described by graphs. GNNs are neural networks that can be directly applied to graphs, and provide an easy way to do node-level, edge-level, and graph-level prediction tasks. In this thesis we tend to observe various types of GNNs and the use of it.Then we tend to survey the use of GNNs on cybersecuirty aspects which means how well we can use GNNs in the field of cybersecurity effectively. We tend to use GNN and it's use through some experiments which we have shown at the later part of this thesis.

Chapter 1

Introduction

1.1 Background

Graph Neural Networks(GNNs) have proven to be an effective tool for vulnerability discovery that outperforms learning-based methods working directly on source code. Unfortunately, these neural networks are uninterruptible models, whose decision process is completely opaque to security experts, which obstructs their practical adoption. Recently, several methods have been proposed for explaining models of machine learning. However, it is unclear whether these methods are suitable for GNNs and support the task of vulnerability discovery. If we look into the aspects of cybersecurity [17] , we can see that every single thing around the globe is connected to the cyberspace and as a result, we need to secure and enhance each and everything we have related to this and that's where GNN comes in. In this thesis, we use some of the GNN features to resolve the security issues and we explain how to enhance it.
Figure 1.1 is taken from the paper "Graph Neural Networks: A Review of Methods and Applications" by "Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang".

1.2 Research Objective

The broad research objective of this thesis is:

To survey usage of GNN in solving problems in cyber security domain and to evaluate effectiveness of such use.

The main area in which our thesis revolves is how to use the GNN features in the many aspects of cybersecurity possible. For an example, we described how BOTNET-DETECTION[19] works on account of using it in the field of predicting number of bots in social networks.

1.3 Contribution of the Thesis

This Thesis is basically an approach to explore the GNN features and how we approach some of it, how we can analyse it's use and how to dissolve it's vulnerabilities. We basically tend to analyse

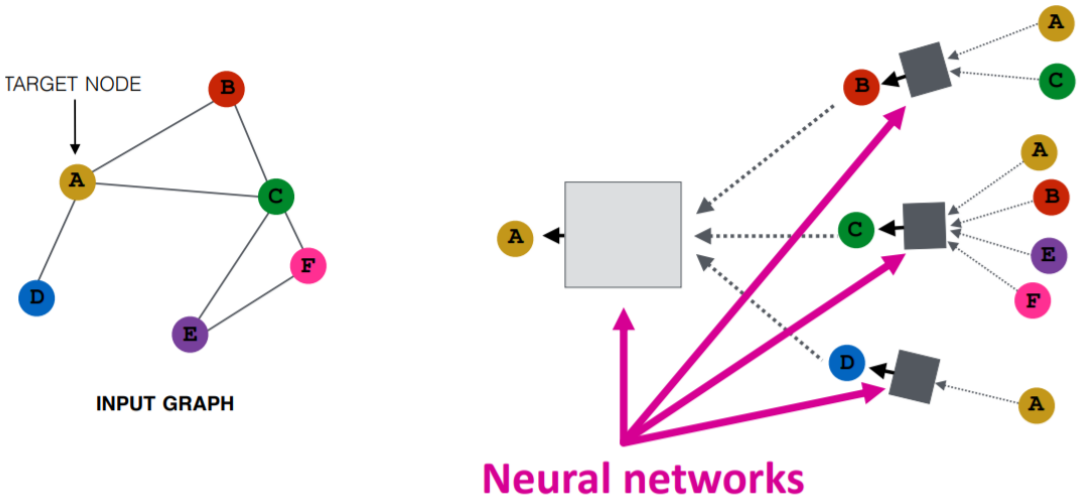


Figure 1.1: Graph Neural Networks

how much of the GNN we can use to dissolve the cyber security issues that is coming now a days.

1.4 Outline of the Thesis

The basic idea of this thesis is how one can approach the cybersecurity areas by the use of GNN. We first start by describing the GRAPH NEURAL NETWORKS following by the various aspects of GNN. then we went to more details about GNN and cybersecurity and in that phase we described how we can use various models of GNN to enhance the cybersecurity aspects. Then we described some of it's experiments by using data-sets. We start this discussion by explaining various types of graph neural networks followed by the various applications of GNNs. then we tend to show how important it is to collaborate cyber security and GNN together and how efficiently it is getting used. Following that, we evaluated some data set based experiments on GNN and it's applications and after that we conclude with the future work part.

Chapter 2

Graph Neural Networks

2.1 Introduction

Classical deep learning methods has mainly been confined to data defined on Euclidean domains, such as grids (e.g., images) and sequences (e.g., speech, text). However, for many of the real world problem areas, non-Euclidean domains such as graphs are natural choice of data representation. Graph Neural Networks (GNNs) [4, 2, 3, 1, 5] are a class of deep learning methods suitable for making inferences over graph data in the field of non-Euclidean data types. With GNNs predictions can be made at the node-level, edge-level, and graph-level, Scarselli et al. [11] introduced the Graph Neural Network (GNN) model as an extension of existing neural network models for processing graph data.

2.2 GNN Architecture

Traditionally, data-sets in Deep Learning applications such as computer vision and Natural Language Processing(NLP) are typically represented in the Euclidean space [20]. Recently though there is an increasing number of non-euclidean data that are represented as graphs. To this end, Graph Neural Networks (GNNs) are an effort to apply deep learning techniques in graphs. The term GNN is typically referred to a variety of different algorithms and is not tied to a single architecture. As we will see, a plethora of different architectures have been developed over the years.

The basic idea behind most GNN architectures is graph convolution. In essence, we try to generalize the idea of convolution into graphs [12].

Figure 2.1 is taken from the paper "Graph Neural Networks: A Review of Methods and Applications" by "jie Zhou,Ganqu Cui, Zhengyan Zhang,Cheng Yang,Zhiyuan Liu and Maosong Sun".

2.2.1 Graph Convolution

Graph convolution predicts the features of the node in the next layer as a function of the neighbours' features[12]. It transforms the node's features in a latent space that can be used for a variety of reasons. But, what can we actually do with these latent node features vectors? Typically all applications fall into one of the following categories:

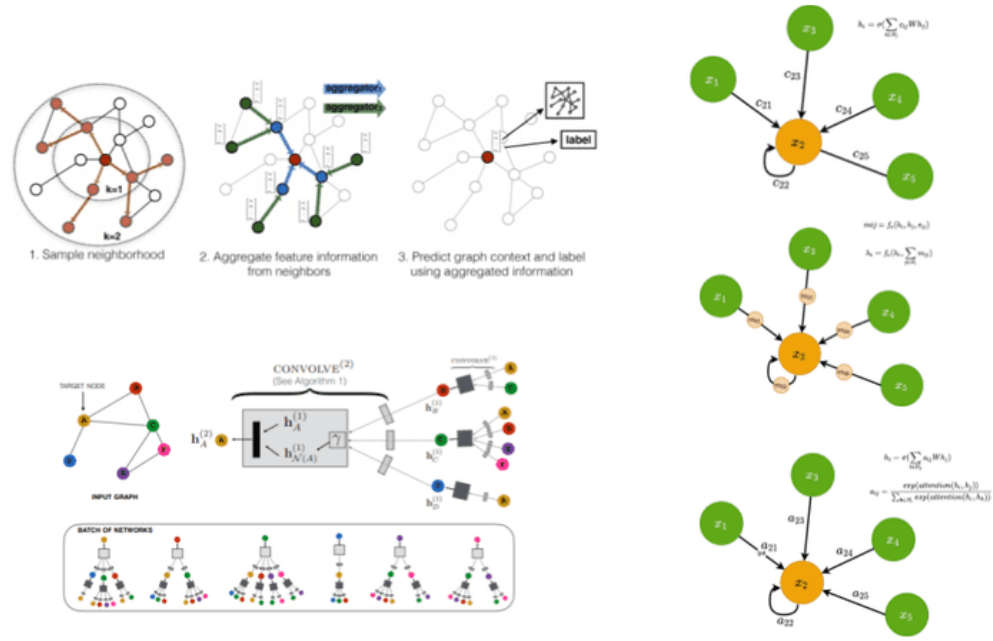


Figure 2.1: GNN Architecture

1. Node classification
2. Edge classification
3. Graph classification

There are basically many type of graph neural architectures based on behaviour of that system. Many methods are involved also to develop the representations of the graphs also. Based on representations and behaviours, there are many classifications of the GNN named as GCN,MPNN etc. We shall discuss them one by one in the later section of this thesis [12].

Figure 2.2 is taken from the paper "Semi-Supervised Classification with Graph Convolutional Networks" by "Thomas N. Kipf and Max Welling".

2.2.2 Gated Graph Neural Networks

Introduction

Gated Graph Sequence Neural Networks (GGS-NNs) [2]is a novel graph-based neural network model. GGS-NNs modifies Graph Neural Networks to use gated recurrent units and modern optimization techniques and then extend to output sequences. It is a subsequent form of GNN but here, we use Gated Recurrent Units and unroll the recurrence and use backpropagation through time in order to compute gradients [2]. This requires more memory than the Almeida-Pineda algorithm(It It is a supervised learning technique, meaning that the desired outputs are known beforehand, and the task of the network is to learn to generate the desired outputs from the inputs), but it removes the need to constrain parameters to ensure convergence.

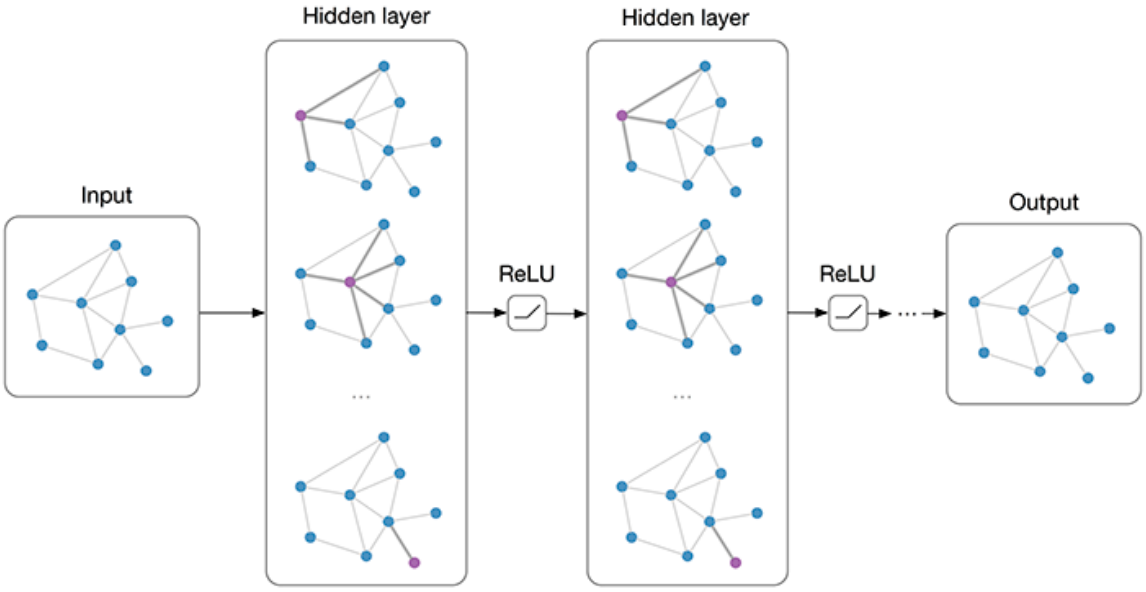


Figure 2.2: Graph Convolutional network

Node Label

In GNNs, there is no point in initializing node representations because the contraction map constraint ensures that the fixed point is independent of the initialization. This is no longer the case with GG-NNs, which lets us incorporate node labels as additional inputs, and in fact, we generalize node labels to be node attribute vectors l_v, which we use to specify node annotations.

Figure 2.3 is taken from the paper "Gated Graph Sequence Neural Networks" by "Li, Yujia and Tarlow, Daniel and Brockschmidt, Marc and Zemel".

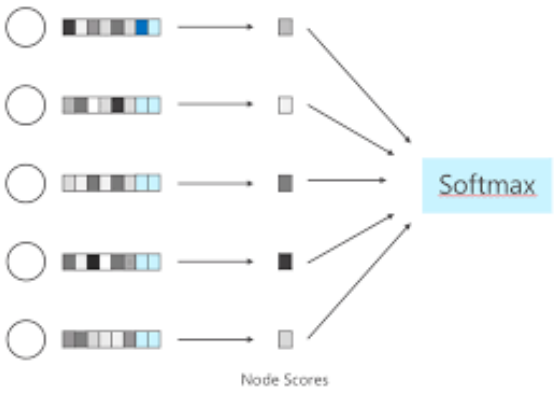


Figure 2.3: Gated Graph Neural Network

Propagation Model

We consider a matrix $A \in R^{D|\nu| \times 2D|\nu|} \ldots (1)$, that determines how nodes in the graph communicate with each other. The sparsity structure corresponds to the edges of the graph, and the parameters in each sub-matrix are determined by the edge type and direction.

$A_v \in A$ is the sub-matrix of A containing the rows corresponding to node v. Equation 1 is the initialization step, which copies node labels into the first components of the hidden state and pads the rest with zeros. We also need to pass the information between different nodes of the graph via incoming and outgoing edges with parameters dependent on the edge type and direction. The remaining are GRU-like updates that incorporate information from the other nodes and from the previous timestep to update each node's hidden state. z and r are the update and reset gates, $\sigma(x) = \frac{1}{(1+e^{-x})}$ is the logistic Sigmoid functionWe initially experimented with a vanilla recurrent neural network-style update, but in preliminary experiments we found this GRU-like propagation step to be more effective.

Gated Graph Sequence Networks

There are two settings for training GGS-NNs: specifying all intermediate labels L(k), or training the full model end-to-end given only L(1), graphs and target outputs sequences [7]. The former can improve performance when we have domain knowledge about specific intermediate information that should be represented in the internal state of nodes, while the latter is more general.Both are described below.

Sequence Outputs With Observed Annotations

Consider the task of making a sequence of predictions for a graph, where each prediction is only about a part of the graph. For such tasks, in order to ensure we predict an output for each part of the graph exactly once, it suffices to have one bit per node, indicating whether the node has been "explained" so far. While this example is artificially simple, it conveys the point that in some settings, a small number of annotations are sufficient to capture the state of the output procedure. When this is the case, we may want to directly input this information into the model via labels indicating target intermediate annotations. At training time, given the annotations $L(k)$ the sequence prediction task decomposes into single step prediction tasks [7]. We can then train the GG-NNs as before. At test time, predicted annotations from one step will be used as input to the next step.

Sequence Outputs With Latent Annotations

More generally, when intermediate node annotations, $L(k)$ are not available during training, we treat them as hidden units in the network, and train the whole model jointly by backpropagating through the whole sequence.

2.2.3 Relational Graph Convolutional Networks

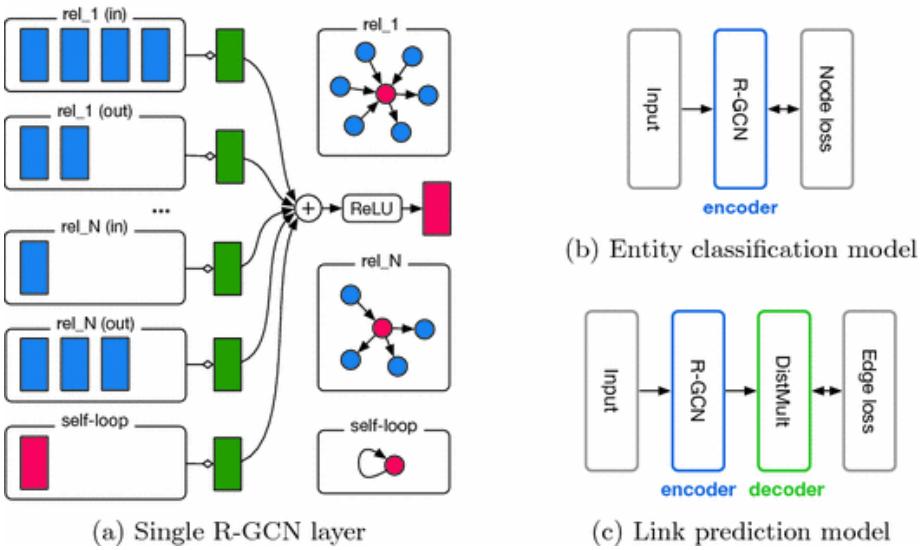
Introduction

Knowledge graphs enable a wide variety of applications, including question answering and information retrieval. RGCNs are related to a recent class of neural networks operating on graphs, and

are developed specifically to deal with the highly multi-relational data characteristic of realistic knowledge bases. We demonstrate the effectiveness of R-GCNs as a stand-alone model for entity classification.

We introduce the following notation: we denote directed and labeled multi-graphs as $G = (V; E; R)$ with nodes (entities) v_i belongs to V and labeled edges (relations) $(v_i; r; v_j) \in E$, where $r \in R$ is a relation type [7].

Figure 2.4 is taken from the paper "Relational Graph Convolutional Network" by "chlichtkrull, Michael and Kipf, Thomas N. and Bloem, Peter and Berg, Rianne van den and Titov, Ivan and Welling, Max".



(a) Single R-GCN layer

(b) Entity classification model

(c) Link prediction model

Figure 2.4: Relational Graph Convolutional Network

Explanation

To begin with the explanation, we can use a massage passing framework as

$$h_i^{l+1} = \sigma(\sum_{m \in M_i} (g_m(h_i^{(l)}, (h_j^{(l)}))$$

..(1)

where $h_i^{(l)} \in R^{d^{(l)}}$ is the hidden state of node v_i in the l-th layer of the neural network, with $d(l)$ being the dimensionality of this layer's representations. Incoming messages of the form $g_m(.,.)$ are accumulated and passed through an element-wise activation function $\sigma(.)$, such as the ReLU(.) $= max(0, .)^2$. M_i denotes the set of incoming messages for node v_i and is often chosen to be identical to the set of incoming edges. $g_m(.,.)$ is typically chosen to be a (message-specific) neural network-like function or simply a linear transformation $g_m(h_i, h_j) = Wh_j$.

This type of transformation has been shown to be very effective at accumulating and encoding features from local, structured neighborhoods, and has led to significant improvements in areas such as graph classification.

By the use of the following architecture, we can develop a propagation model for calculating the forward-pass update of an entity or node denoted by v_i in a relational (directed and labeled) multi-graph: $h_i^{(l+1)} = \sigma(\sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)} + W_0^{(l)} h_i^{(l)}) \ldots (2)$ where N_i^r denotes the set of neighbor indices of node i under relation $r \in R$. $c_{i,r}$ is a problem-specific normalization constant that can either be learned or chosen in advance (such as $c_{i,r} = |N_i^r|$). Intuitively, (2) accumulates transformed feature vectors of neighboring nodes through a normalized sum. Different from regular GCNs, we introduce relation-specific transformations, i.e. depending on the type and direction of an edge.

To ensure that the representation of a node at layer l + 1 can also be informed by the corresponding representation at layer l, we add a single self-connection of a special relation type to each node in the data. Note that instead of simple linear message transformations, one could choose more flexible functions such as multi-layer neural networks (at the expense of computational efficiency). We leave this for future work.

A neural network layer update consists of evaluating (2) in parallel for every node in the graph. In practice, (2) can be implemented efficiently using sparse matrix multiplications to avoid explicit summation over neighborhoods. Multiple layers can be stacked to allow for dependencies across several relational steps. We refer to this graph encoder model as a relational graph convolutional network (R-GCN).

2.2.4 Relational Graph Attention Networks

Introduction

Convolutional Neural Networks (CNNs) have been successfully applied to tackle problems such as image classification, semantic segmentation or machine translation where the underlying data representation has a grid-like structure [14]. These architectures efficiently reuse their local filters, with learnable parameters, by applying them to all the input positions. However, many interesting tasks involve data that can not be represented in a grid-like structure and that instead lies in an irregular domain.

However, many interesting tasks involve data that can not be represented in a grid-like structure and that instead lies in an irregular domain. This is the case of 3D meshes, social networks, telecommunication networks, biological networks or brain connectors. Such data can usually be represented in the form of graphs.

Based on these theories, we introduce an attention-based architecture to perform node classification of graph-structured data [14]. The idea is to compute the hidden representations of each node in the graph, by attending over its neighbors, following a self-attention strategy. The attention architecture has several interesting properties: (1) the operation is efficient, since it is parallelizable across node-neighbor pairs; (2) it can be applied to graph nodes having different degrees by specifying arbitrary weights to the neighbors; and the model is directly applicable to inductive learning problems, including tasks where the model has to generalize to completely unseen graphs. We validate the proposed approach on four challenging benchmarks: Cora, Citeseer and Pubmed citation networks as well as an inductive protein-protein interaction data-set, achieving or matching state-of-the-art results that highlight the potential of attention-based models when dealing with arbitrarily structured graphs.

Figure 2.5 is taken from the paper "Graph Attention Networks" by "Velickovic, Petar and Cucurull, Guillem,Casanova, Arantxa ,Romero, Adriana,Lio, Pietro and Bengio, Yoshua".

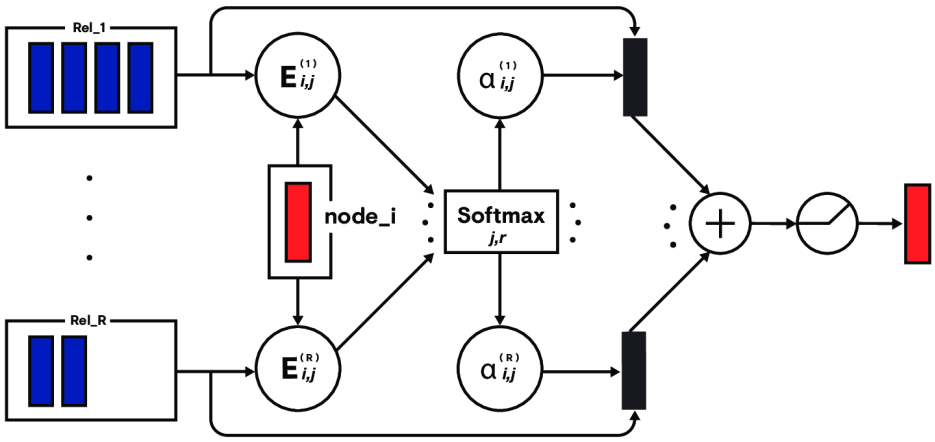


Figure 2.5: Relational Graph Attention Network

Explanation

We will start by describing a single graph attentional layer, as the sole layer for an example. The input to our layer is a set of node features, $h = (\overrightarrow{h_1}, \overrightarrow{h_2}, \ldots, \overrightarrow{h_n}), \overrightarrow{h_i} \in R^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of node features (of potentially different cardinality $F^{'}$), $h = (\overrightarrow{h_1}^{'}, \ldots, \overrightarrow{h_n}^{'}), (\overrightarrow{h}^{'}_i) \in R^{F^{'}}$ as its output.

In order to obtain sufficient expressive power to transform the input features into higher-level features, at least one learnable linear transformation is required. To that end, as an initial step, a shared linear transformation, parametrized by a weight matrix,$W \in R^{F^{'} \times F}$, is applied to every node. We then perform self-attention on the nodes—a shared attentional mechanism a : $R^{F^{'}} \times R^{F^{'}}$. computes attention coefficients $e_i j = a(W\overrightarrow{h_i}, W\overrightarrow{h_j}) \ldots (1)$ that indicate the importance of node j's features to node i. In its most general formulation, the model allows every node to attend on every other node, dropping all structural information. We inject the graph structure into the mechanism by performing masked attention—we only compute e_{ij} for nodes $j \in N_i$, where N_i is some neighborhood of node i in the graph. In all our experiments, these will be exactly the first-order neighbors of i (including i). To make coefficients easily comparable across different nodes, we normalize them across all choices of j using the softmax function: $a_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in N_i} exp(e_{ik})}$. once we calculate the a_{ij} by using the Leaky ReLU non linearity,the normalized attention coefficients are used to compute a linear combination of the features corresponding to them, to serve as the final output features for every node.

2.2.5 Relational Graph Isomorphism Networks

A generalisation of Graph Isomorphism Networks to several edge types.

Introduction

Two graphs which contain the same number of graph vertices connected in the same way are said to be isomorphic. Having gained the basic knowledge for a maximally powerful GNN, we can develop a simple architecture of GIN that can satisfies a theorem on Weisfeiler-Lehman test of isomorphism which generalizes the WL test and hence achieves maximum discriminative power among GNNs [15].

Explanation

The Theorem is as follows:

Let $A : G \to R$ to the power d be a GNN. With a sufficient number of GNN layers, A maps any graphs G1 and G2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:

(a) A aggregates and updates node features in an iterative manner with $h_v^{(k)} = \phi(h_v^{(k1),f(h_u^{(k1)}:u\in N(v)}))$, where the functions f, which operates on multisets, and ϕ are injective.

(b) A's graph-level readout, which operates on the multiset of node features $h_v^{(k)}$, is injective.

According to this theory, The GIN update node representations as $h_v^{(k)} = MLP^{(k)}((1+\epsilon^{(k)}).h_v^{(k-1)} + \sum_{u\in N(v)} h_u^{(k-1)})$.

Node embeddings learned by GIN can be directly used for tasks like node classification and link prediction. For graph classification tasks we propose the following "readout" function that, given embeddings of individual nodes, produces the embedding of the entire graph.

An important aspect of the graph-level readout is that node representations, corresponding to subtree structures, get more refined and global as the number of iterations increases. A sufficient number of iterations is key to achieving good discriminative power. Yet, features from earlier iterations may sometimes generalize better. To consider all structural information, we use information from all depths/iterations of the model. We achieve this by an architecture similar to Jumping Knowledge.

Figure 2.6 is taken from the paper "How Powerful are Graph Neural Networks?" by "Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka".

The WL Test

The Weisfeiler-Lehman test is an efficient algorithm under the context of GNN. It has an one dimensional form called "naive vertex refinement". It is an analogous to neighbor aggregation in GNNs.

WL iteratively aggregate the label of neighborhoods and hashes aggregated labels into unique new labels. The graph isomorphism problem asks whether two graphs are topologically identical. This is a challenging problem: no polynomial-time algorithm is known for it yet Apart from some corner case the Weisfeiler-Lehman (WL) test of graph isomorphism is an effective and computationally efficient test that distinguishes a broad class of graph). Its 1-dimensional form, "naïve vertex refinement", is analogous to neighbor aggregation in GNNs. The WL test iteratively aggregates the labels of nodes and their neighborhoods, and hashes the aggregated labels into unique new labels. The algorithm decides that two graphs are non-isomorphic if at some iteration the labels of the nodes between the two graphs differ.

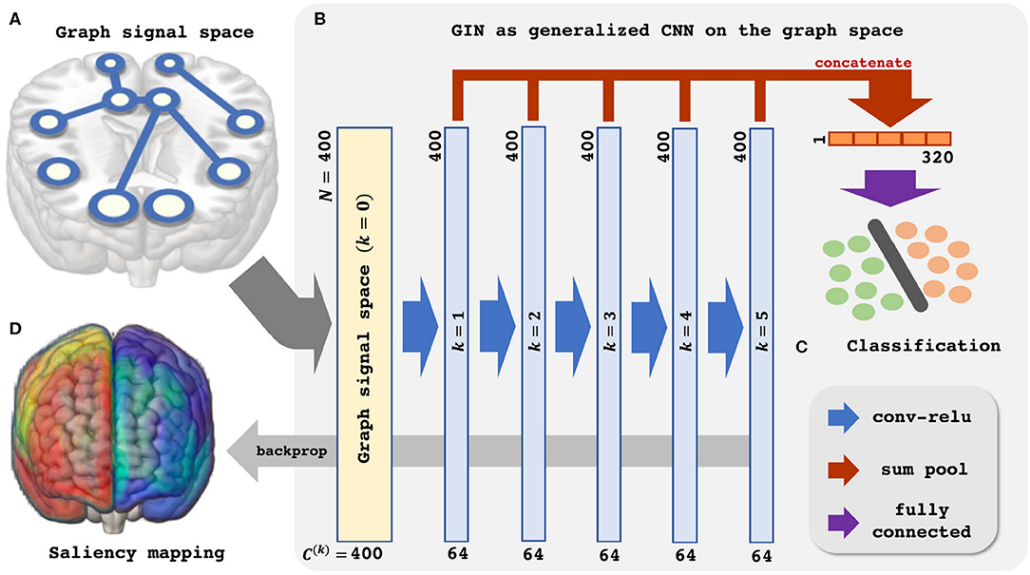


Figure 2.6: Relational Graph Isomorphism Network

Relational Pooling

One way to motivate provably more powerful GNNs is by considering the failure cases of the WL algorithm. This algorithm can not distinguish between a connected six cycle and a set of two triangles. To address this limitation,we consider augmenting MPGNNs with unique node ID features. If we use MP-GNN(A, X) to denote an arbitrary MP-GNN on input adjacency matrix A and node features X, then adding node IDs is equivalent to modifying the MP-GNN to the following:MP-GNN(A, $X \bigoplus I$), where I is the $d \times d$-dimensional identity matrix and denotes column-wise matrix concatenation. In other words, we simply add a unique, one-hot indicator feature for each node. allow a MP-GNN to identify when two nodes share a neighbor, which would make the two graphs distinguishable.

2.2.6 Graph Neural Network with Edge MLPs

A variant of RGCN in which messages on edges are computed using full MLPs, not just a single layer. While many more GNN variants exist, the four main aspects of GNN are R-GIN,R-GCN etc and these are broadly representative of general trends. It is notable that in all of these models, the information passed from one node to another is based on the learned weights and the representation of the source of an edge. In contrast, the representation of the target of an edge is only updated, treated as another incoming message, or used to weight the relevance of an edge. Sometimes unnamed GNN variants of the above mentioned are used for replacing the linear layers to compute the messages for each edge by MLPs applied to the concatenation of the representations of source and target nodes. In the experiments, this will be called GNN-MLP, formally defined as follows: $h_v^{(t+1)} = \sigma(\sum_u -> v \in \epsilon \frac{1}{cvl} MLPl(h_u^{(t)} || h_v^{(t)})$.

2.2.7 Relational Graph Dynamic Convolutional Networks

A new variant of RGCN in which the weights of convolutional layers are dynamically computed. At first, we describe the equation of R-GCN which is as follows: $h_v^{(t+1)} = \sigma(\sum_u -> v \in \epsilon \frac{1}{cvl} Wl(h_u^{(t)})$. To replace the learnable message transformation W‘ by the result of some learnable function f that operates on the target representation: $h_v^{(t+1)} = \sigma(\sum_u -> v \in \epsilon f(h_v^{(t)}; \Theta f, l) h_u^{(t)})$. However, for a representation size D, f would need to produce a matrix of size D2 from D inputs. Hence, if implemented as a simple linear layer, f would have on the order of O(D3) parameters, quickly making it impractical in most contexts.

2.2.8 Graph Neural Networks with Feature-wise Linear Modulation

It is basically new extension of RGCN with FiLM layers. In the equation of RGDCN, we saw that the message passing layer is a linear transformation conditioned on the target node representation, focusing on separate chunks of the node representation at a time. In the extreme case in which the dimension of each chunk is 1.

In the graph setting, we can use each node's representation as an input that determines an element-wise affine transformation of incoming messages, allowing the model to dynamically up-weight and down-weight features based on the information present at the target node of an edge. This yields the following update rule, using a learnable function to compute the parameters of the affine transformation.

Chapter 3

GNN Applications

3.1 Introduction

Graphs and their study have received a lot of attention since ages due to their ability of representing the real world in a fashion that can be analysed objectively. Indeed, graphs can be used to represent a lot of useful, real world data-sets such as social networks, web link data, molecular structures, geographical maps, etc [17]. Apart from these cases which have a natural structure to them, non-structured data such as images and text can also be modelled in the form of graphs in order to perform graph analysis on them. Data in many problem domain can be represented as graph. The kind problems that GNNs can solve are classified into the following categories:

1. Node Classification:
2. Graph Classification:
3. Graph Visualization:
4. Graph Clustering:

3.2 Node Classification

Node Classification is a common machine learning task applied to graph: training a model to learn in which class a node belongs. There are two major classes of classification problems: binary and multi-class. In Binary-class classifications, the given data-set is categorized into two classes and in Multi-class classification, the given data-set is categorized into several classes. Concretely, Node Classification models are used to predict a non-existing node property based on other node properties. The Node Classification model does not rely on relationship information.

The training process follows this outline:

1. The input graph is split into two parts: the train graph and the test graph.
2. The train graph is further divided into a number of validation folds, each consisting of a train part and a validation part.

3. Each model candidate is trained on each train part and evaluated on the respective validation part.
4. The training process uses a logistic regression algorithm, and the evaluation uses the specified metrics. The first metric is the primary metric.
5. The model with the highest average score according to the primary metric will win the training.
6. The winning model will then be retrained on the entire train graph.
7. The winning model is evaluated on the train graph as well as the test graph.
8. The winning model is retrained on the entire original graph.

Finally, the winning model will be registered in the Model Catalog.

Figure 3.1 is taken from the paper "Graph Neural Networks: A Review of Methods and Applications" by "Jie Zhou,Ganqu Cui,Zhengyan Zhang,Cheng Yang,Zhiyuan Liu,Maosong Sun".

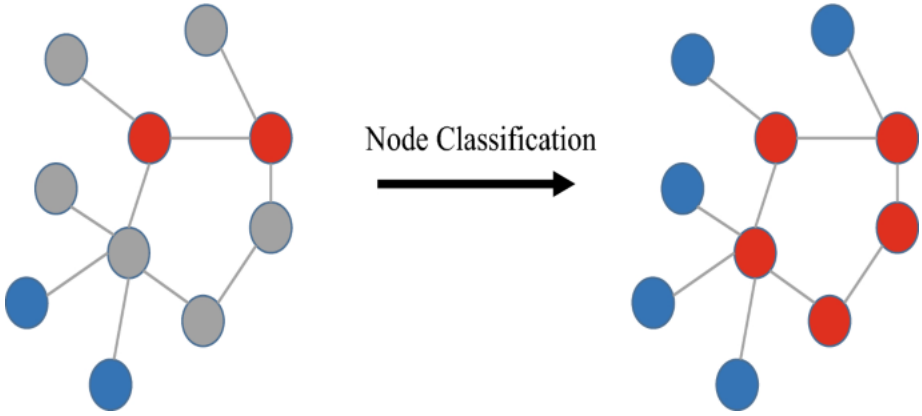


Figure 3.1: Node Classification

3.3 Graph Classification

Graph classification is a problem with practical applications in many different domains. To solve this problem, one usually calculates certain graph statistics (i.e., graph features) that help discriminate between graphs of different classes. When calculating such features, most existing approaches process the entire graph. In a graphlet-based approach, for instance, the entire graph is processed to get the total count of different graphlets or subgraphs. In many real-world applications, however, graphs can be noisy with discriminative patterns confined to certain regions in the graph only. For that, we can use attention-based graph classification. The use of attention allows us to focus on small but informative parts of the graph, avoiding noise in the rest of the graph.

Figure 3.2 is taken from the paper "Graph Neural Networks: A Review of Methods and Applications" by "Jie Zhou,Ganqu Cui,Zhengyan Zhang,Cheng Yang,Zhiyuan Liu,Maosong Sun".

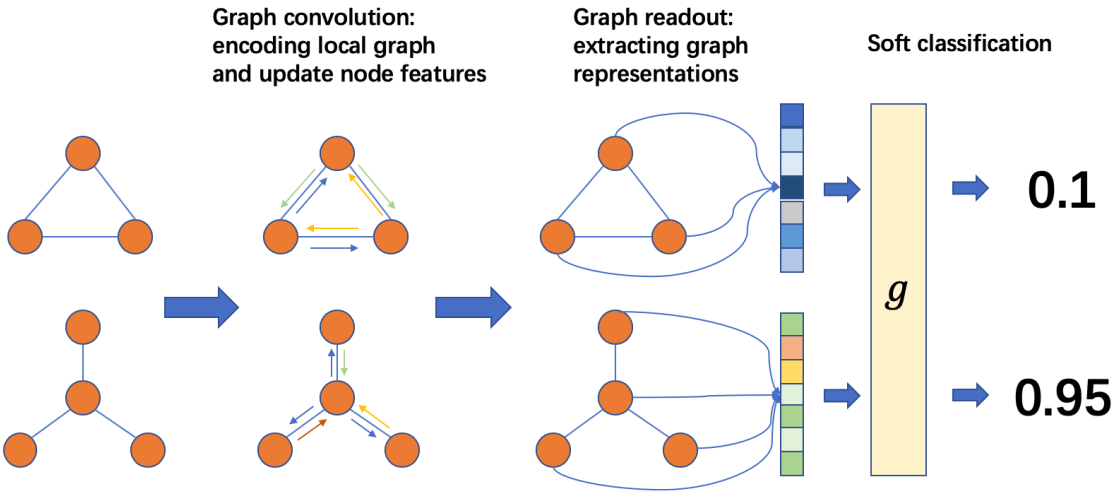


Figure 3.2: Graph Classification

3.4 Graph Visualisation

Graph visualization is the visual representation of the nodes and edges of a graph. Dedicated algorithms, called layouts, calculate the node positions and display the data on two (sometimes three) dimensional spaces. There are some Graph visualization tools that provide user-friendly web interfaces to interact and explore graph data.

These graph visualizations are simply visualizations of data modeled as graphs. Any type of data asset that contains information about connections can be modeled and visualized as a graph, even data initially stored in a tabular way.

Some benefits of graph visualisation are:

1. It will need less time assimilating information from data.
2. There are higher chances to discover insights by interacting with data.
3. One can achieve a better understanding of a problem and also it is also an effective form of communication.

Graph Neural Networks are increasingly being used in application areas such as: social network, knowledge graph, recommender systems, life science etc. We Discuss some of them below.

3.5 Computer Vision

Using regular CNNs, machines can distinguish and identify objects in images and videos. Although there is still much development needed for machines to have the visual intuition of a human. Yet, GNN architectures can be applied to image classification problems.

One of these problems is scene graph generation, in which the model aims to parse an image into a semantic graph that consists of objects and their semantic relationships. Given an image, scene graph generation models detect and recognize objects and predict semantic relationships between pairs of objects.

However, the number of applications of GNNs in computer vision is still growing. It includes human-object interaction, few-shot image classification, and more.

Figure 3.3 is taken from the paper "Deep Learning on Graphs For Computer Vision — CNN, RNN, and GNN" by "Huan Ling".

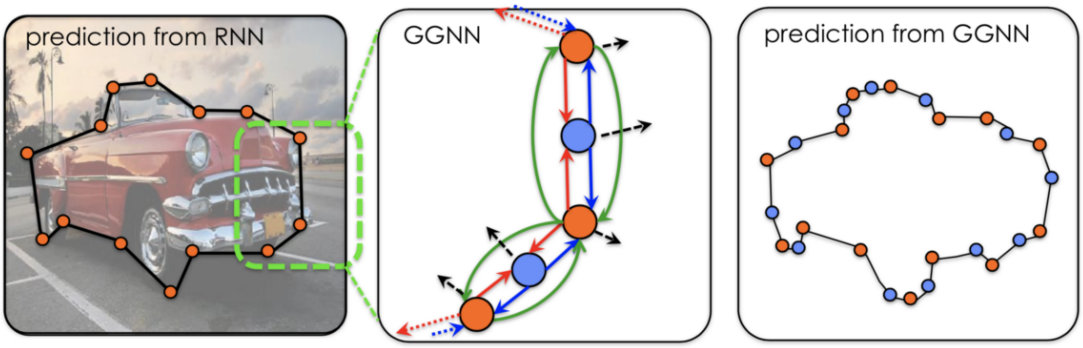


Figure 3.3: Computer Vision using GNN

3.6 Natural Language Processing

In NLP, we know that the text is a type of sequential data which can be described by an RNN or an LSTM. However, graphs are heavily used in various NLP tasks, due to their naturalness and ease of representation.

Recently, there has been a surge of interest in applying GNNs for a large number of NLP problems like text classification, exploiting semantics in machine translation, user geolocation, relation extraction, or question answering.

We know that every node is an entity and edges describe relations between them. In NLP research, the problem of question answering is not recent. But it was limited by the existing database. Although, with techniques like GraphSage (Hamilton et al.), the methods can be generalized to previously unseen nodes.

Figure 3.4 is taken from the paper "Effective attention modeling for aspect-level sentiment classification" by "He et al".

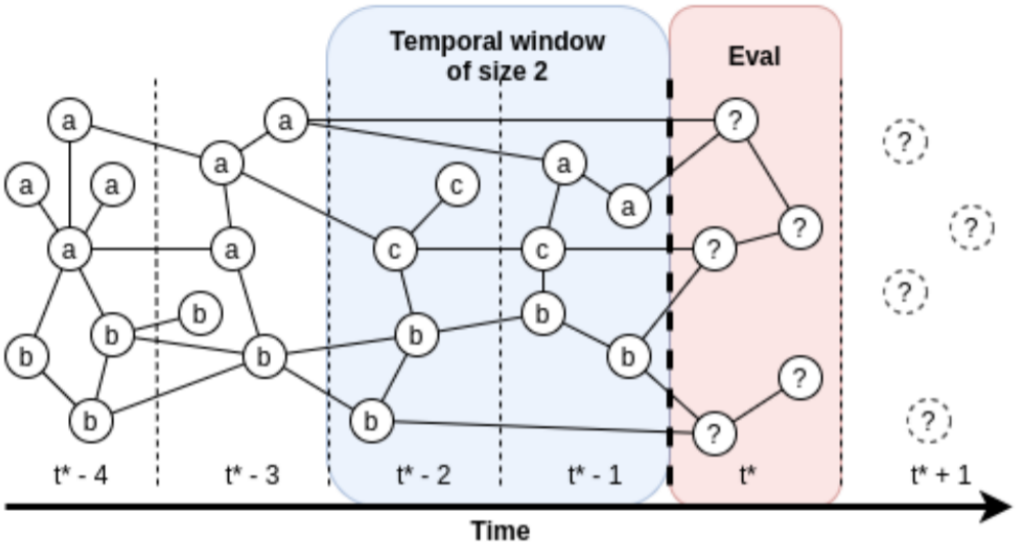


Figure 3.4: NLP explanation using LSTM

3.7 Medical Science

Electronic health records (EHRs) have been widely used to help physicians to make decisions by predicting medical events such as diseases, prescriptions, outcomes, and so on. How to represent patient longitudinal medical data is the key to making these predictions. Recurrent neural network (RNN) is a popular model for patient longitudinal medical data representation from the view of patient status sequences, but it cannot represent complex interactions among different types of medical information, i.e., temporal medical event graphs, which can be represented by graph neural network (GNN)[8].

In this section we will try to illustrate the RGNN model that is a hybrid of Recurrent Neural Network(RNN) and Graph Neural Network(GNN).RNN and GNN are first individually used to represent patient longitudinal medical data from two views, and then they are combined organically. The decomposed LSTM, a state-of-the-art method for next-period prescription prediction, is used to represent patient status sequences, and GNN is used to represent temporal event graphs[8].

1. Modeling Electronic Health Records In the medical domain, medical event prediction is a promising research topic[8]. The main task of medical event prediction is to predict future medical events including risk of diseases , prescriptions, mortality rate etc but as much as the EHR concerns, our main focus is on next-period prescription prediction.

2. As described above, we use a HYBRID method for the EHR prediction. While GNN is used to represent the temporal medical event graph. The LSTM mainly focuses on patient status at each time, and the latter one mainly focuses on the whole medical events. This may be the main reason that both decomposed LSTM and GNN outperform the basic LSTM that considers different types of medical information separately. Because patient medical data is

time-sequential data, RGNN using TG(time-aware graph) is a little better than RGNN using CG(co-occurrence graph).

Figure 3.5 is taken from the paper "A hybrid method of recurrent neural network and graph neural network for next-period prescription prediction" by "Sicen Liu,Tao Liu,Haoyang Ding,Buzhou Tang,Xiaolong Wang,Qingcai Chen Jun Yan and Yi Zhou".

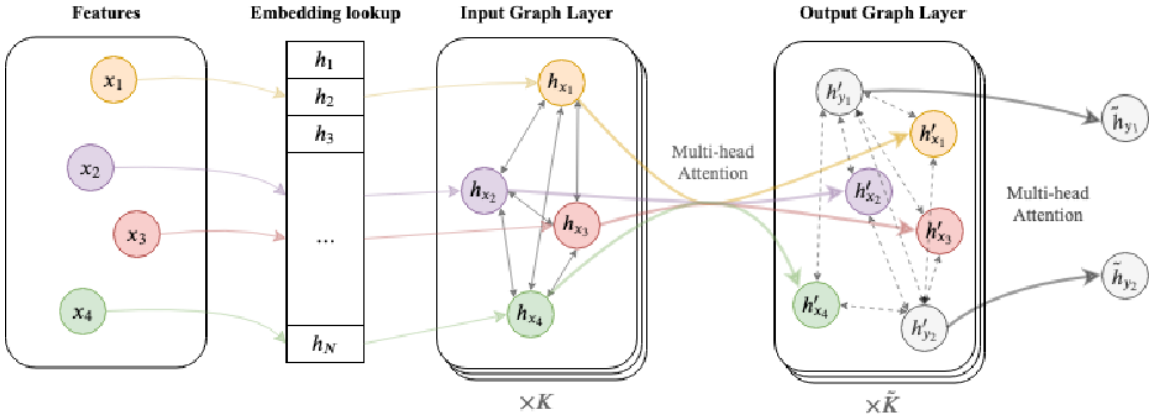


Figure 3.5: Electronic Health Record Generation

3.8 Computer Network

As the computer applications are increasing the use of computer networks are becoming very important and so if we utilise the use of networks with MACHINE LEARNING, it will be easier to extract and secure the networks with much more informations rather than traditional security patches.

1. Resource Requirement Prediction in NFV [10] Network function virtualization (NFV) proposes to replace physical middleboxes with more flexible virtual network functions (VNFs). To dynamically adjust to everchanging traffic demands, VNFs have to be instantiated and their allocated resources have to be adjusted on demand. Deciding the amount of allocated resources is non-trivial. Existing optimization approaches often assume fixed resource requirements for each VNF instance. However, this can easily lead to either waste of resources or bad service quality if too many or too few resources are allocated. To solve this problem, we train machine learning models on real VNF data, containing measurements of performance and resource requirements. For each VNF, the trained models can then accurately predict the required resources to handle a certain traffic load. We integrate these machine learning models into an algorithm for joint VNF scaling and placement and evaluate their impact on resulting VNF placements. Our evaluation based on real-world data shows that using suitable machine learning models effectively avoids over- and underallocation of resources, leading to up to 12 times lower resource consumption and better service quality with up to 4.5 times lower total delay than using standard fixed resource allocation.

2. IoT Firmware Version Prediction [17]

   The firmware information for IoT devices includes the manufacturer, the device type, the device model and the firmware version, etc. Identifying firmware information helps build firmware knowledge graph for many security applications, such as homologous analysis and vulnerability detection of firmware. The traditional firmware information identifying method only utilizes the content-based information, lacks the utilization of the structure information of the firmware, and more importantly, it lacks the use of timing information. Lacking of structural information can reduce prediction accuracy, and lacking of timing information will make it difficult to predict the firmware version. In order to address the disadvantages of the existing method, GNN is used which abstracts the directories or files (components) of the firmware into the nodes of the graph and abstracts the relationships between the nodes into the edges of the graph. Timing information such as component creation time and component version are also attached to the node properties to introduce the time sequence features. As a result, the experimental results show that the accuracy of this method is better than that of random forest for the all four tasks (manufacture, device type, device model and firmware version identification). Particularly, and the accuracy rate is greatly improved in the firmware version identification task.

Chapter 4

GNN and Cyber Security

4.1 Botnet Detection

4.1.1 Introduction

In computer networks, botnets are the network compromised computers that are used to perform various malicious activities. botnets can be inserted in the networks by various means and can jam the network interceptors in no time.

4.1.2 How does it works?

Botnets receives commands from a botmaster through centralised command and control or decentralized peer to peer structures. As (`c&c`) protocol is detectable, botnets uses P2P method in a more specific way as this is harder to detect [19]. Botnets has fast mixing rates that diffuses information. There are many detection approaches of botnets and one of them is on the mixing rates, though a major obstacle is that the massive scale of network communication makes it hard to differentiate botnet communication patterns from background internet traffic. For that, many human power as well as multiple pre-filters were required. This is why we tend to propose to add GNN to identify the botnets [19]. As nodes update their states and exchange information by passing messages to their neighboring nodes so the model can automatically identify node dependencies in the graph after many layers of message passing. So we do not need any explicit filters or any manual labour.

Figure 4.1 is taken from the paper "AUTOMATING BOTNET DETECTION WITH GRAPH NEURAL NETWORKS" by "Jiawei Zhou,Zhiying Xu,Alexander M. Rush and Minlan Yu".

4.1.3 Explanation

First we need to define a communication graph as G={V,A } where v is the node set consisting on n unique nodes observed in traffic traces and $A \in R^{n \times n}$ is a symmetric adjacency matrix with a_ij=1 represents an edge between nodes v_i and v_j and 0 otherwise. We use D=diag(d1,d2.....dn) with $d_i = \sum_{j=1}^{n} aij$ to denote the diagonal node degree matrix.

The GNN can be used as a node vector representation that captures the important context of node. Let $X^{(l)} \in R^{n \times h}$ denote the node feature matrix after layer l and the feature vector (row) for node i represented as xi(l). The vectors are updated every layer, so as to construct a hierarchical

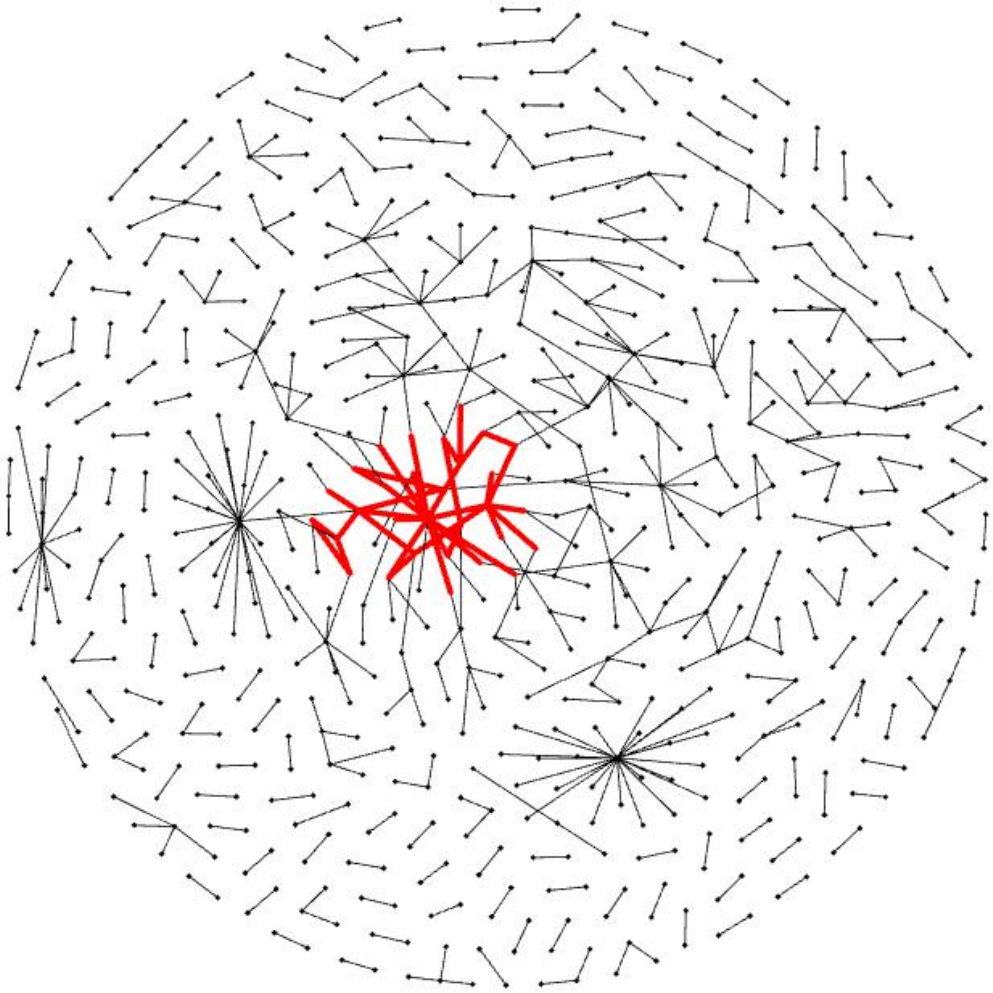


Figure 4.1: Botnet Dataset(Cora dataset)

profile with higher-level vectors representing broader and more abstract properties. At each GNN layer, the representation of each node is first transformed via a learned matrix $W^{(l)}$

$\tilde{x_i}^{(l)} \leftarrow {x_i}^{(l-1)} W^{(l)}$...(1)

Then each node's representation is averaged with the representations of its direct neighbors, which allows the node representation to include its neighbor information as a way to effectively explore the graph structure: ${x_i}^{(l)} \longleftarrow \sum_{j=1}^{n} \frac{a_{ij}}{\sqrt{d_i d_j}} \tilde{x_j}^{(l)}, \forall i \in (1, 2, 3.....n)$..... (2) where the normalization is commonly used to prevent numerical instabilities for deep models, and is shown to have performance gains. Furthermore, a non-linear function is applied at the end to finish updating the hidden node representations for the current layer. More compactly, we can express the above update in matrix form as $X^{(l)} \longleftarrow \sigma(\bar{A} X^{(l-1)} W^{(l)})$. With the normalised adjacency

matrix $\bar{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ And the nonlinear function σ which is typically RelU.A separate linear transformation to map forward the last layer representation is needed which is described as- $X^{(l)} \longleftarrow \sigma(X^{(l-1)}U^{(l)} + \bar{A}X^{(l-1)}W^{(l)}))$. Where $U^{(l)}$ is a learnable transform Matrix at layer l. The top knot represents after layer l are then inputted to a linear layer followed by the softmax function for the final classification. The updating procedure are frequently summarised as a message passing Framework as node features are passed to its neighbouring nodes in every layer. with the stack of L layers, the final representation of each node would be able to learn useful properties within its L-hop neighbourhood for the downstream task . The form of A compliments impacts how the neighbouring node features are normalised before aggregation, and different choices lead to different GNN model variants. To utilise the fast mixing property of botnet topologies, we propose to use a random walk style normalization $\bar{A} = D^{-}1A$ Which only involves the degree of the source nodes to equate the normalized adjacency Matrix to the corresponding probability transition matrix. Second, since we want feature initialization agnostic to any ordering of nodes for purely topological learning, we set the first layer input to all ones. We generate a collection of data-sets including 4 synthetic botnet topologies, DE BRUIJN, KADEMLIA, CHORD, and LEET-CHORD, as well as 2 real botnet topologies we captured, C2 and P2P. The background network graph contains about 140k nodes and 700k edges (undirected) on average. For each of the synthetic botnet topologies, we generate graphs containing 100/1k/10k botnet nodes, and the real botnets contain about 3k botnet nodes [19].

We compare the GNN model with a non-learning specialized detection method, BotGrep and LOGISTIC REGRESSION MODEL(LR) which takes in the following constructed features for each node: its own degree, and the mean, max, min of its neighbors' degrees. Note that BotGrep is a specialized multi-stage algorithm for topological botnet detection, which utilizes the fast-mixing property of random walks within the botnet community and relies on several hand-tuned heuristics. This is in contrast with our GNN detection method which is fully automatic but data-driven.

4.1.4 Results

The following results are shown in table 2 and table 3. the Logistic regression(LR) Model performs poorly in most of the cases indicating that it is not enough to utilise local information up to two hop neighbours and there are no hidden representations to allow more Complex learning. Note, the reported results of BoTGrep are from previous work based on a single graph instance, While our results are averaged over all the graphs in the test set. Still the end-to-end GNN method achieves comparable or better results with Botgrep showing higher detection rates and lower false positive rates in most cases which validates the application on automated botnet detection based on network topology. Moreover, although the GNN models are trained on graphs with 10k botnet nodes, the detection performance on 1k and 100 botnet nodes does not deteriorate too much regardless of the worsening class imbalance problem (no tuning is applied for the detection threshold), which also supports the robustness of the automated approach. If we consider the number of layers for the synthetical topologies, the the order will be CHORD > KADEMLIA > LEET-CHORD > DE BRUIJN. And if we consider the eighen values then, DE BRUIJN > LEET-CHORD > KADEMLIA > CHORD. For the real botnets, we found that there is a sharp phase-transition after 3 layers with the model starting to perform very well from almost nothing, and more layers bring little gain. As can be seen in Table 3, the poor LR baseline and GNN-2 results are consistent, indicating 2-hop neighbor information is not adequate. As for the unnecessary need of deeper models as for other synthetic botnets, it can be explained by the fact that these botnets contain star-shaped attack

traffics towards a victim, such as DDoS attack and spam. Since most nodes in the botnet are able to reach their victim within few hops traveling through a hub node (because of the star-shaped topology), it makes sense that a relatively shallow model can detect this pattern.

4.2 Software Vulnerability Identification

4.2.1 Introduction

Exploitable vulnerabilities in software have attracted tremendous attention in recent years because of the potentially high severity impact on computer security and information safety. To identify vulnerability machine learning is an exceptional answer [20]. Let us find how we can use deep learning to identify such vulnerabilities in softwares. We tend to use deep learning because deep learning techniques extract features automatically, resulting in relieving experts from tedious feature engineering tasks. The abstract feature representations automatically extracted by deep learning methods often demonstrate better generalization abilities than manually extracted features.

4.2.2 Methods of using deep learning to detect vulnerability

There are already many researches based on deep learning that are used to detect any vulnerabilities in softwares. We generalise them into four game changing methods and they are discussed below.

Game Changer 1-Automatically learning semantic features for defect prediction

Gamer changer 1 is the pioneer study to adopt a deep belief network (DBN) to learn the semantic representations of a program. It aims to use the high-level semantic representations learned by the neural networks as defective features. In particular, it enables the automated learning of feature sets indicative of defective code, without relying on manual feature engineering [16]. This method is not only suitable for within-project defect prediction but also applicable for cross-project defect prediction. Application Security Testing(AST) is used as the representation of the program to feed into a DBN for data training. A data processing method is proposed with four steps. The first step is to parse the source code into a token; the second step is to map the token to an integer identifier; the third step is to use DBN to generate semantic features automatically; the final step is to use DBN is to establish a defect prediction model. Experiments were conducted on open-source Java projects data-set to assess the performance of this proposed method. The empirical studies demonstrate that the method can effectively and automatically learn semantic features from Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP) [16].

Game Changer 2-end to end prediction from buffer overruns from raw source code

Game changer 2 is the first to provide an end-to-end solution for detecting buffer error vulnerabilities. Empirical studies demonstrate that a neural network is capable and expressive enough to directly learn vulnerability-relevant features from raw source code without code analysis. A novel neural model is developed to relax the code analysis constraint by constructing and customizing the memory networks [16]. The proposed neural network equips with built-in memory blocks to memorize very long-range code dependencies. Hence, this network modification is crucial for identifying buffer error vulnerabilities. For performance evaluation, experiments are conducted on a self-generated data set. Experimental results show that this method can accurately detect different

types of buffer overflows. In terms of solving overflow tasks, the memory networks model is superior to other models. However, this method still has limitations for further improvements. The first limitation is that it fails to detect the buffer overflow issues residing in external functions because the predefined code in external files is excluded in the input data. The second limitation is that each row must contain some data assignment for this model to work. It is not easy to apply this method out of the box for the source code containing conditional statements because attention values are calculated to find the most relevant code positions.

Game Changer 3-VulDeePecker:A deep learning based system for vulnerability detection

Game changer 3 is the first study to apply a bidirectional Long Short Term Memory (BiLSTM) model for vulnerability detection. The BiLSTM extracts and learns long-range dependencies from code sequences. Its training data is derived from the code gadget representing the program fed to the BiLSTM. There are three stages to process the code gadgets. The first stage is to extract the corresponding program slices of library/API function calls. The second stage is to generate and label code gadgets. The third stage is to convert the code gadgets into vectors. After that, the dependencies across a long-range are captured from the code gadgets. In this paper, the granularity of detection is at the slice-level. For evaluating the performance of VulDeePecker, a set of experiments is conducted on open-source projects and the SARD dataset [16] . The experimental results show that VulDeePecker can handle multiple vulnerabilities, and human experience can help improve the effectiveness of VulDeePecker. Besides, VulDeePecker is more effective than other static analysis tools that require experts' defined rules for detecting vulnerabilities. However, this method has limitations for further improvement. The first limitation is that VulDeePecker only can deal with C/CCC programs. The second limitation is that VulDeePecker can only deal with vulnerabilities related to library/API function calls. The third limitation is that the dataset for performance evaluation is small-scale as it only contains two vulnerability types. The BiLSTM model used in VulDeePecker has motivated researches to adopt this paradigm. The use of BiLSTM enables research works to inspect the code dependencies across long-range. VulDeePecker proposes to use fine-grained code gadgets as the representation of a program instead of capturing data dependence relations only.

Game Changer 4-Cyber vulnerability intelligence for IOT binary

Game changer 4 proposes a deep learning-based vulnerability detector for binary code. It aims to broaden the application eld of vulnerability detection by mitigating the unavailability of source code. For data training, binary segments are fed to a bidirectional LSTM neural network with attention (Att-BiLSTM) [16]. The data processing consists of three steps. First, the binary segments were obtained by applying the IDA Pro tool on the original binary code. The second step extracts functions from binary segments and mark them as 'vulnerable' or 'not vulnerable.' The third step takes the binary segment as a binary feature before feeding it to the embedding layer of the Att-BiLSTM. Moreover, the granularity of detection is at the function-level. For evaluating the performance of the proposed method, many experiments are conducted on an open-source project data set. The experiment results indicate that the proposed method outperforms the source code-based vulnerability detection approaches on binary code. However, this method has limitations for further improvement. In particular, the detection accuracy is relatively low, given that the

detection accuracy is less than 80 percent in each data set. It also overlooks the function inlining scenario when the binary code's structure often changes due to function inlining.

Figure 4.2 is taken from the paper "Software Vulnerability Analysis and Discovery Using Deep Learning Techniques: A Survey" by "PENG ZENG, GUANJUN LIN, LEI PAN, YONGHANG TAI and JUN ZHANG".

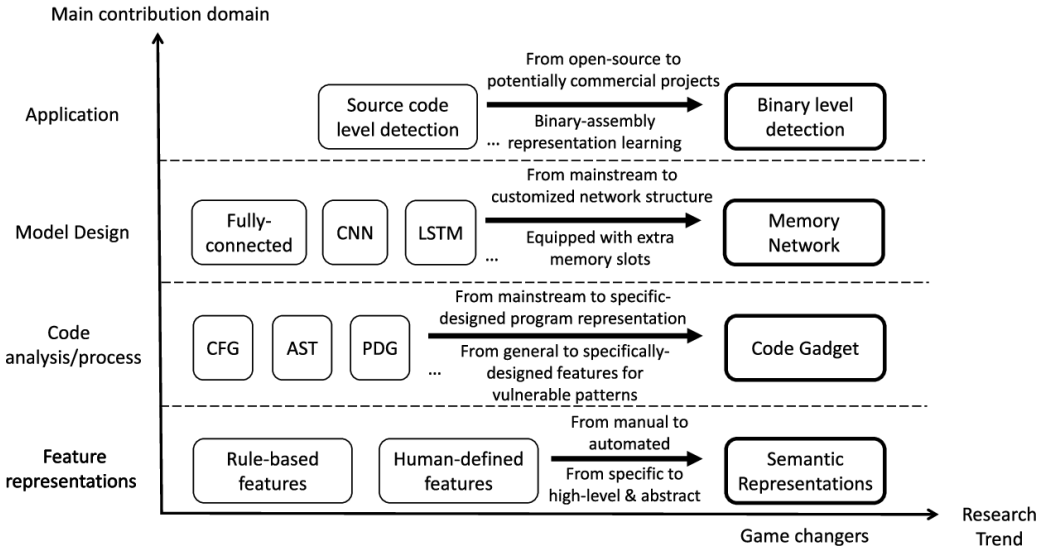


Figure 4.2: Research Trend vs Contribution domain

4.3 Flow-based Network Attack Detection

4.3.1 Introduction

Now a days, network systems are becoming our daily part and so as the security of these became very much important. One of the major threat to the network is Distributed Denial of Service attack. It is an attack where multiple systems are used to target a single victim causing Denial of Service (DoS) of the victim.There has been an exponential Increase of such attacks causing huge damage every year. They are the most commonly occurring attacks due to the fact that anyone can use simple and easy to use free DDoS traffic generating tools available to launch them. Some of the main target industries are media, entertainment, software, technology, security, financial services and gaming industries [13].

4.3.2 What is flow?

A flow is a stream of network packet data having one or more similar features like source IP, destination IP, source port, destination port and protocol. As DDoS comes with large number of packets,so flow is a vital detector for this.

4.3.3 Categories of Flow based techniques

There are basically two types of flow techniques and they are-

Packet Header Feature Extraction Based

The category includes the solutions in which the detection is based on various features such as source IP address, destination IP address or protocol extracted from the header of packets. Cheng, Yin, Liu et. al. gave a formula for IP Address Feature Value (IAFV) to detect DDoS attacks in a given flow of incoming packets. They gave a unique idea that a network flow can be analyzed efficiently by classifying the incoming packets of the flow by source and destination IP address. The IAFV algorithm is as follows:

Input: Packets of Flow F, a sample interval delta t, a criteria to stop algorithm Q, a source IP address S, a destination IP address D, an IP address class set SD, SDS and SDD, an IP address features IAFV. Output: IAFV results

1. Initialize the variables;
2. While (Q is not fulfilled)
3. Read the T, 5 and D of an IP packet from F;
4. End While
5. if (time exceeds the decided delta t)
6. Add all packets with different destination and same source in SDD.
7. Count m
   m is the number of the elements in SDD.
8. End if
9. calculate IAFV
10. return IAFV

Mathematical Formulation Based

This category consists of solutions where the detection is based on mapping mathematical concepts into incoming network packets belonging to a unique flow. Zhongmin and Xinsheng mapped the very well known and acknowledged concept of probability that is correlation coefficient on to network traffic. Their main idea was that normally the frequency of IP addresses to a destination is in a stable range but in DDoS the source IP addresses frequency becomes unstable and random since network packets belonging to random and unique IP addresses are used to attack a single target system in DDoS intervals and observed that range of value of correlation coefficient is steady normally while during attack circumstances, the value of correlation coefficient is abnormally reduced since there will be larger set of unique source IP addresses per unit time [13].

Figure 4.3 is taken from the paper "NF-GNN: Network Flow Graph Neural Networks for Malware Detection and Classification" by "Julian Busch,Anton Kocheturov,Siemens Technology,Volker Tresp,Siemens AG and Thomas Seidl".

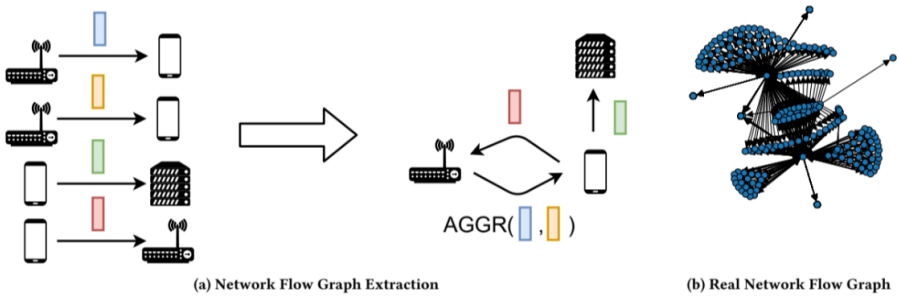


Figure 4.3: Flow Based Network Attack Detection

4.4 Ranking Attack Graphs

A large computer system can run many software components and as a result causes some loopholes to deal with. These loopholes are used to exploit the system.

4.4.1 What is Attack Graphs?

Attack graph is a type of a graph where each node represents a system and edges represents an attack to reach the system. As for a larger system,its complexity will become hard to visualise and as a result, a overview scheme is required. A page rank algorithm is introduced to analyse the rank attack graph. It uses color coding of nodes to highlight important portions. GNN can be trained to simulate the page rank function by using relatively small set of simple sets.GNN can work on small number of training samples and generalize over unseen examples that may be contaminated by noise by the use of approximation algorithm [9]. Attack graph basically works and looks similar to web graph but on to some extent,they may have some differences. Web graph are of linked structured and for ranked graph,it is more of a tree structured,this is the basic difference between these two. Thew complexity of page Rank is $(N \log \frac{1}{\epsilon})$ where N is number of links and ϵ is the expected precision. An attack graph basically describes the security related attributes of the modeled system using graph representations. In the graph representation, the root node indicates the starting node of a system leaf nodes are the final states that the intruder have using any intrusion technique and the intermediate states are the intermediate nodes.

4.4.2 Explanation

Let a function fw1 that expresses the dependence of the state Sn of node n on its neighbors can be represented by- $S_n = f_{w1}(s_u, e_{xy})$, $u \in S(n)$, $x = n \bigvee y = n$ The state S_n is then put through a parameterized output network gw2, which is usually also implemented as an MLP parameterized by w2, to produce an output (e.g.a rank). o(n)=gw2(Sn) f_{w1} and g_{w2} can be implemented by MLP. A sigmoid function can be used as the transfer function that stretches the output from fwi to the desired target. f_{w1} and g_{w2} are thus parameterized by weight sets w1 and w2 of the underlying MLPs. The main objective of this theorem is to verify the effectiveness [of the proposed ranking graph whether, (i) If GNN can learn a ranking scheme applied to attack graphs and (ii) if it can

generalize the results to unseen data. The advantage of using GNN is the insensitivity to noise in a dataset where existing approaches like page rank or rule based approaches are sensitive to noise.

Figure 4.4 is taken from the paper "Ranking Attack Graphs with Graph Neural Networks" by "Liang Lu, Rei Safavi-Naini, Markus Hagenbuchner, Willy Susilo, Jeffrey Horton,Sweah Liang Yong and Ah Chung Tsoi".

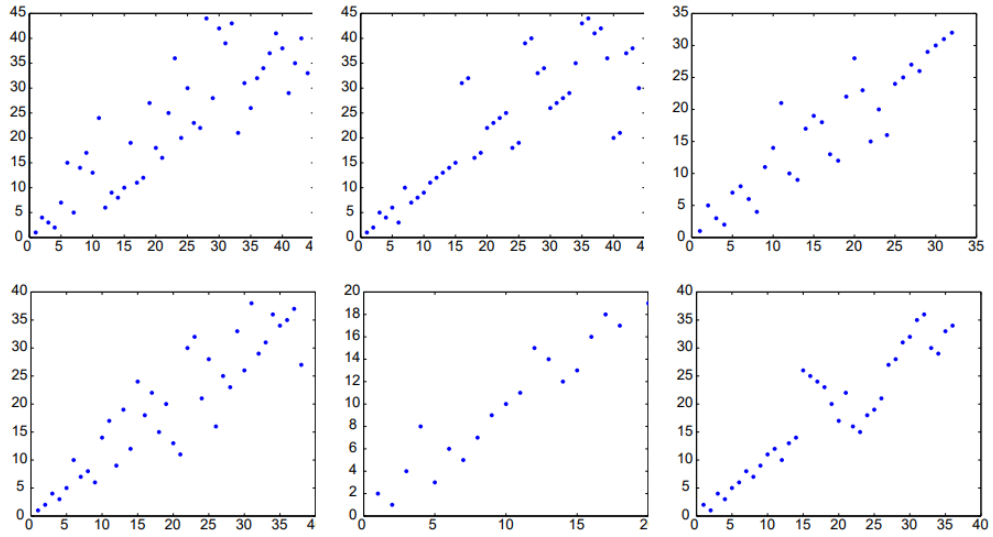


Figure 4.4: Relative Position Diagram when trained on real-world attack graphs

4.5 Malware Detection and Classification

4.5.1 Introduction

Malware is any product tenaciously intended to make harm a PC, server, customer or PC organization. With the fast advancement and development of the web, malware has turned out to be one of the major digital dangers in Nowadays. As the decent variety of malware is expanding nowadays, antivirus tools are not capable of fulfilling the need of protection and results in millions of hosts being hacked [6]. In addition to that, the skills required for malware development is decreasing due to high availability of attacking tools on the Internet. Therefore, to protect computer system from malware is one significant tasks of cybersecurity for a single user as well as for businesses because even a single attack can result in huge information and financial loss [6].

4.5.2 Malware Classification

The main types of malwares are-

1. Adware: It is the slightest risky and the most rewarding malware, it shows advertisements on PC.

2. Spyware: As it implies from the name, the malware that uses for spying. Some run of the mill activities of spyware incorporate following inquiry history to send customized advertisements, following exercises to offer them to the outsiders in this way.

3. Virus: This is the most straightforward type of the software. It is basically any bit of programming that is stacked and propelled without client's authorization while replicating itself or contaminating changing other programming. Regularly this is spread by sharing records or programming between PCs.

4. Worm: It is a program that imitates itself and obliterates data and records on the PC.

5. Trojan: It is a kind of malicious code and computer software to look authorized but can control the system or machine.

6. RootKit: An assortment of vindictive programming created to enable access to a framework or on particular area of the system.

7. Backdoors: It is a method to convert the bypassing normal authentication and encryption.

8. Keyloggers: It is totally depend on Keyboard working style like the action of keyboard typically covertly unaware their actions are being method and monitored.

9. Ransomware: This is malware software but extreme use of this software is for Accounts section like access to system until a sum of money is paid.

10. Browser Hijacker: It is a type of undesirable programming that changes a program's setting without client authorization, to infuse the profitless promoting into program.

4.5.3 Detection Methodologies using Machine Learning

The possible machine learning techniques as well as deep learning techniques that are being used so far are discussed below- In year 2001, M. Schultz et al discussed about the idea of malware detection with machine learning and data mining techniques. Results mentioned below very much relatively better than the old traditional signature-based detection methods.

(a) Data acquisition: M. Schultz et al. in 2001 utilizes dataset of 4,267 programs fragmented into 1,001 clean programs and 3,266 malicious binaries.

(b) Information Preprocessing and Feature choice: There are no copy programs in the dataset and each model in the set is marked either malignant or generous with the assistance of an antivirus scanner. Furthermore, it additionally has three sorts of static highlights for preparing AI models so as to recognize and group malware. These kinds of highlights are as per the following: i) Portable Executable (PE): A library inside Bin-Utils that was libBFD is used to remove data from the versatile executable header. A portion of the highlights like size of document, names of powerfully connected libraries and progressively connected libraries capacity calls were gotten from the PE header. A portion of different highlights like rundown of DLLs were utilized by double and furthermore tally framework calls inside each powerfully connected libraries.

ii) String Sequences: Those highlights using string were furthermore isolated depending upon how these strings were encoded inside the records. In view of the experimentation, M. Schultz

et. al in 2001 found that cord designs in every single spotless document were comparable and this made it not quite the same as malware records as malware records had various examples. Along these lines of location isn't not quite the same as signature- based detection. In any case, M. Schultz et. al utilized it as a component for the classification model. The significant issue with these kinds of features is that they are lacking in robustness as utilizing shrewd procedures they can without much of a stretch be changed, so M. Schultz et. al likewise utilized sequence of Bytes as another Component. iii) Sequence of Bytes: Utilized n-gram set up together techniques as for executable records, using ngram and device called hexdump hexadecimal archives can be gained from the twofold records. On the off chance that we separate these features and different features, M. Schultz et. al found that Sequences of bytes were generally important as it had machine code executable when diverged from asset data, for instance, conservative executable highlights.

(c) Machine learning calculations Ripper algorithm : It is a standard based student that assembles a lot of guidelines that recognize the classes while limiting the measure of blunder. This calculation creates an identification model made out of asset decides that is worked to distinguish future instances of malignant executable records. This calculation utilizes libBFD data as highlights. Naive Bayes : Naive Bayes is a direct framework for building categorizers: models that dispense class names to give models, addressed as vectors of highlight esteems, where the class imprints are drawn from some constrained set. In various valuable applications, parameter estimation for credulous Bayes models uses the methodology for most extraordinary probability. We can utilize every one of our highlights to group malware or not. For our needs, we can utilize it to discover the probability of being malware given every one of the highlights.

Multi-Naïve Bayes: The Multinomial Naive Bayes categorizer is appropriate for order with discrete highlights e.g. Word counts for content arrangement. The multinomial appropriation ordinarily requires whole number component checks. By applying multinomial distribution on the feature set which seems to be a reasonable reason to classify malware or not.

Many machine learning algorithms for vulnerability detection are there and they are-

1. SVM : SVM assembles a hyperplane or set of hyperplanes in a high or boundless dimensional space, which can be used for arrangement. A decent partition is practiced by the hyperplane that has the greatest partition to the nearest getting ready data motivation behind any class (called useful edge), since when everything is said in done greater edge, lower speculation mistake of characterizer.

2. KNN : KNN may be utilized for arrangement and relapse issues. Although in our problem set, it is used to classify malwares in view of those k preparing models or instances, which are in majority with respect to the input i.e. which class it closely associated with. There are only two classes which the input can be associated with one is malware detected or not. KNN basically using for Classification techniques but in Malware Detection many terminally says and according to our study also its evaluate depends on the "ease to interpret output, calculation time and predictive power".

3. J48 Decision Tree : Decision tree is a structure that consolidates a root hub, branches and leaf hubs. Each hub connotes a test quality, each branch implies the consequence of the test, and each leaf hub holds class name. Decision tree doesn't require any area learning yet it

utilizes the idea of data entropy and it is anything but difficult to fathom, and the learning and characterization steps of choice tree are straightforward and quick.

4. Multi-layer Perceptron : For malware detection, multi-layer perception can be used. Basically Perception multiplies with weights and adds bias in one layer only. It is a linear categorizer also. A Multiple Layer Perceptron can be thought of, therefore, as deep artificial neural network.

4.6 An Attention-Based Graph Neural Network for Spam Bot Detection in Social Networks

4.6.1 Introduction

In recent years, the use of social platforms has increased significantly and as a result the vulnerability of the social accounts are also taken into consideration. Numerous malicious behaviors like malicious links ads etc are spreading in the web. The spam bot is a type of account that uses automated programs to spread malicious connections,phising links or unsolicited context in social networks. The growing number of users and the less secured account types make them ideal to target for spam bots. Manually managing large number of spam accounts will cause high cost and so spammers use the Twitter API to create custom programs. Almost 9 percent to 15 percent active twitter accounts are spam bots[18]. Just by cosidering the behaviours of one account, it is can not be said whether the account is a spam bot or not as they tend to imitate the legitimate users and their tweet or post frequencies.

4.6.2 The Method of spam Detection

In this part, we first describe the type of this problem to analyse it. Then we model the approach to a solution.

Formulation Of The Problem

Spam bot detection is a binary classification problem and to resolve this, we need a classifier that can accurately assign labels to accounts in the test set. For an example, lets say a social graph G=(V,E). The main purpose is to build the classifier function f: N-¿Y where N denotes the account set that are classified into the correct class with the credibility label Y by using the multi element information in account propagation. Both account information and network structure information are integrated is this process.

Subgraph Construction

The model of the user follow graph and user retweet graph are introduced.

User follow graph

Firstly lets take user follow graph as a directed graph $G_f = (v, E_f)$ where each node $u_i \in V$ represents users and each edge $(u_i, u_j) \in E_f$ represents a follow relationship. Now Lets take E_1 as a set of undirected edges e.g $E_1 = (ui, uj)|(ui, uj) \in E_f and (u_j, u_i) \notin E_f$ and E2 as bidirectional set

of edges. As the figure follows, according to the social networks definition, edge(u1,u2) represents user u1 is a follower of user u2. The bidirectional edge(u2,u3) represents user u2,u3 are mutual friends and follow each other. There is no connection between u1 and u3. Our goal is to find all neighbor relations of user ui as Ef= E1 U E2. In an unidirectional relationship,if the user ui has many incoming unidirectional edges, the user tends to be marked as a legitimate account as the anomaly accounts in social platforms are unlikely to attract legitimate users whereas if the user ui has many unidirectional outgoing neighbors then ui is likely to be a spam bot as it needs to pay attention to a large number of other accounts to gain information.

User Retweet Sub Graph

Lets take another directed graph Gr=(V, Er) as user retweet sub graph where each node $u_i \in V$ represents a user in social networks. From the following figure, if user u2 tends to retweet the tweet of user u1 then we are considering the reverse edges according to the figure. According to the figure, if u2,u4,u5 all retweet u1's and u1 retweets nobody then u1 may be the influential node and if user u1 is anomaly account then u2,u4,u5 also tend to be marked as anomaly accounts.

Gat Based Model

The Graph Attention level mechanism is basically used here for the spam bot detection. Here the attention mechanism used to learn the importance of different nodes. In a given relationship, since each neighbor has different degrees of influence on the user, the purpose of node level attention mechanism is to learn the importance of neighbors to this particular node and assign different attention values to all neighbors.

Chapter 5

Tools and Data Sets

5.1 Tools

5.1.1 GraphSage

GraphSage[1] algorithm can be considered as a stochastic generalization of graph convolutions, and it is especially useful for massive, dynamic graphs that contain rich feature information. It is basically a framework which is used to generate low-dimensional vector representations for nodes, and is especially useful for graphs that have rich node attribute information. It is an iterative algorithm that learns graph embeddings for every node in a certain graph. The goal of GraphSAGE is to learn a representation for every node based on some combination of its neighbouring nodes.

5.1.2 Graph Convolutional Networks

This is a TensorFlow implementation of Graph Convolutional Networks[2] for the task of (semi-supervised) classification of nodes in a graph. The choice of convolutional architecture is motivated via a localized first-order approximation of spectral graph convolutions. The model scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes. We use a two layered GCN for our experiments.

5.1.3 PyTorch Geometric

PyG (PyTorch Geometric[3]) is a library built upon the popular PyTorch deep learning platform. It has features to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data. PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

[1] https://github.com/williamleif/GraphSAGE
[2] https://github.com/tkipf/gcn/
[3] https://pytorch-geometric.readthedocs.io/en/latest/

5.2 Data Sets

We use our experiments starting with node classification and link predictions using gcn and following them by using graph neural network to find botnets in a network server. for the first two experiments, we used CORA datasets and for the botnet detection technique, we used 'chord','debru','kadem','leet','c2','p2p' datasets.

Chapter 6

Experiments

6.1 Node Classification with Graph Neural Networks

6.1.1 Tools Used

SOFTWARE USED

Pycharm community edition 2021.3.2(Windows 64 bit)

PYTHON VERSION

Python 3.9.0

LIBRARIES USED

1. pytorch
2. dgl(deep graph library)
3. iteratetools
4. numpy
5. scipy

6.1.2 Introduction

Many data-sets in various machine learning (ML) applications have structural relationships between their entities, which can be represented as graphs. Such application includes social and communication networks analysis, traffic prediction, and fraud detection. Graph representation Learning aims to build and train models for graph data-sets to be used for a variety of ML tasks.

This example demonstrate a simple implementation of a Graph Neural Network (GNN) model. The model is used for a node prediction task on the Cora dataset to predict the subject of a paper given its words and citations network. At first we import all the libraries mentioned.

Note that, we implement a Graph Convolution Layer from scratch to provide better understanding of how they work.

```
import dgl
import torch
import torch.nn as nn
import torch.nn.functional as F
import itertools
import numpy as np
import scipy.sparse as sp
```

Figure 6.1: Importing Libraries

6.1.3 Prepare the Dataset

The dataset on which we will going to work is imported through the dgl library.

The Cora dataset consists of 2,708 scientific papers classified into one of seven classes. The citation network consists of 5,429 links. Each paper has a binary word vector of size 1,433, indicating the presence of a corresponding word. A DGL Dataset object may contain one or multiple graphs. The Cora data-set used in this tutorial only consists of one single graph.

A DGL graph can store node features and edge features in two dictionary-like attributes called ndata and edata. In the DGL Cora dataset, the graph contains the following node features:

1. `train_mask`: A boolean tensor indicating whether the node is in the training set.
2. `val_mask`: A boolean tensor indicating whether the node is in the validation set.
3. `test_mask`: A boolean tensor indicating whether the node is in the test set.
4. `label`: The ground truth node category.
5. `feat`: The node features.

6.1.4 Building The Model

After loading the dataset, we defined a two layer gcn where each layer computes new node representations by aggregating neighbor information. To build a multi-layer GCN you can simply stack dgl.nn.GraphConv modules, which inherit torch.nn.Module. DGL provides implementation of many popular neighbor aggregation modules. One can easily invoke them with one line of code.

6.1.5 Training the GCN

Next, we tend to train the gcn to predict the net loss. We look about 24 number of results that we took in a interval of 5. We tend to use the gpu for training and for that, we load the model and the graph both. I use my laptop graphics card(NVIDIA GTX 1650) for the training purpose.

6.1.6 Plotting the graphs

We use `display_learning_curves(history)` to display the features into the graphs.

```
def train(g, model):
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    best_val_acc = 0
    best_test_acc = 0

    features = g.ndata['feat']
    labels = g.ndata['label']
    train_mask = g.ndata['train_mask']
    val_mask = g.ndata['val_mask']
    test_mask = g.ndata['test_mask']
    for e in range(100):
        # Forward
        logits = model(g, features)

        # Compute prediction
        pred = logits.argmax(1)

        # Compute loss
        # Note that you should only compute the losses of the nodes in the training set.
        loss = F.cross_entropy(logits[train_mask], labels[train_mask])

        # Compute accuracy on training/validation/test
        train_acc = (pred[train_mask] == labels[train_mask]).float().mean()
        val_acc = (pred[val_mask] == labels[val_mask]).float().mean()
        test_acc = (pred[test_mask] == labels[test_mask]).float().mean()

        # Save the best validation accuracy and the corresponding test accuracy.
        if best_val_acc < val_acc:
            best_val_acc = val_acc
            best_test_acc = test_acc

        # Backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if e % 5 == 0:
            print('In epoch {}, loss: {:.3f}, val acc: {:.3f} (best {:.3f}), test acc: {:.3f} (best {:.3
                e, loss, val_acc, best_val_acc, test_acc, best_test_acc))
model = GCN(g.ndata['feat'].shape[1], 16, dataset.num_classes)
train(g, model)
```

Figure 6.2: Training And Testing Accuracies on the dataset

6.2 Link Prediction using Graph Neural Networks

6.2.1 Tools Used

SOFTWARE USED

Pycharm community edition 2022.1.2 (Windows 64 bit)

PYTHON VERSION

Python 3.9.0

LIBRARIES USED

1. pytorch
2. dgl(deep graph library)
3. iteratetools
4. numpy
5. scipy

6.2.2 Introduction

Many applications such as social recommendation, item recommendation, knowledge graph completion, etc., can be formulated as link prediction, which predicts whether an edge exists between two particular nodes. This practical shows an example of predicting whether a citation relationship, either citing or being cited, between two papers exists in a citation network. It formulates the edge prediction problem as a binary classification problem as follows:

1. Treat the edges in the graph as positive examples.
2. Sample a number of non-existent edges (i.e.node pairs with no edges betweenthem) as negative examples.
3. Divide the positive examples and negative examples into a training set and a test set.
4. Evaluate the model with any binary classification metric such as Area Under Curve (AUC).

At first we import all the libraries mentioned.

Note that, we implement a Graph Convolution Layer from scratch to provide better understanding of how they work.

```
import dgl.data

dataset = dgl.data.CoraGraphDataset()
g = dataset[0]
```

Out:

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
```

Figure 6.3: Loading the dataset

6.2.3 Preparing The Dataset

The dataset on which we will going to work is imported through the dgl library. The Cora dataset consists of 2,708 scientific papers classified into one of seven classes. The citation network consists of 5,429 links. Each paper has a binary word vector of size 1,433, indicating the presence of a corresponding word.

A DGL Dataset object may contain one or multiple graphs. The Cora dataset used in this tutorial only consists of one single graph. A DGL graph can store node features and edge features in two dictionary-like attributes called ndata and edata. In the DGL Cora dataset, the graph contains the following node features:

1. `train_mask`: A boolean tensor indicating whether the node is in the training set.
2. `val_mask`: A boolean tensor indicating whether the node is in the validation set.
3. `test_mask`: A boolean tensor indicating whether the node is in the test set.
4. `label`: The ground truth node category.
5. `feat`: The node features.

6.2.4 Prepare Training and Testing sets

It randomly picks 10% of the edges for positive examples in the test set, and leave the rest for the training set. It then samples the same number of edges for negative examples in both sets.

```
# Split edge set for training and testing
u, v = g.edges()

eids = np.arange(g.number_of_edges())
eids = np.random.permutation(eids)
test_size = int(len(eids) * 0.1)
train_size = g.number_of_edges() - test_size
test_pos_u, test_pos_v = u[eids[:test_size]], v[eids[:test_size]]
train_pos_u, train_pos_v = u[eids[test_size:]], v[eids[test_size:]]

# Find all negative edges and split them for training and testing
adj = sp.coo_matrix((np.ones(len(u)), (u.numpy(), v.numpy())))
adj_neg = 1 - adj.todense() - np.eye(g.number_of_nodes())
neg_u, neg_v = np.where(adj_neg != 0)

neg_eids = np.random.choice(len(neg_u), g.number_of_edges())
test_neg_u, test_neg_v = neg_u[neg_eids[:test_size]], neg_v[neg_eids[:test_size]]
train_neg_u, train_neg_v = neg_u[neg_eids[test_size:]], neg_v[neg_eids[test_size:]]
```

Figure 6.4: Training and Testing the Dataset

6.2.5 Define a GraphSAGE Model

We now build a model consisting of two GraphSAGE layers, each computes new node representations by averaging neighbor information. DGL provides dgl.nn.SAGEConv that conveniently creates a GraphSAGE layer.

The model then predicts the probability of existence of an edge by computing a score between the representations of both incident nodes with a function (e.g. an MLP or a dot product).

6.2.6 Positive Graph, Negative Graph, and `apply_edges`

DGL recommends to treat the pairs of nodes as another graph, since one can describe a pair of nodes with an edge. In link prediction, we will have a positive graph consisting of all the positive examples as edges, and a negative graph consisting of all the negative examples. The positive graph and the negative graph will contain the same set of nodes as the original graph. This makes it easier to pass node features among multiple graphs for computation. As we will see later, we can directly feed the node representations computed on the entire graph to the positive and the negative graphs for computing pair-wise scores.

The benefit of treating the pairs of nodes as a graph is that you can use the `DGLGraph.apply_edges` method, which conveniently computes new edge features based on the incident nodes' features and the original edge features (if applicable).

DGL provides a set of optimized builtin functions to compute new edge features based on the original node/edge features. For example, `dgl.function.u_dot_v` computes a dot product of the incident nodes' representations for each edge.

```
train_pos_g = dgl.graph((train_pos_u, train_pos_v), num_nodes=g.number_of_nodes())
train_neg_g = dgl.graph((train_neg_u, train_neg_v), num_nodes=g.number_of_nodes())

test_pos_g = dgl.graph((test_pos_u, test_pos_v), num_nodes=g.number_of_nodes())
test_neg_g = dgl.graph((test_neg_u, test_neg_v), num_nodes=g.number_of_nodes())
```

Figure 6.5: Dividing Positive And Negetive Graphs

6.2.7 Training loop

After we defined the node representation computation and the edge score computation, can go ahead and define the overall model, loss function, and evaluation metric.

6.3 Automatic Botnet Detection Using Graph Neural Networks

6.3.1 Tools used-

SOFTWARE USED-

Pycharm community edition 2022.1.2 (Windows 64 bit)

PYTHON VERSION-

Python 3.9.0

LIBRARIES USED-

1. io
2. os
3. sys
4. shutil
5. tqdm

[1]

Introduction

In computer networks, botnets are the network compromised computers that are used to perform various malicious activities.

[1] https://github.com/harvardnlp/botnet-detection

```
# ----------- 3. set up loss and optimizer -------------- #
# in this case, loss will in training loop
optimizer = torch.optim.Adam(itertools.chain(model.parameters(), pred.parameters()), lr=0.01)

# ----------- 4. training -------------------------------- #
all_logits = []
for e in range(100):
    # forward
    h = model(train_g, train_g.ndata['feat'])
    pos_score = pred(train_pos_g, h)
    neg_score = pred(train_neg_g, h)
    loss = compute_loss(pos_score, neg_score)

    # backward
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if e % 5 == 0:
        print('In epoch {}, loss: {}'.format(e, loss))

# ----------- 5. check results ------------------------ #
from sklearn.metrics import roc_auc_score
with torch.no_grad():
    pos_score = pred(test_pos_g, h)
    neg_score = pred(test_neg_g, h)
    print('AUC', compute_auc(pos_score, neg_score))


# Thumbnail credits: Link Prediction with Neo4j, Mark Needham
# sphinx_gallery_thumbnail_path = '_static/blitz_4_link_predict.png'
```

Out:

```
In epoch 0, loss: 0.6930302381515503
In epoch 5, loss: 0.6705640554428101
In epoch 10, loss: 0.6015239953994751
In epoch 15, loss: 0.5163750648498535
In epoch 20, loss: 0.4807712435722351
In epoch 25, loss: 0.44074806571006775
In epoch 30, loss: 0.4119892120361328
In epoch 35, loss: 0.38509640097618103
In epoch 40, loss: 0.3585395812988281
In epoch 45, loss: 0.333467572927475
In epoch 50, loss: 0.30942121148109436
In epoch 55, loss: 0.2853710651397705
In epoch 60, loss: 0.2618500590324402
In epoch 65, loss: 0.23874278366565704
In epoch 70, loss: 0.21666289865970612
In epoch 75, loss: 0.1949688047170639
In epoch 80, loss: 0.17378462851047516
In epoch 85, loss: 0.15328189730644226
In epoch 90, loss: 0.13332721590995789
In epoch 95, loss: 0.11427388340234756
AUC 0.857493766986366
```

Figure 6.6: Results

6.3.2 How does it works?

Botnets receives commands from a botmaster through centralised command and control or decentralized peer to peer structures. As (`c&c`) protocol is detectable, botnets uses P2P method in a more specific way as this is harder to detect. Botnets has fast mixing rates that diffuses information. There are many detection approaches of botnets and one of them is on the mixing rates, though a major obstacle is that the massive scale of network communication makes it hard to differentiate botnet communication patterns from background internet traffic. For that, many human power as well as multiple pre-filters were required. This is why we tend to propose to add GNN to identify the botnets. As nodes update their states and exchange information by passing messages to their neighboring nodes so the model can automatically identify node dependencies in the graph after many layers of message passing. So we do not need any explicit filters or any manual labour.

6.3.3 Loading the Botnet Dataset

We provide standard and easy-to-use data-set and data loaders, which automatically handle the dataset downloading as well as standard data splitting, and can be compatible with most of the graph learning libraries by specifying the `graph_format` argument. The choices for dataset name are (indicating different botnet topologies):

1. 'chord' (synthetic, 10k botnet nodes)
2. 'debru' (synthetic, 10k botnet nodes)
3. 'kadem' (synthetic, 10k botnet nodes)
4. 'leet' (synthetic, 10k botnet nodes)
5. 'c2' (real, 3k botnet nodes)
6. 'p2p' (real, 3k botnet nodes)

The choices for dataset `graph_format` are (for different graph data format according to different graph libraries):

1. 'pyg' for PyTorch Geometric
2. 'dgl' for DGL
3. 'nx' for NetworkX
4. 'dict' for plain python dictionary

Based on different choices of the above argument, when indexing the botnet dataset object, it will return a corresponding graph data object defined by the specified graph library.

The data loader handles automatic batching and is agnostic to the specific graph learning library.

```
from botdet.data.dataset_botnet import BotnetDataset
from botdet.data.dataloader import GraphDataLoader

botnet_dataset_train = BotnetDataset(name='chord', split='train', graph_format='pyg')
botnet_dataset_val = BotnetDataset(name='chord', split='val', graph_format='pyg')
botnet_dataset_test = BotnetDataset(name='chord', split='test', graph_format='pyg')

train_loader = GraphDataLoader(botnet_dataset_train, batch_size=2, shuffle=False, num_workers=0)
val_loader = GraphDataLoader(botnet_dataset_val, batch_size=1, shuffle=False, num_workers=0)
test_loader = GraphDataLoader(botnet_dataset_test, batch_size=1, shuffle=False, num_workers=0)
```

Figure 6.7: Importing Botnet DataSets

6.3.4 To Evaluate a Model Predictor

We prepare a standardized evaluator for easy evaluation and comparison of different models. First load the data-set class with Botnet-Dataset and the evaluation function `eval_predictor`. Then define a simple wrapper of your model as a predictor function (see examples), which takes in a graph from the dataset and returns the prediction probabilities for the positive class (as well as the loss from the forward pass, optionally).

We mainly use the average F1 score to compare across models. For example, to get evaluations on the chord test set:

6.3.5 To Train a Graph Neural Network for Topological Botnet Detection

We provide a set of graph convolutional neural network (GNN) models here with PyTorch Geometric, along with the corresponding training script (note: the training pipeline was tested with PyTorch 1.2 and torch-scatter 1.3.1). Various basic GNN models can be constructed and tested by specifying configuration arguments:

1. Number of layers, hidden size
2. Node updating model each layer (e.g. direct message passing, MLP, gated edges, or graph attention)
3. Message normalization
4. Residual hops
5. Final layer type etc. (check the model API and the training script)

```
from botdet.data.dataset_botnet import BotnetDataset
from botdet.eval.evaluation import eval_predictor
from botdet.eval.evaluation import PygModelPredictor

botnet_dataset_test = BotnetDataset(name='chord', split='test', graph_format='pyg')
predictor = PygModelPredictor(model)    # 'model' is some graph learning model
result_dict_avg, loss_avg = eval_predictor(botnet_dataset_test, predictor)

print(f'Testing --- loss: {loss_avg:.5f}')
print(' ' * 10 + ', '.join(['{}: {:.5f}'.format(k, v) for k, v in result_dict_avg.items()]))

test_f1 = result_dict_avg['f1']
```

Figure 6.8: Training The Datasets

Chapter 7

Conclusion and Future Work

There are rapid advancements in AI,GNN,ML technologies for Cyber Security controls particularly around Intrusion Detection Systems. The Cyber Security industry is struggling to meet demand for professionals so the technology can help ease the pressure and ensure as technology evolves so does the dynamic prevention of attacks.

However, this technology will ultimately be used by hackers to perform more sophisticated attacks. Botnets that utilise these algorithms could passively find a way to penetrate a network and exploit a vulnerability. The GNN algorithms need to learn about other GNN's and whether one is malicious. That will be the key to using AI in Cyber Security, it must also recognise when it's own technology is used against it.

In conclusion, GNN and ML is a complex topic of data science technologies that requires deep understanding of mathematical equations and neural receptors and one needs to get more used to it to build a more secure systems that can help to solve the vulnerabilities among the network ports and the cyber world.

Bibliography

[1] The amazing applications of graph neural networks. https://insidebigdata.com/2021/06/26/the-amazing-applications-of-graph-neural-networks/. Accessed: 12-12-2021.

[2] Graph neural network and some of gnn applications – everything you need to know. https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications. Accessed: 12-12-2021.

[3] Graph neural networks. https://gnn.seas.upenn.edu/. Accessed: 12-12-2021.

[4] Graph neural networks: A review of methods and applications. https://arxiv.org/pdf/1812.08434.pdf. Accessed: 12-12-2021.

[5] How to get started with graph machine learning. https://gordicaleksa.medium.com/how-to-get-started-with-graph-machine-learning-afa53f6f963a. Accessed: 12-12-2021.

[6] J. Busch, A. Kocheturov, V. Tresp, and T. Seidl. NF-GNN: network flow graph neural networks for malware detection and classification. *CoRR*, abs/2103.03939, 2021.

[7] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks, 2015. cite arxiv:1511.05493Comment: Published as a conference paper in ICLR 2016. Fixed a typo.

[8] S. Liu, T. Liu, H. Ding, B. Tang, X. Wang, Q. Chen, J. Yan, and Y. Zhou. A hybrid method of recurrent neural network and graph neural network for next-period prescription prediction. *International Journal of Machine Learning and Cybernetics*, 11:2849 – 2856, 2020.

[9] L. Lu, R. Safavi-Naini, M. Hagenbuchner, W. Susilo, J. Horton, S. L. Yong, and A. C. Tsoi. Ranking attack graphs with graph neural networks. In F. Bao, H. Li, and G. Wang, editors, *Information Security Practice and Experience*, pages 345–359, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[10] R. Mijumbi, S. Hasija, S. Davy, A. Davy, B. Jennings, and R. Boutaba. Topology-aware prediction of virtual network function resource requirements. *IEEE Transactions on Network and Service Management*, 14(1):106–120, 2017.

[11] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[12] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks, 2017.

[13] L. Su, Y. Yao, Z. Lu, and B. Liu. Understanding the influence of graph kernels on deep learning architecture: A case study of flow-based network attack detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 312–318, 2019.

[14] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2017.

[15] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks?, 2018.

[16] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang. Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, 8:197158–197172, 2020.

[17] W. Zhang, H. Li, H. Wen, H. Zhu, and L. Sun. A graph neural network based efficient firmware information extraction method for iot devices. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[18] C. Zhao, Y. Xin, X. Li, H. Zhu, Y. Yang, and Y. Chen. An attention-based graph neural network for spam bot detection in social networks. *Applied Sciences*, 10(22), 2020.

[19] J. Zhou, Z. Xu, A. M. Rush, and M. Yu. Automating botnet detection with graph neural networks. *CoRR*, abs/2003.06344, 2020.

[20] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *CoRR*, abs/1909.03496, 2019.