Project 1: Training GNNs

Abstract

The aim of this project is to get familiar with training and investigating GNNs. The first part focusses on the transductive learning setting. Here, we are given a single graph with partial information and the goal is to predict the missing pieces. The second part is an inductive task. In the inductive setting, we are given a collection of graphs and try to learn a generalisable predictor applicable to graphs not yet shown during training. Most classical machine learning applications focus on the inductive setting.

1 Administrative

- Work in teams of 3
- Start: 11.10.2023
- Hand-in: 01.11.2023 (23:59:59)

1.1 Learning Objectives

- Get familiar with the basics of PyTorch Geometric
- Practice visualizing and gathering statistics about graphs
- Understand backbones of a GNN
- Apply several models on a graph task, evaluate and compare their performance

1.2 Deliverables

- PDF Report $\leq$ 3 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 (should run in 1 pass)
  - A single notebook with code and comments for the solutions to Tasks in Section 2.
  - A single notebook with code and comments for the solutions to Tasks in Section 3.
  - A single notebook with code and comments for the solutions to Tasks in Section 4.
  - 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.

2 Transductive Learning

For this part, we will use the *Cora* dataset as introduced by Yang et al. [2016]. You can load the dataset from `torch_geometric.datasets` using `Planetoid(name='Cora')`. Cora is a citation graph connecting publications (the nodes) and their feature vectors (extracted from the publications). We suggest exploring the different dimensions of the data to aid in the completion of the subsequent tasks.

The task is to assign the publications to one of seven classes, when only observing the labels for a subset of nodes (use the `split='public'` argument when loading the dataset).

2.1 Data Exploration (4 Points)

Explore the data and the graph. To develop an architecture, you should be familiar with the dimensions, scale, and properties of your data to make more informed decisions about the size of the model you would like to fit.

- How many nodes and edges does the graph have?
- How many features does each node have?
- What is the longest shortest path in the graph (diameter)?
- What other graph statistics could be insightful when developing a model?

Deliverable

- Provide your implementation in the Jupyter Notebook.
- Add your explanations and the computed statistics to your report.

2.2 Label Propagation with full observations (5 Points)

For this part you are allowed to observe all labels on the full graph. For each node, perform a classification by extracting all its immediate neighbours and predicting the majority label found in the neighbourhood. Specifically, given a node i with label y_i and its direct neighbourhood $\mathcal{N}(i)$ (connected by exactly one edge), compute the nearest neighbour classifier on the graph:

$$\hat{y}_i = \textit{Majority}\big(\{y_j \mid j \in \mathcal{N}(i)\}\big) \tag{1}$$

If there is a tie, choose from the tied options uniformly at random (you can compute several times with different random seeds and provide a standard deviation).

Deliverable

- Provide your implementation in the Jupyter Notebook
- Add your explanations and the computed classification accuracy (with standard deviation over three runs) to the report. Accuracy should be provided in a table (it will be extended with accuracy from other models in the following tasks).

2.3 Baseline without Graph Structure (5 Points)

Each node (publication) in the *Cora* dataset has a feature vector. Let's find out how predictive those features are by classifying the publications without using the citation information provided in the graph. For this task, use thetrain/val/test split provided by loading the dataset using PyTorch Geometric with `Planetoid(name='Cora', split='public')`.

Pick a machine learning algorithm of your choice (e.g. MLP, Logistic Regression, Random Forests, Gradient Boosted Trees, or a Support Vector Machine) and train it to perform the classification task. Tune your hyperparameters on the validation set and report the accuracy on the test set.

Deliverable

- Provide your implementation in the Jupyter Notebook.
- Add your explanations to the reports, add the classification accuracy on the test set to the same table as started in Subtask 2.2, and provide standard deviations over three runs.

2.4 Untrained GNNs (8 Points)

Now let's use a Graph Neural Network to learn both from the locally available feature information and the structural information provided by the underlying graph.

Using Pytorch Geometric, build a small GNN, which could be trained to perform the node level classification. Your network should take node features and the edge index of the graph as input, then compute node embeddings by passing through a set of graph convolution layers (e.g., `GCNConv`), and finally make the prediction by applying a classification head (e.g., a single `torch.nn.Linear`) with a softmax. For now, **don't train the network!**

First, initialise your GNN and compute embeddings for all nodes in the graph. Visualise them using a dimensionality reduction method such as T-SNE or PCA, assigning colours based on the ground-truth labels. Next, generate and visualise some random vectors of the same size as the embedding vectors. Finally, visualise the real feature vectors provided in the datasets. Compare the visualisations.

Now, take the node embeddings produced by the randomly initialised GNN and train a machine learning algorithm of your choice to predict the target from these random projections. Next, train the same type of model on projections created from only the node features (e.g., use an untrained MLP to create embeddings simply from the node features, not taking into account edges between nodes). Finally, train the same type of model on truly random node embeddings (e.g., use the random vectors you visualised previously). Report the test set accuracy in the comparison table. Do you see a difference in classification quality between embeddings produced by the randomly initialised GNN, the randomly initialised MLP, and truly random embeddings?

Deliverable

- Provide your implementation in the Jupyter Notebook.
- Visualisation plot of random network projections, truly random embeddings, and raw node feature vectors, all coloured by class assignment.
- Add your explanations to the report, add the test set classification accuracy based on predictions from random projections of an untrained GNN to the same table as started in Subtask 2.2, provide standard deviations over three runs. Add comparisons with truly random predictions and random projections of node features without structural information.

2.5 Trained GNNs (5 Points)

Train the GNN you designed in the previous task in Sec. 2.4. Optimise hyperparameters on the validation set and report your test set performance in the comparison table. You should aim for at least 80% classification accuracy.

Deliverable

- Provide your implementation in the Jupyter Notebook.
- Add your explanations and the test set classification accuracy to the same table as started in Subtask 2.2, provide standard deviations over three runs of your final architecture.

2.6 Visualizing Graph Attention Networks (5 Points)

Let's try to understand a bit better how information flows through our GNN. Build a GNN based on your architecture from the previous Subtask 2.5. Replace the graph convolutions with *Graph Attention Network* operators [Veličković et al., 2018]. PyTorch Geometric provides a readily available implementation: `torch_geometric.nn.GATConv`.

Train and tweak the architecture to achieve at least 80% accuracy on the test set. Then perform a forward pass and extract the attention weights of your graph convolution. The Pytorch Geometric implementation of `GATConv` provides an argument in its `forward` method to retrieve these weights by passing: `return_attention_weights=True`).

Sample nodes from the test set. For a given test point, visualise the immediate neighbourhood (one-hop neighbouring nodes) and try to visualise the extracted attention weights, the ground-truth labels of the neighbouring nodes, and the sampled test points ground-truth label. What can you learn about network predictions? Do attention weights channel information in a reasonable fashion?

Deliverable

- Provide your implementation in the Jupyter Notebook
- Add you explanations to the report, add the test set classification accuracy of your *GAT* based network to the same table as started in Subtask 2.2, provide standard deviations over three runs of your final architecture.
- Visualisation of extracted attention weights and labels in the neighbourhood of node from the test set.

3 Inductive Learning

For this part, we will use the *TUDataset* dataset as introduced by Morris et al. [2020]. Specifically, we explore the `ENZYMES` subset. You can load the dataset from `torch_geometric.datasets` using `TUDataset(name='ENZYMES', cleaned=False)`. Each graph represents a molecule (an enzyme) and the goal is to classify them (6 classes) by their EC number, which is based on the chemical reactions they catalyze[1].

Familiarise yourself with the options on how to load the data using the PyTorch Geometrics Wrapper. Since no specific split was published for this dataset, we ask you to define a train/test split yourself. To provide the final results, you should train and evaluate your model on three different random perturbations of the dataset, where the test set contains *100* of the total 600 graphs. Report your results using standard classification accuracy. Is there a better metric?

3.1 Baseline performance (10 Points)

You should develop a pipeline to train a simple GNN that can reach an accuracy of 40% or better, essentially matching the performance reported alongside the dataset publication [Morris et al., 2020].

3.2 Push performance (5 Points)

Now dive deeper: try to develop features, tweak your GNN, explore the literature around the dataset and try to push the accuracy to around 60%.

3.3 State of the art (5 Bonus Points)

These points will not be counted towards the total sum of achievable points, thus if you obtain them, they are a bonus.

For the very eager ones, you can explore the literature around the dataset and take inspiration from state of the art GNN implementations tailored for inductive graph classification and aim to achieve above 70% classification accuracy.

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.

[1] EC: Enzyme Commission number

4 Custom Message Passing (10 Points)

PyTorch Geometric provides an abstraction to implement new message passing operators. Using a provided notebook (you can find it on Moodle) that contains a skeleton for a `MessagePassing` class, implement the GraphSAGE operator as presented by Hamilton et al. [2018][2].

For a given *central* node v with current embedding h_v^{l-1}, the message passing update rule to tranform $h_v^{l-1} \rightarrow h_v^l$ is

$$h_v^{(l)} = W_l \cdot h_v^{(l-1)} + W_r \cdot AGG(\{h_u^{(l-1)}, \forall u \in N(v)\}), \tag{2}$$

where W_1 and W_2 are learnable weight matrices and $u \in N(v)$ is a *neighboring* node. For simplicity, we use mean aggregation:

$$AGG(\{h_u^{(l-1)}, \forall u \in N(v)\}) = \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u^{(l-1)}. \tag{3}$$

Note that we add a *skip connection* to the GraphSage implementation using the term $W_l \cdot h_v^{(l-1)}$.

Before implementing this update rule, think about how different parts of the above formulas correspond to the functions in the template: 1) `forward`, 2) `message`, and 3) `aggregate`. When looking at the template, as a hint, note that the aggregation function is already given (i.e. mean aggregation). Now the question remains: what are the messages passed by each neighbour node, and when do we call the `propagate` function?

Implement a small training pipeline around your operator and run the said pipeline on the `TUDataset(name='ENZYMES', cleaned=False)` same as in the previous Task (Sec. 3). Compare the performance of your implementation with the `torch_geometric.nn.SAGEConv` layer.

Using your custom implementation of a message passing operator, investigate what impact the lack of permutation invariance would have on its performance. For example, instead of mean aggregation, you could use a sequence model such as *GRU* [Chung et al., 2014] in the `aggregate` function of your PyTorch Geometric `MessagePassing` class implementation. Make sure to define a node order (e.g., sorted by index) and always adhere to it before running the sequence model to fully break the permutation invariance and avoid introducing permutation augmentations.

Can you come up with your own twist to message passing?

Deliverable

- Extend the provoperator,yter Notebook with your message passing implementation and a training pipeline around it.
- Add your explanations to the report, add a table presenting the performance of your operator, and compare it to an existing implementation provided in PyTorch Geometric.

[2] https://arxiv.org/abs/1706.02216

References

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs, 2018.

C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. Tudataset: A collection of benchmark datasets for learning with graphs, 2020.

S. University. Cs224w: Machine learning with graphs, 2023. URL https://web.stanford.edu/class/cs224w/.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks, 2018.

Z. Yang, W. W. Cohen, and R. Salakhutdinov. Revisiting semi-supervised learning with graph embeddings, 2016.