Molecule Retrieval with Natural Language Queries

Challenge ALTeGraD - January 2024

Halvard BARILLER
École Normale Supérieure Paris-Saclay
Paris, France

Victor DENG
École Normale Supérieure
Paris, France

Keywords: Contrastive Representation Learning

1 Introduction

Multimodality in machine learning involves the interaction of information from different types of data, requiring models that can understand and relate these varied forms of input. This work aims at bridging such a gap: more specifically, we propose an approach to retrieve specific molecular structures, represented as graphs, from natural language queries. The fundamental challenge lies in the difference in information representation between the two domains: the unstructured and contextual text data and the highly structured molecular graphs.

Our approach introduces a dual-encoder framework, employing contrastive learning to co-train a text encoder and a molecule encoder. The text encoder is designed to interpret the nuances and semantic richness of natural language, while the molecule encoder is tasked with understanding and processing the graph-based representation of molecules. Through contrastive learning, the model is trained to align the representations of each modality in a shared representation space where text queries can be effectively mapped to their relevant molecular structures.

2 Dataset

The dataset consists of pairs of molecules and molecule descriptions in natural language. The molecules are described as undirected, unweighted graphs with a substructure token associated to each node; each substructure token has an associated Mol2vec [Jaeger et al. 2018] embedding.

The dataset was split with 26,408 pairs for training, 3,301 pairs for validation and 3,301 pairs for testing.

3 Text Encoder

3.1 BERT-like Models

All the text encoders that we tried were pretrained BERT-like models fetched from Hugging Face (https://huggingface.co/). The baseline uses DistilBERT [Sanh et al. 2020], which is a smaller and faster version of the original BERT model [Devlin et al. 2019], which was trained on general text. We tried using SciBERT [Beltagy et al. 2019], which was specifically trained on scientific text, and finally RoBERTa [Liu et al. 2019], which is a significantly improved version of BERT.

3.2 SciBERT vs. RoBERTa

We settled on RoBERTa for our best performing models. However, we also considered using SciBERT instead, because SciBERT is specifically trained on scientific text. We tried to tokenize some input molecule descriptions, and the SciBERT tokenizer has less tendency to break down words into too small parts, contrary to RoBERTa or DistilBERT. Nevertheless, RoBERTa still outperformed SciBERT, possibly due to a better overall representation of input text thanks to significant improvements over vanilla BERT (on which SciBERT is based), or the fact that RoBERTa is a larger model (110M parameters for BERT, 125M for RoBERTa) trained on a larger corpus.

4 Molecule Encoder

We tested a number of graph encoders, which are presented below. All models use as node features the provided Mol2vec embeddings of the substructure tokens composing each molecule.

4.1 GCN

The baseline uses a Graph Convolutional Network (GCN) [Kipf and Welling 2017] as molecule encoder. The update of one convolutional layer is written as:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W} \tag{1}$$

where $\mathbf{X}$ and $\mathbf{X}'$ are the input and output feature matrices, $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix of the graph with inserted self-loops, $\hat{\mathbf{D}}$ is its diagonal degree matrix and $\mathbf{W}$ is a matrix of trainable parameters (the bias is omitted in the equation for clarity).

The baseline uses three GCN convolution layers with ReLU activation in-between layers, followed by a mean readout and a two-layer MLP. The dimension of the intermediate GCN layers and the hidden dimension of the final MLP are configurable, and they are both set to 300 by default. The output dimension matches the embedding dimension of the baseline text encoder DistilBERT, which is 768.

We extended the baseline by making the number of GCN convolution layers and MLP layers configurable. We experimented with a network containing 5 GCN convolution layers with a hidden dimension of 300 and a 3-layer MLP with a hidden dimension of 600. We call this version the "extended baseline" in the performance discussion in Section 6.
The baseline makes use of the `torch_geometric.nn.GCNConv` class of PyTorch Geometric.

4.2 GIN

Graph Isomorphism Networks (GINs) [Xu et al. 2019] are another type of graph neural networks. The update at layer k for a node v is written as:

$$h_v^{(k)} = \text{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)h_v^{k-1} + \sum_{u\in\mathcal{N}(v)} h_u^{(k-1)}\right) \tag{2}$$

where $h_v^{(k)}$ is the hidden state of node v at layer k, $\mathcal{N}(v)$ denotes the neighbors of v, $\text{MLP}^{(k)}$ is a MLP and $\epsilon^{(k)}$ is a scalar that can be fixed or trainable. We chose to fix all the $\epsilon^{(k)}$ to 0.

The readout is the concatenation of sum readouts of the input features and all the GIN convolution layers.

We make use of the `torch_geometric.nn.GINConv` class from PyTorch Geometric. We implemented the readout function by ourselves.

4.3 Graphormer

Graphormer [Ying et al. 2021] is a transformer architecture for graphs that takes into account the graph structure of the input. Indeed, the Graphormer architecture modifies the transformer architecture, in which all nodes attend to all other nodes, to integrate graph structure information in two ways:

- First, a centrality encoding is added to the input vectors x_i. This corresponds to adding a learnable vector that depends on the in-degree $\deg^-(v_i)$ of node v_i and another learnable vector that depends on the out-degree $\deg^+(v_i)$ of node x_i. The input vectors of the transformer architecture therefore become:

$$h_i^{(0)} = x_i + z^-_{\deg^-(v_i)} + z^+_{\deg^+(v_i)} \tag{3}$$

  In our case, the input graphs are undirected, so Eqn. 3 simplifies to:

$$h_i^{(0)} = x_i + z_{\deg(v_i)} \tag{4}$$

  where $\deg(v_i)$ is the degree of node v_i.
- Second, the attention weights are modified to take into account the graph structure of the input graph. Indeed, the attention weights A_{ij} are defined as:

$$A_{ij} = \frac{(h_iW_Q)(h_jW_K)^T}{\sqrt{d}} + b_{\phi(v_i,v_j)} \tag{5}$$

  where W_Q is the query projection matrix, W_K is the key projection matrix, d is the dimension of the features that the attention block receives as input, $\phi(v_i,v_j)$ is an integer-valued function that quantifies the connectivity of nodes v_i and v_j and the b_k are learnable scalars. Like in [Ying et al. 2021], we used the shortest path distance between v_i and v_j as $\phi(v_i,v_j)$.

In [Ying et al. 2021], an additional term c_{ij} that encodes edge features is added to the attention weights A_{ij}. As the graphs involved in our task do not have edge features, we omitted this term in our implementation.

Finally, a virtual node v connected to all the other nodes of the graph (but that does not affect the computation of shortest paths) is added to the graph. Message passing is performed with this virtual node just like all the other nodes, except that distance biases $b_{\phi(v,v_i)}$ and $b_{\phi(v_i,v)}$ that involve the virtual node v are replaced with a learned constant. Finally, the final graph readout simply consists in considering the hidden representation of the virtual node.

The model was implemented from scratch, except for the graph transform that adds the virtual node, which was adapted from the PyTorch Geometric transform `torch_geometric.transforms.virtual_node.VirtualNode` in order to function on batches of graphs.

4.4 GraphSAGE

GraphSAGE [Hamilton et al. 2018] is a GNN framework for inductive node embedding, based on a Sample & Aggregate strategy. First, the GraphSAGE algorithm randomly samples local neighbors from a node and aggregates the associated features. It then uses these obtained features to learn a set of aggregator functions, each learning to aggregate feature information from a node's local neighborhood in order to generate a node embedding. This approach has the specificity of not learning the embedding of each node directly, but rather functions allowing its computation: this allows the algorithm to extend to unseen data, which is the case in our task as each molecule is only seen once.

Formally, we define $\mathcal{N}^k(v)$ a uniformly sampled subset of size k from the set of neighbors $\mathcal{N}(v)$ for node v. The Message-Passing scheme is defined as follows:

$$m_v^{(t)} = \text{AGGREGATE}^{(t)}\left(\left\{h_u^{(t)} \mid u\in\mathcal{N}^k(v)\right\}\right) \tag{6}$$

$$h_v^{(t+1)} = \sigma\left(W^{(t)}\left[h_v^{(t)} \,||\, m_v^{(t)}\right]\right) \tag{7}$$

$$h_v^{(t+1)} = \frac{h_v^{(t+1)}}{||h_v^{(t+1)}||_2} \tag{8}$$

where $m_v^{(t)}$ and $h_v^{(t)}$ are respectively the incoming message and the representation of a node v at time t, and $W^{(t)}$ is the

corresponding weight matrix.
Different aggregator functions can be used to learn a node's local neighbourhood information. In our case, we used the basic Mean aggregator:

$$(6) + (7) \implies h_v^{(t+1)} = \sigma\left(W^{(t)} \frac{h_v^{(t)} + \sum_{u \in \mathcal{N}^k(v)} h_u^{(t)}}{\deg(v) + 1}\right) \quad (9)$$

We used the module class `torch_geometric.nn.SAGEConv` from PyTorch Geometric.

Following the architecture of the original paper, we used a depth of $K = 2$ with dropout with $p = 0.5$ and ReLU activation between the layers.

4.5 GAT

The Graph Attention Networks (GAT) [Veličković et al. 2018] are a type of GNN based on attention mechanisms. The rationale is that during the Message-Passing scheme, some neighbors might carry a more important message than others. GATs thus apply self-attention on the nodes in order to assign different weights to neighboring nodes when aggregating information. By doing so, GAT can capture intricate relationships and dependencies within the graph from a local perspective. Formally, we define the attention coefficients α_{ij} for the nodes $v_j \in \mathcal{N}(v_i)$ and the Message-Passing as follows:

$$\alpha_{ij}^{(t)} = \frac{\exp\left(\text{LeakyReLU}\left(a^T\left[W^{(t)}h_i^{(t)} || W^{(t)}h_j^{(t)}\right]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(a^T\left[W^{(t)}h_i^{(t)} || W^{(t)}h_k^{(t)}\right]\right)\right)} \quad (10)$$

$$h_i^{(t+1)} = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(t)} W^{(t)} h_j^{(t)}\right) \quad (11)$$

where $h_v^{(t)}$ is the representation of a node v at time t, $[\cdot||\cdot]$ denotes concatenation and a is a trainable vector.

We used the module class `torch_geometric.nn.GATv2Conv` from PyTorch Geometric.

Following the original architecture, we stacked 3 attention layers (with respectively 4, 4 and 6 attention heads), and applied dropout and ELU activations between each.

4.6 Architectures

Each of these models provides a node-level representation as output. We therefore aggregate these node features into graph-level features, making use of the `torch_ geometric.nn. global_mean_pool` class (except for Graphormer for which aggregation is done by extracting the embedding of the virtual node).

We then add a two-layer MLP with a ReLU activation after the readout to obtain a representation in dimension 768 that matches the dimension of the text embedding.

The final architectures used are as follows:

- GIN: `num_layers=6`, `num_node_features=300`, `interm_hidden_dim=600`, `hidden_dim=300`, `out_interm_dim=600`, `out_dim=768`
- Graphormer: `num_layers=6`, `num_node_features=300`, `hidden_dim=768`, `num_heads=32`
- GraphSAGE: `num_layers=2`, `num_node_features=300`, `nout = 768`, `nhid = 300`, `nhid_ff=600`
- GAT: `num_layers=3`, `num_node_features=300`, `nout = 768`, `nhid = 256`, `nhid_ff=600`

5 Method

5.1 Contrastive Representation Learning

The approach that we adopted for our task (which is also the general approach adopted in the baseline) is Contrastive Learning, where a text encoder (for the natural language descriptions of molecules) and a graph encoder (for the graph descriptions of molecules) with the same embedding space are trained to pull the embeddings of corresponding pairs of molecules and molecule descriptions together and push non-corresponding pairs apart: the model learns to align these embeddings coming from different modalities.

For the training phase, we first optimized the following contrastive loss taken from the baseline:

$$\mathcal{L}((t_i)_i, (m_j)_j) = \frac{1}{N}\sum_{i=1}^{N} \text{CE}((t_i^T m_j)_j, i) + \frac{1}{N}\sum_{j=1}^{N} \text{CE}((m_j^T t_i)_i, j) \quad (12)$$

where N is the number of training text-molecule pairs,

$$\text{CE}((l_1, ..., l_N), i) = -\log \frac{\exp(l_i)}{\sum_{j=1}^{N} \exp(l_j)} \quad (13)$$

is the cross entropy between class i and logits $(l_i)_{1 \leq i \leq N}$, and $(t_i)_{1 \leq i \leq N}$ and $(m_i)_{1 \leq i \leq N}$ are respectively the N embeddings of the textual descriptions and the N corresponding embeddings of molecules. Optimizing this loss will increase the dot products $t_i^T m_i$, $1 \leq i \leq N$ while decreasing the dot products $t_i^T m_j$ for $i \neq j$, hence aligning the pairs (t_i, m_i) while disaligning the pairs (t_i, m_j) for $i \neq j$.

However, decreasing the dot product $t_i^T m_j$ can also be done while keeping t_i and m_j aligned (for instance by decreasing the norm of either t_i or m_j), while one possible similarity metric for text embedding-molecule embedding pairs is the cosine similarity, that would therefore not always decrease if the dot product decreases. We therefore tried adding a cosine contrastive loss to the baseline contrastive loss, yielding the following augmented loss:

$$\mathcal{L}_{aug} = \mathcal{L} + \mathcal{L}_{cos}$$

$$\text{where } \mathcal{L}_{cos}((t_i)_i, (m_j)_j) = \mathcal{L}\left(\left(\frac{t_i}{||t_i||_2}\right)_i, \left(\frac{m_j}{||m_j||_2}\right)_j\right) \quad (14)$$

Finally, we introduced a temperature parameter τ following the methodology described in Radford et al. [2021].

Ranking molecules for a given textual description. Given a textual description of a molecule and a list of molecules, we rank the molecules by measuring the similarity between the text embedding and each molecule embedding. More specifically, if $\text{sim}(t, m)$ is a similarity metric, given a textual description with embedding t and molecules $1, ..., N$ with respective embeddings $m_1, ..., m_N$, we rank the molecules by decreasing $\text{sim}(t, m_i)$.

We tested several similarity metrics. The similarity metric used in the baseline is the cosine similarity:

$$\cos((t, m)) = \frac{\langle t, m \rangle}{||t||_2 ||m||_2}. \tag{15}$$

We also experimented with the following similarities:

- The dot product similarity: $\text{dot}(t, m) = \langle t, m \rangle$.
- The adjusted cosine similarity: given the mean text embedding $\bar{t}$ and the mean molecule embedding $\overline{m}$,

$$\text{adjcos}(t, m) = \cos((t - \bar{t}, m - \overline{m})) \tag{16}$$

- The negative Euclidean distance, or negative Minkowski distance with $p = 2$. This distance turned out to perform particularly poorly (see below for discussion on curse of dimensionality).
- An average similarity defined as the average of the five previous similarity scores.
- A normalized average similarity, which is the average of the normalized cosine, adjusted cosine and dot product similarities, where we define the normalized similarity score from a similarity score sim by: given t and m and molecule embeddings $m_1, ..., m_N$,

$$\overline{\text{sim}}(t, m) = \frac{\text{sim}(t, m)}{\max_{1 \le i \le N} \text{sim}(t, m_i)}. \tag{17}$$

Euclidean distance and curse of dimensionality. The Euclidean distance mentioned above performs particularly poorly because of the curse of dimensionality: in high dimension (which is our case, since our embedding space has dimension 768), distances between points often become similar and lose meaning. Figure 1 in Appendix A.1 shows the distribution of pairwise Minkowski distances between text and graph embeddings of the test set (using a model that uses a 6-layer GIN as its graph encoder and SciBERT as its test encoder, trained for 22 epochs, and having a LRAP of 0.56 to 0.58 on the validation set depending on the similarity metric or combination of metrics used): the distances concentrate around 23 or 24 (and there are millions of pairwise distances in this interval), to the point that numerical errors that differ between the `euclidean` and `minkowski` metric parameters of `sklearn.metrics.pairwise.pairwise_distances` alter the rankings significantly.

5.2 Optimization and hyperparameters

5.2.1 Training time.
The baseline trains the model for 5 epochs only. We trained all our models for at least 30 epochs, managing to substantially improve the performance of the baseline, with a Label-Ranking Average Precision (LRAP) of around 0.54 for the base model after 20 epochs compared to 0.3480 for the baseline score.

Initially, we implemented early stopping after 7 epochs without improving the validation loss (and later the LRAP score) in order to prevent overfitting. However, we realized that the validation loss or LRAP score would continue improving but very slowly (early stopping would stop the training too early), so we eventually abandoned the idea of automatic early stopping and instead manually stopped the trainings after they started to become too long or when we noticed no significant improvement over a long range of time. We still kept the model with the best validation loss then validation LRAP for each training.

5.2.2 Batch.
When training models, we saturated the batch size (i.e. we used the biggest batch size that fit in VRAM) in order to improve performance, as this leads to a quadratic increase in the number of negative text-molecule pairs seen for each batch. [Chen et al. 2020; Radford et al. 2021]

5.2.3 Optimizers.
We used the optimizer `AdamW` [Loshchilov and Hutter 2017] with $(\beta_1, \beta_2) = (0.9, 0.98)$ and a weight decay of $\lambda = 0.01$.
The baseline approach used a single optimizer for the whole model with a learning rate of $lr = 4 \cdot 10^{-5}$. We experimented with this method at first, but eventually uncoupled the optimizers between the encoders: the rationale behind it was that we were solely fine-tuning the text encoder, and hence requiring a lighter approach than for the graph encoder which was trained from scratch. We used the same configuration as previously described for the optimizer of the graph encoder. As for the text encoder, we reduced the learning rate to $lr = 1 \cdot 10^{-5}$, and excluded the Layer Normalization and Bias parameters from the weight decay [van Laarhoven 2017].

Moreover, we adjusted the learning rates throughout the training process using the `LinearLR` scheduler in PyTorch, starting at 100% of the original learning rate and ending at 30% or 20% of the initial learning rate, depending on the runs.

To improve optimization times, we used automatic mixed precision provided by PyTorch. This consists in automatically performing some operations in the `torch.float32` type and others in the lower precision `torch.float16` type. This has the advantage of speeding up some computations, since operations in `torch.float16` type are performed faster. To prevent gradient underflows or overflows (since some small non-zero or very large gradients that can be stored in `torch.float32` may not be storable in `torch.float16`), automatic scaling of the loss and gradients is applied, multiplying the gradients and loss by a big enough factor (to avoid underflows) or small enough factor (to avoid overflows).

5.3 Ensemble Methods

One of the key aspect in our approach has been the use of Ensemble Methods. As described in the previous sections, we trained a variety of encoders on this Contrastive Learning task. Each of these models used a different architecture and was carefully tuned so as to achieve the highest performance possible. In an attempt to leverage the diversity of information learned by each of them and reduce the variability, we considered two different aggregation methods.

5.3.1 Soft Ranking. We first implemented a soft ranking between the models. To do so, we computed the pairwise similarities obtained with each model based on the normalized average similarity (cf. Section 5.1). These measurements were thus identically scaled, rendering them comparable, and we summed them to obtain overall similarity coefficients.

5.3.2 Hard Ranking. We tested a second approach doing Hard Ranking. After obtaining the matrix of similarities for each model, we computed the elements' rank row-wise (smallest element ranked as 1, increasing order) and summed the rank matrices.

5.4 Practical details

Apart from PyTorch, Numpy, Scipy and Pandas, we used the Hugging Face Transformers library[1] for the pretrained word embeddings and PyTorch Geometric[2] (and NetworkX[3] at one point) to implement the graph encoders.

We also used Weights and Biases[4] to track our trainings, with different metrics plotted, such as the validation LRAP or the training loss.

Finally, we used Kaggle's free GPUs (NVIDIA T4 and P100) as well as private VMs with NVIDIA RTX 3090 GPUs for training.

6 Numerical Results and Performance

6.1 Numerical Results

Table 2 summarizes the performance of the different graph encoders when combined with the RoBERTa text encoder. Some models were stopped at a certain time when we considered that the learning was becoming negligible based on the monitoring of the loss and LRAP on the validation set.

Examples of training curves (training loss, validation loss and LRAP on validation set) for some models are shown in Appendix A.2.

6.2 Ensemble Results

6.3 Training Resource Usage

The average training time for each epoch is around 6 minutes for most models on a NVIDIA RTX 3090 GPU. Graphormer however takes much longer to train, at around 10 minutes per epoch on a RTX 3090 GPU, with additionally a much larger number of epochs required to achieve good performance (see Table 2). The Graphormer and the GAT were also more memory intensive, forcing us to reduce the batch size from 32 to 16.

Interestingly, our models trained slower on NVIDIA T4 and P100 GPUs (Kaggle) than on NVIDIA RTX 3090 or 3080 GPUs (private VMs).

7 Discussion

During our work, we considered a variety of directions for potential improvement. Due to budget constraints related to computing power, we have unfortunately not been able to explore them all as much as we would have liked. We discuss these ideas below.

7.1 Data Preprocessing

Our approach involved very little data preprocessing. Towards the end, we noticed a discrepancy in the losses between the training and the validation set when looking at our different trainings. Indeed, it seemed that despite the large quantity of training examples, the model was presenting some light signs of overfitting as the training loss continued to decrease while the validation loss stagnated. We thus considered proceeding with some data augmentation on the training set to make the learning process more robust.

However, as the data is highly technical, such a process would require a cautious set-up. Regarding the text augmentation, we would have liked to try in particular word replacement using Contextual Word Embedding. We also began experiments where we increased the maximum sequence length for tokenization in order to avoid discarding important portions of some inputs, but this idea came towards the end of the challenge and we did not have enough performance measurements to draw conclusions on the impact of this technique (this required training models from scratch).

The graph augmentation is even more delicate: while it can make sense to add or remove some edges or nodes in a general graph context, such manipulation would have a significant impact on our task as this could severely damage the molecular structure of our data. Solutions to this problem have been proposed [Magar et al. 2021] where data

[1]https://huggingface.co/docs/transformers/index
[2]https://pyg.org/
[3]https://networkx.org/
[4]https://wandb.ai

	Number of Epochs						
Models	1	30	60	90	140	190	Best Performance
GCN - Baseline	24.85	68.44	-	-	-	-	68.44
GCN - Extended baseline	12.75	66.15	72.52				72.52
GIN	**31.85**	74.29	79.52	81.51	84.23	-	84.23
Graphormer	01.12	55.48	65.91	76.25	78.05	82.57	82.57
GraphSAGE	24.35	**76.74**	**79.71**	**81.59**	**85.14**	-	**85.14**
GAT	24.96	71.69	76.54	79.08	79.55	80.31	80.31

Table 1. LRAP on Validation Set

Models	Soft Ranking	Hard Ranking
GIN + Graphormer + SAGE	**88.63**	88.35
GIN + Graphormer + SAGE + GAT	87.21	86.66

Table 2. LRAP on Validation Set for Ensemble Methods

augmentation is performed in a chemically-sound manner, thus preserving the integrity of the molecules.

7.2 Hyperparameter Tuning

As described in Section 5.2.3, we eventually considered an approach that uses two different optimizers, one for each encoder. However, we did not witness substantial improvement when uncoupling the learning rates despite our expectations. With more computing power, we think that performing rigorous tuning with hyperparameter sweeps would have allowed to better calibrate the learning rates and other optimizer parameters.

7.3 Embedding Dimension

The last tuning we would have liked to do is the tuning of the embedding space $\mathcal{E}$. In the baseline, the latter was such that $\mathcal{E} = \mathbb{R}^d$ where $d = 768$, as this is the standard dimension for Transformers like BERT or RoBERTa. However, we think that considering other values for d could actually have an impact on the performance in aligning the encoders' representation. Once again, this idea came towards the end and despite wanting to test other dimensions (512 or 1024 for example), this meant training models from scratch and was therefore too expensive in terms of both time and money.

8 Conclusion

During this data challenge, we were able to get hands-on experience with model training, including training monitoring with Weights and Biases and finding the right hyperparameters. We learned to use different computing resources available online, including using virtual machines from cloud providers. We were able to obtain rather good results by selecting well-performing text and graph encoders, using the right hyperparameters to train them, and using ensemble methods to aggregate the predictions of different well-performing models.

References

Iz Beltagy, Kyle Lo, and Arman Cohan. 2019. SciBERT: A Pretrained Language Model for Scientific Text. arXiv:1903.10676 [cs.CL]

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. arXiv:2002.05709

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

William L. Hamilton, Rex Ying, and Jure Leskovec. 2018. Inductive Representation Learning on Large Graphs. arXiv:1706.02216

Sabrina Jaeger, Simone Fulle, and Samo Turk. 2018. Mol2vec: Unsupervised Machine Learning Approach with Chemical Intuition. *Journal of Chemical Information and Modeling* 58, 1 (2018), 27–35. https://doi.org/10.1021/acs.jcim.7b00616 arXiv:https://doi.org/10.1021/acs.jcim.7b00616 PMID: 29268609.

Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907 [cs.LG]

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

Rishikesh Magar, Yuyang Wang, Cooper Lorsung, Chen Liang, Hariharan Ramasubramanian, Peiyuan Li, and Amir Barati Farimani. 2021. AugLiChem: Data Augmentation Library of Chemical Structures for Machine Learning. arXiv:2111.15112 [cs.LG]

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*.

PMLR, 8748–8763.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL]

Twan van Laarhoven. 2017. L2 Regularization versus Batch and Weight Normalization. *ArXiv* abs/1706.05350 (2017).

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks? arXiv:1810.00826 [cs.LG]

Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Bad for Graph Representation? arXiv:2106.05234 [cs.LG]

A Appendix

A.1 Curse of dimensionality

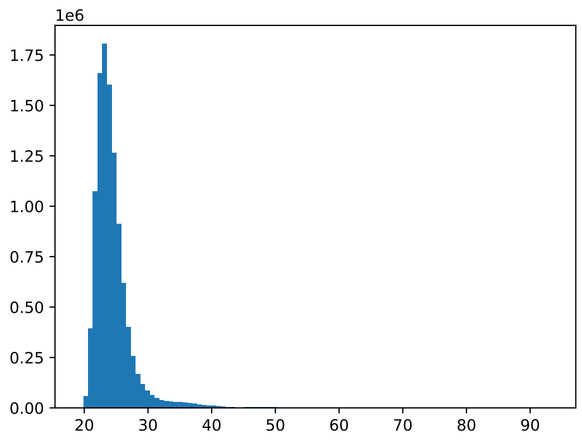
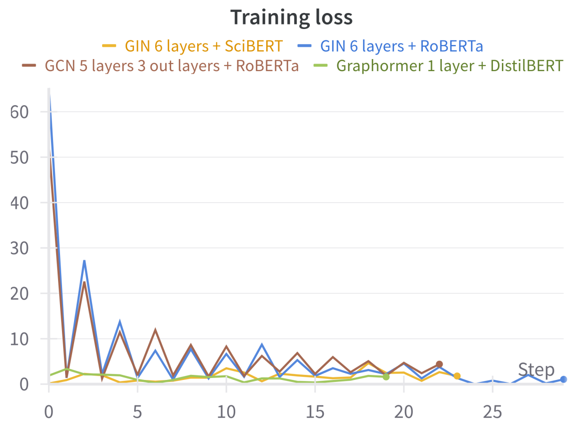
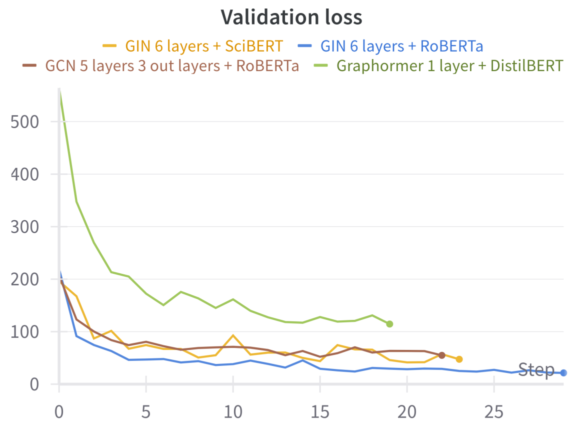
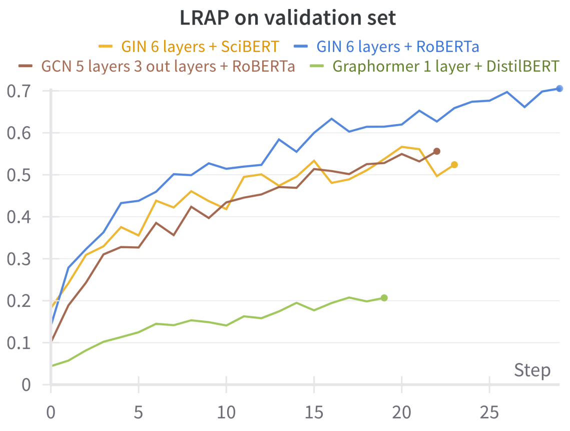


Figure 1. Distribution of pairwise Euclidean distances between text and graph embeddings of the test set (using GIN and SciBERT).

A.2 Examples of training curves


Training loss
GIN 6 layers + SciBERT
GIN 6 layers + RoBERTa
GCN 5 layers 3 out layers + RoBERTa
Graphormer 1 layer + DistilBERT
Step


Validation loss
GIN 6 layers + SciBERT
GIN 6 layers + RoBERTa
GCN 5 layers 3 out layers + RoBERTa
Graphormer 1 layer + DistilBERT
Step


LRAP on validation set
GIN 6 layers + SciBERT
GIN 6 layers + RoBERTa
GCN 5 layers 3 out layers + RoBERTa
Graphormer 1 layer + DistilBERT
Step