A brief comparison between graph neural networks applied to Anti-Money Laundering task
Blockchain and Cryptocurrencies Course Project

Simone Marasi
Master's Degree in Artificial Intelligence, University of Bologna
simone.marasi@studio.unibo.it

Abstract

For my project I tackled the anti-money laundering problem with the end goal to classify illicit transactions and create a comparison between different architectures, in particular to compare different types of Graph Neural Networks (GNNs), to identify what are the key features and approaches that enable good performances in this given context. To do so, I tested a set of different models of increasing complexity against the same task, finding out that making use of the attention mechanism and graph sampling are two good approaches to achieve better performances.

1 Introduction

As part of my course project for the Blockchain and Cryptocurrencies course, I tackled an Anti-Money Laundering (AML) problem which is a crucial one when we deal with cryptocurrencies transactions, due its decentralized and less monitored nature. In this case, I focused on classification of illicit transactions, that is the primary goal of this kind of task. This task is very important for financial institutions to achieve compliance with legal requirements to actively monitor for and report suspicious activities. My goal was to create a comparison between different approaches, to find out which processes are better suited to solve this type of problem.

Weber et al. in their paper (Weber et al., 2019) investigated this field trying to apply some machine learning and deep learning techniques to identify possible suspicious transactions and labelling nodes which made them as illicit. In previous works it has been tried machine learning models in order to classify the nature of a node, obtaining quite good results for example with the use of Random Forest classifier, which outperforms even the Graph Convolutional Network approach. Also a temporal GCN (Evolve GCN (Pareja et al., 2020)) has been used, but with a very small improvement in performances. Other works, even more recently, have tackled the same problem improving the results. It is the case of (Lo et al., 2023), that used self supervised node embedding in GNNs.

My approach to this task was to implement several graph network solutions, that were structurally different, in order to highlight the differences:

- built-in PyTorch Geometric GNNs (GCN, GAT, GATv2, Chebyshev and GraphSage approaches);
- manual implementation of GAT, inspired however from the PyG model;

The problem described can be thought of in terms of node classification, in fact the final goal is to classify whether a node is considered licit or illicit. Moreover, the approach to use during the train of the graph is the transductive learning, where the model has already encountered both the training and the test input data (self-supervised approach). The drawback of this approach is that if a new node is added to the graph, we need to retrain the entire model from scratch.

I followed this approach since Graph Neural Networks differ in how they handle graph information and I think this could sensibly affect the final results.

To achieve my comparison I ran a total of six experiments based on the GNN architectures yet mentioned on the Elliptic Dataset. All experiments share the same data processing pipeline and the same training hyperparameters.

By comparing the results from my experiments I found out that attention is the main enabler of a good performance. In fact both GAT implementation and GATv2 improves a lot the metrics with respect to the simple GCN approach. Also using different strategies, such as graph sampling, could help to obtain good results. All the results are reported in Section 5.

2 Background

In this section I briefly explain the core differences between the models used and their peculiarity in the chosen domain.

2.1 Spectral methods

This kind of methods is maybe the most straightforward to understand in terms of idea. In fact, it has to do with the classic convolutional operator in the spectral domain and it uses the Fourier transform (and its inverse) to perform the operations needed to learn the parameters of the network. It acts globally so the main drawbacks could be the fact that it does not have the notion of locality inherently and that for large graphs obviously is quite inefficient. To this type of network belongs both ChebNets and the classical GCNs.

The former uses the Chebyshev expansion of order K to try to deal with the problem of locality. In fact in this way only a k-hop neighborhood is considered for the computation of the convolutional operations. GCNs are much more computationally effective but can not handle edge features and message passing between nodes, as spatial methods can do. Using only a 1-hop neighborhood, as the GCN does, lead to a smaller capacity to learn some generalization and for this reason it was the kind of network which gave the worst results.

2.2 Spatial methods

With spatial methods, instead, convolutions are executed directly on the graph, making some preliminary transformation on node features and then updating them based on node information and those of their neighbors. The spatial method used in the project is GAT (Graph Attention Network) in its three variants. Two of them are directly implemented using PyTorch Geometric built-in layers and one is made by hand and was inspired by one of them. The key behind the strength of GATs is the use of a learnable coefficient in the node-wise update rule and applying an attention function to it. The main problem in the standard GAT is that in the scoring function the learned layers *W* and *a* are applied consecutively, and can be collapsed into a single linear layer. The GATv2 modifies the order of operations. So, they apply *a* layer after the nonlinearity and the *W* layer after the concatenation. This dynamic attention provides also a much better robustness to noise. The difference is noticeable from the following formulas:

GAT:

$$e(h_i, h_j) = \text{LeakyReLU}(a^T[Wh_i||Wh_j])$$

GATv2:

$$e(h_i, h_j) = a^T\text{LeakyReLU}(W[h_i||h_j])$$

2.3 Sampling methods

Another type of GNN methods are sampling methods. They are generally used and well-performing in huge graphs due their ability to sample only a subset of them to conduct propagation instead of using all neighborhood information. One of the well known networks using these methods is GraphSage. In the script I used the PyG built-in SageConv layer to investigate how they perform on the Elliptic dataset.

3 System description

The PyTorch Geometric built-in networks were chosen picking one of them for each type of architecture to examine. Moreover, for the GAT network, a manual version was also implemented, inspired by the source code of the built-in one. Going one step deeper, it consists in a series of layers of attention layers stacked with the last one producing raw scores as output (it does not use any activation function). Each layer is implemented through different steps:

- Linear projection and regularization;
- Calculation of edge attention scores;
- Neighborhood aggregation;
- Residual and skip connections and concatenation of scores;

The different phases are better detailed below:

3.1 Linear projection and regularization

In this phase the input is passed through a dropout layer and then the node features are projected into *NH* independent output features (where *NH* is the number of attention head of the current layer).

3.2 Calculation of edge attention scores

Then it is applied the scoring function (element-wise product) for source and target scores, and then they are duplicated following only the edge index connections instead of preparing all possible combinations of scores. Then scores are softmaxed only over their neighborhoods, calculating in this way the attention scores taking into account only near feature vectors and getting rid of other edge scores that include nodes that are not in the neighborhood.

3.3 Neighborhood aggregation

After having computed attention scores it is performed the element-wise (aka Hadamard) product between projected node features and attention scores. Weighted and projected neighborhood feature vectors for every target node are then summed up.

3.4 Residual and skip connections and concatenation of scores

Last step for each layer is the one that adds residual connections to output features (projecting input feature vectors into dimension that can be added to output). Just in case it is not the last layer of the stack it concatenates the scores, obtaining the final feature representation with the expected shape of *(N, NH*FOUT)* where *N* is the number of nodes, *NH* the number of heads and *FOUT* the number of output features. Otherwise, if it is the last layer it computes the average of the scores, with the shape *(N, NFOUT)* and it returns them as raw scores (without applying the activation function as in the previous layers)

4 Data

I ran my experiments on the Elliptic public dataset, which maps Bitcoin transactions to real entities belonging to 2 categories: licit or illicit transactions. This is composed of 3 csv files: the first file maps nodes with their label (licit/illicit/unknown), the second file introduces the edges between 2 nodes by using their transaction ids and the last one has nodes ids with 166 features. A node in the graph represents a transaction, an edge represents a flow of Bitcoins between one transaction and the other. Each node has 166 features and has been labelled as being created by a "licit" (e.g. those made by regulated crypto exchanges), "illicit" (e.g. those made by dark markets) or "unknown" entity. There are 203.769 nodes and 234.355 edges in the graph, we have 2% (4.545) of the nodes that are labelled as illicit and 21% (42.019) are labelled licit. The remaining transactions are not labelled with regard to licit versus illicit. There are also 49 distinct time steps. The first 94 features represent local information about the transaction and the remaining 72 features are aggregated features obtained using transaction information one-hop backward/forward from the center node.

The dataset was parsed into a suitable tabular data structure (Pandas DataFrame). The training dataset has been split using 65% of the whole samples for training purposes using the others 15% and 20% for validation and test purposes respectively. It is also worth pointing out that it is guaranteed that the label proportions are maintained along the splits, thus avoiding unjustifiable boosting of results.

5 Experimental setup and results

We ran six experiments representative of the GNN approach: GCN, GAT, GATv2, custom GAT, Chebyshev network and GraphSage network.

5.1 Training details

I used the following training hyperparameters for all the models in exam:

- **Binary Cross Entropy** as a loss function;
- **Adam optimizer** with a learning rate of 1e-3;
- **800 epochs**;

5.2 Metrics

To evaluate the performance of the models I used the same metrics of the original paper (Weber et al., 2019) to have a possible comparison between the results obtained. These metrics are considered only for the illicit class, and they were precision, recall, F1 score and micro-averaged F1 score.

- **Precision:** it focuses on False Positive errors (in this case licit transactions classified as illicit). Precision does well in cases like this in which we would need to avoid false negatives.
- **Recall:** it focuses instead on False Negative errors, giving us an indication of missed positive predictions, meaning illicit transactions classified instead as licit.
- **F1 score:** it tries to find the balance between precision and recall by calculating their harmonic mean.
- **Micro-avg F1 score:** it is included for completeness and indicates the proportion of correctly classified observations out of all observations (both licit and illicit).

Model	Precision	Recall	F1	F1 Micro AVG
GCN	0.832	0.457	0.59	0.94
GAT	0.787	0.683	0.731	0.952
SAGE	0.931	0.788	0.853	0.974
Cheb	**0.942**	0.795	**0.862**	**0.976**
GATv2	0.891	**0.804**	0.845	0.972
Custom GAT	0.861	0.762	0.808	0.966

Table 1: Metrics obtained for all models tested on illicit class

5.3 Results

You can find the training results on the test set in Table 1 and by looking at the plot in Figure 1 to have an overall view of the results obtained.

6 Discussion

From the plot and the result's table we can notice that the GCN network is the worst in almost all metrics computed. Very bad results are reached in terms of recall due to its nature to consider only 1-hop neighborhood, keeping its learning capacity quite bad. For this kind of network the results are in line with those reported in the original paper (Weber et al., 2019). All GAT implementations overcome these results but with noticeable differences among them. In particular, the GATv2 network obtains the best result. This variant fixes an inherent problem of the simple GAT layers as described previously in the theory section.
The custom implementation of GAT instead performs better than the built-in GAT in all metrics considered. It is worth noting that the manual implementation of GAT takes more time to train, as expected, due to its not optimized nature. In general, the two best models are those using the GraphSage and Chebyshev methods. They obtain around 86% of F1 score, 94% of Precision and almost 80% in Recall, being overtaken in this score only by 1% by GATv2. The strength of these models lies in the fact that it manages to generalize based on sampling of the neighborhood of nodes, in the case of GraphSage and the use of a k-neighborhood in the case of Chebyshev network, as described in the previous section. The choice of kernel size is a hyperparameter that has been tuned to obtain the best possible result.

7 Conclusion

In conclusion, I can say that my experiments provided enough evidence to say that the simple GCN method does not provide good performance and the use of k-neighbors is necessary to improve a lot the performances of the model. Exploring different ways of sampling nodes and finally trying to re-implement manually the GAT network inspiring to PyTorch code, I have been able to compare results and to highlight strengths and weaknesses of the different models used.

I am happy with results regarding the manual implementation where I notice that they are in line or, in some trainings, even better than the built-in GAT network but since the model is not optimized it is much slower during the training. Some optimization of the code may result in better performances even in terms of speed.

8 Links to external resources

You can find the full code of the project here: https://github.com/simonemarasi/aml-elliptic-gnn

References

Wai Weng Lo, Gayan K Kulatilleke, Mohanad Sarhan, Siamak Layeghy, and Marius Portmann. 2023. Inspection-l: self-supervised gnn node embeddings for money laundering detection in bitcoin. *Applied Intelligence*, pages 1–12.

Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks

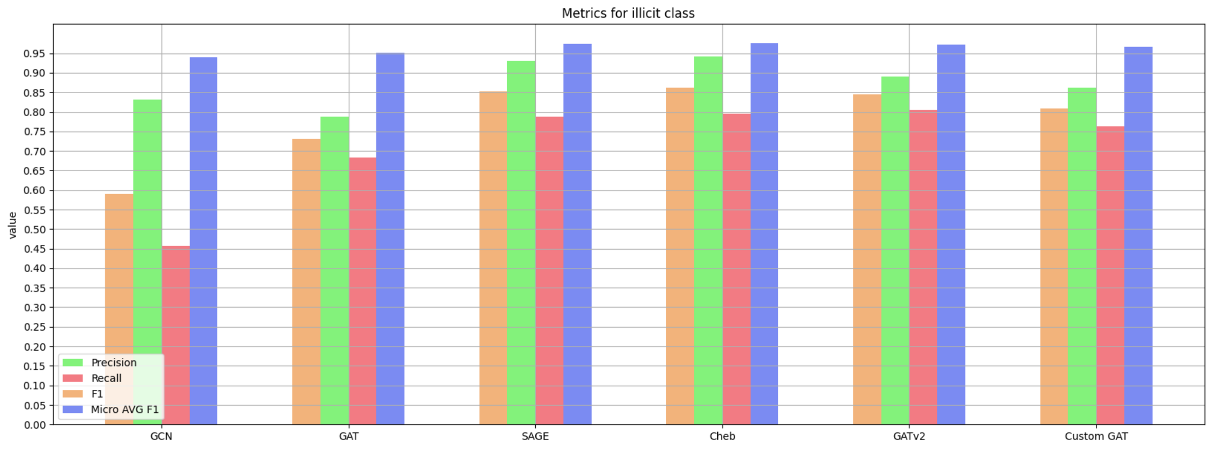

Figure 1: Comparison between metrics of different models on illicit class

for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5363–5370.

Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I Weidele, Claudio Bellei, Tom Robinson, and Charles E Leiserson. 2019. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. *arXiv preprint arXiv:1908.02591*.