Graph link prediction

Francesco Bottalico [787587]
f.bottalico15@studenti.uniba.it

Abstract

The aim of this project work is to explore the use of autoencoders for the task of link prediction on graphs. Autoencoders are able to reconstruct the starting graph while finding new latent links. We will evaluate different types of graph encoders, based on different theoretical approaches (convolution and attention), on different graph datasets. We present a way to train these models on very sparse graphs, using different methods of training, namely reconstruction-based and contrastive-based, the latter will exploit the partitioning of the original graph to learn better node embeddings. In order to use this approach, a simple method of graph clustering is presented.

1 Introduction

The task of *link prediction* on graph datasets consist on finding latent connections between two nodes. Two different settings of training and evaluation are possible for this task: transductive or inductive. In the first case both in training and testing we work on the same graph, so nodes and links in the training set are also visibile in the evaluation phase, differently, in the other setting we train our model on a graph to then evaluate it on a different one, with different nodes, in this work we will be in the first scenario. As for the model, this project will focus on the use of **Graph Autoencoders**, first introducted in Kipf and Welling (2016), an unsupervised way to discover latent links between nodes by using an autoencoder based architecture to reconstruct the original graph. We will try different types of encoders, convolution-based (Kipf and Welling, 2017) and attention-based (Veličković et al., 2018), then propose some modifications to enhance their performances. Since graph datasets have very sparse connections, we will show how to cope with this unbalance problem during training. Training will be done using a reconstruction-based loss and a contrastive-based loss (Chen et al., 2020), in this case we will try to exploit the partitioning of the original graph into subgraphs to learn different embeddings for the same nodes, to then aggregate them (Ucar, 2023). In order to partition the graph, three different methods are implemented: random edge splitting, Fiedler disjoint clustering and Fiedler clustering.

2 Related Work

As already said, we will use **Graph Autoencoders** (Kipf and Welling, 2016) with different encoders using various type of layers: graph convolutional layers (Kipf and Welling, 2017), graph attention layers (Veličković et al., 2018), graph convolutional layers with normalization (Ahn and Kim, 2021), graph transformers (Dwivedi and Bresson, 2021). To train these models we will use a reconstruction-based loss function as done in the original paper, then we will also try to use the contrastive framework presented in Chen et al. (2020) to enhance node embeddings. For partitioning the graph, other than using a randomized approach, we implement a very simple clustering method to optimally split the graph using the **Fiedler eigenvector**[1].

3 Datasets

Every dataset that will be used should be specified by two matrices: an adiacency matrix A of size$N \times N$ and a features matrix X of size $N \times D$, where N are the number of nodes in the dataset and D are the number of features for each node. Notice that the graphs are undirected, so A is a symmetrical matrix.

3.1 Preprocessing

The graph must be randomly splitted in training and test set, in such a way that there are no edges in common. This can be done through the `split_data` function, which is not used during

[1] https://en.wikipedia.org/wiki/Algebraic_connectivity

the experiment in favor of the `kfold` function, which splits the graph's positive edges in k disjoint sets, then returns an iterator on each fold. Specifically, for each iteration, we have: the adjacency matrix without the test edges, the set of training edges, a set of positive and negative test edges and a set of positive and negative validation edges (by default 5% of the training edges).

Negative sampling We assume we hare in a *closed world* scenario, that is if there isn't a connection between two nodes then we can take that as a *negative sample*. This is how negative edges in test and validation set are generated.

4 Training method

The rationale behind the training method of the autoencoders is quite simple: given the matrices A and X, we want to reconstruct A, the output of our model will be a matrix $\hat{A}$. Since A is a boolean matrix ($a_{ij} = 1$ if $i \to j$, 0 otherwise) we can interpret values of $\hat{A}$ (that will be in range $[0, 1]$) as probabilities and simply use the **binary cross entropy** loss in order to maximize the probability of positive edges while minimizing the negative ones. However this approach has mainly two problems:

1. Since A is very likely to be a sparse matrix, we have much more negative samples over positive ones.
2. We must avoid overfitting, as we don't want only to reconstruct the original adjacency matrix, but also discover latent links.

Unbalance problem To solve the first problem we simply weight more **false negative** errors over **false positive** ones. This is done by editing the binary cross entropy formula in order to weight more positive examples

$$\mathcal{L}_r = \sum_{i=0}^{N}\sum_{j=0}^{N} w(a_{ij}\cdot\log\hat{a}_{ij})+(1-a_{ij})\cdot\log(1-\hat{a}_{ij}) \tag{1}$$

w is the weight computed in the following way

$$w = \frac{TN}{TP} = \frac{N^2 - TP}{TP} \tag{2}$$

where TP are the number of positive edges and TN are the number of negative edges. Note that we should normalize the new cross entropy formula by multiplying it by a normalization factor $\frac{1}{2TN}$.

Avoid overfitting In order to avoid the second problem, many precautions have been taken:

1. Using a weight decay regularization during training
2. Keeping the encoder and decoder architecture simple
3. Applying dropout in attention-based models
4. Using *early stopping*

Many of these thing will be further explained in section 6. As for the early stopping, after every iteration metrics are computed on the validation set and, if there are no improvements after k iterations, training is stopped. The parameter k is called *patience_max* (30 by default) and early stopping is enabled after a warm up of 50 epochs.

4.1 Contrastive learning

Other than using only the just explained reconstruction loss, we introduce what we call **contrastive loss**. By optimizing this loss we are able to move close together in the latent space, vector representations of the same graph's node but, to be able to apply this concept, we need a way to learn different embeddings of the same nodes. This is done by partitioning the training graph into disjoint subgraphs via different methods, then using the same model on the different subgraphs to obtain different representations of a node. This happens because in different subgraphs a node can have a different set of neighbors, so for each forward step on a subgraph's adjacency matrix we obtain a limited representation of a node. It also allows the model to focus on different parts of the original graphs and so avoid overfitting.

The graph is partitioned in such a way that $A = \sum_{i=0}^{k} A_i$, where A_i are the subgraph's adjacency matrices and k is the number of partitions. It is also worth noting that self-loops are not present in A, those are added to A or A_i just before training, so that if a node doesn't have neighbors its representation depends only on its features.

The contrastive loss is defined as

$$\mathcal{L}_c = \frac{1}{J}\sum_{\{h^{(a)},h^{(b)}\}\in P} p(h^{(a)}, h^{(b)}) \tag{4}$$

where p is the loss between two subgraphs

$$p(h^{(a)}, h^{(b)}) = \frac{1}{2N}\sum_{i=0}^{N}[l(h_i^{(a)}, h_i^{(b)})+l(h_i^{(b)}, h_i^{(a)})] \tag{5}$$

$$l(h_i^{(a)}, h_i^{(b)}) = -\log \frac{\exp(sim(h_i^{(a)}, h_i^{(b)})/\tau)}{\sum_{k=0}^{N} \exp(sim(h_i^{(a)}, h_k^{(b)})/\tau) + \sum_{\substack{k=0 \\ k \neq i}}^{N} \exp(sim(h_i^{(a)}, h_k^{(a)})/\tau)} \tag{3}$$

and l is the loss between every pair of a node's representation, shown in equation 3. sim is a similarity function (we will use the **cosine similarity**), τ is the temperature coefficient, useful when we have lots of pairs to obtain a probability distribution with an higher entropy. This loss gives us the ability to make the different representations of a node closer together at the expense of other pairs.

It is also worth noting that h could be the embedding of each node or a projection of them as explained in Chen et al. (2020).

Graph partitioning Three different algorithms are implemented to partition the graph into k subgraphs, so that every partition does not have edges in common:

1. **Random edge splitting**: Edges are randomly splitted in the different subgraphs, in the same way done during the preprocessing phase to generate testing and validation data.

2. **Fiedler disjoint clustering**: The first eigenvector associated to the smallest eigenvalue greater than 0 of the graph's Laplacian matrix it's called Fiedler eigenvector. This gives us information about the connectivity of every node of the graph, so we can use this information to cluster nodes of the graph. Then every subgraph will contain only a cluster of nodes and only the edges inside of the cluster are keeped, those going from nodes of two different clusters will be removed.

3. **Fiedler clustering**: The same as above but edges between nodes in two different clusters are not removed, in this case there will be some common edges.

Note that the second method causes a loss of information, especially with higher values of k, as we lose some positve links. We also notice that in the first and third method we can have up to k different representations for each node, everyone of them can take into account a different neighborhood for every note depending on how the original graph is splitted. In the second case we will only have two representations of a node, one of the node with its neighbors and the other without them, no matter what is the value of k so, in this case, the contrastive loss 4 will be a little bit different to make the computation more efficient, as we don't need to compute it on every subgraph's combination. In fact, if a node is present in a subgraph, it will be connected in the same way as the original graph, if not present, will only be connected to itself.

4.2 Loss function

Our final loss function will be defined as

$$\mathcal{L} = \mathcal{L}_r + \alpha \mathcal{L}_c \tag{6}$$

where $\mathcal{L}_r$ is the reconstruction loss defined in equation 1 and $\mathcal{L}_c$ is the contrastive loss described in equation 4. α is a scaling factor for the contrastive loss, if $\alpha = 0$ contrastive loss is disabled while, if enabled, the default value is $\alpha = 0.5$.

If we splitted the original graph into subgraphs, $\mathcal{L}_r$ is the mean of the reconstruction losses over all the subgraphs.

4.3 Loss complexity

It is useful to analyze the complexity of the model training while using the contrastive loss. In this case we make k indipendent forward steps (using the same model and without updating the weights) to compute the losses, of which the contrastive one requires J iterations, with $J = \binom{k}{2}$ and k the number of subgraphs. If we use **Fiedler disjoint clustering** the contrastive loss can be computed in only one step. Even though this is computationally efficient, we won't use this technique because of the loss of information we have by removing edges between different clusters, especially with $k > 2$. Additionally, having more than two node's representations can be useful to learn more meaningful embeddings.

5 Models

During the testing phase, four different models have been used, every of them with a different *encoder*. The *decoder* is defined as $D(H) = \sigma(H \cdot H^T)$, where H is the embedding matrix of size $N \times D$ with D the dimension of the embeddings, σ is the sigmoid function.

Every model also has a *projection layer* that applies a non-linear transformation to node embeddings, defined as a 2-layer MLP with an input and output size equal to the dimension of the embeddings and an hidden size equal to the double of that. A **ReLU** function is applied between the two layers.

5.1 Convolutional Autoencoder (GCAE)

The first model has an encoder composed of two graph convolutional layers (Kipf and Welling, 2016), with an output size of 128 and 64, **ReLU** as the activation function. The convolution operation is defined as

$$GCN(A, X) = \sigma(\hat{A}XW) \tag{7}$$

σ is the activation function, $\hat{A}$ is the normalized adjacency matrix, W is the weight matrix.

5.2 Normalized Convolutional Autoencoder (GNCAE)

The same as the convolutional autoencoder, but we insert a normalization layer like

$$GNCN(A, X) = \sigma(s\hat{A}g(XW)) \tag{8}$$

with s scaling factor and g the normalization function.

5.3 Attention Autoencoder (GATAE)

In this case we have two **Graph Attention Layers** (Veličković et al., 2018) of size 128 and 64, the first has 4 attention heads whose output is concatenated and passed through the *ELU* activation function, the second layer has only 1 attention head.

5.4 Graph Transformer (GTAE)

We implement a graph transformer like described in Dwivedi and Bresson (2021). The encoder is a stack of transformer encoders, in our case we only have a one-layer stack as increasing this number tends to make the model too complex and causes overfitting. The hiddens size of the encoder is 64 with 4 attention heads and the output is projected in a 32-dimensional space. The input features are summed to a **positional embedding** specific for every node to add spatial informations, this is obtained by computing the **Laplacian positional encodings**.

6 Experimental settings

Every model is trained for a maximum of 300 epochs (500 for the Graph Transformer), with a patience of 30 epochs for the early stopping which is enabled after a warm up of 50 epochs. As for the optimizer, we used **Adam** with a learning rate of 10^{-2} (10^{-3} for training the Graph Transformer) and a weight decay of 10^{-5}.

For the contrastive training we used $\alpha = 0.5$ in equation 6, $\tau = 0.5$ in equation 3 and we split the original graph into two partitions, since the contrastive learning complexity grows quickly as we increase the number of subgraphs. We have choosen that value of τ because we have to maximize the probability of one pair over the other $2(N-1)$, so for a big value of nodes we want to rapidly unbalance the probability distribution towards the true pair (high entropy).

We will evaluate every model on four datasets: **Cora**, **Citeseer**, **Pubmed**, **Facebook**. The first three are citation graphs while the last one is a social network, all of them are undirected graphs and are choosen to be of various size. Other informations about them can be retrieved in Table 1. All the experiments where executed ten times, using a **10-fold cross validation**.

	Cora	Citeseer	Pubmed	Facebook
Nodes	2708	3327	19717	547
Edges	10556	9228	88651	9626
Features	1433	3703	500	262

Table 1: Informations about the datasets

6.1 Evaluation

Model's performances are evaluated using two types of metrics: accuracy and ranking metrics. For the first we evaluate the **AUC**, the area under the TPR-FPR curve, in order to evaluate the ability to retrieve latent links without adding too much false positives, then we evaluate the **AP** (Average Precision), that is the area under the Precision-Recall curve. We also compute ranking based metrics, in particular the **MRR** (Mean Reciprocal Rank), **MR** (Mean Rank), **Hits@k** (with $k = 1, 3, 10$). These are computed by "corrupting" every edge (u, v) in the test set by editing one between u or v with a node u' (or v') such that (u', v) (or (u, v')) is not a positive link. The ranking metrics are then computed on a list containing the positive edge along with 100 negative ones.

Cora							
	AUC	AP	MRR	MR	H@1	H@3	H@10
GCN	0.916±0.01	0.918±0.01	0.411±0.03	9.904±0.89	0.266±0.03	0.455±0.04	0.736±0.03
GAT	0.920±0.01	**0.928±0.01**	**0.482±0.02**	9.338±0.70	**0.332±0.02**	**0.561±0.02**	**0.783±0.02**
GNCN	**0.923±0.01**	0.927±0.01	0.465±0.02	**9.042±0.80**	0.311±0.02	0.541±0.02	0.778±0.02
GT	0.901±0.01	0.910±0.01	0.423±0.03	11.374±1.28	0.278±0.03	0.487±0.04	0.722±0.04
GCNC	**0.929±0.01**	**0.938±0.01**	**0.497±0.03**	**8.617±0.85**	**0.341±0.03**	**0.588±0.04**	**0.797±0.03**
GATC	0.925±0.01	0.929±0.01	0.459±0.02	8.726±0.40	0.296±0.02	0.549±0.03	0.784±0.02
GNCNC	0.907±0.01	0.914±0.01	0.457±0.02	10.455±0.42	0.300±0.02	0.535±0.02	0.769±0.01

Citeseer							
	AUC	AP	MRR	MR	H@1	H@3	H@10
GCN	0.883±0.02	0.891±0.02	0.409±0.06	11.747±1.17	0.267±0.06	0.470±0.06	0.701±0.04
GAT	0.895±0.01	0.908±0.01	0.516±0.02	10.674±0.53	0.388±0.02	0.586±0.03	0.748±0.01
GNCN	**0.941±0.01**	**0.950±0.01**	**0.588±0.03**	**7.165±0.80**	**0.455±0.04**	**0.668±0.04**	**0.846±0.02**
GT	0.889±0.01	0.899±0.01	0.420±0.03	11.178±0.70	0.276±0.03	0.483±0.04	0.711±0.02
GCNC	0.924±0.00	0.931±0.01	0.522±0.02	8.256±0.58	0.374±0.03	0.608±0.02	0.802±0.01
GATC	0.922±0.01	0.931±0.01	0.554±0.02	8.074±0.38	0.416±0.04	0.636±0.03	0.811±0.02
GNCNC	**0.934±0.00**	**0.948±0.00**	**0.598±0.02**	**7.882±0.48**	**0.462±0.02**	**0.690±0.02**	**0.835±0.02**

Pubmed							
	AUC	AP	MRR	MR	H@1	H@3	H@10
GCN	**0.930±0.01**	**0.935±0.01**	**0.456±0.03**	**8.833±0.37**	**0.300±0.03**	**0.528±0.03**	**0.779±0.02**
GAT	0.910±0.01	0.911±0.01	0.384±0.03	10.195±1.15	0.219±0.04	0.453±0.03	0.745±0.02
GNCN	0.922±0.01	0.922±0.01	0.390±0.01	9.543±0.56	0.222±0.02	0.459±0.02	0.769±0.01
GT	0.917±0.01	0.913±0.01	0.376±0.04	9.969±0.69	0.215±0.03	0.441±0.06	0.738±0.03
GCNC	**0.947±0.01**	**0.944±0.01**	**0.484±0.03**	**7.294±0.39**	**0.325±0.02**	**0.558±0.03**	**0.823±0.02**
GATC	0.938±0.01	0.934±0.01	0.418±0.02	7.738±0.30	0.231±0.03	0.524±0.03	0.818±0.02
GNCNC	0.932±0.01	0.933±0.01	0.410±0.01	7.943±0.56	0.232±0.02	0.519±0.02	0.799±0.01

Facebook							
	AUC	AP	MRR	MR	H@1	H@3	H@10
GCN	**0.964±0.01**	**0.963±0.01**	**0.322±0.02**	**8.100±0.37**	**0.141±0.02**	**0.372±0.03**	**0.779±0.02**
GAT	0.959±0.01	0.955±0.01	0.297±0.02	8.791±0.71	0.129±0.02	0.329±0.03	0.735±0.03
GNCN	0.955±0.01	0.947±0.01	0.275±0.01	8.842±0.46	0.095±0.01	0.314±0.02	0.733±0.02
GT	0.961±0.00	0.960±0.00	0.304±0.01	8.413±0.46	0.117±0.01	0.358±0.01	0.775±0.02
GCNC	**0.962±0.01**	**0.959±0.01**	**0.312±0.01**	**8.400±0.30**	**0.123±0.01**	**0.373±0.01**	**0.770±0.01**
GATC	0.961±0.00	0.957±0.01	0.296±0.01	8.541±0.39	0.120±0.01	0.333±0.02	0.748±0.01
GNCNC	0.959±0.01	0.945±0.01	0.266±0.02	9.826±0.57	0.092±0.01	0.301±0.04	0.716±0.02

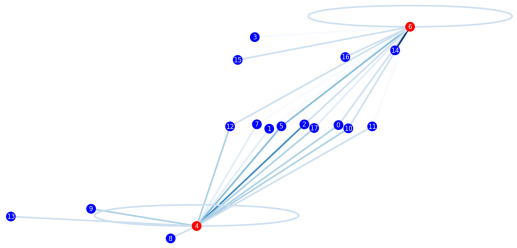
Table 2: Final results on every dataset, models ending in "C" are those trained using contrastive learning.

7 Results

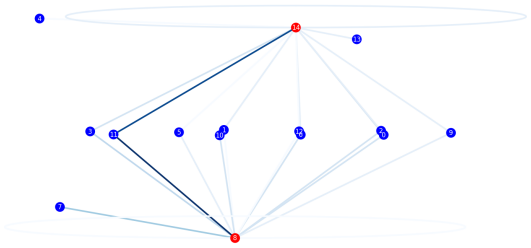
Results are shown in Table 2. As for the models, we can see that **GCNC** works significantly better then the others on *Citeseer*, probably because this is a very sparse dataset and has many isolated nodes, which leads us to believe that this model works better in this type of environment. Generally speaking, models based on attention do not seem to work much better than convolutional ones, with the exception of *Cora* where **GAT** works a little bit better than the others, this problem regarding attention-based models will be addressed in the next section. Regarding contrastive learning, we can see that it increases model's performances:
On *Cora* all models (with the exception of GCNC) work better.
On *Citeseer* the contrastive learning significantly increases performances of models that originally didn't work well, while GNCN performs worse. This is probably because splitting the graph gives us the ability to address the sparsity problem of this dataset.
On *Pubmed*, like in the other cases, contrastive learning increases the performances of every model.
On *Facebook* however, contrastive learning does not gives us any performace boost, probably because this is already a small graph so partitioning is not useful.

7.1 Attention models advantages and limitations

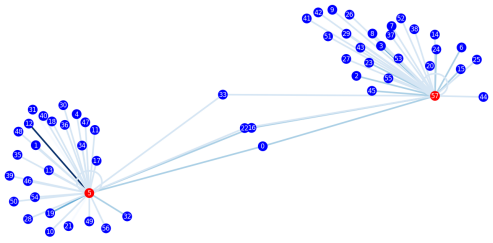
Attention-based models have the main advantage that attention coefficients are autonomously learned and can be analyzed to get a trivial explaination of the predictions. By analyzing these coefficients we



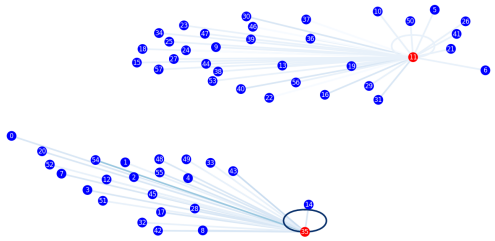
(a) Score = 1.0; Node 5 may be relevant in the prediction



(b) Score = 1.0; Node 11 may be relevant in the prediction



(c) Score = 0.7377; Few common neighbors, large neighborhood



(d) Score = 0.3036; No common neighbors

Figure 1: Nodes in red have a positive link in the test set, attention across their neighborhood is shown through link color. Upper graphs show high score predictions while the lower ones show low score predictions

are able to see that, when a link is successfully predicted, the two involved nodes have some mutual neighbors and the attention of both is focused on some of them. On the contrary, when a positive link is missed, the two nodes do not have common neighbors or all the coefficients are fairly balanced. These things can be visually seen in Figure 1. Since the sum of node attention coefficients is equal to one, we can interpret them as a probability distribution, whom in that case has a **low entropy** (balanced distribution). In this specific case the attention operation becomes a mere average of the neighbor's features and probably works worse than the classical graph convolution, especially when the node has a large neighborhood.

8 Conclusions

We have seen how to train different types of graph autoencoders for the unsupervised task of graph link prediction by exploiting the reconstruction loss, how much the predicted graph is similar to the input graph, and the contrastive loss, applied when partitioning the original graph. Since it is very important to avoid overfitting in this type of context, we have seen how to do it and that the dividing the graph into subgraphs may help avoiding this issue, probably because the model learns an embedding for every node taking into account different neighborhoods for each forward step. Lastly, we noticed that attention-based model do not have a large performace gap over the convolution-based ones, even though their attention mechanism could be useful to trivially comprehend their predictions.

A Paired T-Tests

Cora		
GNCN - GAT	AUC	1.460
GCNC - GCN	MRR	6.327
GATC - GAT	AUC	2.433
Citeseer		
GCNC - GCN	AUC	19.953
GATC - GAT	AUC	13.140
Pubmed		
GCNC - GCN	AUC	8.273
GATC - GAT	AUC	13.626
GNCNC - GNCN	AUC	4.866
Facebook		
GATC - GAT	AUC	0.973
GNCNC - GNCN	AUC	1.946

Table 3: t-values for some of the most significant experiments

References

Seong Jin Ahn and MyoungHo Kim. 2021. Variational graph normalized autoencoders. In *Proceedings of*

the 30th ACM International Conference on Information: Knowledge Management.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations.

Vijay Prakash Dwivedi and Xavier Bresson. 2021. A generalization of transformer networks to graphs.

Thomas N. Kipf and Max Welling. 2016. Variational graph auto-encoders.

Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks.

Talip Ucar. 2023. Ness: Node embeddings from static subgraphs.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph attention networks.