ETH zürich

BIOMEDICAL INFORMATICS

Project 2: GNNs on knowledge and 3D graphs

Abstract

The aim of this project is to get more experience with GNNs by using them in more complex applied scenarios. The first part covers prediction tasks on knowledge graphs. The second part focuses on classification of 3D objects represented in different ways.

Administrative

- Work in teams of 3.
- Start: 15.11.2023.
- Hand-in: 06.12.2023 (23:59:59).

Learning Objectives

- Practice using GNNs in various applications.
- Explore relational and 3D graph data.
- Compare the utility of different data representations.
- Investigate the effectiveness of various data augmentation techniques.

Deliverables

- PDF Report ≤ 4 pages (not counting references).
  - Make sure to mention all team members, indicate the student numbers and contact details.
  - We ask you to use the NeurIPS Conference Template found here. In LaTeX, you can load the style file with `\usepackage[preprint]{neurips_2023}` by passing the `preprint` option (for example, to not show line numbers).
- Jupyter Notebooks.
  - Three notebooks with code and comments for the solutions to tasks in sections 1.1, 1.2, and 1.3.
  - Three notebooks with code and comments for the solutions to tasks in sections 2.1, 2.2, and x2.3.
  - You can of course use additional Python files for helper functions and classes; however, the results should all be computed and shown in the notebooks.
  - Notebooks and Code should run on Google Colab.

1 Knowledge graphs

In this task, we explore the application of GNNs on knowledge graphs. Unlike the graphs explored in Project 1, knowledge graphs are directional, which means the model architecture has to be adapted. For example, one could use the relational graph convolution layers provided in PyTorch Geometric as RGCNConv.

We consider two tasks: entity classification and link prediction. Entity classification is essentially node classification. Link prediction amounts to predicting for given nodes u and v whether an edge (u, v) exists in the graph and if so, what type of relation it represents.

Entity classification

We start with the *AIFB* dataset available in PyTorch Geometric as Entities('AIFB') presented in Schlichtkrull et al. [2018]. Later in the task, we will also consider a similar but bigger *AM* dataset – Entities('AM').

In this section, we use simple classification accuracy as the performance metric. To aid the comparison of different models, put their performance on validation and test sets into a table. Unless stated otherwise, accuracy should be averaged over 3 runs, with the standard deviation also provided.

1.1 AFIB dataset

1.1.1 Data exploration (0.5 points)

Explore the data:

- (if applicable) check for missing data,
- collect graph statistics (number of nodes, edges, classes, features, graph diameter, etc.),
- visualize the graph or its part.

1.1.2 Random baseline (0.5 points)

Build a model that given a node predicts its class uniformly at random among all possible classes. Having such a baseline is often a good practice for testing the training and evaluation pipeline.

1.1.3 Feature baseline (1 point)

Similar to Project 1, build a model that predicts the class of a node based only on its features (any model is fine: linear regression, boosted trees, SVM, MLP, etc.). This model helps to sanity-check the models that take the graph structure into account.

1.1.4 GNN (4 points)

Using the R-GCN blocks, build a simple GNN to learn both from the node's feature and the graph structure. At this stage, it only needs to outperform the baseline models.

1.1.5 Oversmoothing (7 points)

Now, increase the depth of your model to increase the size of the neighborhood that's aggregated into a node's hidden state. How does the performance change? For at least 5 different depths, plot the number of model parameters (GNN) layers against their performance.

Here, to save time, measure accuracy over just one run.

Do you observe the same effect when increasing the model width (i.e., the number of neurons in one layer)? Plot it on the same figure and compare.

Take one of your deeper models and show that your network experiences oversmoothing (of course here, the network should not be explicitly adapted to avoid it). Compute an appropriate metric to quantify oversmoothing (revisit the lecture and tutorial on oversmoothing). Can you reduce oversmoothing and get the network to learn better (e.g., by using *PairNorm* [Zhao and Akoglu, 2020] or other discussed techniques)? Note, that the goal here is not to get a network with great performance. You should just demonstrate the *reduction* in oversmoothing independent of whether this improved model learning and generalization to the test set.

Deliverables for task 1.1:

- Jupyter Notebook with the implementation.
- In the report: brief comments for each subtask (description of the model, explanations of observed effects, etc.).
- In the report: a table comparing the performance of the models (similar to Project 1).
- In the report: a figure comparing the number of parameters and the performance of the models.

1.2 AM dataset

1.2.1 Data exploration and baselines (2 points)

Perform the usual due diligence: explore the data, evaluate a random baseline, and train and evaluate a feature baseline.

1.2.2 Is bigger = better? (5 points)

Take two architectures from section 1.1.5 with the biggest performance difference (e.g., the smallest and the biggest with respect to the number of layers). Train these models on the *AM* dataset. Is the difference in performance the same? Why or why not?

1.2.3 Better GNN (5 points)

Design a GNN that performs better than the previous two models. In this task, do not change the data – instead, adjust the model, the training procedure, etc. You should achieve at least 80% accuracy.

Compare the number of parameters and the training time of the three models. Does the new model perform better because of the design choices or the training length? Explain.

Deliverables for task 1.2:

- Jupyter Notebook with the implementation.
- In the report: brief comments for each subtask (description of the model, explanations of observed effects, etc.).
- In the report: a table comparing the performance of the models (similar to Project 1).

Link prediction

Now, consider the link prediction task. We use the *FB15k-237* dataset proposed in Schlichtkrull et al. [2018], which is available in PyTorch Geometric as FB15k_237.

The link prediction performance can be evaluated using *mean reciprocal rank* (MRR) and *Hits at n* (H@n) metrics (see Bordes et al. [2013]).

1.3 FB15k-237

1.3.1 Data exploration and baselines (2 points)

Perform the usual due diligence: explore the data, evaluate the random baseline, and train and evaluate the feature baseline.

1.3.2 GNN (5 points)

Similar to the previous task, build a GNN. Perform a hyperparameter search. Compare the performance with baselines. You should achieve the MRR of 2.50 and H@10 of 0.40.

1.3.3 Augmentations (6 points)

Propose augmentations to the knowledge graph to improve the performance. Train the baseline and GNN models on the augmented data. How did the performance change? Is it strictly comparable with the performance of the original data?

1.4 Contrastive Learning (10 points)

In this task, the goal is to leverage contrastive learning to perform a link prediction using node representations alone without any task-specific training.

This goes as follows:

- Implement a neural network that can compute node representations of a graph $\mathcal{G}$. Meaning, given a node $v \in \mathcal{G}$ we train $f_\theta(v) \mapsto \mathbf{v} \in \mathbb{R}^d$ to compute node representations $\mathbf{v}$. Where f_θ can obviously be a GNN (e.g., R-GCN).
- The network should be trained using a contrastive learning objective (e.g., *Triplet Loss*[1] or *InfoNCE*[2]). You should come up with a way to build positive (and negative) node pairs (v_s, v_t), then obtain their representations $(\mathbf{v_s}, \mathbf{v_t})$ and feed them into the contrastive objective of your choice.
- We have now learned task-agnostic node representations $\mathbf{v}$. To perform a link prediction (as our downstream application) using our learned node representations we need to learn a scoring function $s(v_s, e_r, v_t)$ for a given triplet of source node v_s, target node v_t, and relationship e_r of type r. As proposed by Yang et al. [2015] we learn a scoring function $s(v_s, e_r, v_t) = \mathbf{v_s}^T W_r \mathbf{v_t}$ where $W_r \in \mathbb{R}^{d \times d}$ is a different trainable matrix for each relationship type r. You should train the scoring function with a binary crossentropy loss, predicting 1 for correct triplets and 0 for corrupted triplets (you need to sample them). Note, in this part we do not train the function f_θ anymore, our node representations should remain fixed. The only trainable components in this stage are the matrices W_r.
- Finally, perform the evaluation using your learned scoring function and pretrained node embeddings.
- **Important note:** do not train this pipeline end-to-end, i.e. do not directly optimize the parameters of your node embedding network f_θ with the scoring function. The goal is to learn task-agnostic representations and investigate their usefulness for downstream applications such as link prediction. It is inherent to the task, that the optimization is performed in two distinct steps and the node representations remain fixed during downstream training.

Report your performance on the final link prediction task and compare it to an end-to-end optimization performance from the previous steps. What's the influence of your positive pair sampling method in contrastive pretraining on the final performance?

Deliverables for task 1.3:

- Jupyter Notebook with the implementation.
- In the report: brief comments for each subtask (description of the model, explanations of observed effects, etc.).
- In the report: a table comparing the performance of the models (similar to Project 1).

[1] https://pytorch.org/docs/stable/generated/torch.nn.TripletMarginLoss.html
[2] https://github.com/RElbers/info-nce-pytorch

2 Graphs in 3D

In this task, we explore the use of graphs for 3D tasks such as object classification. Specifically, we consider the *ModelNet10* dataset from Wu et al. [2015] (available in PyTorch Geometric as modelnet). Here, as an exception we evaluate models using only 1 run for computational reasons.

3D data can be represented in many different ways: with voxels, point clouds, meshes, or even implicitly encoded using neural networks Park et al. [2019]. Choosing the best representation for a particular application can drastically improve the performance and simplify the implementation of machine learning algorithms.

In this section, we do not chase state-of-the-art performance, as it heavily relies on big computational resources. Instead, we focus on comparing the principles of the models used on different data representation types.

2.1 Voxel format

One way to represent a volumetric object is to create a "volumetric image".

First, consider a 2D object. It can be represented with an "image" – a 2D matrix, with 1 assigned to the pixels that the object occupies, and 0 to which it does not. Now, for the 3D case, one can follow a similar procedure – create a 3D matrix with 0 and 1 encoding whether the object occupies a certain voxel (i.e. volumetric pixel) or not.

2.1.1 Data exploration and random baseline (3 points)

Collect the statistics about the dataset, and check for missing data. Do you preprocess the data? Why and how?

Build and evaluate a random baseline.

2.1.2 Voxelization (2 points)

Voxelize the dataset. Visualize a sample from the original dataset and its voxelized counterpart.

2.1.3 CNN (8 points)

Train a 3D convolutional neural network (CNN) for object classification. How does the voxel size affect the performance of your model? Explain and report the accuracy for at least 3 different voxel sizes.

2.2 Point clouds

Another way to represent the volumetric object is a point cloud – a set of points represented with their coordinates.

2.2.1 Point cloud format (2 points)

Convert the dataset to the point cloud format (i.e. you can use the PyTorch Geometric sampling transform). Add a visualization of the point cloud sample to the voxelized sample.

2.2.2 Coordinate baseline (3 points)

Train a GNN on the point cloud as a graph without edges. Use the point coordinates (x, y, z) as node features.

2.2.3 PointNet (10 points)

One of the most well-known architectures for object classification from point clouds is PointNet Qi et al. [2017]. Train and evaluate a PointNet model (e.g., the PyTorch implementation is available at pointNet.pytorch). How does sampling size affect its performance? Explain and report the accuracy

for at least 3 different voxel sizes. Compare the properties and the performance of the CNN and the PointNet.

2.3 Meshes

Finally, a 3D shape can be represented as a mesh which is essentially a 3D graph.

2.3.1 GNN (6 points)

Develop a GNN that uses just the object mesh for the classification. Compare its performance to the CNN and the PointNet.

For voxels and point clouds, you can adjust the resolution of the object representation (with the voxel size and the number of points sampled respectively). Is it possible to control the resolution of the mesh? How?

2.3.2 GNN with coordinates (6 points)

Now, develop a GNN that uses the coordinates (x, y, z) as node features. Compare the performance of the new GNN with the previous models. What is the major difference between the PointNet and the GNN with coordinates? Propose a dataset augmentation that would drastically reduce the GNN (with the coordinate features) performance while not affecting PointNet's performance (no need to test it).

2.3.3 Rotation-invariant GNNs (10 points)

Modify the GNN with coordinate features to be able to handle the transformations. Compare its performance to the other models.

Deliverable

- Provide a Jupyter Notebook for each of the three subtasks.
- Add your explanations to the report, add a table, and present numbers that reflect improvements due to significant architectural changes you made. At the very least, your table should contain a row for each of the three mentioned subtasks, where you managed to go above the threshold.
- Visualization of voxel, point cloud, and mesh representation of an object from the dataset.

References

A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26, 2013.

J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

C. R. Qi, H. Su, K. Mo, and L. J. Guibas. PointNet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*, pages 593–607. Springer, 2018.

Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d ShapeNets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

B. Yang, W. tau Yih, X. He, J. Gao, and L. Deng. Embedding entities and relations for learning and inference in knowledge bases, 2015.

L. Zhao and L. Akoglu. PairNorm: Tackling oversmoothing in GNNs, 2020.