DEPARTMENT OF ELECTRICAL ENGINEERING & ELECTRONICS

Final report for project 'Graph Matching Networks for Similarity Learning'

Author: Xiaowei Shi (201522273)

Project Supervisor: Xinping Yi

Project Assessor: Valerio Selis

Abstract

This project present and implement a deep neural network to predict similarity between graph-structured data by leveraging a graph matching network (GMN). This network learns a similarity metric through an attention-based GMN trained by input graph pairs. This GMN adopts a message-passing graph neural network to iteratively generate more distinguished node embedding and leverage the attention mechanism to incorporate correlative information of another graph in the input pairs into the node embedding. During the model training stage, both synthetic data and real-world datasets were used to generate input graph pairs which are then modified and labelled based on a classic graph similarity index: graph edit distance (GED). Through experiments conducted with different aggregation types and different sizes of node feature and edge feature dimensions, this model achieves varying degrees of effectiveness in predicting the similarity. This project observed these results and also conjectured further verified the impact of choosing different node embedding dimensions and different aggregation operation types on the mean and upper and lower limits of performance.

1 Project Specification

The following included a copy of the original project specifications and verification table.

1.1 Project specifications

3 | Project Specification

3.1 Aims

There are three major aims: First, one uses GNN to generate graph embeddings for similarity learning, while the second makes use of the newly proposed application for cross-graph attention matching (GMN) in similarity graph matching calculations.
Using the above two models to conduct tests on GMN performance in multiple applications, the new graph similarity learning model (GMN) is then applied to malicious intrusion detection of software systems, and finally, run experiments to evaluate the results from GMN performance in multiple applications. As a standard, the GMN should outperform baseline models as well as model-agnostic models such as Siamese networks for all tasks.

1.2 Verification table

3.2 Detailed Solution

This project can be broken down into the following work packages shown in Table 1.:

Table 1: Work Packages.

Work-Packages	Work-Packages	Work-Packages
1 The graph embedding model (GEM)	• The graph encoder • The message passing layers • Graph aggregator • Putting together	Either of the two forms must make the node ordering unchanged.
2 The graph matching networks(GMN)	Similarity functions • Cross-graph attention • Graph matching layer and GMN	
3 Training	•Labelled data examples •Training on pairs •Training on triplets	Try at least 3 methods of loss function.
4 The graph attention networks(GAN)	•A few graph manipulation primitives •Dataset for training •Fixed dataset for evaluation	Given two graphs, calculating the edit distance between them is the number of actions required to convert one graph to another.
5 Apply on heterogeneous graphs and	• Configs • Evaluate • Build the model	Observing improvement in

evaluate	- set up placeholders - build the computation graphs - build the metrics and statistics • Train pipeline • Run	performance after training for 5,000 steps, the loss should go down and pair AUC and triplet accuracies should be going up indicating the training would work.
6 Test the model by creating some visualizations and adjusting parameters	• Split the batched graphs into individual graphs and visualize them. • Build the computation graph for visualization.	The attention pattern should be uniform at first and then gets more concentrated after several layers.

2 Introduction

2.1 Introduction of graph and graph matching problem

Graph-structured data exists in various application scenarios, due to being a natural and ubiquitous way of representation to describe complicated data structures. It falls into the category of non-Euclidean structure data type shown in Figure 1., whose number of neighbour nodes may be different and maybe arranged irregularly, unlike text and image. This type of data includes layout structures, knowledge maps, social networks, chemical molecular structures, and some others [1].

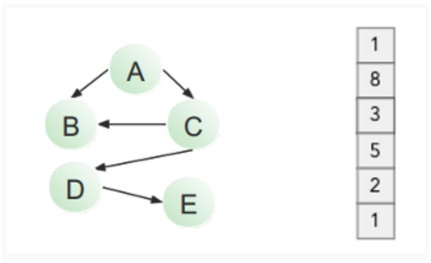


Figure 1. Grid-like data (right) compared with non-euclidean data (left).

A graph Matching problem is trying to optimally construct some certain correlation between two input graph-structured objects and minimize their node and edge disagreements, which is not only one of the key challenges in the various research area but also a preliminary and inevitable task of many complicated graph-based research tasks.

Graphs appear in application areas of computer vision [2, 3], bioinformatics [4], structural layout analysis [3], pattern recognition [5], cheminformatics [6], cyber security [7], source code/binary code analysis [8], social network analysis [9], POI retrievals [10] and some others shown in Figure 2. Some tasks can be solved only based on this task, for example, node classification task [11, 12, 13], graph classification task [14, 15, 2], graph generation task [16, 17, 4], and many others.

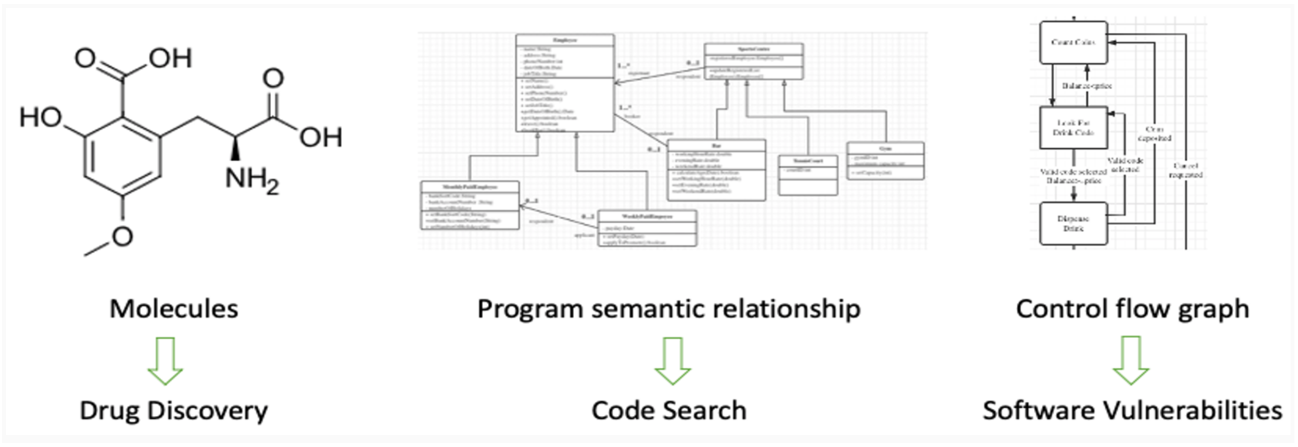


Figure 2. Real-world examples of graph-structured data and their applications.

2.2 Scope

Based on different goals in real application scenarios of graph matching, general graph matching problems can be classified into two different categories: termed as exact matching and inexact matching at first [18]. The former problem requires finding a rigorous corresponding relationship between input graphs or their subgraphs shown in Figure 3. As for the latter problem, this requirement is relaxed greatly into only optimally finding the bijections between vertices that optimize some affinity or distortion criterion. This is the reason why this kind of problem is also referred to as Error correcting graph matching in the former literature published on IEEE [19]. After all, these two categories of graph matching problems have the same input where they all have graph-structured data pairs as input but different output. The output of the first one mainly can be represented as a correspondence matrix [20], while the output of the second class is to compute a similarity score usually represented as a scalar value [21, 22] or represented as binary -1 (not similar) and 1 (similar) [8] shown in Figure 4. This project focus on the second category, where the input is a pair of graphs, and it is mainly expected to compute a binary similarity score as output, so this project can also be categorised as a graph-graph classification task.

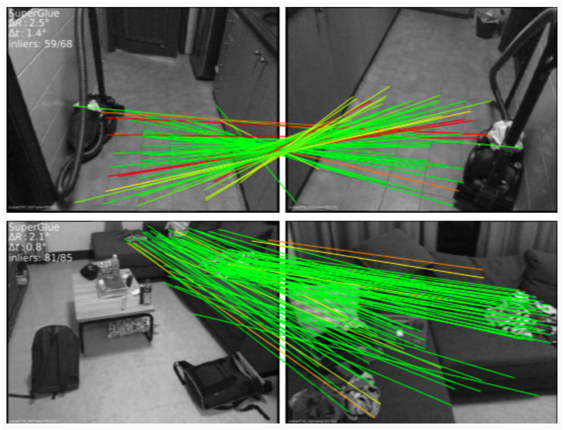


Figure 3. An example of the first category of graph matching problem: is to find a rigorous corresponding relationship between nodes of input pairs [23].

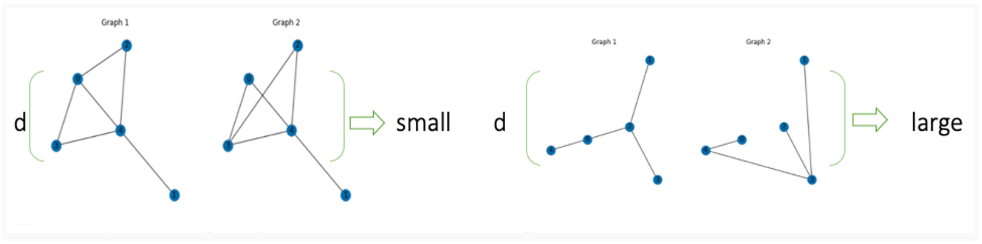


Figure 4. An example of the second category of graph matching problems: is to find a similarity score between input pairs.

3 Literature review

According to former works of literature, both categories of graph matching problems are non-deterministic polynomial-time hardness (NP-hard) [22, 24], which means that neither of these tasks is computationally feasible to provide a promising solution within an acceptable time in the real-world environments at industrial standards.

Considering the great importance and inherent difficulty of graph matching problems, this problem has been studied extensively. A number of approximate algorithms based on the theoretical study and empirical knowledge of experts have been proposed to find sub-optimal solutions in a reasonable amount of time [24, 25, 26, 22, 18]. Loiola [24] has proposed copious past research on both exact algorithms and heuristic algorithms. Foggia and Raveaux [26, 25] have presented an approximation algorithm based on bipartite graph matching; its main principle is to perform bipartite matching between each subgraph of two graphs to calculate an approximation of the graph edit distance. Riesen and Yan [22, 18] have summarized and stated two main categories of approximate methods: Bipartite and QAP-based.

However, it still suffers from poor scalability and heavy reliance on expert knowledge, so it remains a challenging and open research area to be resolved. Thus, a new Interdisciplinary area, which is Deep Graph Learning (DGL), began to rise [27, 28, 15]. Kipf and Welling [29] have attempted to adapt semi-supervised deep learning from images to non-Euclidean data in an end-to-end fashion, in which the input is the original data instead of extracted features, and the output is the final results without the need to interpret. Rong [30] has summarized the main achievements in DGL and introduced its scalability and robustness. Wu [31] has summarized and categorised advanced GNNs into recurrent GNNs, convolutional GNNs, graph autoencoders, and spatial-temporal GNNs.

There have been many GNN models proposed since then for learning efficient node embeddings for downstream tasks [2, 3, 4, 5]. For example, node classification task [11, 12, 13], graph classification task [14, 15, 2], graph generation task [16, 17, 4] and many others [8, 3].
GNN-based models were highly successful at learning graph representations for downstream tasks, showing that GNNs are a powerful class of deep learning models.

Researchers seeking to solve graph-related problems began to adopt GNN-based models and to improve the matching accuracy and efficiency after obtaining excellent results and great success with GNN-based models in the above-mentioned applications and many other graph-related tasks [13, 14, 15, 17, 18]. During the training phase, these models attempt to learn the mapping between input graph pairs and ground-truth correspondences in supervised learning and therefore are performed more time-efficient than traditional approximation methods during the reasoning phase.

4 Industrial relevance

Learning a similarity measure between any pair of graph-structured objects is one of the fundamental problems in a variety of applications, from similarity graph search in databases [32] to binary function analysis [8], unknown malware detection [8], Semantic Code Retrieval [33], Malicious Account Detection [34], Inquiry-POI Retrieval [10], etc.

4.1 GMN for Malicious Account Detection

Massive online services are popular targets for cyber attacks. By creating malicious accounts, attackers can spread spam and make huge profits. This basically has a negative impact on the ecosystem. In many cases of abuses, for example, robot accounts, are used for billions of junk mail sent via electronic mail systems [34]. What is more serious is that in the financial system like Alipay [34], when a large number of accounts are hijacked by malicious users or a group of malicious users, these malicious users can cash out to cheat profits, causing great harm to the whole financial system [34]. Effective and accurate detection of such malicious accounts plays an important role in such systems.

4.2 GMN for malware detection

Attackers create, use, and sell malware for many reasons, but they are most commonly used to steal individuals, finance, or business information. Although the motivation is different, the network attackers always concentrate their tactics, technology, and programs on the access rights of privileges and accounts to perform their tasks. End-to-end deep learning proves its effectiveness and provides a powerful tool for malware detection. Effectively prevent potential danger and loss of life and property [8].

4.3 GMN for Inquiry-POI Retrieval

The growing interest in travelling abroad, also increases the interest points in multiple languages to find (POIs) requirements. This is even better than when travelling abroad in an unfamiliar language better find local restaurants and attractions. Multilingual POI search allows the user to find required through the multilingual query language of POI, making it a map of today's global applications (such as Baidu Maps) of a component [10]. The cross-attention module fuses the representation of two types of nodes for the Query-POI correlation score [10]. This allows better handling of correlation rankings between multilingual queries and POIs of different prevalence [10]. In the real world, numerous experiments on large-scale real-world datasets have proved the superiority and effectiveness of GMN and its variants [10].

5 Theory

5.1 Problem definition

Learning to measure the similarity between any pair of graph structure objects is one of the basic problems in a variety of applications, from similarity graph search in databases [32] to binary function analysis [8], Unknown malware detection [8], semantic code retrieval [33], etc. According to different application backgrounds, similarity measures can be defined by different measures of structural similarities, such as graph editing distance (GED) [22], maximum common subgraph (MCS) [21], and even more roughly binary similarity (i.e., similarity or not) [33]. GED corresponds to the MCS problem under the fitness function [22]. In addition to GED calculation, learning binary label $S \in \{-1, 1\}$ (similar or dissimilar) between a pair of graphs can be regarded as a task of graph-graph classification learning and has been widely studied in many practical applications, including binary code analysis, source code analysis, malware detection, etc. Basically, the graph similarity problem aims to calculate the similarity score between a pair of graphs, which indicates how similar the pair of graphs are.

5.2 General framework of GNN

The basic idea of graph neural networks is to iteratively update node representation by combining neighbour representation with their own representation. You [35] introduced the general framework of graph neural networks. Starting with the initial node representation $H_0 = X_0$, in each layer, There are two main functions:

• AGGREGATE, which attempts to AGGREGATE information from the neighbours of each node;

• COMBINE, which attempts to update the node representation by combining aggregate information from neighbours with the current node representation.

Mathematically, it can be defined as the general framework of graph neural network as follows:

Initialization:

$$H_0 = X_0$$

For k = 1, 2, ⋯, K,:

$$a_k^v = \text{AGGREGATE}_k \{H_{k-1}^u : u \in N(v)\}$$

$$H_k^v = \text{COMBINE}_k \{H_{k-1}^v , a_k^v \}$$

Where N(v) is the neighbor set of the V^{th} node. Node representation H_k^v at the last layer can be regarded as the final node representation, which can be used for downstream tasks.

5.3 Cross-Graph Convolutional Network

Wang et al., in an article published in IEEE [36], claim that this is the first work using GNN for depth graph matching learning. By taking advantage of GNN's efficient learning ability, node embedding can be updated using structural similarity information between two graphs, and the graph matching problem, namely the quadratic allocation problem, is transformed into a linear allocation problem that can be easily solved [36].

The authors of this paper propose a graph matching model with substitution loss based on cross-graph affinity, namely PCA-GM [36]. PCA-GM consists of three steps:

- First, in order to enhance the learning node embedding of individual graphs using standard messaging networks, namely, graph convolutional networks [36]
- Second, in order to enhance the learning node embedding of individual graphs using standard messaging networks, namely, graph convolutional networks [36]
- It not only aggregates information from local neighbours but also combines information from similar nodes in another graph [36]. The formula is as follows:

$$H^{(1)}{}_{(k)} = \text{CrossGConv}\,(S^{\wedge}, H^{(1)}{}_{(k-1)}, H^{(2)}{}_{(k-1)})$$
$$H^{(2)}{}_{(k)} = \text{CrossGConv}\,(S^{\wedge}{}_{T}, H^{(2)}{}_{(k-1)}, H^{(1)}{}_{(k-1)})$$

Where $H^{(1)}{}_{(k)}$ and $H^{(2)}{}_{(k)}$ is the kth layer node embedding of $G^{(1)}$ and $G^{(2)}$; K represents the KTH iteration; S^ represents the prediction allocation matrix calculated from the shallow node embedding layer; CrossGConv stands for cross graph convolution network.

5.4 Message-passing Neural Network

Another very popular graph neural network architecture is the neural messaging Network (MPNN) [37], which was originally proposed for learning molecular graph representation. However, MPNN is actually very general, providing a general framework for graph neural networks and can also be used for node classification tasks. The basic idea of MPNN is to formalize the existing graph neural network as a general framework for neural message delivery between nodes. In MPNN, there are two important functions, including messages and updates:

$$m^k_i = \sum_{i \in N^{(j)}} M^k \left(H^{k-1}_i, H^{k-1}_j, e_j\right)$$
$$H^k_i = U^k \left(H^{k-1}_i, m^k_i\right)$$

M_k defines the message between the k-level nodes i and j, which depends on the representation of the two nodes and their edge information. Uk is the node update function in layer K, which combines aggregate messages from neighbours with the node representation itself.
The Sum AGGREGATE function is the Sum of all messages from neighbours. The COMBINE function is the same as the node update function described above.

5.5 Graph Attention Network

For the target node i, the importance of neighbour j is determined by the weight of their edge $e_{i\,j}$ (normalized by their node degree). In practice, however, input diagrams can be noisy. Edge weights may not reflect the true strength between two nodes. Therefore, a more principled approach is to automatically learn the importance of each neighbour.

Graph Attention Networks [12] build on this idea and try to learn the importance of each neighbour based on the Attention mechanism [38, 39]. Attention mechanisms have been widely used for a variety of tasks in natural language understanding (such as machine translation and question-and-answer) and computer vision (such as visual question-and-answer and image captioning).
Graph attention layer. The attention layer of the figure defines how to represent the hidden nodes of layer K-1:

$$H_{k-1} \in RN\times F$$

Moving to a new node means

$$H_k \in RN\times F'$$

In order to ensure sufficient representation power to convert lower-level node representation to higher-level node representation, a shared linear transformation is applied to each node, expressed as $W \in RF \times F'$. Then, self-attention is defined on the nodes, and the attention coefficient of any pair of nodes is measured by the shared attention mechanism A: $RF' \times RF' \rightarrow R$

$$e_{ij} = a(W\, H^{k-1}{}_{i}\, ,\, W\, H^{k-1}{}_{j}\,)$$

e_{ij} represents the strength of the relationship between nodes i and j. Each node could theoretically allow it to focus on every other node on the graph, but this would ignore graph structure information, and a more logical solution would be to focus only on each node's neighbours [12].

6 Design

GMN in this project is derived from the combination of different characteristics of various basic graph neural networks mentioned above. It uses a similar cross graph matching network based on standard messaging GNN to iteratively generate more discriminative node embeddings:

$$H^{(l)} = \{h^{(l)}_i\} \quad v_i \in V^{(l)}, \; l = \{1, 2\}$$

Used for two input diagrams. Intuitively speaking, it updates the node embedding of an input graph by combining soft attention with the attention association information of another input graph, which is similar to the attention mechanism and cross graph convolutional network mentioned above.

$$H^{(1)}_{(k)} = \text{CrossGConv}(\hat{S}, H^{(1)}_{(k-1)}, H^{(2)}_{(k-1)})$$
$$H^{(2)}_{(k)} = \text{CrossGConv}(\hat{S}^{\top}, H^{(2)}_{(k-1)}, H^{(1)}_{(k-1)})$$

Where $H^{(1)}_{(k)}$ and $H^{(2)}_{(k)}$ are nodes embedded in kth layer of graph $G_{(1)}$ and $G_{(2)}$. K represents the Kth iteration; S^ represents the prediction allocation matrix calculated from the shallow node embedding layer; CrossGConv stands for cross graph convolution network.

Subsequently, in order to calculate the similarity score, GMN adopts the following aggregation operation [40] to provide a graph-level embedding vector for each output, i.e

$$h_G^{(l)}, l = \{1, 2\}$$

And the existing similarity function is applied to the final similarity prediction, namely:

$$S(h_G^{(1)}, h_G^{(2)}) = f_s(h_G^{(1)}, h_G^{(2)})$$

The f_s can be any existing similarity function such as Euclid, cosine or similar hamming function:

Euclidean distance:

$$d_H(x,y)=\sqrt{\sum (xi - yi)2}$$

Hamming distance:

$$d_H(x,y)=\sum Hi=1 \mid [x_i \neq y_i]$$

$$h_G^{(l)} = MLP_{\theta 1}(\sum_{vi} \in V^{(l)} \sigma (MLP_{\theta 2}(h^{(l)}{}_i)) \odot MLP_{\theta 3}(h^{(l)}{}_i)), l = \{1, 2\}$$

Where σ represents the activation function. ⊙ indicates element by element multiplication. MLPθ 1, MLPθ 2 and MLPθ 3 are the MLP networks to be trained. Graph Matching Networks are built on Graph node representation learning but focus more on the interaction of two graphs from low-level nodes to high-level graphs.

It works in three steps explained below and illustrated in Figure 5.:

Firstly, 2 separate MLPs progressively transform embeddings without changing the connectivity. Secondly, a message-passing layer to let the model be aware of graph connectivity using the attention mechanism, which is shown in Figure 6., to get cross-graph messages. Thirdly, aggregate all neighbouring messages via any existing aggregation function (Sum, Mean, or Max).

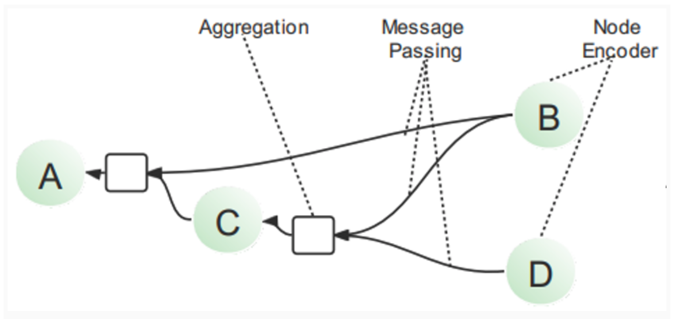


Figure 5. An illustration of three steps of GNN.

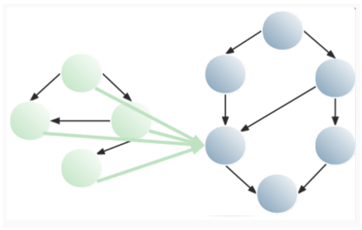


Figure 6. An illustration of how the attention mechanism works.

7 Experimental method

7.1 Dependencies and imports required

The code of this project is all written in python and used mainly PyTorch. All the required modules can be accessed through the GitHub link listed at the end of this report in the requirements.txt file.

Install all the requirements via “pip install -r requirements.txt”
And following is a copy of the file:
Pytorch =1.5,

networkx >= 2.3,

torch-sparse==0.6.7 (pip install torch-sparse),

torch-cluster==1.4.5 (pip install torch-cluster),

torch-geometric==1.3.2 (pip install torch-geometric)

(pip install dgl-cu101)

numpy>=1.16.4

six>=1.12

7.2 Measurements (Lost function)

Based on different supervision of training samples, for example, the binary label of ground-truth between two graphs or the relative similarity between three graphs, this project adopts the same loss function as Li [8]: Two margin-based loss functions, namely pair Loss Function and Triplet Loss Function and different similarity function f_s:

$$S(h_G^{(1)}, h_G^{(2)}) = f_s(h_G^{(1)}, h_G^{(2)})$$

Where f_s can be any existing similarity function such as Euclidean, cosine, or Hamming similarity function:

Euclidean distance:

$$d_H(x,y)=\sqrt{\sum (xi - yi)2}$$

Hamming distance:

$$d_H(x,y)=\sum Hi=1 \mid [x_i \neq y_i]$$

7.3 Datasets

This project used two real-world datasets the QM7 dataset and the QM7b dataset [41, 42].
"QM7 dataset is a subset of GDB-13, a database of nearly 1 billion stable and synthesizable organic molecules, consisting of all molecules with up to 23 atoms (including seven heavy atoms C, N, O, and S), for a total of 7,165 molecules." [41, 42].
Download link: data/qm7.mat (17.9 MB)
"QM7b dataset is an extension of the QM7 dataset for multitasking learning, in which 13 additional properties (e.g., polarization, HOMO and LUMO eigenvalues, excitation energy) must be predicted at different theoretical levels (ZINDO, SCS, PBE0, GW) [41, 42]. It also includes other molecules that contain chlorine atoms, 7,211 in all" [41, 42].
Download link: data/qm7b.mat (16.1 MB)

The following is an example of the configuration:

```
encoder= {'node_hidden_sizes': [32], 'node_feature_dim': 1, 'edge_hidden_sizes': [16]}
aggregator= {'node_hidden_sizes': [128], 'graph_transform_sizes': [128], 'input_size': [32],
'gated': True, 'aggregation_type': 'sum'}
graph_embedding_net= {'node_state_dim': 32, 'edge_state_dim': 16, 'edge_hidden_sizes': [64,
64], 'node_hidden_sizes': [64], 'n_prop_layers': 5, 'share_prop_params': True,
'edge_net_init_scale': 0.1, 'node_update_type': 'gru', 'use_reverse_direction': True,
'reverse_dir_param_different': False, 'layer_norm': False, 'prop_type': 'matching'}
graph_matching_net= {'node_state_dim': 32, 'edge_state_dim': 16, 'edge_hidden_sizes': [64,
64], 'node_hidden_sizes': [64], 'n_prop_layers': 5, 'share_prop_params': True,
'edge_net_init_scale': 0.1, 'node_update_type': 'gru', 'use_reverse_direction': True,
'reverse_dir_param_different': False, 'layer_norm': False, 'prop_type': 'matching', 'similarity':
'dotproduct'}
model_type= matching
data= {'problem': 'graph_edit_distance', 'dataset_params': {'n_nodes_range': [20, 20],
'p_edge_range': [0.2, 0.2], 'n_changes_positive': 1, 'n_changes_negative': 2,
'validation_dataset_size': 1000}}
training= {'batch_size': 20, 'learning_rate': 0.0001, 'mode': 'pair', 'loss': 'margin', 'margin': 1.0,
'graph_vec_regularizer_weight': 1e-06, 'clip_value': 10.0, 'n_training_steps': 500000,
'print_after': 100, 'eval_after': 10}
evaluation= {'batch_size': 20}
seed= 8
```

The measurement of accuracy was made after 40000 iterations.

```
iter 1000, loss 1.0307, sim_pos -1.0216, sim_neg -1.1651, sim_diff 0.1434, val/pair_auc 0.6844,
val/triplet_acc 0.6910, time 26.29s

iter 10000, loss 0.9144, sim_pos -0.8796, sim_neg -1.0608, sim_diff 0.1813, val/pair_auc 0.7836,
val/triplet_acc 0.8140, time 23.59s

iter 20000, loss 0.6656, sim_pos -0.8212, sim_neg -1.6955, sim_diff 0.8743, val/pair_auc 0.8254,
val/triplet_acc 0.8280, time 23.39s

iter 30000, loss 0.4941, sim_pos -0.6990, sim_neg -2.1143, sim_diff 1.4153, val/pair_auc 0.8360,
val/triplet_acc 0.8310, time 23.21s

iter 40000, loss 0.7887, sim_pos -1.0414, sim_neg -1.5750, sim_diff 0.5337, val/pair_auc 0.8406,
val/triplet_acc 0.8490, time 23.17s
```

8 Results and calculations

The column following Table 2. shows different combinations of dimensionality of node embedding and edge embedding. For example, '16, 8' means this model is trained with 16 being the node embedding dimension and 8 being the edge one. The model AUC (Area under curve) at different iterations (50000 epochs in total), which is the area under the ROC curve drawn for each classification model as an indicator of different classification models, is shown in the table to compare the impact of different embedding dimensionality and aggregation type on the model training.

Table 2. Performance of Different models under the combination of dimensions embedded in different nodes and different aggregation operation types

	Sum	Mean	Max
16, 8	0.73	0.70	0.69
32, 8	0,74	0.70	0.73
64, 8	0.70	0.70	0.67

16, 16	0.73	0.74	0.74
32, 16	0.75	0.75	0.75
64, 16	0.76	0.73	0.73
16, 32	0.76	0.74	0.73
32, 32	0.74	0.71	0.75
64, 32	0.74	0.77	0.73
16, 64	0.76	0.78	0.75
32, 64	0.75	0.76	0.73
64, 64	0.75	0.76	0.79

9 Discussion

Through experiments conducted with different aggregation types and different sizes of node feature and edge feature dimensions, this model achieves varying degrees of effectiveness in predicting the similarity. The impact of choosing different node embedding dimensions and different aggregation operation types on the mean and upper and lower limits of performance is examed. The following Figure is the performance distribution diagram under the combination of dimensions embedded in different nodes and different types of aggregation operations.

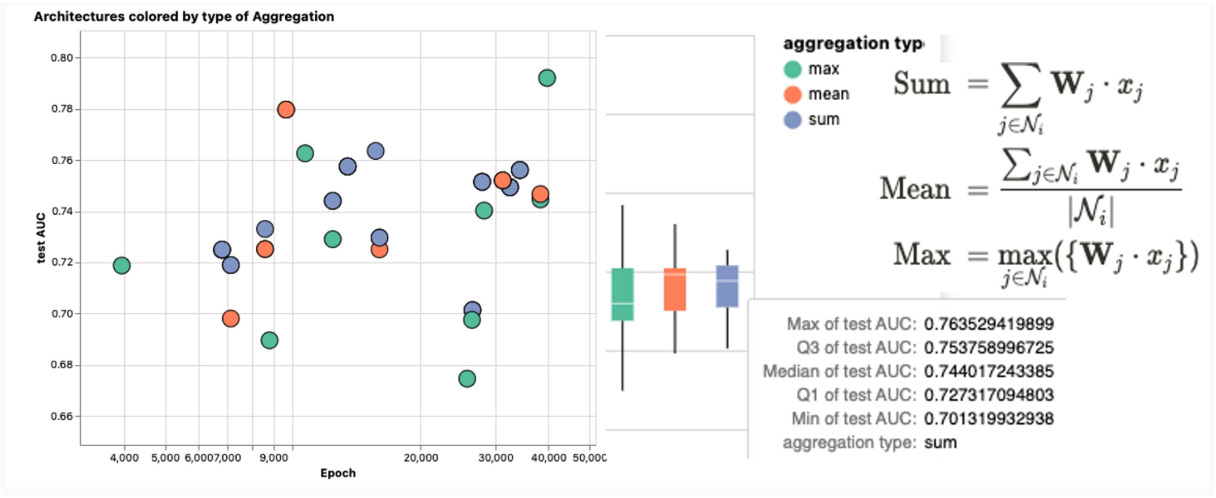
9.1 Comparison of performance based on different embedded dimensions

Experiments with different data sets and different ways of constructing composite graphs show that models with higher dimensions tend to have better mean and lower limit performance. The model is sensitive, and different parameters have a significant influence on the results. Based on the observation of the graph, the trend is not obvious, but in general, under the same aggregation operation, the larger the embedding dimension, the better the performance distribution. But the marginal effect of increasing dimensions on performance is very small. The choice of embedding dimensions that lead to good enough performance is not constant but depends on the data being applied. In another piece of literature, Dwivedi [43] introduced a GNN benchmark testing framework. In this paper [43], the author used ZINC, PATTERNCLUSTER, MINISTCIFAR10, and TSP data sets to conduct experiments, and the distribution of specific performance results had very similar results. It is a bold inference that more parameters may mean higher performance. The parameter efficiency of the GNN model is very high. Even with a few parameters (with Node embedding Dimension being 16), high-performance models can be found [43].

9.2 Comparison of performance effects of aggregation operations

By observing and comparing the performance distribution of aggregation operations based on learning representations of different graph attributes, it is difficult to observe which aggregation operation is completely superior to other operations. Max operation has the highest performance when the node embedding dimension and edge embedding dimension are both 64, but the worst performance when the node embedding dimension and edge embedding dimension are 64 and 8, respectively. The Mean operation has a relatively optimal mean value, while the Sum operation has optimal stability. In the work of You [35], You tried to obtain the best GNN design in three different tasks (graph-IMDB, Node-Smallworld, and

Node-CIteseer) and came to the conclusion that sum performed best as an aggregation operation in general.
In the work of [35], aggregators were also studied by ablation study. Mean and max-pooling are used, but sum does not perform well in this scenario. This paper is also further sorted by representation ability, with the sum operation capturing the complete multiple sets, the mean operation capturing the proportion and distribution of elements of a given type, and the Max operation aggregator ignoring multiplicity [35].
Embedded aggregations should use smooth aggregations that sort the nodes and provide a constant number of nodes. In addition, an ideal property of an aggregation operation is that similar inputs provide similar aggregate outputs, and dissimilar inputs yield distinct aggregate outputs that can be distinguished [35]. When neighbourhood aggregation is performed, the mean or maximum value remains the same, and by induction, the same node representation will always get everywhere. Therefore, in this case, mean and Max pool aggregators cannot capture any structural information [35]. The maximum pool treats multiple nodes with the same function as only one node, resulting in its inability to capture the exact structure or distribution [35].
In summary, no aggregation operation is consistently the best choice. When a node has a highly variable number of neighbours or tasks that need to normalize the characteristics of the local neighbourhood, mean operation may be very useful [35]. The Max operation can be useful when a task needs to highlight a single salient feature. Since Sum is not normalized, it provides a balance between the two. Both provide a representation of the local distribution of features and highlight outliers. In general practice, with no special requirements, the Sum operation is usually the more balanced choice.



$$\text{Sum} = \sum_{j \in \mathcal{N}_i} \mathbf{W}_j \cdot x_j$$

$$\text{Mean} = \frac{\sum_{j \in \mathcal{N}_i} \mathbf{W}_j \cdot x_j}{|\mathcal{N}_i|}$$

$$\text{Max} = \max_{j \in \mathcal{N}_i}(\{\mathbf{W}_j \cdot x_j\})$$

```
     Max of test AUC: 0.763529419899
      Q3 of test AUC: 0.753758996725
  Median of test AUC: 0.744017243385
      Q1 of test AUC: 0.727317094803
     Min of test AUC: 0.701319932938
    aggregation type: sum
```

Figure 7. Performance distribution diagram under the combination of dimensions embedded in different nodes and different aggregation operation types.

9.3 Potential improvements

Firstly, with graph matching problems involving input graph pairs, the corresponding features between the two graphs are the basis for graph matching and graph similarity learning. Existing models cause additional computing overhead that can't be ignored, and in the future, it may be possible to develop more fine-grained cross graph features.

Secondly, a Heterogeneous graph (HG), also known as a heterogeneous information network (HIN), as shown in the figure below, contains different types of nodes and edges, which have become ubiquitous in the real world and are commonly seen in knowledge graph scenarios. Embedding in learning low-dimensional space while preserving heterogeneous structure and semantics for downstream tasks.

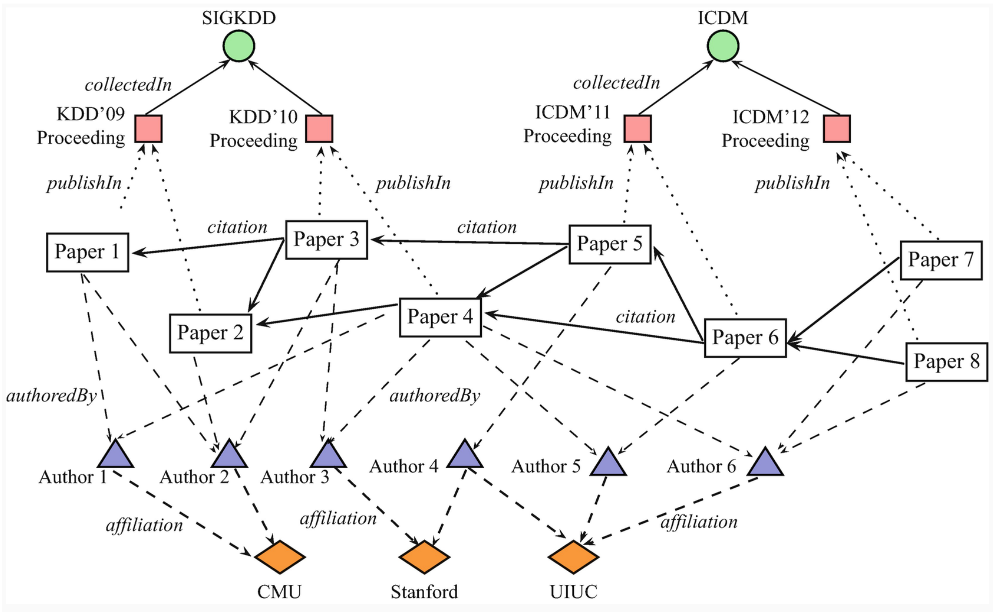


Figure 8. An example of a Heterogeneous Graph that where contains five different types of nodes [44].

10 Conclusions

Graph matching networks (GMNs) is utilized in this project to predict the similarity between graph-structured data by using deep neural networks.
In an attention-based GMN, a similarity measure is learned based on input graph pairs. A message-passing graph neural network is used to iteratively produce more distinct node embeddings, and an attention mechanism is used for incorporating relevant information from the other graph in the input pair into the node embeddings. The input graph pairs are created during the model training phase using synthetic and real datasets and are then modified and labelled according to a classic graph similarity index: the graph edit distance (GED). The model has different prediction effects on similarity depending on aggregation types, node features, and edge feature sizes. As a result, the project observed and speculated on

these results and also verified how different node embedding dimensions and aggregation methods affect the performance mean and upper and lower limits.

References

[1] P. K. Singh, “Data with Non-Euclidean Geometry and Its Characterization” Journal of Artificial Intelligence and Technology, 07, 2005, 2(1), 3–8.

[2] H. Gao, S. Ji, “Graph U-Nets” International Conference on Machine Learning, PMLR, pp 2083–2092

[3] A. Patil, Manyi Li, H. Zhang, “LayoutGMN: Neural Graph Matching for Structural Layout Similarity” 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 12, 2020

[4] B. Samanta, D. Abir, G. Jana, P.K. Chattaraj, N. Ganguly, M.G. Rodriguez, “Nevae: A deep generative model for molecular graphs” Proceedings of the AAAI Conference on Artifcial Intelligence, vol 33, pp 1110–1117

[5] K. He, X. Zhang, S. Ren, J. Sun, “Deep residual learning for image recognition” Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

[6] Y. Li, L. Zhang, Z. Liu, “Multi-objective de novo drug design with conditional
graph generative model” Journal of cheminformatics 10(1):1–24

[7] S. Wang, Z. Chen, X. Yu, D. Li, J. Ni, L. Tang, J. Gui, Z. Li, H. Chen, P.S. Yu, " Heterogeneous graph matching networks for unknown malware detection" Proceedings of the Twenty-Eighth International Joint Conference on Artifcial Intelligence, IJCAI, pp 3762–3770

[8] Y. Li, C. Gu, T. Dullien, O. Vinyals, P. Kohli, "Graph matching networks for learning the similarity of graph structured objects." International Conference on Machine Learning, PMLR, pp 3835–3845

[9] S. Bourigault, C. Lagnier, S. Lamprier, L. Denoyer, P. Gallinari, "Learning social network embeddings for predicting information diffusion" Proceedings of the 7th ACM international conference on Web search and data mining, pp 393–402

[10] H. Jizhou, W. Haifeng , S. Yibo, F. Miao, H. Zhengjie, "HGAMN: Heterogeneous Graph Attention Matching Network for Multilingual POI Retrieval at Baidu Ma" Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data MiningAugust 2021 Pages 3032–3040

[11] W. L. Hamilton, "Inductive representation learning on large graphs" NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems, 12, 2017, Pages 1025–1035.

[12] P. Velickovic, "Graph attention networks" International Conference on Learning Representations, 04, 2018.

[13] Y. Chen, L. Wu, M. Zaki, "Iterative deep graph learning for graph neural networks: Better and robust node embeddings" NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing SystemsDecember 2020 Article No.: 1620Pages 19314–19326

[14] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, J. Leskovec, "Hierarchical

graph representation learning with differentiable pooling" Advances in Neural Information Processing Systems, pp 4800–4810

[15] Y. Ma, S. Wang, C.C. Aggarwal, J. Tang, "Graph convolutional networks with eigenpooling" ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, ACM, pp 723–731

[16] M. Simonovsky, N. Komodakis, "Graphvae: Towards generation of small graphs using variational autoencoders" arXiv preprint arXiv:180203480

[17] J. You, R. Ying, X. Ren, W. Hamilton, J. Leskovec "Graphrnn: Generating realistic graphs with deep auto-regressive models" International Conference on Machine Learning, PMLR, pp 5708–5717

[18] J. Yan, X.C. Yin, W. Lin, C. Deng, H. Zha, X. Yang, " A short survey of recent advances in graph matching" Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval, pp 167–174

[19] H. Bunke, "Error correcting graph matching: on the influence of the underlying cost function," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 21, no. 9, pp. 917-922, Sept. 1999, doi: 10.1109/34.790431.

[20] J. Yan, S. Yang, E. Hancock, "Learning for graph matching and related combinatorial optimization problems" Bessiere C (ed) Proceedings of the TwentyNinth International Joint Conference on Artificial Intelligence, IJCAI-20, International Joint Conferences on Artificial Intelligence Organization, pp 4988–4996

[21] H. Bunke, "On a relation between graph edit distance and maximum common subgraph" Pattern Recognition Letters 18(8):689–694

[22] K. Riesen "Structural Pattern Recognition with Graph Edit Distance Approximation Algorithms and Applications" Springer

[23] Thinklab-SJTU, "GitHub page - Thinklab-SJTU" 07, 2021. [Online]. Available: https://github.com/Thinklab-SJTU/ThinkMatch/blob/master/docs/images/superglue.png [Accessed Apr. 26, 2022].

[24] E.M. Loiola, N.M.M. de Abreu, P.O. Boaventura-Netto, P. Hahn, T. Querido, "A survey for the quadratic assignment problem" European journal of operational research 176(2):657–690

[25] R. Romain, "A graph matching method and a graph matching distance based on subgraph assignments" Pattern Recognition LettersVolume 31Issue, 04, 2010 pp 394–406

[26] P. Foggia, G. Percannella, M. Vento, "Graph matching and learning in pattern recognition in the last 10 years" International Journal of Pattern Recognition and Artificial Intelligence 28(01):1450,001

[27] T. Derr, Y. Ma and J. Tang, "Signed Graph Convolutional Networks," 2018 IEEE International Conference on Data Mining (ICDM), 2018, pp. 929-934, doi: 10.1109/ICDM.2018.00113.

[28] M. Yao, Z. Guo, Z. Ren, E. Zhao, J.g Tang, D. Yin, "Streaming graph neural networks" SIGIR 2020.

[29] T.N. Kipf, M. Welling, "Semi-supervised classifcation with graph convolutional networks" 5th International Conference on Learning Representations, 04, 2017, ICLR

[30] Y. Rong, Y. Bian, T. Xu, W. Xie, Y. Wei, W. Huang, J. Huang, “Self-supervised graph transformer on large-scale molecular data” Advances in Neural Information Processing Systems 33

[31] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," in IEEE Transactions on Neural Networks and Learning Systems, vol. 32, no. 1, pp. 4-24, 01, 2021, doi: 10.1109/TNNLS.2020.2978386.

[32] X. Yan, J. Han, "gSpan: graph-based substructure pattern mining," 2002 IEEE International Conference on Data Mining, 2002. Proceedings., 2002, pp. 721-724, doi: 10.1109/ICDM.2002.1184038.

[33] X. Ling, L. Wu, Wang S, G. Pan, T. Ma, F. Xu, A.X. Liu, C. Wu, S. Ji, “Deep graph matching and searching for semantic code retrieval” ACM Transactions on Knowledge Discovery from Data (TKDD)

[34] Z. L, C. C, “Heterogeneous Graph Neural Networks for Malicious Account Detection” 2018, CIKM, pp.2077-2085

[35] J. You, R. Ying, J. Leskovec, “Design Space for Graph Neural Networks” NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing SystemsDecember 2020 Article No.: 1427Pages 17009–17021

[36] R. Wang, J. Yan and X. Yang, "Learning Combinatorial Embedding Networks for Deep Graph Matching," 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 3056-3065, doi: 10.1109/ICCV.2019.00315.

[37] J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, G.E. Dahl, “Neural message passing for quantum chemistry” Precup D, Teh YW (eds) Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW,

Australia, 6-11 August 2017, PMLR, Proceedings of Machine Learning Research, vol 70, pp 1263–1272

[38] D. Bahdanau, K. Cho, Y. Bengio, “Neural machine translation by jointly learning to align and translate” 3rd International Conference on Learning Representations

[39] A. Vaswani, N. Shazeer, N. Parmar, J, Uszkoreit, L, Jones, A.N. Gomez,u. Kaiser, I. Polosukhin, “Attention is all you need” Proceedings of the 31st International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, NIPS’17, p 6000–6010

[40] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, “Gated graph sequence neural Networks” International Conference on Learning Representations (ICLR)

[41] L. C. Blum, J.-L. Reymond, 970 Million Druglike Small Molecules for Virtual Screening in the Chemical Universe Database GDB-13, J. Am. Chem. Soc., 131:8732, 2009. [Online]. Available: https://pubs.acs.org/doi/10.1021/ja902302h [Accessed: Apr. 26, 2022].

[42] M. Rupp, A. Tkatchenko, K.-R. Müller, O. A. von Lilienfeld: Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning, Physical Review Letters, 108(5):058301, 2012. [Online]. Available: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.108.058301 [Accessed: Apr. 26, 2022].

[43] V.P. Dwivedi, C.K. Joshi, T. Laurent, Y. Bengio, X. Bresson, “Benchmarking Graph Neural Networks” Chemistry. ChemRxiv. Cambridge: Cambridge Open Engage, 2021

[44] X. Kong, P.S. Yu "Graph Classification in Heterogeneous Networks" [Online]. Available: https://doi.org/10.1007/978-1-4939-7131-2_176 [Accessed: Apr. 26, 2022].

[45] Y. Lin, "GitHub page - Graph-Matching-Networks" 03, 2021. [Online]. Available: https://github.com/Lin-Yijie/Graph-Matching-Networks [Accessed Apr. 26, 2022].

Appendix - Code Embed

The code in this work is a Graph Matching Network that operates on small unidirectional graphs. A PyTorch implementation of Graph Matching Networks built on top of the implementation by Lin-Yijie [45].
Code of this project available:
https://github.com/ItsShi/GraphNeuralNetworkforSimilarityLearning
The code of this project is all written in python and used mainly PyTorch. All the required modules can be accessed through the GitHub link listed at the end of this report in the 'requirements.txt' file. Install all the requirements via "pip install -r requirements.txt".
And following is a copy of the file:
Pytorch =1.5,
networkx >= 2.3,
torch-sparse==0.6.7 (pip install torch-sparse),
torch-cluster==1.4.5 (pip install torch-cluster),
torch-geometric==1.3.2 (pip install torch-geometric)
(pip install dgl-cu101)
numpy>=1.16.4

six>=1.12

The following is a copy the all of the code:

```
# Graph Matching Networks for the Similarity Learning

## Some dependencies and imports

#These are all the dependencies that will be used in this notebook.


gpu_info = !nvidia-smi
gpu_info = '\n'.join(gpu_info)
if gpu_info.find('failed') >= 0:
 print('Not connected to a GPU')
else:
 print(gpu_info)

from psutil import virtual_memory
ram_gb = virtual_memory().total / 1e9
print('Your runtime has {:.1f} gigabytes of available RAM\n'.format(ram_gb))

if ram_gb < 20:
 print('Not using a high-RAM runtime')
else:
 print('You are using a high-RAM runtime!')


!pip uninstall torch  #1.10.0+cu111
!pip install torch==1.5
!nvcc --version
print("\n")

import torch
import torch.nn as nn
print(torch.___version___)

# !pip install torch-scatter
# !pip install torch-sparse
# !pip install torch-geometric
# import torch_geometric

#dgl
# !git init
# !git clone https://github.com/dmlc/dgl.git
# !git submodule init
# !git submodule update

!pip install dgl-cu101

# #gpu
# if torch.cuda.is_available():
#   !pip install -q torch-scatter -f https://data.pyg.org/whl/torch-1.9.0+cu111.html
#   !pip install -q torch-sparse -f https://data.pyg.org/whl/torch-1.9.0+cu111.html #gpu
#   !pip install -q git+https://github.com/pyg-team/pytorch_geometric.git
#   import torch_geometric

# else:
#   !pip install -q torch-scatter -f https://data.pyg.org/whl/torch-1.10.0+cpu.html
#   !pip install -q torch-sparse -f https://data.pyg.org/whl/torch-1.10.0+cpu.html  #cpu pytorch1.10.0
#   !pip install -q git+https://github.com/pyg-team/pytorch_geometric.git
#   import torch_geometric

#pascal VOC dataset
# !pip install opencv-python
# !pip3 install torchvision


# !pip install https://github.com/fastai/fastai/archive/master.zip


# !apt update && apt install -y libsm6 libxext6


# !pip3 install
http://download.pytorch.org/whl/cu80/torch-0.3.0.post4-cp36-cp36m-linux_x86_64.whl

# !mkdir data

# #Wget: supports downloading via HTTP, HTTPS, and FTP
# !wget http://pjreddie.com/media/files/VOCtrainval_06-Nov-2007.tar -P data/

# print("\n")
# !wget https://storage.googleapis.com/coco-dataset/external/PASCAL_VOC.zip -P data/

# #tar(tarball): for collecting many files into one archive file, "tape archive"

# !tar -xf data/VOCtrainval_06-Nov-2007.tar -C data/


# !unzip data/PASCAL_VOC.zip -d data/


# !rm -rf data/PASCAL_VOC.zip data/VOCtrainval_06-Nov-2007.tar

#compare existing deep graph matching algorithms under different datasets
#provides a unified data interface and an evaluating platform
#pygmtools supports 5 datasets, including PascalVOC, Willow-Object, SPair-71k, CUB2011 and
IMC-PT-SparseGM.
# !pip uninstall Pillow
# !pip install Pillow==7.2.0
# !pip install pygmtools
#dataset.py: to download dataset and process the dataset into a json file, and also save train set and
test set.
```

```
#benchmark.py:  to fetch data from json file and evaluate prediction result.
#dataset_config.py: Fixed dataset settings, mostly dataset path and classes.

'''
evaluation metrics include matching_precision (p), matching_recall (r) and f1_score (f1).
Also, to measure the reliability of the evaluation result,
we define coverage (cvg) for each class in the dataset as number of evaluated pairs in the class /
number of all possible pairs in the class.
Therefore, larger coverage refers to higher reliability.
'''

'''
#example
from pygmtools.benchmark import Benchmark

# Define Benchmark on PascalVOC.
bm = Benchmark(name='PascalVOC', sets='train',
        obj_resize=(256, 256), problem='2GM',
        filter='intersection')

# Random fetch data and ground truth.
data_list, gt_dict, _ = bm.rand_get_data(cls=None, num=2)
'''

# from google.colab import drive
# drive.mount('/content/drive')

'''
import matplotlib
import networkx
import numpy
import pandas
# import scikitlearn
import scipy
import texttable
import tqdm
import Cython

print(matplotlib. __version__)
print(networkx. __version__)
print(numpy. __version__)
print(pandas. __version__)
print(matplotlib. __version__)
print(scipy. __version__)
print(texttable. __version__)
print(tqdm. __version__)
print(Cython. __version__)

# matplotlib==3.3.4
# networkx==2.5
# numpy==1.20.1
# pandas==1.2.3
# scikit-learn==0.24.1
# scipy==1.6.1
# texttable==1.6.3
# tqdm==4.59.0
# Cython==0.29.23

'''

#%loadpy '-/Users/xiaoweishi/Downloads/Graph-Matching-Networks-main/segment.py'

#background:


#training and learning:
You can choose pairwise training or ternary training
Pairwise training requires labels positive (similar) or negative (dissimilar)
And ternary training only needs to be relatively similar, i.e. whether G1 is closer to G2 or G3

Edge-based pairwise loss using Euclidean similarity
#
'''


$$
L_\mathrm{pair} = \mathbb{E}_{(G_1, G_2, t)}[\max\{0, \gamma - t(1 - d(G_1, G_2))\}]
$$




### Graph embedding model


#### The graph encoder


$$d(G_1, G_2) = d_H(embed(G_1), embed(G_2)),$$


1. $embed$ maps any graph  $G$  into an $H$ -dimensional vector:

$embed(G_1) = $

1) mapping:

$$\begin{array}{rcl}
h_i^{(0)} &=& \mathrm{MLP_{node}}(x_i) \\
e_{ij} &=& \mathrm{MLP_{edge}}(x_{ij})
\end{array}
$$

2) iterative message passing:

$$\begin{array}{rcl}
m_{i\rightarrow j} &=& f_\mathrm{message}(h_i^{(t)}, h_j^{(t)}, e_{ij}) \\
```

h_i^{(t+1)} &=& f_\mathrm{node}(h_i^{(t)}, \sum_{j:(j,i)\in E} m_{j\rightarrow i})
\end{array}
$$

3) aggregate node representations to get graph representations

$$h_G = \mathrm{MLP_G}\left(\sum_{i\in V} h_i^{(T)}\right).$$
$$h_G = \mathrm{MLP_G}\left(\sum_{i\in V} \sigma(\mathrm{MLP_{gate}}(h_i^{(T)})) \odot \mathrm{MLP}(h_i^{(T)})\right).$$

2. d_H is an existing distance metric\:

Euclidean distance: $d_H(x, y) = \sqrt{\sum_{i=1}^H (x_i - y_i)^2}$,

or Hamming distance: $d_H(x, y)=\sum_{i=1}^H \mathbb{I}[x_i \ne y_i]$

Adam

```
import torch.nn as nn

class GraphEncoder(nn.Module):
  """Encoder module that projects node and edge features to some embeddings."""

  def __init__(self,
          node_feature_dim,
          edge_feature_dim,
          node_hidden_sizes=None,
          edge_hidden_sizes=None,
          name='graph-encoder'):

    super(GraphEncoder, self).__init__()

    self._node_feature_dim = node_feature_dim
    self._edge_feature_dim = edge_feature_dim
    self._node_hidden_sizes = node_hidden_sizes if node_hidden_sizes else None
    self._edge_hidden_sizes = edge_hidden_sizes
    self._build_model()

  def _build_model(self):
    layer = []
    layer.append(nn.Linear(self._node_feature_dim, self._node_hidden_sizes[0]))
    for i in range(1, len(self._node_hidden_sizes)):
      layer.append(nn.ReLU())
      layer.append(nn.Linear(self._node_hidden_sizes[i - 1], self._node_hidden_sizes[i]))
    self.MLP1 = nn.Sequential(*layer)

    if self._edge_hidden_sizes is not None:
      layer = []
      layer.append(nn.Linear(self._edge_feature_dim, self._edge_hidden_sizes[0]))
      for i in range(1, len(self._edge_hidden_sizes)):
            layer.append(nn.ReLU())
            layer.append(nn.Linear(self._edge_hidden_sizes[i - 1], self._edge_hidden_sizes[i]))
          self.MLP2 = nn.Sequential(*layer)
      else:
          self.MLP2 = None

  def forward(self, node_features, edge_features=None):
      """Encode node and edge features.
      Args:
        node_features: [n_nodes, node_feat_dim] float tensor.
        edge_features: if provided, should be [n_edges, edge_feat_dim] float
          tensor.
      Returns:
        node_outputs: [n_nodes, node_embedding_dim] float tensor, node embeddings.
        edge_outputs: if edge_features is not None and edge_hidden_sizes is not
          None, this is [n_edges, edge_embedding_dim] float tensor, edge
          embeddings; otherwise just the input edge_features.
      """
      if self._node_hidden_sizes is None:
          node_outputs = node_features
      else:
          node_outputs = self.MLP1(node_features)
      if edge_features is None or self._edge_hidden_sizes is None:
          edge_outputs = edge_features
      else:
          edge_outputs = self.MLP2(edge_features)

      return node_outputs, edge_outputs



#### The message passing layers

def unsorted_segment_sum(data, segment_ids, num_segments):
  """
  Computes the sum along segments of a tensor. Analogous to tf.unsorted_segment_sum.
  :param data: A tensor whose segments are to be summed.
  :param segment_ids: The segment indices tensor.
  :param num_segments: The number of segments.
  :return: A tensor of same data type as the data argument.
  """
  # -------------print("\ndata: {}\nsegment_ids: {}\nnum_segments: {}".format(data,
segment_ids, num_segments))
  assert all([i in data.shape for i in segment_ids.shape]), "segment_ids.shape should be a prefix of
data.shape"

  # Encourage to use the below code when a deterministic result is
  # needed (reproducibility). However, the code below is with low efficiency.

  # tensor = torch.zeros(num_segments, data.shape[1]).cuda()
  # for index in range(num_segments):
  #    tensor[index, :] = torch.sum(data[segment_ids == index, :], dim=0)
  # return tensor
  #----------------print("\ndata.shape:{} and segment_ids.shape: {}".format(data.shape,
segment_ids.shape))
  if len(segment_ids.shape) == 1:
    if torch.cuda.is_available():
      s = torch.prod(torch.tensor(data.shape[1:])).long().cuda()
      segment_ids = segment_ids.repeat_interleave(s).view(segment_ids.shape[0], *data.shape[1:])
    else:
      s = torch.prod(torch.tensor(data.shape[1:])).long()
      segment_ids = segment_ids.repeat_interleave(s).view(segment_ids.shape[0], *data.shape[1:])
  #----------------- print("\ndata.shape:{} and segment_ids.shape: {}".format(data.shape,
segment_ids.shape))
  assert data.shape == segment_ids.shape, "data.shape and segment_ids.shape should be equal"

  shape = [num_segments] + list(data.shape[1:])
  if torch.cuda.is_available():
    tensor = torch.zeros(*shape).cuda().scatter_add(0, segment_ids, data)
  else:
    tensor = torch.zeros(*shape).scatter_add(0, segment_ids, data)

  tensor = tensor.type(data.dtype)
  return tensor


def graph_prop_once(node_states,
          from_idx,
          to_idx,
          message_net,
          aggregation_module=None,
          edge_features=None):
  """One round of propagation (message passing) in a graph.
  Args:
    node_states: [n_nodes, node_state_dim] float tensor, node state vectors, one
      row for each node.
    from_idx: [n_edges] int tensor, index of the from nodes.
    to_idx: [n_edges] int tensor, index of the to nodes.
    message_net: a network that maps concatenated edge inputs to message
      vectors.
    aggregation_module: a module that aggregates messages on edges to aggregated
      messages for each node.  Should be a callable and can be called like the
      following,
      `aggregated_messages = aggregation_module(messages, to_idx, n_nodes)`,
      where messages is [n_edges, edge_message_dim] tensor, to_idx is the index
      of the to nodes, i.e. where each message should go to, and n_nodes is an
      int which is the number of nodes to aggregate into.
    edge_features: if provided, should be a [n_edges, edge_feature_dim] float
      tensor, extra features for each edge.
  Returns:
    aggregated_messages: an [n_nodes, edge_message_dim] float tensor, the
      aggregated messages, one row for each node.
  """
  from_states = node_states[from_idx]
  to_states = node_states[to_idx]
  edge_inputs = [from_states, to_states]

  if edge_features is not None:
    edge_inputs.append(edge_features)

  edge_inputs = torch.cat(edge_inputs, dim=-1)
  messages = message_net(edge_inputs)
  #----------------------print("messages: {}\nto_idx: {}\n".format(messages, to_idx))
  tensor = unsorted_segment_sum(messages, to_idx, node_states.shape[0])
  return tensor



class GraphPropLayer(nn.Module):
  """Implementation of a graph propagation (message passing) layer."""

  def __init__(self,
         node_state_dim,
         edge_state_dim,
         edge_hidden_sizes,  # int
         node_hidden_sizes,  # int
         edge_net_init_scale=0.1,
         node_update_type='residual',
         use_reverse_direction=True,
         reverse_dir_param_different=True,
         layer_norm=False,
         prop_type='embedding',
         name='graph-net'):
    """Constructor.
    Args:
     node_state_dim: int, dimensionality of node states.
     edge_hidden_sizes: list of ints, hidden sizes for the edge message
      net, the last element in the list is the size of the message vectors.
     node_hidden_sizes: list of ints, hidden sizes for the node update
      net.
     edge_net_init_scale: initialization scale for the edge networks.  This
      is typically set to a small value such that the gradient does not blow
      up.
     node_update_type: type of node updates, one of {mlp, gru, residual}.
     use_reverse_direction: set to True to also propagate messages in the
      reverse direction.
     reverse_dir_param_different: set to True to have the messages computed
      using a different set of parameters than for the forward direction.
     layer_norm: set to True to use layer normalization in a few places.
     name: name of this module.
    """
    super(GraphPropLayer, self).__init__()

    self._node_state_dim = node_state_dim
    self._edge_state_dim = edge_state_dim
    self._edge_hidden_sizes = edge_hidden_sizes[:]

    # output size is node_state_dim
```

```
        self._node_hidden_sizes = node_hidden_sizes[:] + [node_state_dim]
        self._edge_net_init_scale = edge_net_init_scale
        self._node_update_type = node_update_type

        self._use_reverse_direction = use_reverse_direction
        self._reverse_dir_param_different = reverse_dir_param_different

        self._layer_norm = layer_norm
        self._prop_type = prop_type
        self.build_model()

        if self._layer_norm:
            self.layer_norm1 = nn.LayerNorm()
            self.layer_norm2 = nn.LayerNorm()

    def build_model(self):
        layer = []
        layer.append(nn.Linear(self._node_state_dim*2 + self._edge_state_dim,
self._edge_hidden_sizes[0]))
        for i in range(1, len(self._edge_hidden_sizes)):
            layer.append(nn.ReLU())
            layer.append(nn.Linear(self._edge_hidden_sizes[i - 1], self._edge_hidden_sizes[i]))
        self._message_net = nn.Sequential(*layer)

        # optionally compute message vectors in the reverse direction
        if self._use_reverse_direction:
            if self._reverse_dir_param_different:
                layer = []
                layer.append(nn.Linear(self._node_state_dim*2 + self._edge_state_dim,
self._edge_hidden_sizes[0]))
                for i in range(1, len(self._edge_hidden_sizes)):
                    layer.append(nn.ReLU())
                    layer.append(nn.Linear(self._edge_hidden_sizes[i - 1], self._edge_hidden_sizes[i]))
                self._reverse_message_net = nn.Sequential(*layer)
            else:
                self._reverse_message_net = self._message_net

        if self._node_update_type == 'gru':
            if self._prop_type == 'embedding':
                self.GRU = torch.nn.GRU(self._node_state_dim * 2, self._node_state_dim)
            elif self._prop_type == 'matching':
                self.GRU = torch.nn.GRU(self._node_state_dim * 3, self._node_state_dim)
        else:
            layer = []
            if self._prop_type == 'embedding':
                layer.append(nn.Linear(self._node_state_dim * 3, self._node_hidden_sizes[0]))
            elif self._prop_type == 'matching':
                layer.append(nn.Linear(self._node_state_dim * 4, self._node_hidden_sizes[0]))
            for i in range(1, len(self._node_hidden_sizes)):
                layer.append(nn.ReLU())
                layer.append(nn.Linear(self._node_hidden_sizes[i - 1], self._node_hidden_sizes[i]))
            self.MLP = nn.Sequential(*layer)
```

```
def _compute_aggregated_messages(
      self, node_states, from_idx, to_idx, edge_features=None):
    """Compute aggregated messages for each node.
    Args:
     node_states: [n_nodes, input_node_state_dim] float tensor, node states.
     from_idx: [n_edges] int tensor, from node indices for each edge.
     to_idx: [n_edges] int tensor, to node indices for each edge.
     edge_features: if not None, should be [n_edges, edge_embedding_dim]
      tensor, edge features.
    Returns:
     aggregated_messages: [n_nodes, aggregated_message_dim] float tensor, the
      aggregated messages for each node.
    """

    aggregated_messages = graph_prop_once(
      node_states,
      from_idx,
      to_idx,
      self._message_net,
      aggregation_module=None,
      edge_features=edge_features)

    # optionally compute message vectors in the reverse direction
    if self._use_reverse_direction:
      reverse_aggregated_messages = graph_prop_once(
        node_states,
        to_idx,
        from_idx,
        self._reverse_message_net,
        aggregation_module=None,
        edge_features=edge_features)

      aggregated_messages += reverse_aggregated_messages

    if self._layer_norm:
      aggregated_messages = self.layer_norm1(aggregated_messages)

    return aggregated_messages

def _compute_node_update(self,
              node_states,
              node_state_inputs,
              node_features=None):
    """Compute node updates.
    Args:
     node_states: [n_nodes, node_state_dim] float tensor, the input node
      states.
     node_state_inputs: a list of tensors used to compute node updates.  Each
      element tensor should have shape [n_nodes, feat_dim], where feat_dim can
      be different.  These tensors will be concatenated along the feature
      dimension.
     node_features: extra node features if provided, should be of size
      [n_nodes, extra_node_feat_dim] float tensor, can be used to implement
```

```
        different types of skip connections.
    Returns:
      new_node_states: [n_nodes, node_state_dim] float tensor, the new node
        state tensor.
    Raises:
      ValueError: if node update type is not supported.
    """
    if self._node_update_type in ('mlp', 'residual'):
        node_state_inputs.append(node_states)
    if node_features is not None:
        node_state_inputs.append(node_features)

    if len(node_state_inputs) == 1:
        node_state_inputs = node_state_inputs[0]
    else:
        node_state_inputs = torch.cat(node_state_inputs, dim=-1)

    if self._node_update_type == 'gru':
        node_state_inputs = torch.unsqueeze(node_state_inputs, 0)
        node_states = torch.unsqueeze(node_states, 0)
        _, new_node_states = self.GRU(node_state_inputs, node_states)
        new_node_states = torch.squeeze(new_node_states)
        return new_node_states
    else:
        mlp_output = self.MLP(node_state_inputs)
        if self._layer_norm:
            mlp_output = nn.self.layer_norm2(mlp_output)
        if self._node_update_type == 'mlp':
            return mlp_output
        elif self._node_update_type == 'residual':
            return node_states + mlp_output
        else:
            raise ValueError('Unknown node update type %s' % self._node_update_type)

  def forward(self,
            node_states,
            from_idx,
            to_idx,
            edge_features=None,
            node_features=None):
    """Run one propagation step.
    Args:
      node_states: [n_nodes, input_node_state_dim] float tensor, node states.
      from_idx: [n_edges] int tensor, from node indices for each edge.
      to_idx: [n_edges] int tensor, to node indices for each edge.
      edge_features: if not None, should be [n_edges, edge_embedding_dim]
        tensor, edge features.
      node_features: extra node features if provided, should be of size
        [n_nodes, extra_node_feat_dim] float tensor, can be used to implement
        different types of skip connections.
    Returns:
      node_states: [n_nodes, node_state_dim] float tensor, new node states.
    """
    aggregated_messages = self._compute_aggregated_messages(
      node_states, from_idx, to_idx, edge_features=edge_features)

    return self._compute_node_update(node_states,
                                     [aggregated_messages],
                                     node_features=node_features)

#### Graph aggregator

class GraphAggregator(nn.Module):
  """This module computes graph representations by aggregating from parts."""

  def __init__(self,
               node_hidden_sizes,
               graph_transform_sizes=None,
               input_size=None,
               gated=True,
               aggregation_type='sum',
               name='graph-aggregator'):
    """Constructor.
    Args:
     node_hidden_sizes: the hidden layer sizes of the node transformation nets.
      The last element is the size of the aggregated graph representation.
     graph_transform_sizes: sizes of the transformation layers on top of the
      graph representations.  The last element of this list is the final
      dimensionality of the output graph representations.
     gated: set to True to do gated aggregation, False not to.
     aggregation_type: one of {sum, max, mean, sqrt_n}.
     name: name of this module.
    """
    super(GraphAggregator, self).__init__()

    self._node_hidden_sizes = node_hidden_sizes
    self._graph_transform_sizes = graph_transform_sizes
    self._graph_state_dim = node_hidden_sizes[-1]
    self._input_size = input_size
    #  The last element is the size of the aggregated graph representation.
    self._gated = gated
    self._aggregation_type = aggregation_type
    self._aggregation_op = None
    self.MLP1, self.MLP2 = self.build_model()

  def build_model(self):
    node_hidden_sizes = self._node_hidden_sizes
    if self._gated:
      node_hidden_sizes[-1] = self._graph_state_dim * 2

    layer = []
    layer.append(nn.Linear(self._input_size[0], node_hidden_sizes[0]))
    for i in range(1, len(node_hidden_sizes)):
      layer.append(nn.ReLU())
      layer.append(nn.Linear(node_hidden_sizes[i - 1], node_hidden_sizes[i]))
    MLP1 = nn.Sequential(*layer)
    if (self._graph_transform_sizes is not None and
        len(self._graph_transform_sizes) > 0):
      layer = []
      layer.append(nn.Linear(self._graph_state_dim, self._graph_transform_sizes[0]))
      for i in range(1, len(self._graph_transform_sizes)):
        layer.append(nn.ReLU())
        layer.append(nn.Linear(self._graph_transform_sizes[i - 1],
self._graph_transform_sizes[i]))
      MLP2 = nn.Sequential(*layer)

    return MLP1, MLP2

  def forward(self, node_states, graph_idx, n_graphs):
    """Compute aggregated graph representations.
    Args:
     node_states: [n_nodes, node_state_dim] float tensor, node states of a
      batch of graphs concatenated together along the first dimension.
     graph_idx: [n_nodes] int tensor, graph ID for each node.
     n_graphs: integer, number of graphs in this batch.
    Returns:
     graph_states: [n_graphs, graph_state_dim] float tensor, graph
      representations, one row for each graph.
    """

    node_states_g = self.MLP1(node_states)

    if self._gated:
      gates = torch.sigmoid(node_states_g[:, :self._graph_state_dim])
      node_states_g = node_states_g[:, self._graph_state_dim:] * gates

    graph_states = unsorted_segment_sum(node_states_g, graph_idx, n_graphs)

    if self._aggregation_type == 'max':
      # reset everything that's smaller than -1e5 to 0.
      graph_states *= torch.FloatTensor(graph_states > -1e5)
    # transform the reduced graph states further


    if (self._graph_transform_sizes is not None and
        len(self._graph_transform_sizes) > 0):
      graph_states = self.MLP2(graph_states)

    return graph_states

#### Putting them together

class GraphEmbeddingNet(nn.Module):
  """A graph to embedding mapping network."""

  def __init__(self,
         encoder,
         aggregator,
         node__state__dim,
         edge__state__dim,
         edge__hidden__sizes,
         node__hidden__sizes,
         n__prop__layers,
         share__prop__params=False,
         edge__net__init__scale=0.1,
         node__update__type='residual',
         use__reverse__direction=True,
         reverse__dir__param__different=True,
         layer__norm=False,
         layer__class=GraphPropLayer,
         prop__type='embedding',
         name='graph-embedding-net'):
    """Constructor.
    Args:
     encoder: GraphEncoder, encoder that maps features to embeddings.
     aggregator: GraphAggregator, aggregator that produces graph
      representations.
     node__state__dim: dimensionality of node states.
     edge__hidden__sizes: sizes of the hidden layers of the edge message nets.
     node__hidden__sizes: sizes of the hidden layers of the node update nets.
     n__prop__layers: number of graph propagation layers.
     share__prop__params: set to True to share propagation parameters across all
      graph propagation layers, False not to.
     edge__net__init__scale: scale of initialization for the edge message nets.
     node__update__type: type of node updates, one of {mlp, gru, residual}.
     use__reverse__direction: set to True to also propagate messages in the
      reverse direction.
     reverse__dir__param__different: set to True to have the messages computed
      using a different set of parameters than for the forward direction.
     layer__norm: set to True to use layer normalization in a few places.
     name: name of this module.
    """
    super(GraphEmbeddingNet, self).___init___()

    self._encoder = encoder
    self._aggregator = aggregator
    self._node__state__dim = node__state__dim
    self._edge__state__dim = edge__state__dim
    self._edge__hidden__sizes = edge__hidden__sizes
    self._node__hidden__sizes = node__hidden__sizes
    self._n__prop__layers = n__prop__layers
    self._share__prop__params = share__prop__params
    self._edge__net__init__scale = edge__net__init__scale
    self._node__update__type = node__update__type
    self._use__reverse__direction = use__reverse__direction
    self._reverse__dir__param__different = reverse__dir__param__different
    self._layer__norm = layer__norm
    self._prop__layers = []
    self._prop__layers = nn.ModuleList()
    self._layer__class = layer__class
    self._prop__type = prop__type
    self.build_model()

  def _build_layer(self, layer_id):
    """Build one layer in the network."""
    return self._layer_class(
      self._node_state_dim,
      self._edge_state_dim,
      self._edge_hidden_sizes,
      self._node_hidden_sizes,
      edge_net_init_scale=self._edge_net_init_scale,
      node_update_type=self._node_update_type,
      use_reverse_direction=self._use_reverse_direction,
      reverse_dir_param_different=self._reverse_dir_param_different,
      layer_norm=self._layer_norm,
      prop_type=self._prop_type)
    # name='graph-prop-%d' % layer_id)

  def _apply_layer(self,
                   layer,
                   node_states,
                   from_idx,
                   to_idx,
                   graph_idx,
                   n_graphs,
                   edge_features):
    """Apply one layer on the given inputs."""
    del graph_idx, n_graphs
    return layer(node_states, from_idx, to_idx, edge_features=edge_features)

  def build_model(self):
    if len(self._prop_layers) < self._n_prop_layers:
      # build the layers
      for i in range(self._n_prop_layers):
          if i == 0 or not self._share_prop_params:
              layer = self._build_layer(i)
          else:
              layer = self._prop_layers[0]
          self._prop_layers.append(layer)

  def forward(self,
              node_features,
              edge_features,
              from_idx,
              to_idx,
              graph_idx,
              n_graphs):
    """Compute graph representations.
    Args:
     node_features: [n_nodes, node_feat_dim] float tensor.
     edge_features: [n_edges, edge_feat_dim] float tensor.
     from_idx: [n_edges] int tensor, index of the from node for each edge.
     to_idx: [n_edges] int tensor, index of the to node for each edge.
     graph_idx: [n_nodes] int tensor, graph id for each node.
      n_graphs: int, number of graphs in the batch.
    Returns:
      graph_representations: [n_graphs, graph_representation_dim] float tensor,
        graph representations.
    """

    node_features, edge_features = self._encoder(node_features, edge_features)
    node_states = node_features

    layer_outputs = [node_states]

    for layer in self._prop_layers:
      # node_features could be wired in here as well, leaving it out for now as
      # it is already in the inputs
      node_states = self._apply_layer(
          layer,
          node_states,
          from_idx,
          to_idx,
          graph_idx,
          n_graphs,
          edge_features)
      layer_outputs.append(node_states)

    # these tensors may be used e.g. for visualization
    self._layer_outputs = layer_outputs
    return self._aggregator(node_states, graph_idx, n_graphs)

  def reset_n_prop_layers(self, n_prop_layers):
    """Set n_prop_layers to the provided new value.
    This allows us to train with certain number of propagation layers and
    evaluate with a different number of propagation layers.
    This only works if n_prop_layers is smaller than the number used for
    training, or when share_prop_params is set to True, in which case this can
    be arbitrarily large.
    Args:
      n_prop_layers: the new number of propagation layers to set.
    """
    self._n_prop_layers = n_prop_layers

  @property
  def n_prop_layers(self):
    return self._n_prop_layers

  def get_layer_outputs(self):
    """Get the outputs at each layer."""
    if hasattr(self, '_layer_outputs'):
      return self._layer_outputs
    else:
      raise ValueError('No layer outputs available.')
```

### Graph matching networks

#### A few similarity functions

$$d(G\_1, G\_2) = d\_H(embed\\_and\\_match(G\_1, G\_2))$$

1. $embed\\_and\\_match(G\_1, G\_2)$ =

1) mapping:

$$\begin{array}{rcl}
h\_i^{(0)} &=& \mathrm{MLP\_{node}}(x\_i) \\\\
e\_{ij} &=& \mathrm{MLP\_{edge}}(x\_{ij})
\end{array}
$$

2) passing cross-graph messages:
$$\begin{array}{rcl}
m\_{i\rightarrow j} &=& f\_\mathrm{message}(h\_i^{(t)}, h\_j^{(t)}, e\_{ij}) \\\\
\end{array}
$$
cross graph attention weight:
$$\begin{array}{rcl}
a\_{i\rightarrow j} &=& \frac{\exp(s(h\_i^{(t)}, h\_j^{(t)}))}{\sum\_j \exp(s(h\_i^{(t)}, h\_j^{(t)}))} ,
GAT: a\_{i\rightarrow j} &=& \frac{\exp(LeakyReLU([Wh\_i||Wh\_j]))}{\sum\_j
\exp(LeakyReLU([Wh\_i||Wh\_j]))}\\\\
a\_{j\rightarrow i} &=& \frac{\exp(s(h\_i^{(t)}, h\_j^{(t)}))}{\sum\_i \exp(s(h\_i^{(t)}, h\_j^{(t)}))}
\end{array}
$$
take the difference of attention-weighted sum of node representations:
$$\begin{array}{rcl}
\mu\_i &=& \sum\_j a\_{i\rightarrow j} (h\_i^{(t)} - h\_j^{(t)}) = h\_i^{(t)} - \sum\_j a\_{i\rightarrow
j} h\_j^{(t)}, GAT: \mu\_i &=& \sum\_j a\_{i\rightarrow j} Wh\_i^{(t)}\\\\
\mu\_j &=& \sum\_i a\_{j\rightarrow i} (h\_j^{(t)} - h\_i^{(t)}) = h\_j^{(t)} - \sum\_i a\_{j\rightarrow
i} h\_i^{(t)}.
\end{array}
$$
$$
h\_i^{(t+1)} = f\_\mathrm{node}\left(h\_i^{(t)}, \sum\_{j:(j,i)\in E} m\_{j\rightarrow i},
\mu\_i\right). GAT: h\_i^{(t+1)} = \sigma \left(\mu\_i\right)
$$
3) aggregate node representations to get graph representations

$$h\_G = \mathrm{MLP\_G}\left(\sum\_{i\in V} h\_i^{(T)}\right).$$
$$h\_G = \mathrm{MLP\_G}\left(\sum\_{i\in V} \sigma(\mathrm{MLP\_{gate}}(h\_i^{(T)})) \odot
\mathrm{MLP}(h\_i^{(T)})\right).$$

2. d_H is an existing distance metric\\:

Euclidean distance:

$d\_H(x, y) = \sqrt{\sum\_{i=1}^H (x\_i - y\_i)^2}$

or Hamming distance:

$d_H(x, y)=\sum_{i=1}^H \mathbb{I}[x_i \ne y_i]$

or dot product:

or cosine:

```
def pairwise_euclidean_similarity(x, y):
  """Compute the pairwise Euclidean similarity between x and y.
  This function computes the following similarity value between each pair of x_i
  and y_j: s(x_i, y_j) = -|x_i - y_j|^2.
  Args:
    x: NxD float tensor.
    y: MxD float tensor.
  Returns:
    s: NxM float tensor, the pairwise euclidean similarity.
  """
  s = 2 * torch.mm(x, torch.transpose(y, 1, 0))
  diag_x = torch.sum(x * x, dim=-1)
  diag_x = torch.unsqueeze(diag_x, 0)
  diag_y = torch.reshape(torch.sum(y * y, dim=-1), (1, -1))

  return s - diag_x - diag_y


def pairwise_dot_product_similarity(x, y):
  """Compute the dot product similarity between x and y.
  This function computes the following similarity value between each pair of x_i
  and y_j: s(x_i, y_j) = x_i^T y_j.
  Args:
    x: NxD float tensor.
    y: MxD float tensor.
  Returns:
    s: NxM float tensor, the pairwise dot product similarity.
  """
  return torch.mm(x, torch.transpose(y, 1, 0))


def pairwise_cosine_similarity(x, y):
  """Compute the cosine similarity between x and y.
  This function computes the following similarity value between each pair of x_i
  and y_j: s(x_i, y_j) = x_i^T y_j / (|x_i||y_j|).
  Args:
    x: NxD float tensor.
    y: MxD float tensor.
  Returns:
    s: NxM float tensor, the pairwise cosine similarity.
  """
  x = torch.div(x, torch.sqrt(torch.max(torch.sum(x ** 2), 1e-12)))
  y = torch.div(y, torch.sqrt(torch.max(torch.sum(y ** 2), 1e-12)))
  return torch.mm(x, torch.transpose(y, 1, 0))
```

```
PAIRWISE_SIMILARITY_FUNCTION = {
  'euclidean': pairwise_euclidean_similarity,
  'dotproduct': pairwise_dot_product_similarity,
  'cosine': pairwise_cosine_similarity,
}


def get_pairwise_similarity(name):
  """Get pairwise similarity metric by name.
  Args:
    name: string, name of the similarity metric, one of {dot-product, cosine,
      euclidean}.
  Returns:
    similarity: a (x, y) -> sim function.
  Raises:
    ValueError: if name is not supported.
  """
  if name not in PAIRWISE_SIMILARITY_FUNCTION:
      raise ValueError('Similarity metric name "%s" not supported.' % name)
  else:
      return PAIRWISE_SIMILARITY_FUNCTION[name]

#### Cross-graph attention

def compute_cross_attention(x, y, sim):
  """Compute cross attention.
  x_i attend to y_j:
  a_{i->j} = exp(sim(x_i, y_j)) / sum_j exp(sim(x_i, y_j))
  y_j attend to x_i:
  a_{j->i} = exp(sim(x_i, y_j)) / sum_i exp(sim(x_i, y_j))
  attention_x = sum_j a_{i->j} y_j
  attention_y = sum_i a_{j->i} x_i
  Args:
    x: NxD float tensor.
    y: MxD float tensor.
    sim: a (x, y) -> similarity function.
  Returns:
    attention_x: NxD float tensor.
    attention_y: NxD float tensor.
  """
  a = sim(x, y)
  a_x = torch.softmax(a, dim=1)  # i->j
  a_y = torch.softmax(a, dim=0)  # j->i
  attention_x = torch.mm(a_x, y)
  attention_y = torch.mm(torch.transpose(a_y, 1, 0), x)
  return attention_x, attention_y




def batch_block_pair_attention(data,
                    block_idx,
                    n_blocks,
                    similarity='dotproduct'):
  """Compute batched attention between pairs of blocks.
  This function partitions the batch data into blocks according to block_idx.
  For each pair of blocks, x = data[block_idx == 2i], and
  y = data[block_idx == 2i+1], we compute
  x_i attend to y_j:
  a_{i->j} = exp(sim(x_i, y_j)) / sum_j exp(sim(x_i, y_j))
  y_j attend to x_i:
  a_{j->i} = exp(sim(x_i, y_j)) / sum_i exp(sim(x_i, y_j))
  and
  attention_x = sum_j a_{i->j} y_j
  attention_y = sum_i a_{j->i} x_i.
  Args:
   data: NxD float tensor.
   block_idx: N-dim int tensor.
   n_blocks: integer.
   similarity: a string, the similarity metric.
  Returns:
   attention_output: NxD float tensor, each x_i replaced by attention_x_i.
  Raises:
   ValueError: if n_blocks is not an integer or not a multiple of 2.
  """
  if not isinstance(n_blocks, int):
    raise ValueError('n_blocks (%s) has to be an integer.' % str(n_blocks))

  if n_blocks % 2 != 0:
    raise ValueError('n_blocks (%d) must be a multiple of 2.' % n_blocks)

  sim = get_pairwise_similarity(similarity)

  results = []

  # This is probably better than doing boolean_mask for each i
  partitions = []
  for i in range(n_blocks):
    partitions.append(data[block_idx == i, :])

  for i in range(0, n_blocks, 2):
    x = partitions[i]
    y = partitions[i + 1]
    attention_x, attention_y = compute_cross_attention(x, y, sim)
    results.append(attention_x)
    results.append(attention_y)
  results = torch.cat(results, dim=0)

  return results

#### Graph matching layer and graph matching networks

class GraphPropMatchingLayer(GraphPropLayer):
  """A graph propagation layer that also does cross graph matching.
  It assumes the incoming graph data is batched and paired, i.e. graph 0 and 1
  forms the first pair and graph 2 and 3 are the second pair etc., and computes
  cross-graph attention-based matching for each pair.
  """

  def forward(self,
              node_states,
              from_idx,
              to_idx,
              graph_idx,
              n_graphs,
              similarity='dotproduct',
              edge_features=None,
              node_features=None):
    """Run one propagation step with cross-graph matching.
    Args:
      node_states: [n_nodes, node_state_dim] float tensor, node states.
      from_idx: [n_edges] int tensor, from node indices for each edge.
      to_idx: [n_edges] int tensor, to node indices for each edge.
      graph_idx: [n_onodes] int tensor, graph id for each node.
      n_graphs: integer, number of graphs in the batch.
      similarity: type of similarity to use for the cross graph attention.
      edge_features: if not None, should be [n_edges, edge_feat_dim] tensor,
        extra edge features.
      node_features: if not None, should be [n_nodes, node_feat_dim] tensor,
        extra node features.
    Returns:
      node_states: [n_nodes, node_state_dim] float tensor, new node states.
    Raises:
      ValueError: if some options are not provided correctly.
    """
    aggregated_messages = self._compute_aggregated_messages(
        node_states, from_idx, to_idx, edge_features=edge_features)

    cross_graph_attention = batch_block_pair_attention(
        node_states, graph_idx, n_graphs, similarity=similarity)
    attention_input = node_states - cross_graph_attention

    return self._compute_node_update(node_states,
                                     [aggregated_messages, attention_input],
                                     node_features=node_features)


class GraphMatchingNet(GraphEmbeddingNet):
  """Graph matching net.
  This class uses graph matching layers instead of the simple graph prop layers.
  It assumes the incoming graph data is batched and paired, i.e. graph 0 and 1
  forms the first pair and graph 2 and 3 are the second pair etc., and computes
  cross-graph attention-based matching for each pair.
  """
  def __init__(self,
               encoder,
               aggregator,
               node_state_dim,
               edge_state_dim,
               edge_hidden_sizes,
               node_hidden_sizes,
               n_prop_layers,
               share_prop_params=False,
               edge_net_init_scale=0.1,
               node_update_type='residual',
               use_reverse_direction=True,
               reverse_dir_param_different=True,
               layer_norm=False,
               layer_class=GraphPropLayer,
               similarity='dotproduct',
               prop_type='embedding'):
    super(GraphMatchingNet, self).__init__(
        encoder,
        aggregator,
        node_state_dim,
        edge_state_dim,
        edge_hidden_sizes,
        node_hidden_sizes,
        n_prop_layers,
        share_prop_params=share_prop_params,
        edge_net_init_scale=edge_net_init_scale,
        node_update_type=node_update_type,
        use_reverse_direction=use_reverse_direction,
        reverse_dir_param_different=reverse_dir_param_different,
        layer_norm=layer_norm,
        layer_class=GraphPropMatchingLayer,
        prop_type=prop_type,
    )
    self._similarity = similarity

  def _apply_layer(self,
                   layer,
                   node_states,
                   from_idx,
                   to_idx,
                   graph_idx,
                   n_graphs,
                   edge_features):
    """Apply one layer on the given inputs."""
    return layer(node_states, from_idx, to_idx, graph_idx, n_graphs,
                 similarity=self._similarity, edge_features=edge_features)
```

## Training

### Training on pairs (loss)

Euclidean distance: $d_H(x, y) = \sqrt{\sum_{i=1}^H (x_i - y_i)^2}$, or Hamming distance: $d_H(x, y)=\sum_{i=1}^H \mathbb{I}[x_i \ne y_i]$

```
def euclidean_distance(x, y):
  """This is the squared Euclidean distance."""
  return torch.sum((x - y) ** 2, dim=-1)


def approximate_hamming_similarity(x, y):
  """Approximate Hamming similarity."""
  return torch.mean(torch.tanh(x) * torch.tanh(y), dim=1)


def pairwise_loss(x, y, labels, loss_type='margin', margin=1.0):
  """Compute pairwise loss.
  Args:
    x: [N, D] float tensor, representations for N examples.
    y: [N, D] float tensor, representations for another N examples.
    labels: [N] int tensor, with values in -1 or +1.  labels[i] = +1 if x[i]
      and y[i] are similar, and -1 otherwise.
    loss_type: margin or hamming.
    margin: float scalar, margin for the margin loss.
  Returns:
    loss: [N] float tensor.  Loss for each pair of representations.
  """

  labels = labels.float()

  if loss_type == 'margin':
    return torch.relu(margin - labels * (1 - euclidean_distance(x, y)))
  elif loss_type == 'hamming':
    return 0.25 * (labels - approximate_hamming_similarity(x, y)) ** 2
  else:
    raise ValueError('Unknown loss_type %s' % loss_type)
```

### Training on triplets (loss)

margin, hamming

```
def triplet_loss(x_1, y, x_2, z, loss_type='margin', margin=1.0):
  """Compute triplet loss.
  This function computes loss on a triplet of inputs (x, y, z).  A similarity or
  distance value is computed for each pair of (x, y) and (x, z).  Since the
  representations for x can be different in the two pairs (like our matching
  model) we distinguish the two x representations by x_1 and x_2.
  Args:
    x_1: [N, D] float tensor.
    y: [N, D] float tensor.
    x_2: [N, D] float tensor.
    z: [N, D] float tensor.
    loss_type: margin or hamming.
    margin: float scalar, margin for the margin loss.
  Returns:
    loss: [N] float tensor.  Loss for each pair of representations.
  """
  if loss_type == 'margin':
      return torch.relu(margin +
                        euclidean_distance(x_1, y) -
                        euclidean_distance(x_2, z))
  elif loss_type == 'hamming':
      return 0.125 * ((approximate_hamming_similarity(x_1, y) - 1) ** 2 +
                      (approximate_hamming_similarity(x_2, z) + 1) ** 2)
  else:
      raise ValueError('Unknown loss_type %s' % loss_type)

## Datasets


import abc
import collections

"""A general Interface"""

GraphData = collections.namedtuple('GraphData', [
  'from_idx',
  'to_idx',
  'node_features',
  'edge_features',
  'graph_idx',
  'n_graphs'])

class GraphSimilarityDataset(object):
  """Base class for all the graph similarity learning datasets.
 This class defines some common interfaces a graph similarity dataset can have,
 in particular the functions that creates iterators over pairs and triplets.
 """

  @abc.abstractmethod
  def triplets(self, batch_size):
      """Create an iterator over triplets.
  Args:
    batch_size: int, number of triplets in a batch.
  Yields:
    graphs: a `GraphData` instance.  The batch of triplets put together.  Each
      triplet has 3 graphs (x, y, z).  Here the first graph is duplicated once
      so the graphs for each triplet are ordered as (x, y, x, z) in the batch.
      The batch contains `batch_size` number of triplets, hence `4*batch_size`
      many graphs.
  """
      pass

  @abc.abstractmethod
  def pairs(self, batch_size):
      """Create an iterator over pairs.
  Args:
    batch_size: int, number of pairs in a batch.
  Yields:
    graphs: a `GraphData` instance.  The batch of pairs put together.  Each
      pair has 2 graphs (x, y).  The batch contains `batch_size` number of
      pairs, hence `2*batch_size` many graphs.
    labels: [batch_size] int labels for each pair, +1 for similar, -1 for not.
  """
    pass


## Graph edit distance task

### Graph manipulation primitives


import networkx as nx

"""Graph Edit Distance Task"""


# Graph Manipulation Functions
def permute_graph_nodes(g):
  """Permute node ordering of a graph, returns a new graph."""
  n = g.number_of_nodes()
  new_g = nx.Graph()
  new_g.add_nodes_from(range(n))
  perm = np.random.permutation(n)
  edges = g.edges()
  new_edges = []
  for x, y in edges:
    new_edges.append((perm[x], perm[y]))
  new_g.add_edges_from(new_edges)
  return new_g


def substitute_random_edges(g, n):
  """Substitutes n edges from graph g with another n randomly picked edges."""
  g = copy.deepcopy(g)
  n_nodes = g.number_of_nodes()
  edges = list(g.edges())
  # sample n edges without replacement
  e_remove = [
    edges[i] for i in np.random.choice(np.arange(len(edges)), n, replace=False)
  ]
  edge_set = set(edges)
  e_add = set()
  while len(e_add) < n:
    e = np.random.choice(n_nodes, 2, replace=False)
    # make sure e does not exist and is not already chosen to be added
    if (
        (e[0], e[1]) not in edge_set
        and (e[1], e[0]) not in edge_set
        and (e[0], e[1]) not in e_add
```

```
            and (e[1], e[0]) not in e_add
    ):
        e_add.add((e[0], e[1]))

  for i, j in e_remove:
    g.remove_edge(i, j)
  for i, j in e_add:
    g.add_edge(i, j)
  return g

### Dataset for training, fixed dataset for evaluation


import contextlib
import networkx as nx
import numpy as np


class GraphEditDistanceDataset(GraphSimilarityDataset):
  """Graph edit distance dataset."""

  def __init__(
        self,
        n_nodes_range,
        p_edge_range,
        n_changes_positive,
        n_changes_negative,
        permute=True,
  ):
    """Constructor.
  Args:
   n_nodes_range: a tuple (n_min, n_max).  The minimum and maximum number of
    nodes in a graph to generate.
   p_edge_range: a tuple (p_min, p_max).  The minimum and maximum edge
    probability.
   n_changes_positive: the number of edge substitutions for a pair to be
    considered positive (similar).
   n_changes_negative: the number of edge substitutions for a pair to be
    considered negative (not similar).
   permute: if True (default), permute node orderings in addition to
    changing edges; if False, the node orderings across a pair or triplet of
    graphs will be the same, useful for visualization.
  """
    self._n_min, self._n_max = n_nodes_range
    self._p_min, self._p_max = p_edge_range
    self._k_pos = n_changes_positive
    self._k_neg = n_changes_negative
    self._permute = permute

  def _get_graph(self):
    """Generate one graph."""
    n_nodes = np.random.randint(self._n_min, self._n_max + 1)
    p_edge = np.random.uniform(self._p_min, self._p_max)
    # do a little bit of filtering
    n_trials = 100
    for _ in range(n_trials):
      g = nx.erdos_renyi_graph(n_nodes, p_edge)
      if nx.is_connected(g):
        return g

    raise ValueError("Failed to generate a connected graph.")

  def _get_pair(self, positive):
    """Generate one pair of graphs."""
    g = self._get_graph()
    if self._permute:
      permuted_g = permute_graph_nodes(g)
    else:
      permuted_g = g
    n_changes = self._k_pos if positive else self._k_neg
    changed_g = substitute_random_edges(g, n_changes)
    return permuted_g, changed_g

  def _get_triplet(self):
    """Generate one triplet of graphs."""
    g = self._get_graph()
    if self._permute:
      permuted_g = permute_graph_nodes(g)
    else:
      permuted_g = g
    pos_g = substitute_random_edges(g, self._k_pos)
    neg_g = substitute_random_edges(g, self._k_neg)
    return permuted_g, pos_g, neg_g

  def triplets(self, batch_size):
    """Yields batches of triplet data."""
    while True:
      batch_graphs = []
      for _ in range(batch_size):
        g1, g2, g3 = self._get_triplet()
        batch_graphs.append((g1, g2, g1, g3))
      yield self._pack_batch(batch_graphs)

  def pairs(self, batch_size):
    """Yields batches of pair data."""
    while True:
      batch_graphs = []
      batch_labels = []
      positive = True
      for _ in range(batch_size):
        g1, g2 = self._get_pair(positive)
        batch_graphs.append((g1, g2))
        batch_labels.append(1 if positive else -1)
        positive = not positive
      packed_graphs = self._pack_batch(batch_graphs)
      labels = np.array(batch_labels, dtype=np.int32)
      yield packed_graphs, labels

  def _pack_batch(self, graphs):
    """Pack a batch of graphs into a single `GraphData` instance.
  Args:
   graphs: a list of generated networkx graphs.
  Returns:
   graph_data: a `GraphData` instance, with node and edge indices properly
    shifted.
  """
    Graphs = []
    for graph in graphs:
      for inergraph in graph:
        Graphs.append(inergraph)
    graphs = Graphs
    from_idx = []
    to_idx = []
    graph_idx = []

    n_total_nodes = 0
    n_total_edges = 0
    for i, g in enumerate(graphs):
      n_nodes = g.number_of_nodes()
      n_edges = g.number_of_edges()
      edges = np.array(g.edges(), dtype=np.int32)
      # shift the node indices for the edges
      from_idx.append(edges[:, 0] + n_total_nodes)
      to_idx.append(edges[:, 1] + n_total_nodes)
      graph_idx.append(np.ones(n_nodes, dtype=np.int32) * i)

      n_total_nodes += n_nodes
      n_total_edges += n_edges

    GraphData = collections.namedtuple('GraphData', [
      'from_idx',
      'to_idx',
      'node_features',
      'edge_features',
      'graph_idx',
      'n_graphs'])

    return GraphData(
      from_idx=np.concatenate(from_idx, axis=0),
      to_idx=np.concatenate(to_idx, axis=0),
      # this task only cares about the structures, the graphs have no features.
      # setting higher dimension of ones to confirm code functioning
      # with high dimensional features.
      node_features=np.ones((n_total_nodes, 8), dtype=np.float32),
      edge_features=np.ones((n_total_edges, 4), dtype=np.float32),
      graph_idx=np.concatenate(graph_idx, axis=0),
      n_graphs=len(graphs),
    )




# Use Fixed datasets for evaluation
@contextlib.contextmanager
def reset_random_state(seed):
    """This function creates a context that uses the given seed."""
    np_rnd_state = np.random.get_state()
    rnd_state = random.getstate()
    np.random.seed(seed)
    random.seed(seed + 1)
    try:
        yield
    finally:
        random.setstate(rnd_state)
        np.random.set_state(np_rnd_state)




class FixedGraphEditDistanceDataset(GraphEditDistanceDataset):
    """A fixed dataset of pairs or triplets for the graph edit distance task.
  This dataset can be used for evaluation.
  """

    def __init__(
            self,
            n_nodes_range,
            p_edge_range,
            n_changes_positive,
            n_changes_negative,
            dataset_size,
            permute=True,
            seed=1234,
    ):
        super(FixedGraphEditDistanceDataset, self).__init__(
            n_nodes_range,
            p_edge_range,
            n_changes_positive,
            n_changes_negative,
            permute=permute,
        )
        self._dataset_size = dataset_size
        self._seed = seed

    def triplets(self, batch_size):
        """Yield triplets."""

        if hasattr(self, "_triplets"):
            triplets = self._triplets
        else:
```

```
      # get a fixed set of triplets
      with reset_random_state(self._seed):
        triplets = []
        for _ in range(self._dataset_size):
          g1, g2, g3 = self._get_triplet()
          triplets.append((g1, g2, g1, g3))
      self._triplets = triplets

    ptr = 0
    while ptr + batch_size <= len(triplets):
      batch_graphs = triplets[ptr: ptr + batch_size]
      yield self._pack_batch(batch_graphs)
      ptr += batch_size

  def pairs(self, batch_size):
    """Yield pairs and labels."""

    if hasattr(self, "_pairs") and hasattr(self, "_labels"):
      pairs = self._pairs
      labels = self._labels
    else:
      # get a fixed set of pairs first
      with reset_random_state(self._seed):
        pairs = []
        labels = []
        positive = True
        for _ in range(self._dataset_size):
          pairs.append(self._get_pair(positive))
          labels.append(1 if positive else -1)
          positive = not positive
      labels = np.array(labels, dtype=np.int32)

      self._pairs = pairs
      self._labels = labels

    ptr = 0
    while ptr + batch_size <= len(pairs):
      batch_graphs = pairs[ptr: ptr + batch_size]
      packed_batch = self._pack_batch(batch_graphs)
      yield packed_batch, labels[ptr: ptr + batch_size]
      ptr += batch_size

###QM7b

!pip install dgl -q
from dgl.data import QM7bDataset
qm = QM7bDataset()


class QM7b():
 #       1713
 def pairs(self,batch_size):#namedtuple
  while True:
   batch_graphs = []
   batch_labels = []
   positive = True
   for i in range(batch_size):#20
    g1,_ = qm[i]
    g2 = g1.clone()
    if positive:#True
     remove_idx = np.random.choice(np.arange(g2.num_edges()),1,replace=False)
    else:#False
     remove_idx = np.random.choice(np.arange(g2.num_edges()),2,replace=False)
    g2.remove_edges(remove_idx)
    batch_graphs.append((g1, g2))#list
    batch_labels.append(1 if positive else -1)#
    positive = not positive#
   labels = np.array(batch_labels, dtype=np.int32)
   from_idx = []
   to_idx = []
   graph_idx = []
   graphs = []
   num_nodes = 0
   num_edges = 0
   for tuples in batch_graphs:#
    for pair in tuples:#
     graphs.append(pair)#
   batch_graphs = graphs
   for i,g in enumerate(batch_graphs):#
    edge_idx = torch.arange(0,g.num_edges())
    src, dst = g.find_edges(edge_idx)
    src = np.array(src, dtype=np.int32)
    dst = np.array(dst, dtype=np.int32)
    from_idx.append(src)
    to_idx.append(dst)
    num_nodes += g.num_nodes()
    num_edges += g.num_edges()
    graph_idx.append(np.ones(g.num_nodes(), dtype=np.int32) * i)
   GraphData = collections.namedtuple('GraphData',
['from_idx','to_idx','node_features','edge_features', 'graph_idx','n_graphs'])
   packed_graphs = GraphData(
       from_idx=np.concatenate(from_idx, axis=0),
       to_idx=np.concatenate(to_idx, axis=0),
       node_features=np.ones((num_nodes, 8), dtype=np.float32),
       edge_features=np.ones((num_edges, 4), dtype=np.float32),
       graph_idx=np.concatenate(graph_idx, axis=0),
       n_graphs=len(batch_graphs),
   )
   yield packed_graphs, labels

class QM7bVali():
 def pairs(self,batch_size):
  batch_graphs = []
  batch_labels = []
  positive = True
  for i in range(1000):#
```

```
    g1,__ = qm[i+20]
    g2 = g1.clone()
    if positive:#True
      remove__idx =  np.random.choice(np.arange(g2.num__edges()),1,replace=False)
    else:#False
      remove__idx = np.random.choice(np.arange(g2.num__edges()),5,replace=False)
    g2.remove__edges(remove__idx)
    batch__graphs.append((g1, g2))#list
    batch__labels.append(1 if positive else -1)#list
    positive = not positive#
  labels = np.array(batch__labels, dtype=np.int32)

  ptr = 0
  while ptr + batch__size <= len(batch__graphs):
    batch__graphs = batch__graphs[ptr: ptr + batch__size]
    from__idx = []
    to__idx = []
    graph__idx = []
    graphs = []
    num__nodes = 0
    num__edges = 0
    for tuples in batch__graphs:#
      for pair in tuples:#
        graphs.append(pair)#list
    for i,g in enumerate(graphs):#
      edge__idx = torch.arange(0,g.num__edges())
      src, dst = g.find__edges(edge__idx)
      src = np.array(src, dtype=np.int32)
      dst = np.array(dst, dtype=np.int32)
      from__idx.append(src)
      to__idx.append(dst)
      num__nodes += g.num__nodes()
      num__edges += g.num__edges()
      graph__idx.append(np.ones(g.num__nodes(), dtype=np.int32) * i)
    GraphData = collections.namedtuple('GraphData',
['from__idx','to__idx','node__features','edge__features', 'graph__idx','n__graphs'])
    packed__graphs = GraphData(
        from__idx=np.concatenate(from__idx, axis=0),
        to__idx=np.concatenate(to__idx, axis=0),
        node__features=np.ones((num__nodes, 8), dtype=np.float32),
        edge__features=np.ones((num__edges, 4), dtype=np.float32),
        graph__idx=np.concatenate(graph__idx, axis=0),
        n__graphs=len(graphs),
    )
    yield packed__graphs, labels[ptr: ptr + batch__size]
    ptr += batch__size

 def triplets(self, batch__size):

  triplets = []
  for i in range(1000):
    g1,__ = qm[i+20]
    g2 = g1.clone()
    g3 = g1.clone()
    remove_idx =  np.random.choice(np.arange(g2.num_edges()),1,replace=False)
    g2.remove_edges(remove_idx)
    remove_idx =  np.random.choice(np.arange(g2.num_edges()),5,replace=False)
    g3.remove_edges(remove_idx)
    triplets.append((g1, g2, g1, g3))

  ptr = 0
  while ptr + batch_size <= len(triplets):
   batch_graphs = triplets[ptr: ptr + batch_size]

   from_idx = []
   to_idx = []
   graph_idx = []
   graphs = []
   num_nodes = 0
   num_edges = 0
   for tuples in batch_graphs:#
    for pair in tuples:#
     graphs.append(pair)#list
   for i,g in enumerate(graphs):#
    edge_idx = torch.arange(0,g.num_edges())
    src, dst = g.find_edges(edge_idx)
    src = np.array(src, dtype=np.int32)
    dst = np.array(dst, dtype=np.int32)
    from_idx.append(src)
    to_idx.append(dst)
    num_nodes += g.num_nodes()
    num_edges += g.num_edges()
    graph_idx.append(np.ones(g.num_nodes(), dtype=np.int32) * i)
   GraphData = collections.namedtuple('GraphData',
['from_idx','to_idx','node_features','edge_features', 'graph_idx','n_graphs'])
   packed_graphs = GraphData(
       from_idx=np.concatenate(from_idx, axis=0),
       to_idx=np.concatenate(to_idx, axis=0),
       node_features=np.ones((num_nodes, 8), dtype=np.float32),
       edge_features=np.ones((num_edges, 4), dtype=np.float32),
       graph_idx=np.concatenate(graph_idx, axis=0),
       n_graphs=len(graphs),
   )
   yield packed_graphs
   ptr += batch_size
```

## Building the model, and the training and evaluation pipelines

### Configs

```
def get_default_config():
  """The default configs."""
  model_type = 'matching'  # `embedding
  # Set to `embedding` to use the graph embedding net.
  node_state_dim = 32
  edge_state_dim = 16
  graph_rep_dim = 128
  graph_embedding_net_config = dict(
    node_state_dim=node_state_dim,
    edge_state_dim=edge_state_dim,
    edge_hidden_sizes=[node_state_dim * 2, node_state_dim * 2],
    node_hidden_sizes=[node_state_dim * 2],
    n_prop_layers=5,
    # set to False to not share parameters across message passing layers
    share_prop_params=True,  # 判断在信息传递层是否参数共享
    # initialize message MLP with small parameter weights to prevent
    # aggregated message vectors blowing up, alternatively we could also use
    # e.g. layer normalization to keep the scale of these under control.
    edge_net_init_scale=0.1,#                    #
    # other types of update like `mlp` and `residual` can also be used here. gru
    node_update_type='gru', # 'mlp'  `residual`
    # set to False if your graph already contains edges in both directions.
    use_reverse_direction=True,    #
    # set to True if your graph is directed
    reverse_dir_param_different=False,  #
    # we didn't use layer norm in our experiments but sometimes this can help.
    layer_norm=False,#
    # set to `embedding` to use the graph embedding net.
    prop_type=model_type)
  graph_matching_net_config = graph_embedding_net_config.copy()
  graph_matching_net_config['similarity'] = 'dotproduct'  # other: euclidean, cosine
  return dict(
    encoder=dict(
      node_hidden_sizes=[node_state_dim],
      node_feature_dim=1,
      edge_hidden_sizes=[edge_state_dim]),
    aggregator=dict(
      node_hidden_sizes=[graph_rep_dim],
      graph_transform_sizes=[graph_rep_dim],
      input_size=[node_state_dim],
      gated=True,
      aggregation_type='sum'),
    graph_embedding_net=graph_embedding_net_config,
    graph_matching_net=graph_matching_net_config,
    model_type=model_type,
    data=dict(
      problem='graph_edit_distance',
      dataset_params=dict(
        # always generate graphs with 20 nodes and p_edge=0.2.
        n_nodes_range=[20, 20],
        p_edge_range=[0.2, 0.2],#            n_changes_positive=1,  #
n_changes_negative=2,
        validation_dataset_size=1000)),
    training=dict(
      batch_size=20,
      learning_rate=1e-4,
      mode='pair',
      loss='margin',  # other: hamming
      margin=1.0,
      # A small regularizer on the graph vector scales to avoid the graph
      # vectors blowing up.  If numerical issues is particularly bad in the
      # model we can add `snt.LayerNorm` to the outputs of each layer, the
      # aggregated messages and aggregated node representations to
      # keep the network activation scale in a reasonable range.
      graph_vec_regularizer_weight=1e-6,        #                              #
      # Add gradient clipping to avoid large gradients.
      clip_value=10.0,   #
      # '''

      # '''
      # Increase this to train longer.
      n_training_steps=500000,   #
      # Print training information every this many training steps.
      print_after=100,   #
      # Evaluate on validation set every `eval_after * print_after` steps.
      eval_after=10),   #
    evaluation=dict(
      batch_size=20),
    seed=8,
  )
```

### Evaluation

```
from sklearn import metrics

def exact_hamming_similarity(x, y):
  """Compute the binary Hamming similarity."""
  match = ((x > 0) * (y > 0)).float()
  return torch.mean(match, dim=1)


def compute_similarity(config, x, y):
  """Compute the distance between x and y vectors.
  The distance will be computed based on the training loss type.
  Args:
   config: a config dict.
   x: [n_examples, feature_dim] float tensor.
   y: [n_examples, feature_dim] float tensor.
  Returns:
   dist: [n_examples] float tensor.
  Raises:
   ValueError: if loss type is not supported.
  """
  if config['training']['loss'] == 'margin':
    # similarity is negative distance
    return -euclidean_distance(x, y)
  elif config['training']['loss'] == 'hamming':
    return exact_hamming_similarity(x, y)
  else:
    raise ValueError('Unknown loss type %s' % config['training']['loss'])


def auc(scores, labels, **auc_args):
  """Compute the AUC for pair classification.
  See `tf.metrics.auc` for more details about this metric.
  Args:
    scores: [n_examples] float.  Higher scores mean higher preference of being
      assigned the label of +1.
    labels: [n_examples] int.  Labels are either +1 or -1.
    **auc_args: other arguments that can be used by `tf.metrics.auc`.
  Returns:
    auc: the area under the ROC curve.
  """
  scores_max = torch.max(scores)
  scores_min = torch.min(scores)

  # normalize scores to [0, 1] and add a small epislon for safety
  scores = (scores - scores_min) / (scores_max - scores_min + 1e-8)

  labels = (labels + 1) / 2

  fpr, tpr, thresholds = metrics.roc_curve(labels.cpu().detach().numpy(),
scores.cpu().detach().numpy())
  return metrics.auc(fpr, tpr)

### Build the model

def reshape_and_split_tensor(tensor, n_splits):
  """Reshape and split a 2D tensor along the last dimension.
  Args:
    tensor: a [num_examples, feature_dim] tensor.  num_examples must be a
      multiple of `n_splits`.
    n_splits: int, number of splits to split the tensor into.
  Returns:
    splits: a list of `n_splits` tensors.  The first split is [tensor[0],
      tensor[n_splits], tensor[n_splits * 2], ...], the second split is
      [tensor[1], tensor[n_splits + 1], tensor[n_splits * 2 + 1], ...], etc..
  """
  feature_dim = tensor.shape[-1]
  tensor = torch.reshape(tensor, [-1, feature_dim * n_splits])
  tensor_split = []
  for i in range(n_splits):
    tensor_split.append(tensor[:, feature_dim * i: feature_dim * (i + 1)])
  return tensor_split


def build_model(config, node_feature_dim, edge_feature_dim):
  """Create model for training and evaluation.
  Args:
    config: a dictionary of configs, like the one created by the
      `get_default_config` function.
    node_feature_dim: int, dimensionality of node features.
    edge_feature_dim: int, dimensionality of edge features.
  Returns:
    tensors: a (potentially nested) name => tensor dict.
    placeholders: a (potentially nested) name => tensor dict.
    AE_model: a GraphEmbeddingNet or GraphMatchingNet instance.
  Raises:
    ValueError: if the specified model or training settings are not supported.
  """
  config['encoder']['node_feature_dim'] = node_feature_dim
  config['encoder']['edge_feature_dim'] = edge_feature_dim

  encoder = GraphEncoder(**config['encoder'])
  aggregator = GraphAggregator(**config['aggregator'])
  if config['model_type'] == 'embedding':
      model = GraphEmbeddingNet(
          encoder, aggregator, **config['graph_embedding_net'])
  elif config['model_type'] == 'matching':
      model = GraphMatchingNet(
          encoder, aggregator, **config['graph_matching_net'])
  else:
      raise ValueError('Unknown model type: %s' % config['model_type'])

  optimizer = torch.optim.Adam((model.parameters()),
                               lr=config['training']['learning_rate'], weight_decay=1e-5)

  return model, optimizer




### build the dataset

import copy

def build_datasets(config):
  """Build the training and evaluation datasets."""
  config = copy.deepcopy(config)

  if config['data']['problem'] == 'graph_edit_distance':
      dataset_params = config['data']['dataset_params']
      validation_dataset_size = dataset_params['validation_dataset_size']
      del dataset_params['validation_dataset_size']
      training_set = GraphEditDistanceDataset(**dataset_params)
      dataset_params['dataset_size'] = validation_dataset_size
      validation_set = FixedGraphEditDistanceDataset(**dataset_params)
  elif config['data']['problem'] == 'QM7b':
    training_set = QM7b()
    validation_set = QM7bVali()
  else:
      raise ValueError('Unknown problem type: %s' % config['data']['problem'])
  return training_set, validation_set
```

```
def get_graph(batch):
  if len(batch) != 2:
    # if isinstance(batch, GraphData):
    graph = batch
    node_features = torch.from_numpy(graph.node_features)
    edge_features = torch.from_numpy(graph.edge_features)
    from_idx = torch.from_numpy(graph.from_idx).long()
    to_idx = torch.from_numpy(graph.to_idx).long()
    graph_idx = torch.from_numpy(graph.graph_idx).long()
    return node_features, edge_features, from_idx, to_idx, graph_idx
  else:
    graph, labels = batch
    node_features = torch.from_numpy(graph.node_features)
    edge_features = torch.from_numpy(graph.edge_features)
    from_idx = torch.from_numpy(graph.from_idx).long()
    to_idx = torch.from_numpy(graph.to_idx).long()
    graph_idx = torch.from_numpy(graph.graph_idx).long()
    labels = torch.from_numpy(labels).long()
  return node_features, edge_features, from_idx, to_idx, graph_idx, labels

### Let's run it!

import random
import time
import torch
import numpy as np

# Set GPU
#os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
#os.environ["CUDA_VISIBLE_DEVICES"] = "1"
use_cuda = torch.cuda.is_available()
device = torch.device('cuda:0' if use_cuda else 'cpu')
#device = torch.device('cpu')


# Print configure
config = get_default_config()
for (k, v) in config.items():
  print("%s= %s" % (k, v))

# Set random seed
seed = config['seed']
random.seed(seed)
np.random.seed(seed + 1)
torch.manual_seed(seed + 2)
torch.backends.cudnn.deterministic = False
torch.backends.cudnn.benchmark = True


config['data']['problem'] = 'graph_edit_distance'
# config['data']['problem'] = 'QM7b'
config['model_type'] = 'matching'
# config['model_type'] = 'embedding'
```

```
training_set, validation_set = build_datasets(config)

if config['training']['mode'] == 'pair':
 #默认的是pair，默认的batch_size为20
  training_data_iter = training_set.pairs(config['training']['batch_size'])


  first_batch_graphs, _ = next(training_data_iter)
else:
  training_data_iter = training_set.triplets(config['training']['batch_size'])
  first_batch_graphs = next(training_data_iter)

node_feature_dim = first_batch_graphs.node_features.shape[-1]
edge_feature_dim = first_batch_graphs.edge_features.shape[-1]




model, optimizer = build_model(config, node_feature_dim, edge_feature_dim)
model.to(device)

accumulated_metrics = collections.defaultdict(list)


training_n_graphs_in_batch = config['training']['batch_size']
if config['training']['mode'] == 'pair':
  training_n_graphs_in_batch *= 2
elif config['training']['mode'] == 'triplet':
  training_n_graphs_in_batch *= 4
else:
  raise ValueError('Unknown training mode: %s' % config['training']['mode'])

t_start = time.time()
for i_iter in range(config['training']['n_training_steps']):
  model.train(mode=True)
  batch = next(training_data_iter)
  if config['training']['mode'] == 'pair':
    node_features, edge_features, from_idx, to_idx, graph_idx, labels = get_graph(batch)

    labels = labels.to(device)
  else:

    node_features, edge_features, from_idx, to_idx, graph_idx = get_graph(batch)

  graph_vectors = model(node_features.to(device), edge_features.to(device), from_idx.to(device),
to_idx.to(device),graph_idx.to(device), training_n_graphs_in_batch)

  if config['training']['mode'] == 'pair':
    x, y = reshape_and_split_tensor(graph_vectors, 2)


    loss = pairwise_loss(x, y, labels,
                     loss_type=config['training']['loss'],
                     margin=config['training']['margin'])

    is_pos = (labels == torch.ones(labels.shape).long().to(device)).float()
    is_neg = 1 - is_pos
    n_pos = torch.sum(is_pos)
    n_neg = torch.sum(is_neg)



    sim = compute_similarity(config, x, y)
    sim_pos = torch.sum(sim * is_pos) / (n_pos + 1e-8)
    sim_neg = torch.sum(sim * is_neg) / (n_neg + 1e-8)
  else:
    x_1, y, x_2, z = reshape_and_split_tensor(graph_vectors, 4)
    loss = triplet_loss(x_1, y, x_2, z,
                    loss_type=config['training']['loss'],
                    margin=config['training']['margin'])

    sim_pos = torch.mean(compute_similarity(config, x_1, y))
    sim_neg = torch.mean(compute_similarity(config, x_2, z))

  graph_vec_scale = torch.mean(graph_vectors ** 2)
  if config['training']['graph_vec_regularizer_weight'] > 0:
    loss += (config['training']['graph_vec_regularizer_weight'] *
           0.5 * graph_vec_scale)

  optimizer.zero_grad()


  loss.backward(torch.ones_like(loss))  #只支持pytorch1.2及以上
  nn.utils.clip_grad_value_(model.parameters(), config['training']['clip_value'])
  optimizer.step()

  sim_diff = sim_pos - sim_neg
  accumulated_metrics['loss'].append(loss)
  accumulated_metrics['sim_pos'].append(sim_pos)
  accumulated_metrics['sim_neg'].append(sim_neg)
  accumulated_metrics['sim_diff'].append(sim_diff)


  # evaluation
  if (i_iter + 1) % config['training']['print_after'] == 0:
    metrics_to_print = {
       k: torch.mean(v[0]) for k, v in accumulated_metrics.items()}
    info_str = ', '.join(
       ['%s %.4f' % (k, v) for k, v in metrics_to_print.items()])
    # reset the metrics
    accumulated_metrics = collections.defaultdict(list)

    if ((i_iter + 1) // config['training']['print_after'] %
          config['training']['eval_after'] == 0):
       model.eval()
```

```
        with torch.no_grad():
          accumulated_pair_auc = []
          for batch in validation_set.pairs(config['evaluation']['batch_size']):
            node_features, edge_features, from_idx, to_idx, graph_idx, labels = get_graph(batch)
            labels = labels.to(device)
            eval_pairs = model(node_features.to(device), edge_features.to(device),
from_idx.to(device),
                      to_idx.to(device),
                      graph_idx.to(device), config['evaluation']['batch_size'] * 2)

            x, y = reshape_and_split_tensor(eval_pairs, 2)
            similarity = compute_similarity(config, x, y)
            pair_auc = auc(similarity, labels)
            accumulated_pair_auc.append(pair_auc)

          accumulated_triplet_acc = []
          for batch in validation_set.triplets(config['evaluation']['batch_size']):
            node_features, edge_features, from_idx, to_idx, graph_idx = get_graph(batch)
            eval_triplets = model(node_features.to(device), edge_features.to(device),
from_idx.to(device),
                       to_idx.to(device),
                       graph_idx.to(device),
                       config['evaluation']['batch_size'] * 4)
            x_1, y, x_2, z = reshape_and_split_tensor(eval_triplets, 4)
            sim_1 = compute_similarity(config, x_1, y)
            sim_2 = compute_similarity(config, x_2, z)
            triplet_acc = torch.mean((sim_1 > sim_2).float())
            accumulated_triplet_acc.append(triplet_acc.cpu().numpy())

          eval_metrics = {
            'pair_auc': np.mean(accumulated_pair_auc),
            'triplet_acc': np.mean(accumulated_triplet_acc)}
          info_str += ', ' + ', '.join(
            ['%s %.4f' % ('val/' + k, v) for k, v in eval_metrics.items()])
        model.train()
      print('iter %d, %s, time %.2fs' % (
        i_iter + 1, info_str, time.time() - t_start))
      t_start = time.time()
```