ABSTRACT

Graph representation learning has emerged as a powerful technique for real-world problems. Various downstream graph learning tasks have benefited from its recent developments, such as node classification, similarity search, graph classification, and link prediction. However, prior arts on graph representation learning focus on domain specific problems and train a dedicated model for each graph, which is usually non-transferable to out-of-domain data. Inspired by recent advances in pre-training from natural language processing and computer vision, we design Graph Contrastive Coding (GCC) — an unsupervised graph representation learning framework — to capture the universal network topological properties across multiple networks. We design GCC’s pre-training task as subgraph-level instance discrimination in and across networks and leverage contrastive learning to empower the model to learn the intrinsic and transferable structural representations. We conduct extensive experiments on three graph learning tasks and ten graph datasets. The results show that GCC pre-trained on a collection of diverse datasets can achieve competitive or better performance to its task-specific trained-from-scratch counterparts. This suggests that the pre-training and fine-tuning paradigm presents great potential for graph representation learning.

CSCS CONCEPTS

- Information systems → Data mining; Social networks; Computing methodologies → Unsupervised learning; Neural networks; Learning latent representations.

KEYWORDS

graph representation learning, graph neural network, pre-training

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. GCC: Graph Contrastive Coding for Graph Neural Network Pre-Training. In Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’20), August 23–27, 2020, Virtual Event, CA, USA. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403168

1 INTRODUCTION

Hypothesis. Representative graph structural patterns are universal and transferable across networks.

Over the past two decades, the main focus of network science research has been on discovering and abstracting the universal structural properties underlying different networks. For example, Barabasi and Albert show that several types of networks, e.g., World Wide Web, social, and biological networks, have the scale-free property, i.e., all of their degree distributions follow a power law [1]. Leskovec et al. [28] discover that a wide range of real graphs satisfy the densification and shrinking laws. Other common patterns across networks include small world [58], motif distribution [31], community organization [34], and core-periphery structure [6], validating our hypothesis at the conceptual level.

In the past few years, however, the paradigm of graph learning has been shifted from structural pattern discovery to graph representation learning [11, 14, 25, 39–41, 48, 60], motivated by recent advances in deep learning [4, 30]. Specifically, graph representation learning converts the vertices, edges or subgraphs of a graph into low-dimensional embeddings such that vital structural information of the graph is preserved. The learned embeddings from the input graph can be then fed into standard machine learning models for downstream tasks on the same graph.

However, most representation learning work on graphs has thus far focused on learning representations for one single graph or a fixed set of graphs and very limited work can be transferred to out-of-domain data and tasks. Essentially, those representation learning models aim to learn network-specific structural patterns dedicated for each dataset. For example, the DeepWalk embedding model [39]...
learned on the Facebook social graph cannot be applied to other graphs. In view of (1) this limitation of graph representation learning and (2) the prior arts on common structural pattern discovery, a natural question arises here: can we universally learn transferable representative graph embeddings from networks?

The similar question has also been asked and pursued in natural language processing [10], computer vision [17], and other domains. To date, the most powerful solution is to pre-train a representation learning model from a large dataset, that is, self-supervised representation learning. The idea of pre-training is to use the pre-trained model as a good initialization for fine-tuning over (different) tasks on unseen datasets. For example, BERT [10] designs language model pre-training tasks to learn a Transformer encoder [53] from a large corpus. The pre-trained Transformer encoder is then adapted to various NLP tasks [56] by fine-tuning.

Presented Work. Inspired by this and the existence of universal graph structural patterns, we propose to study the potential of pre-training representation learning models for graphs. Ideally, given a (diverse) set of input graphs, such as the Facebook social graph and the DBLP co-author graph, we aim to pre-train a representation learning model from them with a self-supervised task, and then fine-tune it on different graphs with different graph learning tasks, such as node classification on the US-Airport graph. The critical question for graph pre-training here is: how to design the pre-training task such that the universal structural patterns in and across networks can be captured and further transferred?

In this work, we present the Graph Contrastive Coding (GCC) model to learn structural representations across graphs. Conceptually, we leverage the idea of contrastive learning [59] to design GCC’s pre-training task as instance discrimination. Its basic idea is to sample instances from input graphs, treat each of them as a distinct class of its own, and learn to encode and discriminate between these instances. Specifically, there are three questions to answer for GCC such that it can learn the transferable structural patterns: (1) what are the instances? (2) what are the discrimination rules? And (3) how to encode the instances?

In GCC’s pre-training stage, we propose to distinguish vertices according to their local structures (Cf. Figure 1). For each vertex, we sample subgraphs from its multi-hop ego network as instances. GCC aims to distinguish between subgraphs sampled from a certain vertex and subgraphs sampled from other vertices. Finally, for each subgraph, we use a graph neural network (specifically, the GIN model [60]) as the graph encoder to map the underlying structural patterns to latent representations. As GCC does not assume vertices and subgraphs come from the same graph, the graph encoder is forced to capture universal patterns across different input graphs. Given the pre-trained GCC model, we apply it to unseen graphs for various downstream tasks. GCC is able to measure the structural similarity between two vertices from different domains, e.g., a user from Facebook friendship network and a researcher from DBLP co-authorship network, as shown in Figure 1.

To the best of our knowledge, very limited work exists in the field of structural graph representation pre-training to date. A very recent one is to design strategies for pre-training GNNs on labeled graphs with node attributes for specific domains (molecular graphs) [19]. Another recent work is InfoGraph [46], which focuses on learning domain-specific graph-level representations, especially for graph classification tasks. The third related work is by Hu et al. [20], who define several graph learning tasks, such as predicting centrality scores, to pre-train a GCN [25] model on synthetic graphs.

We conduct extensive experiments to demonstrate the performance and transferability of GCC. We pre-train the GCC model on a collection of diverse types of graphs and apply the pre-trained model to three downstream graph learning tasks on ten new graph datasets. The results suggest that the GCC model achieves competitive or better results to the state-of-the-art task-specific graph representation learning models that are trained from scratch. For example, for node classification on the US-Airport network, GCC pre-trained on the Facebook, IMDB, and DBLP graphs outperforms GraphWave [12], ProNE [65] and Struc2vec [43] which are trained directly on the US-Airport graph, empirically demonstrating our hypothesis at the beginning.

To summarize, our work makes the following four contributions:

- We formalize the problem of structural graph representation pre-training across multiple graphs and identify its design challenges.
- We design the pre-training task as subgraph-level instance discrimination to capture universal and transferable structural patterns from multiple input graphs.
- We present the Graph Contrastive Coding (GCC) framework to learn structural graph representations, which leverages contrastive learning to guide the pre-training.
- We conduct extensive experiments to demonstrate that for out-of-domain tasks, GCC can offer comparable or superior performance over dedicated graph-specific models.

2 RELATED WORK

In this section, we review related work of vertex similarity, contrastive learning and graph pre-training.

2.1 Vertex Similarity

Quantifying similarity of vertices in networks/graphs has been extensively studied in the past years. The goal of vertex similarity is to answer questions [26] like “How similar are these two vertices?” or “Which other vertices are most similar to these vertices?” The definition of similarity can be different in different situations. We briefly review the following three types of vertex similarity.
Neighborhood similarity The basic assumption of neighborhood similarity, a.k.a., proximity, is that vertices closely connected should be considered similar. Early neighborhood similarity measures include Jaccard similarity (counting common neighbors), RWR similarity [36] and SimRank [21], etc. Most recently developed network embedding algorithms, such as LINE [48], DeepWalk [39], node2vec [14], also follow the neighborhood similarity assumption.

Structural similarity Different from neighborhood similarity which measures similarity by connectivity, structural similarity doesn’t even assume vertices are connected. The basic assumption of structural similarity is that vertices with similar local structures should be considered similar. There are two lines of research about modeling structural similarity. The first line defines representative patterns based on domain knowledge. Examples include vertex degree, structural diversity [52], structural hole [7], k-core [2], motif [5, 32], etc. Consequently, models of this genre, such as Struc2vec [43] and RoIX [18], usually involve explicit featurization. The second line of research leverages the spectral graph theory to model structural similarity. A recent example is GraphWave [12]. In this work, we focus on structural similarity. Unlike the above two genres, we adopt contrastive learning and graph neural networks to learn structural similarity from data.

Attribute similarity Real world graph data always come with rich attributes, such as text in citation networks, demographic information in social networks, and chemical features in molecular graphs. Recent graph neural networks models, such as GCN [25], GAT [54], GraphSAGE [16, 63] and MPNN [13], leverage additional attributes as side information or supervised signals to learn representations which are further used to measure vertex similarity.

2.2 Contrastive Learning
Contrastive learning is a natural choice to capture similarity from data. In natural language processing, Word2vec [30] model uses co-occurring words and negative sampling to learn word embeddings. In computer vision, a large collection of work [15, 17, 50, 59] learns self-supervised image representation by minimizing the distance between two views of the same image. In this work, we adopt the InfoNCE loss from Oord et al. [35] and instance discrimination task from Wu et al. [59], as discussed in Section 3.

2.3 Graph Pre-training
Skip-gram based model Early attempts to pre-train graph representations are skip-gram based network embedding models inspired by Word2vec [30], such as LINE [48], DeepWalk [39], node2vec [14], and metapath2vec [11]. Most of them follow the neighborhood similarity assumption, as discussed in section 2.1. The representations learned by the above methods are tied up with graphs used to train the models, and can not handle out-of-sample problems. Our Graph Contrastive Coding (GCC) framework differs from these methods in two aspects. First, GCC focuses on structural similarity, which is orthogonal to neighborhood similarity. Second, GCC can be transferred across graphs, even to graphs never seen during pre-training.

Pre-training graph neural networks There are several recent efforts to bring ideas from language pre-training [10] to pre-training graph neural networks (GNN). For example, Hu et al. [19] pre-train GNN on labeled graphs, especially molecular graphs, where each vertex (atom) has an atom type (such as C, N, O), and each edge (chemical bond) has a bond type (such as the single bond and double bond). The pre-training task is to recover atom types and chemical bond types in masked molecular graphs. Another related work is by Hu et al. [20], which defines several graph learning tasks to pre-train a GCN [25]. Our GCC framework differs from the above methods in two aspects. First, GCC is for general unlabeled graphs, especially social and information networks. Second, GCC does not involve explicit featurization and pre-defined graph learning tasks.

3 GRAPH REPRESENTATION PRE-TRAINING
In this section, we formalize the structural graph representation pre-training problem. To address it, we present the Graph Contrastive Coding (GCC) framework.

3.1 The Graph Pre-Training Problem
Conceptually, given a collection of graphs from various domains, we aim to pre-train a model to capture structural patterns across these graphs in an unsupervised manner. The model should be able to benefit various graph learning tasks on different graphs. The underlying assumption is that there exist common and transferable structural patterns across different graphs such as the motif, k-core, and structural hole, as evident in network science literature [5, 7, 32, 52]. One illustrative scenario is that we pre-train a model on Facebook, IMDB, and DBLP graphs with self-supervision, and apply it on the US-Airport network for node classification.

Formally, the structural graph representation pre-training problem is to learn a function $f$ that maps a vertex to a low-dimensional feature vector, such that $f$ has the following two properties:
• First, structural similarity, it maps vertices with similar local network topologies close to each other in the vector space;
• Second, transferability, it is compatible with vertices and graphs unseen during pre-training.

As such, the embedding function \( f \) can be adopted in various graph learning tasks, such as social role prediction, node classification, and graph classification.

Note that the focus of this work is on structural representation learning without node attributes and node labels, making it completely different from the problem setting in graph neural network research. In addition, the goal is to pre-train a structural representation model and apply it to unseen graphs, differing from traditional network embedding \([14, 39, 48]\) and recent attempts on pre-training graph neural networks with attributed graphs as input and applying them within a specific domain \([19]\).

3.2 Graph Contrastive Coding Overview

To pre-train structural representations on graphs, we present the Graph Contrastive Coding (GCC) model. Given a set of graphs, our goal is to pre-train a universal graph neural network encoder to capture the structural patterns behind these graphs. To achieve this, we need to design proper self-supervised tasks and learning objectives for graph structured data. Inspired by the recent success of contrastive learning in CV and NLP tasks to define an instance as an image or a sentence. However, such ideas cannot be directly extended to graph data, as instances in graphs are not clearly defined. Moreover, our pre-training focus is purely on structural representations without additional input features/attributes. This leaves the natural choice of a single vertex as an instance infeasible, as it is not applicable to discriminate between two vertices. To address this issue, we propose to extend one single vertex to its local structure. Specially, for a certain vertex \( v \), we define an instance to be its \( r \)-ego network:

\[
\text{Definition 3.1. A } r \text{-ego network. Let } G = (V, E) \text{ be a graph, where } V \text{ denotes the set of vertices and } E \subseteq V \times V \text{ denotes the set of edges}. \text{ For a vertex } v, \text{ its } r \text{-neighbors are defined as } S_v = \{ u : d(u, v) \leq r \} \text{ where } d(u, v) \text{ is the shortest path distance between } u \text{ and } v \text{ in the graph } G. \text{ The } r \text{-ego network of vertex } v, \text{ denoted by } G_v, \text{ is the sub-graph induced by } S_v.
\]

The left panel of Figure 2 shows two examples of 2-ego networks. GCC treats each \( r \)-ego network as a distinct class of its own and encourages the model to distinguish similar instances from dissimilar instances. Next, we introduce how to define (dis)similar instances.

Q2: Define (dis)similar instances. In computer vision \([17]\), two random data augmentations (e.g., random crop, random resize, random color jittering, random flip, etc) of the same image are treated as a similar instance pair. In GCC, we consider two random data augmentations of the same \( r \)-ego network as a similar instance pair and define the data augmentation as graph sampling \([27]\). Graph sampling is a technique to derive representative subgraph samples from the original graph. Suppose we would like to augment vertex \( v \)'s \( r \)-ego network \( (G_v) \), the graph sampling for GCC follows the three steps—random walks with restart (RWR) \([51]\), subgraph induction, and anonymization \([22, 29]\).

1. **Random walk with restart.** We start a random walk on \( G \) from the ego vertex \( v \). The walk iteratively travels to its neighborhood with the probability proportional to the edge weight. In addition, at each step, with a positive probability the walk returns back to the starting vertex \( v \).

2. **Subgraph induction.** The random walk with restart collects a subset of vertices surrounding \( v \), denoted by \( S_v \). The sub-graph \( G_v \) induced by \( S_v \) is then regarded as an augmented version of the \( r \)-ego network \( G_v \). This step is also known as the Induced Subgraph Random Walk Sampling (ISRWR).

3. **Anonymization** We then anonymize the sampled graph \( G_v \) by re-labeling its vertices to be \( \{1, 2, \ldots, |S_v|\} \), in arbitrary order.

We repeat the aforementioned procedure twice to create two data augmentations, which form a similar instance pair \((x^q, x^k)\). If two subgraphs are augmented from different \( r \)-ego networks, we treat them as a dissimilar instance pair \((x^q, x^k)\) with \( k \neq k_v \). It is worth noting that all the above graph operations—random walk with restart, subgraph induction, and anonymization—are available in the DGL package \([57]\).

In this work, we consider undirected edges.

Vertex order doesn’t matter because most of graph neural networks are invariant to permutations of their inputs \([4]\).
Discussion on graph sampling. In random walk with restart sampling, the restart probability controls the radius of ego-network (i.e., r) which GCC conducts data augmentation on. In this work, we follow Qiu et al. [42] to use 0.8 as the restart probability. The proposed GCC framework is flexible to other graph sampling algorithms, such as neighborhood sampling [16] and forest fire [27].

Discussion on anonymization. Now we discuss the intuition behind the anonymization step in the above data augmentation procedure. This step is designed to keep the underlying structural patterns and hide the exact vertex indices. This design avoids learning a trivial solution to instance discrimination, i.e., simply checking whether vertex indices of two sampled graphs match. Moreover, it facilitates the transfer of the learned model across different graphs as such a model is not associated with a particular vertex set.

Q3: Define graph encoders. Given two sampled subgraphs \( x^q \) and \( x^k \), GCC encodes them via two graph neural network encoders \( f_q \) and \( f_k \), respectively. Technically, any graph neural networks [4] can be used here as the encoder, and the GCC model is not sensitive to different choices. In practice, we adopt the Graph Isomorphism Network (GIN) [60], a state-of-the-art graph neural network model, as our graph encoder. Recall that we focus on structural representation pre-training while most GNN models require vertex features/attributes as input. To bridge the gap, we propose to leverage the graph structure of each sampled subgraph to initialize vertex features. Specifically, we define the generalized positional embedding as follows:

**Definition 3.2. Generalized positional embedding.** For each subgraph, its generalized positional embedding is defined to be the top eigenvectors of its normalized graph Laplacian. Formally, suppose one subgraph has adjacency matrix \( A \) and degree matrix \( D \), we conduct eigen-decomposition on its normalized graph Laplacian \( I - D^{-1/2}AD^{-1/2} = UAU^T \), where the top eigenvectors in \( U \) [55] are defined as generalized positional embedding.

The generalized positional embedding is inspired by the Transformer model in NLP [53], where the sine and cosine functions of different frequencies are used to define the positional embeddings in word sequences. Such a definition is deeply connected with graph Laplacian as follows.

**Fact.** The Laplacian of path graph has eigenvectors: \( u_k(i) = \cos(\pi ki/n - \pi k/2n) \), for \( 1 \leq k \leq n, 1 \leq i \leq n \). Here \( n \) is the number of vertices in the path graph, and \( u_k(i) \) is the entry at \( i \)-th row and \( k \)-the column of \( U \), i.e., \( U = [u_1 \ldots u_n] \).

The above fact shows that the positional embedding in sequence models can be viewed as Laplacian eigenvectors of path graphs. This inspires us to generalize the positional embedding from path graphs to arbitrary graphs. The reason for using the normalized graph Laplacian rather than the unnormalized version is that path graph is a regular graph (i.e., with constant degrees) while real-world graphs are often irregular and have skewed degree distributions. In addition to the generalized positional embedding, we also add the one-hot encoding of vertex degrees [60] and the binary indicator of the ego vertex [42] as vertex features. After encoded by the graph encoder, the final \( d \)-dimensional output vectors are then normalized by their L2-Norm [17].

A running example. We illustrate a running example of GCC in Figure 2. For simplicity, we set the dictionary size to be 3, i.e., \( K = 2 \). GCC first randomly augment two subgraphs \( x^q \) and \( x^k \) from a 2-ego network on the left panel of Figure 2. Meanwhile, another two subgraphs, \( x^{k_1} \) and \( x^{k_2} \), are generated from a noise distribution — in this example, they are randomly augmented from another 2-ego network on the left panel of Figure 2. Then the two graph encoders, \( f_q \) and \( f_k \), map the query and the three keys to low-dimensional vectors — \( q \) and \( \{k_0, k_1, k_2\} \). Finally, the contrastive loss in Eq. 1 encourages the model to recognize \((x^q, x^k)\) as a similar instance pair and distinguish them from dissimilar instances, i.e., \((x^{k_1}, x^{k_2})\).

3.4 GCC Learning

In contrastive learning, it is required to maintain the \( K \)-size dictionary and encoders. Ideally, in Eq. 1, the dictionary should cover as many instances as possible, making \( K \) extremely large. However, due to the computational constraints, we usually design and adopt economical strategies to effectively build and maintain the dictionary, such as end-to-end (E2E) and momentum contrast (MoCo) [17]. We discuss the two strategies as follows.

E2E samples mini-batches of instances and considers samples in the same mini-batch as the dictionary. The objective in Eq. 1 is then optimized with respect to parameters of both \( f_q \) and \( f_k \), both of which can accept gradient updates by backpropagation consistently. The main drawback of E2E is that the dictionary size is constrained by the batch size.

MoCo is designed to increase the dictionary size without additional backpropagation costs. Concretely, MoCo maintains a queue of samples from preceding mini-batches. During optimization, MoCo only updates the parameters of \( f_q \) (denoted by \( \theta_q \)) by backpropagation. The parameters of \( f_k \) (denoted by \( \theta_k \)) are not updated by gradient descent. He et al. [17] propose a momentum-based update rule for \( \theta_k \). More formally, MoCo updates \( \theta_k \) by \( \theta_k \leftarrow m\theta_k + (1 - m)\theta_q \), where \( m \in [0, 1] \) is a momentum hyper-parameter. The above momentum update rule gradually propagates the update in \( \theta_q \) to \( \theta_k \), making \( \theta_k \) evolve smoothly and consistently. In summary, MoCo achieves a larger dictionary size at the expense of dictionary consistency, i.e., the key representations in the dictionary are encoded by a smoothly-varying key encoder.

In addition to E2E and MoCo, there are other contrastive learning mechanisms to maintain the dictionary, such as memory bank [59]. Recently, He et al. [17] show that MoCo is a more effective option than memory bank in computer vision tasks. Therefore, we mainly focus on E2E and MoCo for GCC.

3.5 GCC Fine-Tuning

Downstream tasks. Downstream tasks in graph learning generally fall into two categories — graph-level and node-level, where the target is to predict labels of graphs or nodes, respectively. For graph-level tasks, the input graph itself can be encoded by GCC to achieve the representation. For node-level tasks, the node representation can be defined by encoding its \( r \)-ego network (or subgraphs augmented from its \( r \)-ego network). In either case, the encoded representations are then fed into downstream tasks to predict task-specific outputs.
Freezing vs. full fine-tuning. GCC offers two fine-tuning strategies for downstream tasks — freezing mode and full fine-tuning mode. In the freezing mode, we freeze the parameters of the pre-trained graph encoder $f_g$ and treat it as a static feature extractor, then the classifiers catering for specific downstream tasks are trained on top of the extracted features. In the full fine-tuning mode, the graph encoder $f_g$ is trained end-to-end together with the classifier on a downstream task. More implementation details about fine-tuning are available in Section 4.2.

GCC as a local algorithm. As a graph algorithm, GCC belongs to the local algorithm category [45, 49], in which the algorithms only involve local explorations of the input (large-scale) network, since GCC explores local structures by random walk based graph sampling. Such property enables GCC to scale to large-scale graph learning tasks and to be friendly to the distributed computing setting.

4 EXPERIMENTS

In this section, we evaluate GCC on three graph learning tasks — node classification, graph classification, and similarity search, which have been commonly used to benchmark graph learning algorithms [12, 43, 46, 60, 61]. We first introduce the self-supervised pre-training settings in Section 4.1, and then report GCC transfer learning results on three graph learning tasks in Section 4.2.

4.1 Pre-Training

Datasets. Our self-supervised pre-training is performed on six graph datasets, which can be categorized into two groups — academic graphs and social graphs. As for academic graphs, we collect the Academia dataset from NetRep [44] as well as two DBLP datasets from SNAP [62] and NetRep [44], respectively. As for social graphs, we collect Facebook and IMDB datasets from NetRep [44], as well as a LiveJournal dataset from SNAP [3]. Table 1 presents the detailed statistics of datasets for pre-training.

Pre-training settings. We train for 75,000 steps and use Adam [24] for optimization with learning rate of 0.005, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 \times 10^{-8}$, weight decay of $1e-4$, learning rate warmup over the first 7,500 steps, and linear decay of the learning rate after 7,500 steps. Gradient norm clipping is applied with range $[-1, 1]$. For MoCo, we use mini-batch size of 32, dictionary size of 16,384, and momentum $m$ of 0.999. For E2E, we use mini-batch size of 1,024. For both MoCo and E2E, the temperature $T$ is set as 0.07, and we adopt GIN [60] with 5 layers and 64 hidden units each layer as our encoders. Detailed hyper-parameters can be found in Table 6 in the Appendix.

4.2 Downstream Task Evaluation

In this section, we apply GCC to three graph learning tasks including node classification, graph classification, and similarity search. As prerequisites, we discuss the two fine-tuning strategies of GCC as well as the baselines we compare with.

Fine-tuning. As we discussed in Section 3.5, we adopt two fine-tuning strategies for GCC. We select logistic regression or SVM from the scikit-learn [38] package as the linear classifier for the freezing strategy. As for the full fine-tuning strategy, we use Adam optimizer with learning rate 0.005, learning rate warmup over the first 3 epochs, and linear learning rate decay after 3 epochs.

Baselines. Baselines can be categorized into two categories. In the first category, the baseline models learn vertex/graph representations from unlabeled graph data and then feed them into logistic regression or SVM. Examples include DGK [61], Struc2vec [43], GraphWave [12], graph2vec [33] and InfoGraph [46]. GCC with freezing setting belongs to this category. In the second category, the models are optimized in an end-to-end supervised manner. Examples include DGCNN [67] and GIN [60]. GCC with the full fine-tuning setting belongs to this category. For a fair comparison, we fix the representation dimension of all models to be 64 except graph2vec and InfoGraph. The details of baselines will be discussed later.

4.2.1 Node Classification.

Setup. Node classification task is to predict unknown node labels in a partially labeled network. To evaluate GCC, we sample a subgraph centered at each vertex and apply GCC on it. Then the obtained representation is fed into an output layer to predict the node label. As for datasets, we adopt US-Airport [43] and H-index [64]. US-Airport consists of the airline activity data among 1,190 airports. The 4 classes indicate different activity levels of the airports. H-index is a co-authorship graph extracted from OAG [64]. The labels indicate whether the h-index of the author is above or below the median.

Experimental results. We compare GCC with ProNE [65], GraphWave [12], and Struc2vec [43]. Table 2 represents the results. It is worth noting that, under the freezing setting, the graph encoder in GCC is not trained on either US-Airport or H-Index dataset, which other baselines use as training data. This places GCC at a disadvantage. However, GCC (MoCo, freeze) performs competitively to Struc2vec in US-Airport, and achieves the best performance in H-index where Struc2vec cannot finish in one day. Moreover, GCC can be further boosted by fully fine-tuning on the target US-Airport or H-Index domain.

4.2.2 Graph Classification.

Setup. We use five datasets from Yanardag and Vishwanathan [61] — COLLAB, IMDB-BINARY, IMDB-MULTI, REDDITBINARY and REDDIT-MULTI5K, which are widely benchmarked in recent graph classification models [19, 46, 67]. Each dataset is a set of graphs where each graph is associated with a label. To evaluate GCC on this task, we use raw input graphs as the input of GCC. Then the encoded graph-level representation is fed into a classification layer to predict the label of the graph. We compare GCC with several recent developed graph classification models, including Deep Graph Kernel (DGK) [61], graph2vec [33], InfoGraph [46], DGCNN [67] and GIN [60]. Among these baselines, DGK, graph2vec...
and InfoGraph belong to the first category, while DGCNN and GIN belong to the second category.

**Experimental Results.** Table 3 shows the comparison. In the first category, GCC (MoCo, freeze) performs competitively to InfoGraph in IMDB-B and IMDB-M, while achieves the best performance in other datasets. Again, we want to emphasize that DGK, graph2vec and InfoGraph all need to be pre-trained on target domain graphs, but GCC only relies on the graphs listed in Table 1 for pre-training. In the second category, we compare GCC with DGCNN and GIN. GCC achieves better performance than DGCNN and comparable performance to GIN. GIN is a recently proposed SOTA model for graph classification. We follow the instructions in the paper [60] to provide random guess results for reference.

**Experimental Results.** Table 4 presents the performance of different methods on top-k similarity search task in three co-author datasets. The problem of top-k similarity search is defined as follows. Given two graphs G1 and G2, for example KDD and ICDM co-author graphs, we want to find the most similar vertex v from G1 for each vertex u in G2. In this dataset, the ground truth is defined to be authors publish in both conferences. Note that similarity search is an unsupervised task, so we evaluate GCC without fine-tuning. Especially, we first extract two subgraphs centered at u and v by random walk with restart graph sampling. After encoding them by GCC, we measure the similarity score between u and v to be the inner product of their representations. Finally, by sorting the above scores, we use HITS@10 (top-10 accuracy) to measure the performance of different methods. We compare GCC with RolX [18], Panther++ [66] and GraphWave [12]. We also provide random guess results for reference.

**Experimental Results.** Table 4 presents the performance of different methods on top-k similarity search task in three co-author networks. We can see that, compared with Panther++ [66] and GraphWave [12] which are trained in place on co-author graphs, simply applying pre-trained GCC can be competitive.

**Effect of pre-training.** It is still not clear if GCC’s good performance is due to pre-training or the expression power of its GIN [60] encoder. To answer this question, we fully fine-tune GCC with its GIN encoder randomly initialized, which is equivalent to train a GIN encoder from scratch. We name this model GCC (rand), as shown...
in Table 2 and Table 3. In all datasets except IMDB-B, GCC (MoCo) outperforms its randomly initialized counterpart, showing that pre-training always provides a better start point for fine-tuning than random initialization. For IMDB-B, we attribute it to the domain shift between pre-training data and down-stream tasks.

Contrastive loss mechanisms. The common belief is that MoCo has stronger expression power than E2E [17], and a larger dictionary size $K$ always helps. We also observe such trends, as shown in Figure 3. However, the effect of a large dictionary size is not as significant as reported in computer vision tasks [17]. For example, MoCo ($K = 16384$) merely outperforms MoCo ($K = 1024$) by small margins in terms of accuracy — 1.0 absolute gain in US-Airport and 0.8 absolute gain in COLLAB. However, training MoCo is much more economical than training E2E. E2E ($K=1024$) takes 5 days and 16 hours, while MoCo ($K=16384$) only needs 9 hours. Detailed training time can be found in Table 5 in the Appendix.

Table 5: Momentum ablation.

<table>
<thead>
<tr>
<th>momentum $m$</th>
<th>0.0</th>
<th>0.9</th>
<th>0.99</th>
<th>0.999</th>
<th>0.9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>US-Airport</td>
<td>62.3</td>
<td>63.2</td>
<td>63.7</td>
<td>65.6</td>
<td>61.5</td>
</tr>
<tr>
<td>COLLAB</td>
<td>76.6</td>
<td>75.1</td>
<td>77.4</td>
<td>78.9</td>
<td>79.4</td>
</tr>
</tbody>
</table>

Momentum. As mentioned in MoCo [17], momentum $m$ plays a subtle role in learning high-quality representations. Table 5 shows accuracy with different momentum values on US-Airport and COLLAB datasets. For US-Airport, the best performance is reached by $m = 0.999$, which is the desired value in [17], showing that building a consistent dictionary is important for MoCo. However, in COLLAB, it seems that a larger momentum value brings better performance. Moreover, we do not observe the "training loss oscillation" reported in [17] when setting $m = 0$. GCC (MoCo) converges well, but the accuracy is much worse.

Pre-training datasets. We ablate the number of datasets used for pre-training. To avoid enumerating a combinatorial space, we pre-train with first several datasets in Table 1, and report the 10-fold validation accuracy scores on US-Airport and COLLAB, respectively. For example, when using one dataset for pre-training, we select Academia; when using two, we choose Academia and DBLP (SNAP); and so on. We present ordinary least squares (OLS) estimates of the relationship between the number of datasets and the model performance. As shown in Figure 4, we can observe a trend towards higher accuracy when using more datasets for pre-training. On average, adding one more dataset leads to 0.43 and 0.81 accuracy (%) gain on US-Airport and COLLAB, respectively. 6

Figure 3: Comparison of contrastive loss mechanisms.

(a) US-Airport
(b) COLLAB

(a) US-Airport: $y = 0.4344x + 62.394$
(b) COLLAB: $y = 0.8065x + 74.4737$

Figure 4: Ablation study on pre-training datasets.

5 CONCLUSION

In this work, we study graph representation learning with the goal of characterizing and transferring structural features in social and information networks. We present Graph Contrastive Coding (GCC), which is a graph-based contrastive learning framework to learn structural representations and similarity from data. The pre-trained model achieves competitive performance to its supervised trained-from-scratch counterparts in three graph learning tasks on ten graph datasets. In the future, we plan to benchmark more graph learning tasks and more graph datasets. We also would like to explore applications of GCC on graphs in other domains, such as protein-protein association networks [47].

Acknowledgements. The work is supported by the National Key R&D Program of China (2018YFB1402600), NSFC for Distinguished Young Scholar (61825602), and NSFC (61836013).

REFERENCES


6The effect on US-Airport is positive, but statistically insignificant ($p$-value $= 0.231$), while the effect on COLLAB is positive and significant ($p$-value $< 0.001$).
A APPENDIX

Table 6: Pre-training hyper-parameters for E2E and MoCo.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>E2E</th>
<th>MoCo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size</td>
<td>1024</td>
<td>32</td>
</tr>
<tr>
<td>Restart probability</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Dictionary size K</td>
<td>1023</td>
<td>16384</td>
</tr>
<tr>
<td>Temperature r</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>Momentum m</td>
<td>NA</td>
<td>0.999</td>
</tr>
<tr>
<td>Warmup steps</td>
<td>7,500</td>
<td>7,500</td>
</tr>
<tr>
<td>Weight decay</td>
<td>1e-5</td>
<td>1e-5</td>
</tr>
<tr>
<td>Training steps</td>
<td>75,000</td>
<td>75,000</td>
</tr>
<tr>
<td>Initial learning rate</td>
<td>0.005</td>
<td>0.005</td>
</tr>
<tr>
<td>Learning rate decay</td>
<td>Linear</td>
<td>Linear</td>
</tr>
<tr>
<td>Adam ϵ</td>
<td>1e-8</td>
<td>1e-8</td>
</tr>
<tr>
<td>Adam β₁</td>
<td>0.9</td>
<td>0.9</td>
</tr>
<tr>
<td>Adam β₂</td>
<td>0.999</td>
<td>0.999</td>
</tr>
<tr>
<td>Gradient clipping</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Number of layers</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Hidden units per layer</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Dropout rate</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Degree embedding size</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Figure 5: Pre-training time of different contrastive loss mechanisms and dictionary size K.

A.1 Implementation details

A.1.1 Hardware Configuration.

The experiments are conducted on Linux servers equipped with an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 256GB RAM and 8 NVIDIA 2080Ti GPUs.

A.1.2 Software Configuration.

All models are implemented in PyTorch [37] version 1.3.1, DGL [57] version 0.4.1 with CUDA version 10.1, scikit-learn version 0.20.3 and Python 3.6. Our code and datasets will be available.

A.1.3 Pre-training.

The detailed hyper-parameters are listed in Table 6. Training times of GCC variants are listed in Figure 5. The training time of GCC (E2E) grows sharply with the dictionary size K while GCC (MoCo) roughly remains the same, which indicates that MoCo is more economical and easy to scale with larger dictionary size.

A.2 Baselines

A.2.1 Node Classification.

GraphWave [12] We download the authors’ official source code and keep all the training settings as the same. The implementation requires a networkx graph and time points as input. We convert our dataset to the networkx format, and use automatic selection of the range of scales provided by the authors. We set the output embedding dimension to 64.


Struc2vec [43] We download the authors’ official source code and use default hyper-parameters provided by the authors: (1) walk length = 80; (2) number of walks = 10; (3) window size = 10; (4) number of iterations = 5.

The only modifications we do are: (1) number of dimensions = 64; (2) number of workers = 48 to speed up training.

We find the method hard to scale on the H-index datasets although we set the number of workers to 48, compared to 4 by default. We keep the code running for 24 hours on the H-index datasets and it failed to finish. We observed that the sampling strategy in Struc2vec takes up most of the time, as illustrated in the original paper.


ProNE [65] We download the authors’ official source code and keep hyper-parameters as the same: (1) walk length = 80; (2) number of workers = 48 to speed up training.

We adopt the reported results in these papers. Our experimental setting is exactly the same except for the dimension size. Note that graph2vec uses 1024 and InfoGraph uses 512 as the dimension size.

A.2.2 Graph Classification.

DGK [61], graph2vec [33], InfoGraph [46], DGCNN [67] We adopt the reported results in these papers. Our experimental setting is exactly the same except for the dimension size. Note that graph2vec uses 1024 and InfoGraph uses 512 as the dimension size.

Following GIN, we use 64.

AAPPENDIX

Table 7: Performance of GIN model under various hyper-parameter configurations.

<table>
<thead>
<tr>
<th>batch</th>
<th>dropout</th>
<th>degree</th>
<th>IMDB-B</th>
<th>IMDB-M</th>
<th>COLLAB</th>
<th>RDT-B</th>
<th>RDT-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0</td>
<td>no</td>
<td>72.9</td>
<td>48.7</td>
<td>69.2</td>
<td>83.5</td>
<td>52.2</td>
</tr>
<tr>
<td>32</td>
<td>0.5</td>
<td>no</td>
<td>71.6</td>
<td>49.1</td>
<td>66.2</td>
<td>77.5</td>
<td>51.7</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>no</td>
<td>75.6</td>
<td>50.9</td>
<td>73.0</td>
<td>89.4</td>
<td>54.5</td>
</tr>
<tr>
<td>128</td>
<td>0.5</td>
<td>no</td>
<td>74.6</td>
<td>51.0</td>
<td>72.3</td>
<td>81.8</td>
<td>53.5</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>yes</td>
<td>73.9</td>
<td>51.1</td>
<td>79.7</td>
<td>77.0</td>
<td>46.2</td>
</tr>
<tr>
<td>32</td>
<td>0.5</td>
<td>yes</td>
<td>74.5</td>
<td>50.1</td>
<td>79.1</td>
<td>77.4</td>
<td>45.6</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>yes</td>
<td>73.3</td>
<td>51.0</td>
<td>79.8</td>
<td>76.7</td>
<td>45.6</td>
</tr>
<tr>
<td>128</td>
<td>0.5</td>
<td>yes</td>
<td>73.1</td>
<td>51.5</td>
<td>80.2</td>
<td>77.4</td>
<td>45.8</td>
</tr>
</tbody>
</table>
cross-validation accuracy averaged over the 10 folds was selected. We report the obtained results in Table 7.

Code: https://github.com/weihua916/powerful-gnns.

A.2.3 Top-k Similarity Search.

**Random, RolX [18], Panther++ [66]** We obtain the experimental results for these baselines from Zhang et al. [66].

Code: https://github.com/yuikns/panther/.

**GraphWave [12]** Embeddings computed by the GraphWave method also have the ability to generalize across graphs. The authors evaluated on synthetic graphs in their paper which are not publicly available. To compare with GraphWave on the co-author datasets, we compute GraphWave embeddings given two graphs $G_1$ and $G_2$ and follow the same procedure mentioned in section 4.2.2 to compute the HITS@10 (top-10 accuracy) score.


A.3 Datasets

A.3.1 Node Classification Datasets.

**US-Airport** We obtain the US-Airport dataset directly from Ribeiro et al. [43].

**H-index** We create the H-index dataset, a co-authorship graph extracted from OAG [64]. Since the original OAG co-authorship graph has millions of nodes, it is too large as a node classification benchmark. Therefore, we implemented the following procedure to extract smaller subgraphs from OAG:

1. Select an initial vertex set $V_s$ in OAG;
2. Run breadth first search (BFS) from $V_s$ until $N$ nodes are visited;
3. Return the sub-graph induced by the visited $N$ nodes.

We set $N = 5,000$, and randomly select 20 nodes from top 200 nodes with largest degree as the initial vertex set in step (1).

A.3.2 Graph Classification Datasets.

We download COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY and REDDIT-MULTI5K from Benchmark Data Sets for Graph Kernels [23].

A.3.3 Top-k Similarity Search Datasets.

We obtain the paired conference co-author datasets, including KDD-ICDM, SIGIR-CIKM, SIGMOD-ICDE, from the Zhang et al. [66] and make them publicly available with the permission of the original authors.

---

8https://github.com/leoribeiro/struc2vec/tree/master/graph
9https://www.openacademic.ai/oag/
10https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets
11https://github.com/yuikns/panther