Topological Data Analysis for Graph Classification

Tameez Latib
UCLA
tameezlatib@gmail.com

Abstract

Graph like data structures are abundant in every day life (e.g. molecules, relational data, etc) and thus Graph Neural Networks have become increasingly popular as tools to deal with this structured data. However, the field is still quite recent and many advances are yet to be made. Current approaches try to over-complicate algorithms by including attention layers or employing incredibly deep neural networks to extract necessary graph features. Instead I propose using topological data analysis- a tool to extract relevant graph features- before training with a standard graph convolution architecture. This method has been tested on the graph classification task and achieves SOTA accuracy on multiple datasets

1. Introduction

Graph data typically comes in the form of $G = (V, E, X)$ where V, E are the sets of vertices and edges, and X is a feature matrix- each node can have associated features. For datasets such as PROTEINS, ENYZMES, etc, the node feature is generally a label representing what class the node belongs. Therefore a more compact representation of our graph structure can be A, X, where $A \in \{0,1\}^{n\times n}$ is an adjacency matrix and $X \in \mathbf{R}^{n\times d}$ is still the feature matrix, for a graph with n nodes and d features per node. In the graph classification task, each graph G has an associated label y designating the class, and the goal is to predict y based on G. This has many uses, for example in the PROTEINS dataset we are predicting whether a protein structure is an enyzme or not based on the amino acid graph representation. Due to this representation, standard classifier methods such as decision trees, CNNs, etc have not been effective. However, recent methods inspired by Kipf-Welling's GCN (Graph convolutional network) have achieved reasonable results, and are the norm for graph classification. Looking at the leaderboards (https://paperswithcode.com/task/graph-classification), the vast majority of approaches employ some type of GNN, but modify the underlying architecture with more complicated mechanisms such as attention ([2]), or complex pooling mechanisms ([3], [5]) to deal with the rich topological information. In this report, I try to bridge the gap between the machine learning and mathematics communities, and review Topological Data Analysis (TDA) as an alternative to the previously mentioned methods, and provide a far simpler architecture capable of achieving SOTA accuracy (I.e. this approach outranks all others on multiple datasets).

2. TDA

2.1. Background

Topological data analysis is a tool to extract data from high dimensional datasets. Specifically, it deals with the topology or shape of the data and tries to encapsulate this topology data into a lower dimensional space. Note that this is similar to the premise of GCN or graph embedding models- it's the topology or the structure that we care about and we want to find some way to represent this. The other important thing about TDA is that it can extract features based on mathematical formula- i.e. it does not need to train.

Vietoris Rips Persistence (From the library [4]) for example takes some point cloud and produces triples containing topological features in the form of (birth, death, k) where birth is the appearance of a topological feature, death is the disappearance of that feature, and k is the homology dimension of the feature (a k dimensional hole- that is, a k-dim cycle not on the boundary of a (k+1)-dim object). It applies some function to the data repeatedly, and when such a k-dimensional hole first exists, it is marked as a birth. When that same feature disappears, we mark it as a death. The output of this is also known as a Persistence Diagram, and while it contains many useful features, it should be further processed to a lower dimensional vector space.

Persistence Entropy is one way to process this, by creating entropy measure for each homology dimension. That is, for fixed k, we have $E_k = -\sum_i p_i log(p_i)$ for $p_i = (d_i - b_i)/L$ where d_i, b_i are the ith features death/birth, and $L = \sum_i (d_i - b_i)$

While the topic of TDA has many other facets, the curious reader can see ([4], [1], etc) for more.

2.2. Previous TDA methods

The concept of TDA for use in graph classification is not new. In fact, the current best approach on the NCI1 dataset ([6]) uses TDA and learns metrics based on topology outputs. In fact, this approach exclusively uses graph structure (without attributes) and still received best accuracy! However, this method does not include GCN architecture, and because it does not use graph attributes, it performs poorly on other datasets where the feature matrix is important.

Here, I make use of a new python library ([4]) for all TDA related queries. From here on out, I will denote X_{tda} as the features from TDA with input as the adjacency matrix A. I.e. this is an embedding of graph structure.

2.3. Brief showcase

Table 1. Train/test accuracy using decision tree on X_{tda}

Acc.	ENZYMES	PROTEINS	MUTAG	NCI1
Train	.80	.85	.92	.72
Test	.31	.63	.82	.64
Random	.17	.50	.50	.50

In addition to [6], which shows the promise of TDA, I conduct brief experiment: How accurate can a simple decision tree get with only X_{tda}? Surprisingly, quite well. Testing on the datasets MUTAG, ENZYMES, PROTEINS, and NCI1, with each graph G having an associated $x_{tda} \in \mathbf{R}^8$ (4 homology dimensions, using two metrics including persistence entropy), the results are summarised in table 1. Note that there was a 70/30 training/test split using shuffled data. From the table, these decision trees achieve incredibly high training accuracy- On MUTAG, we have an amazing 82% test accuracy with the decision tree! However, they do suffer from overfitting. At the very least this gives some insight and motivation as to why TDA may be prove useful and sets a benchmark for other methods to beat.

3. Proposed Approach

Figure 1 summarizes the architecture. Note that the input X, A goes through the first branch of the network- a GCN (4 layers), then a single fully connected layer. The second branch of the network uses X_{tda}, passing it through a Fully connected network (2 layers). The two branches are added together as a simple form of data fusion, and then passed into another 3-layer FCN. Relu activation functions were used- others such as sigmoid or tanh did not produce the same results.

3.1. Padding

One problem is that the input does not have fixed length. There are 2 approaches to this. The first is to pad the inputs to a fixed length. The second is to allow variable inputs, but then use some pooling mechanism in the network to get a fixed size at some point in time. The second approach has some benefits, such as avoiding irrelevant operations on zero entries and allowing the network to work for arbitrary sized inputs in the special case that train and test datasets are quite different. However, in this special case, we already expect low accuracy training and testing sets having different distributions. Further, by using graph convolutions as the first layers, we don't see the padded zeros affect any of the outputs. Also we could use sparse multiplication to mitigate efficiency issues. Both approaches were tested (See the Experiments section), and it was determined that the padding version of the network outperformed, with little to no downsides.

3.2. Data augmentation

Since these are graphs, a very simple data augmentation that preserves class data is to apply a permutation. For example, take two nodes i, j and switch them in the adjacency and feature matrices. For the feature matrix, this amounts to swapping the i and j rows, whereas for the adjacency matrix we should swap i and j rows and also the i and j columns to preserve the connections. More generally, take some permutation matrix P (A matrix with entries in $\{0, 1\}$ with row and column sums adding to 1), and a training set sample A, X, and produce PAP, PX as a new training set sample. This creates extra data and helps the network learn more without overfitting.

3.3. Concatenation

A common operation in graph neural networks is to concatenate output layer results, and process this concatenation which contains multiple orders of graph convolutions. The idea is that this version of the output has more features from all orders of graph convolution, which in theory could produce better results. The downside is that this creates larger output layers, which then require more trainable parameters. In this architecture, concatenation occurs after the GCN part, before the FCN_1

3.4. Graph Convolution Rule

The standard graph convolution rule is $H^{(l+1)} = \sigma(D^{-1/2}AD^{-1/2}H^{(l)}W^{(l)})$, where D is a diagonal matrix created placing the column sum of A in the diagonal elements, σ is the activation function, $H^{(l)}$ is the lth layer output, and $W^{(l)}$ are the trainable weights. The idea is that only information between adjacent nodes can be propagated, instead of a FC layer which propagates information

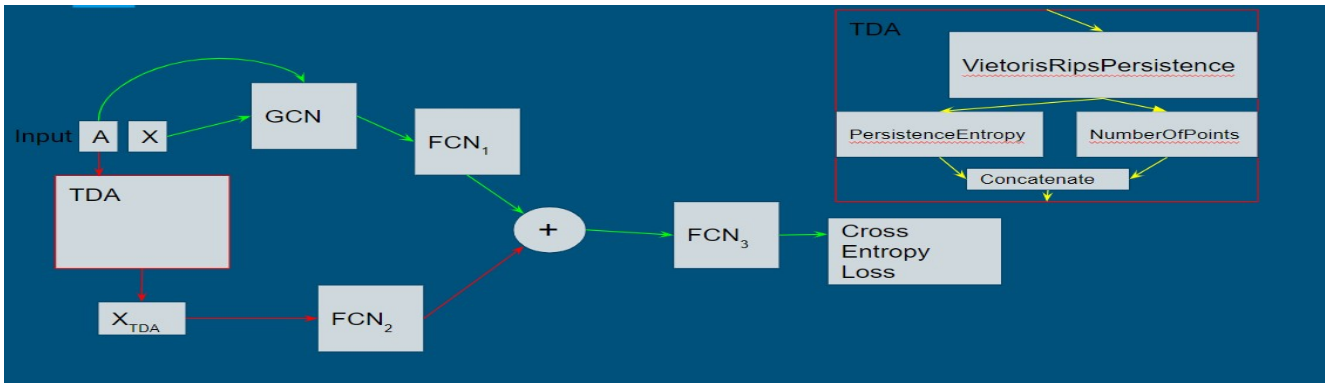

Figure 1. Proposed architecture, TDA net

from all input nodes to all output nodes. However, the extra D matrices weight this propagation and in theory add more topological information. I choose to experiment with both the expression $H^{(l+1)} = \sigma(D^{-1/2}AD^{-1/2}H^{(l)}W^{(l)})$ and $H^{(l+1)} = \sigma(AH^{(l)}W^{(l)})$, and find that for this task the second formulation works better. My insight is that using TDA already includes the topological information in the other branch, so on this branch we may only need to propagate features according to their neighbors. Therefore, we let the weights learn what is important instead of trying to use the Laplacian matrix ($D^{-1/2}AD^{-1/2}$), since the structure is already captured.

4. Implementation Details

All data was taken from https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets and it's important to note the format of this data- a few txt files with edges, graph labels, and node labels. So the first thing we need to do is store this into python data containers somehow. The txt files are processed into adjacency matrices and feature matrices and all stored in a single .npy file so that we don't have to process the data for multiple experimental runs.

Next step is to shuffle the data and separate train/test data. If applicable, we perform data augmentation on the training set to generate more samples. Ideally this would reduce overfitting from the network given its later fully connected layers.

Now we take all samples as input to the TDA section, which extracts useful features. Note that if testing data is unknown, we can always apply TDA at runtime before test data is thrown into the network. However, for convenience, it is easiest to apply TDA all at the same time and generate an X_{tda} for each graph. Now, our data set is a A, X, X_{tda}, y for each graph (Adjacency, node features/labels, topological features, and graph label).

Testing implementations with and without padding, it would appear that the padded version of input performs better, and so we must find the graph with maximum nodes before initializing and training a network. To extrapolate to the case where testing data is unknown, it is reasonable to suggest that training data and testing data are similar, and we may pad to $n_{max} * \beta$ where n_{max} is the maximum number of nodes from all training set graphs, and β adds some wiggle room (e.g. $\beta = 1.1$ for 10% increased size of test set graphs). Once finalised, we pad adjacency matrices and feature matrices with zeros to the bottom and right so that $A \in \mathbf{R}^{n_{max} \times n_{max}}$ and $X \in \mathbf{R}^{n_{max} \times d}$. Note that we don't pad $X_{tda} \in \mathbf{R}^8$, since we only use 4 homology dimension with two transformers.

Lastly, we train with basic SGD plus momentum. Since it's classification task, we use cross entropy loss. I output the train/test accuracy every 5 epochs of training, ideally training for 200 epochs. At first, I used the NCI1 dataset to test the implementation and try different architectures. Once the architecture was confirmed, the exact same architecture (without any hyperparameter changes) was used for the other datasets (ENZYMES, MUTAG, PROTEINS).

5. Experiments

These experiments were all conducted on the NCI1 dataset to find out which architecture works best. Specifically, I considered the cases of padding, concatenation, data augmentation, modifying graph convolution rule, and more/fewer layers, training up to 50 epochs for these test cases (and up to 200 epochs for the final results). Through experimentation, padding was found to be necessary. I believe the pooling functionality used was not extracting the necessary features when considering variable input graphs. So even though we lose the abstraction to allow arbitrarily large inputs, padding the input to fixed length increases accuracy (compare .67% train acc of no padding model to .75% train acc with padding model). Next, we also want to consider the effect of concatenation. From experimental data, concatenation has no considerable impact (on average about 3% improvement across all tests run) given that it requires far more trainable parameters and creates a somewhat inefficient network. Therefore we also choose not to use concatenation. Data augmentation actually did improve results, but it consequently took far longer to train. So I show final results with and without data augmentation. Similarly, inclusion of the Laplacian matrix instead of the adjacency matrix in the graph convolution does not affect performance. I believe this is due to the structural information already being in the TDA branch of the model. It was also found that FC layers after the GCN (before the graph conv and TDA branches merge) improve performance, and thus FCN_1 is included in the architecture in figure 1. Therefore, the final architecture does not use concatenation or the laplacian matrix formulation- but does use padding and optionally data augmentation.

6. Final Results

Table 2. Test accuracy of TDA net with (+DA) or without data augmentation after 200 epochs compared to current leaderboard approaches

Method	ENZYMES	PROTEINS	MUTAG	NCI1
with DA	.88	.91	.94	.88
without	.81	.92	.965	.90
current #1	.78	.85	1.00	.87
current #2	.71	.81	0.967	.86

As a final result of this project, note that this method outperforms current best approaches on ENZYMES, PROTEINS, and NCI1 datasets, whereas it achieves a close 3rd best on MUTAG dataset. Note that it considerably outperforms previous bests on enzymes/proteins. See table 2, where current best approaches are taken from https://paperswithcode.com/task/graph-classification. Furthermore, it outperforms on these datasets using the exact same architecture and structure; once the architecture and hyperparameters were finalised from the 50-epoch test on NCI1, only the dataset was swapped out.

7. Future Work

I believe it would be interesting to see a timing analysis between this method and other 'state of the art' methods; since the network here is relatively lightweight I would imagine far greater performance in this area. Going forward I would like to try a similar approach on other graph tasks like link prediction. However, I imagine this would be more involved since TDA would only encapsulate the graph structure from the adjacency matrix. It's still possible to create some embedding using the information from TDA, and then use that embedding in another algorithm. Nevertheless, I believe that TDA should be further studied for use in GCNs or other models too.

References

[1] N. Atienza, R. Gonzalez-Diaz, and M. Rucco. Persistent entropy for separating topological features from noise in vietoris-rips complexes, 2017. 2

[2] D. Q. Nguyen, T. D. Nguyen, and D. Phung. Universal graph transformer self-attention networks. *arXiv preprint arXiv:1909.11855*, 2019. 1

[3] A. Nouranizadeh, M. Matinkia, M. Rahmati, and R. Safabakhsh. Maximum entropy weighted independent set pooling for graph neural networks. *arXiv preprint arXiv:2107.01410*, 2021. 1

[4] G. Tauzin, U. Lupo, L. Tunstall, J. B. Pérez, M. Caorsi, A. M. Medina-Mardones, A. Dassatti, and K. Hess. giotto-tda: A topological data analysis toolkit for machine learning and data exploration. *Journal of Machine Learning Research*, 22(39):1–6, 2021. 1, 2

[5] Z. Zhang, J. Bu, M. Ester, J. Zhang, C. Yao, Z. Yu, and C. Wang. Hierarchical graph pooling with structure learning. *arXiv preprint arXiv:1911.05954*, 2019. 1

[6] Q. Zhao and Y. Wang. Learning metrics for persistence-based summaries and applications for graph classification. *arXiv preprint arXiv:1904.12189*, 2019. 2