Explaining GNN for Node Classification

Neha Pokharel and Paul Bergmann

Paderborn University

Abstract. We present an explainable approach for node classification of graphs. We preprocess a graph dataset, BAShapeDataset, by computing important graph features such as the number of neighbors, presence of cycles, and the number of neighbors within cycles. We then design a GCN model with two GraphConv layers to learn node representations from the input features. To improve interpretability, we train a Decision Tree classifier on the GCN predictions, considering both GCN features and graph structure.

Keywords: Graph Neural Network · Deep Graph Library · Explainable Approach.

1 Introduction

Making machine learning models predictable is of major importance to make them applicable to security-dependent areas. One well-known task of machine learning is the classification of nodes in a graph, i.e. giving the nodes of the graph a label depending on these features in the graph. This is often done using a Graph Neural Network (GNN), even though this approach is not directly explainable. Thus, other explainable models are used on the predictions of the GNN to understand the decision process of the GNN. In the following, we try to identify house-shaped subgraphs in a graph using a GNN. Afterward, we explain the GNN by clarifying its decision process using a decision tree.

2 Data Analysis

The data set consists of a graph in which nodes are labeled with values from zero to three. Thus, the data set is constructed for node classification. The graph is constructed as follows[1]:

1. A Barabási–Albert (BA) graph is constructed. This is a graph, constructed from an empty graph of m_0 nodes. Afterward, one after another node is added to the graph and randomly connected to $m \leq m_0$ nodes. Thereby, the probability of an edge to the new node is proportional to its degree. All nodes in the BA graph are labeled with zero.
2. Further graphs in which five nodes are constructed such that they have a house structure (house motives). I.e. a C_5 graph with an extra edge, connecting two vertices in the cycle. The middle, bottom, and top of the house motives are labeled one, two, and three, respectively.

3. The house motives are attached to the BA graph by merging one vertex from the motive with one vertex from the BA graph, keeping the label of the motive. Further, random edges are added.

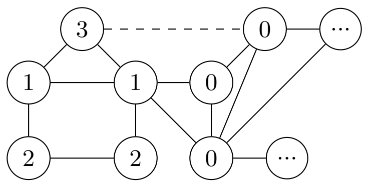

Fig. 1. BA graph with attached house motives. Random edges are displayed dashed.

3 Explanation of GNN

Graph Neural Network (GNN) handles the tasks associated with the graph data. The tasks includes node classification, link prediction and graph classification [2]. This document provides a detailed look at training and evaluating a GNN using the Deep Graph Library(DGL).

A graph is a data structure comprising vertices (nodes) and edges. Edges represent the relationship between nodes. In terms GNNs, a node often carries feature information and the edges define the graph's structure [3]. In a GNN, the graph convolution operation aggregates the feature information from a node's neighborhood. A neighborhood is defined as a set of nodes directly connected to the node via edges. GNNs combine feature information from the node itself and its neighbors to perform a graph convolution. This combination can be a simple average, weighted sum, or operation. The critical aspect is that the result should capture the node's information and structural position in the graph [4] [5].

GNNs are used in various domains, including social network analysis, molecular chemistry, recommendation systems and many more [2]. Anywhere some data can be represented as a graph, GNNs can be applied. However, developing GNN models often involves intricate graph handling, message-passing algorithms, and efficient computation. To address these challenges the Deep Graph Library (DGL) has emerged as a powerful tool for building and training GNNs.

3.1 DGL Library

The Deep Graph Library(DGL) is an open-source Python library for graph neural networks(GNNs). It was designed to provide a comprehensive framework for users interested in applying deep learning techniques to graph-structured data, a common format for many real-world problems ranging from social network analysis to molecular chemistry [6] [7].

DGL is developed by researchers from the Distributed Machine Learning Community (DMLC). The DMLC known for creating other influential machine learning libraries such as Apache MXNet and XGBoost released DGL in 2019 [6].

One of the main goals of DGL is to make it easy for developers and researchers to implement and innovate on graph network architectures. To that end, DGL provides a simple and flexible API for defining and working with graph data and graph computations. It supports creating, manipulating, and transforming graphs and performing operations like convolutions and aggregations on graph nodes and edges [7].

In addition to its API, DGL provides a variety of built-in graph neural network layers and models, such as Graph Convolutional Networks (GCNs), GraphSAGE, and GAT, among others. This allows users to rapidly prototype and experiment with these architectures without implementing them from scratch [8].

DGL is designed to be highly efficient and scalable. It takes advantage of the sparsity of graph data to perform computations more efficiently than dense tensor-based frameworks. It also supports multi-GPU and distributed training for large-scale graphs, making it suitable for research and production settings [8] [7].

Another feature of DGL is its interoperability with popular deep-learning frameworks. DGL can work alongside PyTorch, MXNet, and TensorFlow, allowing users to leverage the strengths of these libraries for tasks such as optimization, regularization, and normalization [8].

Overall, DGL provides a rich and user-friendly framework for applying graph neural networks to a wide range of tasks and is an important tool in any machine learning practitioner or researcher working with graph-structured data that is why we have choosen DGL framework for our mini project.

3.2 Implementation

In our implementation, we utilize the Deep Graph Library (DGL) to implement a Graph Convolutional Network (GCN). Specifically, we define a GCN model with two GraphConv layers, which perform graph convolutions to learn node representations. The DGL library provides convenient functions for graph manipulation, such as adding self-loops to the graph. The model is trained using the Adam optimizer and CrossEntropyLoss criterion.

4 GNN Training & Evaluation

Training a GNN is similar to training other neural network models which involves the iterative process of forward propagation, loss computation, backpropagation, and model weight updates [5].

4.1 GNN Training

Below is the Python function for a single training step.

```
def train():
    model.train() # Set the model to training mode
    logits = model(G, x) # Forward propagation
    # Compute the loss
    loss = criterion(logits[train_indices], y[train_indices])
    optimizer.zero_grad() # Reset gradients
    loss.backward() # Backpropagation
    optimizer.step() # Update model weights
    return loss.item()
```

In the above function, firstly the model is set to training mode. Then, propagate forward by feeding the graph 'G' and node features 'x' to the model obtaining predicted labels (logits). The loss is calculated by comparing the predicted labels with the true labels. Gradients is reset to prevent accumulation, backpropagation is performed to calculate gradients, and the model weights are updated.

This training function is typically executed inside a training loop where over the training data multiple times (epochs) are iterated. Monitoring the model's performance during training is common practice, usually after a set number of epochs.

4.2 GNN Evaluation

Below is the Python function for the evaluation.

```
def test():
    model.eval() # Set the model to evaluation mode
    with torch.no_grad(): # No need to calculate gradients
        logits = model(G, x) # Forward propagation
        # Get predicted classes
        _, indices = torch.max(logits[test_indices], dim=1)
        # Compare with true labels
        correct = torch.sum(indices == y[test_indices])
        # Calculate accuracy
        return correct.item() * 1.0 / len(test_indices)
```

In the above function, the model is set to the evaluation mode, disabling certain layers like Dropout. Then, forward propagation is performed as in training, but the calculation gradients is not needed this time since the model weights is being update. The class with the highest predicted probability is extracted as the model's prediction. Finally, the predictions to the true labels is compared to calculate the accuracy.

5 Explainable Approach

In this section, we explain our approach to explain the GNN. This includes the conversion from graph data to tabular data as well as the machine learning method on the tabular data.

5.1 Conversion from Graph Data to Tabular Data

We converted the graph to tabular data by measuring features for every vertex of the graph. These features we selected are the number of neighbors of the vertex, whether the vertex is in a cycle of length three, whether the vertex is in a cycle of length four, whether the vertex has a neighbor that is in a cycle of length three and whether the vertex has a neighbor that is in a cycle of length four. We selected all of these features for a reason. We assume that a vertex of the BA graph might have usually a higher degree than a vertex of a house motive. When a vertex is in a cycle of length three but not in a cycle of length four it could be a roof vertex. If the vertex is in a cycle of length four but not of length three it might be a bottom vertex. If a vertex is in both, a cycle of length three and four, it might be a middle vertex. When a vertex has no neighbor in a cycle of length three or no neighbor in a cycle of length four, it can not be part of a house motive. For an interpretation, of whether these features were sufficient to predict the GNN see section 6.

5.2 Decision Tree

We decided to use a decision tree to explain the GNN. The decision tree was constructed using the python scikit-learn[9] library. The properties of the model were set, such that the tree does not exceed a height of ten and with criterion entropy. This means that the Shannon entropy of the possible classes is calculated and used to minimize the logarithmic loss. Since the procedure is deterministic, the method is explainable. For every evolution step of the GNN, we retrained the decision tree on the predictions of the training set for the GNN. Afterward, we predicted the results of the GNN using the decision tree on the test data and compared the results. Further, the decision tree for the GNN after 10, 100, and 200 iterations is depicted in a diagram.

6 Conclusion

In the following, we interpret the results achieved through our graph neural network. This is done by analyzing distinct evolution steps of the GNN. Note that the numbers can vary when training the GNN again on the same data. Usually, the results are similar to the following.

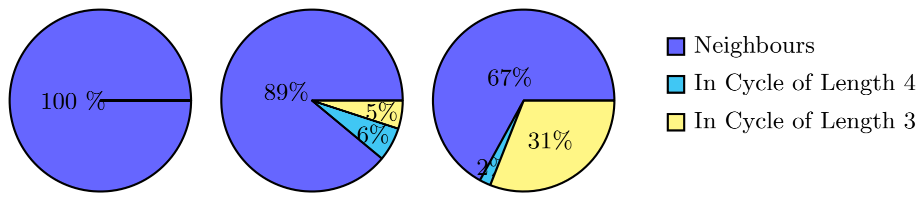

Fig. 2. Charts showing the feature importance after 10, 100, and 200 iterations of the GNN, respectively.

Figure 2 shows the feature importance for distinct training states of the GNN. At the first iterations, when the predictions of the GNN are about 36% accurate, the prediction is almost only based on the number of neighbors of a vertex. We say almost since the decision tree is in 3.7% of the cases wrong when predicting the GNN. If the decision of the GNN would only be based on the number of neighbors, the decision tree should have an accuracy of 100%. The more precise the GNN model gets, the vertex features of being in a cycle of length three or four play an increasing role. Nevertheless, the number of neighbors stays the most important feature. A reason for that might be that the vertices in a house motive tend to have a lesser degree than vertices in the BA graph. Whether a vertex has a neighbor in a cycle of length three or four has no influence on the decision of the GNN, i.e. their feature importance is zero. This is reasoned by the commonness of the property. Already a lot of vertices are either in a cycle of length three or four e.g. all vertices in a house motive. Thus, even if a vertex is not itself in a cycle, it is very common that one of its neighbors is. This holds per default for the house motive an is also usual for the BA graph due to the normally higher degree of the vertices.

Accuracy of GNN over Generations		
Epoch	GNN Accuracy	Decision Tree Accuracy
10	0.3611	0.9630
20	0.3611	1.0000
50	0.3704	0.9815
100	0.6759	0.7130
150	0.8148	0.9907
200	0.9815	0.9907

Table 1. Accuracy of predictions of the GNN over distinct generations and the accuracy of the explainable decision tree of predicting the results of the GNN.

Table 1 shows the accuracy of the GNN and the accuracy of the decision tree on the GNN predictions. At the beginning of the learning process - when the prediction is mainly based on the number of neighbors - and a the end of the learning process, the decision tree seems to be mostly accurate. Around iteration 100 of the GNN, the decision tree seems to be error prone with a precision of only 71% of predicting the GNN right. At this point, the GNN is still increasing its accuracy but is labeling still one of three vertices wrong. The inaccuracy of the decision tree indicates that the predictions of the GNN are mainly independent of the features in the tabular data.

7 Contributions of team members

7.1 Contributions by Neha

- Data Selection

- Implementation of GNN
- Add requirements file, readme file and format based on flake8.
- Documentation

7.2 Contributions by Paul

- Data Analysis
- Data conversion to tabular data
- Explainable model
- Documentation

References

1. "BAShapeDataset x2014; DGL 1.1.1 documentation — docs.dgl.ai." `https://docs.dgl.ai/generated/dgl.data.BAShapeDataset.html`. [Accessed 30-Jun-2023].
2. Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.
3. J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.
4. K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?," *arXiv preprint arXiv:1810.00826*, 2018.
5. K. Xu, M. Zhang, J. Li, S. S. Du, K.-i. Kawarabayashi, and S. Jegelka, "How neural networks extrapolate: From feedforward to graph neural networks," *arXiv preprint arXiv:2009.11848*, 2020.
6. "GitHub - dmlc/dgl: Python package built to ease deep learning on graph, on top of existing DL frameworks. — github.com." `https://github.com/dmlc/dgl`. [Accessed 28-Jun-2023].
7. M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.
8. "Welcome to Deep Graph Library Tutorials and Documentation x2014; DGL 1.1.1 documentation — docs.dgl.ai." `https://docs.dgl.ai/`. [Accessed 28-Jun-2023].
9. "1.10. Decision Trees — scikit-learn.org." `https://scikit-learn.org/stable/modules/tree.html`. [Accessed 30-Jun-2023].