Molecule Retrieval with Natural Language Queries

Sofiane Ezzehi[1] **and Bastien Le Chenadec**[1]

[1]École des Ponts ParisTech

Contribution Statement

1 Introduction

The goal of this challenge is to retrieve molecules from a database using natural language queries. Each sample in the dataset is constituted of a ChEBI description of a molecule, which is a text describing its structure and properties, and an undirected graph representing the molecule with embeddings for each node. The embeddings are pre-computed using the Mol2Vec algorithm [1]. Given a textual query, the goal is to retrieve the molecule that best matches the query. The evaluation metric is the label ranking average precision score (LRAP) which is equivalent to the mean reciprocal rank (MRR) in our case.

The challenging part of this task is to find a way to combine two very different modalities : texts and graphs. One way to achieve this is to use contrastive learning : one model encodes the text and the other encodes the graph. The two encoders are then trained to project similar samples close to each other in the embedding space. This approach has been shown to be effective in many tasks [2, 3].

2 Data

3 Method

In this section, we describe the different models we used to encode the text and the graph. The text encoder and graph encoders are two separate models that are trained jointly using contrastive learning, so that they share the same embedding space.

3.1 Graph Attention Networks

Graph Attention Networks (GAT) [4] have been shown to be effective in many tasks. Like other graph neural networks, GATs aggregate information from the neighbors of each node to compute its embedding. The main difference with other models is that GATs use an attention mechanism to weight the neighbors of each node. Specifically we used the improved version of GATs suggested in [5].

Let G be an undirected graph with N nodes denoted $[\![1, N]\!]$. Let d be the dimension of the node embeddings, and $h_1, \dots, h_N \in \mathbb{R}^d$ be the said embeddings. Let $W \in \mathbb{R}^{d' \times d}$ and $a \in \mathbb{R}^{2d'}$. The attention weights are :

$$e(h_i, h_j) = a^T \, \text{LeakyReLU}([Wh_i \| Wh_j]) \tag{1}$$

where $\|$ denotes the concatenation operator. The attention weights are normalized using the softmax operator :

$$\alpha_{ij} = \frac{\exp(e(h_i, h_j))}{\sum_{k \in \mathcal{N}_i} \exp(e(h_i, h_k))} \tag{2}$$

where $\mathcal{N}_i$ denotes the set of neighbors of node i in G. This mechanism clearly allows the batch processing of graphs with different sizes. The embedding of node i is then computed as :

$$h'_i = \text{LeakyReLU}\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W h_j\right) \tag{3}$$

In general we will use multi-head attention, with K heads, $a^{(1)}, \dots, a^{(K)} \in \mathbb{R}^{2d'/K}$ and $W^{(1)}, \dots, W^{(K)} \in \mathbb{R}^{d'/K \times d}$:

$$h'_i = \text{LeakyReLU}\left(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} W^{(k)} h_j\right) \tag{4}$$

Furthermore, we will stack multiple GAT layers to obtain a deeper model. We may also apply a multi-layer perceptron to the embeddings of the last layer to obtain a more expressive representation.

3.2 DiffPool

A limitation of Graph Neural Networks (GNNs) is the inherently flat structure of the embeddings. This means that the more macroscopic structure of the graph is not very well captured, since these structural aspects are only encoded by the very local message passing layers. GATs, as we have seen, are designed to capture a more nuanced version of these local structures since they use an attention mechanism to weight the neighbors of each node. However, the global structure of the graph is still not very well captured, even when exploring larger radius neighborhoods.

DiffPool [6] is a method that aims to address this issue. It is a differentiable graph pooling layer that can be used to learn a hierarchical representation of the graph. The main idea is to learn a set of cluster assignments as well as new embeddings for the nodes, and then to use these assignments and embeddings to coarsen the graph. The coarsened graph is then passed to another GNN, and the process is repeated until the graph is small enough. The embeddings of the nodes in the last layer are then used as the graph embedding.

Let's briefly describe the two main steps of the DiffPool algorithm. We will denote

1. **Cluster assignment and new embeddings (GAT layers)** : In the original paper [6], the cluster assignments

as well as the new embeddings are learned using GNNs. However, in our case, we will use GATs. This allows for a more expressive representation of the nodes.

More specifically, the new embeddings at layer (l) are denoted $Z^{(l)} \in \mathbb{R}^{n_l \times d}$, and are simply obtained as the output of a GAT,

$$Z^{(l)} = \mathrm{GAT}_{l,\,\mathrm{embed}}\left(A^{(l)}, X^{(l)}\right).$$

The cluster assignments at layer (l) are denoted $S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$, and are obtained by applying a softmax function to the output of another GAT (with independent parameters),

$$S^{(l)} = \mathrm{softmax}\left(\mathrm{GAT}_{l,\,\mathrm{pool}}\left(A^{(l)}, X^{(l)}\right)\right).$$

As we can see from the last equation, the cluster assignments are a probability distribution over the nodes of the graph. This means that the cluster assignments are soft, and that each node can belong to multiple clusters.

2. **Graph coarsening (Diffpool layer)** : The coarsened graph is obtained by simply applying the cluster assignments to the new computed embeddings. More specifically, the adjacency matrix of the coarsened graph $A^{(l+1)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}$ is obtained as

$$A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)}.$$

The new node embeddings $X^{(l+1)} \in \mathbb{R}^{n_{l+1} \times d}$ are obtained as

$$X^{(l+1)} = {S^{(l)}}^T Z^{(l)}.$$

At the end of the process, a multi-layer perceptron is typically applied to the embeddings of the last layer.

We can therefore see that a series of coarsening steps are applied to the graph, which results in a smaller graph that captures the global structure of the original graph. Figure 1 that we have taken from the original paper [6] with slight modifications illustrates the process.

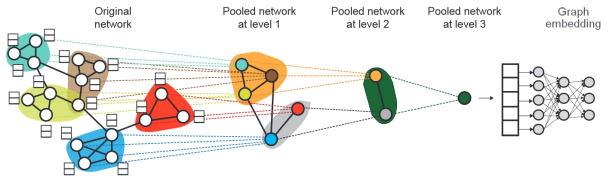

Figure 1: Illustration of the DiffPool method taken from [6]. The last feedforward neural network is used to obtain the graph embedding.

Ying et al. have also shown in [Proposition 1, [6]] that the permutation invariance of the DiffPool method is guaranteed, which is a crucial property for graph embedding methods.

3.3 *Language modelling*

We used a pretrained large language model (LLM) to encode the text. Specifically, we settled on `sentence-transformers/all-MiniLM-L6-v2` which is a distilled version of MiniLM [7] known for its efficiency and relatively small size. The sentence embeddings are obtained by averaging the embeddings of the tokens in the sentence, which yields a 384-dimensional vector.

4 TRAINING

4.1 *Model Evolution throughout the Challenge*

Throughout the challenge, we tried different models and training procedures. We started with a slightly modified version of the baseline method with a "distlibert" language Transformer model trained by distilling BERT base, and ended up, for the final submission, with an ensemble of XX models. We recap in the following table the big landmarks of our model evolution.

N°	Text Encoder	Graph Encoder	Main characteristics	Score
1	DistilBERT	GAT	Baseline	0.5
2	MiniLM	GAT	3 GAT layers 3 attention layers	0.74
3	MiniLM	DiffPool	2 layers	

Table 1: Model evolution throughout the challenge.

4.2 *Loss Functions*

We tried different loss functions to train the models. The two main loss functions we used are the Cross-Entropy loss and the Circle loss [**?**]. Let's briefly describe them

1. **Cross-Entropy loss** :
2. **Circle loss** :

4.3 *Training Procedure and implementation details*

We mainly used the PyTorch library to implement the models. Let's detail in the following the main implementation details of the models.

DiffPool implementation for batch processing: We can easily batch process graphs with different sizes using the DiffPool method. [TODO]

Optimizers and Learning Rate Schedulers: We mainly used the Adam optimizer with a starting learning rate of 10^{-4}. To automatically decrease the learning rate, we used a scheduler. More specifically, we fixed the learning rate to 10^{-4} for the first 10 epochs, and then decreased it by a factor of 0.05 at every epoch until the 100th. We then fixed the learning rate to the value it had at epoch 100 for the rest of the training.

Freezing and Unfreezing We used a two-step training procedure. We first trained the graph encoder for 5 epochs with the text encoder frozen. After that, we trained the whole model for 100 epochs. This allowed the graph encoder to learn a good representation of the graphs before the text encoder starts to learn and gave noticeable improvements in the results and convergence speed.

However, as will be further discussed in the results section, this freezing and unfreezing procedure has to be handled with care, as it can lead to very bad results if the freezing is too long while an ADAM optimizer is used. This is due to the fact that the moving averages of the gradients are updated with biased estimates of the gradients during the frozen phase.

Ensembling A very important part of our approach was to ensemble the models. This led to a significant improvement in the results as we will see in the results section. We used a simple averaging method to ensemble XX models. A tuning of the weights of the models in the ensemble has been done on the validation set.

We tried different ensembling methods, such as the Condorcet fusion method and the XX method, but they did not lead to better results than the simple averaging method. We will discuss these methods in Section 6.

4.4 Trained models' hyperparameters

We present here the hyperparameters of the models we used to train the graph and text encoders and that are all part of the ensemble.

[insert table with hyperparameters of the models]

5 RESULTS

5.1 Results per trained model

5.2 Global results after ensembling

5.3 Execution time

6 OTHER NON-CONCLUSIVE APPROACHES

6.1 PathNN

6.2 Condorcet Fusion

6.3 Alternate training procedure

REFERENCES

[1] Sabrina Jaeger, Simone Fulle, and Samo Turk. Mol2vec: Unsupervised machine learning approach with chemical intuition. *Journal of Chemical Information and Modeling*, 2018. URL https://pubs.acs.org/doi/10.1021/acs.jcim.7b00616.

[2] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey E. Hinton. A simple framework for contrastive learning of visual representations. *arXiv (Cornell University)*, 2 2020. URL http://export.arxiv.org/pdf/2002.05709.

[3] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SIMCSE: Simple Contrastive Learning of Sentence Embeddings. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 1 2021. doi: 10.18653/v1/2021.emnlp-main.552. URL https://doi.org/10.18653/v1/2021.emnlp-main.552.

[4] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Píetro Lió, and Yoshua Bengio. Graph attention networks. *arXiv (Cornell University)*, 2 2018. doi: 10.17863/cam.48429. URL https://arxiv.org/pdf/1710.10903.pdf.

[5] Shaked Brody, Uri Alon, and Eran Yahav. How Attentive are Graph Attention Networks? *arXiv (Cornell University)*, 5 2021. doi: 10.48550/arxiv.2105.14491. URL https://arxiv.org/abs/2105.14491.

[6] Zhitao Ying, Jiaxuan You, Christopher J. Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *neural information processing systems*, 31: 4805–4815, 12 2018. URL https://arxiv.org/pdf/1710.10903.pdf.

[7] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers. *arXiv (Cornell University)*, 2 2020. URL https://arxiv.org/pdf/2002.10957.pdf.