# Learning Graph Representations with Recurrent Neural Network Autoencoders

Aynaz Taheri
University of Illinois at Chicago
ataher2@uic.edu

Kevin Gimpel
Toyota Technological Institute at Chicago
kgimpel@ttic.edu

Tanya Berger-Wolf
University of Illinois at Chicago
tanyabw@uic.edu

## ABSTRACT

Representing and comparing graphs is a central problem in many fields. We present an approach to learn representations of graphs using recurrent neural network autoencoders. Recurrent neural networks require sequential data, so we begin with several methods to generate sequences from graphs, including random walks, breadth-first search, and shortest paths. We train long short-term memory (LSTM) autoencoders to embed these graph sequences into a continuous vector space. We then represent a graph by averaging its graph sequence representations. The graph representations are then used for graph classification and comparison tasks. We demonstrate the effectiveness of our approach by showing improvements over the existing state-of-the-art on several graph classification tasks, including both labeled and unlabeled graphs.

## KEYWORDS

Representation learning, Deep Learning, Recurrent Neural Networks, Graph Classification

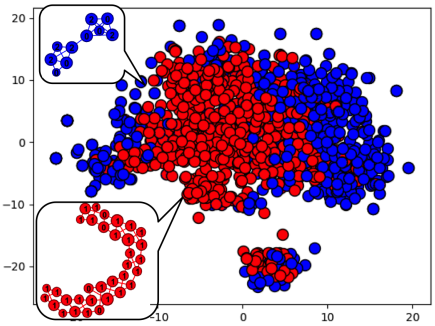


**Figure 1: Learned graph representations in Proteins [5] dataset, which has two classes: enzyme (red) and non-enzyme (blue). The plot is generated by t-SNE [29] and the two highlighted graphs are generated by Gephi [2].**

## 1 INTRODUCTION

We address the problem of comparing and classifying graphs by learning their representations. This problem arises in many domain areas, including bioinformatics, social network analysis, chemistry, neuroscience, and computer vision. For instance, in neuroscience, comparing brain networks represented by graphs helps to identify brains with neurological disorders [43]. In social network analysis, we may need to compare egonetworks to detect anomalies [1] or to identify corresponding egonetworks across multiple social networks. Cutting across domains, we may be interested in understanding how to distinguish the structure of a social network from a biological network, an authorship network, a computer network, or a citation network [33]. In some tasks, we need to quantify the similarity between two graphs. For example, in pattern recognition tasks such as handwriting recognition, we need to measure the similarity between graphs derived from handwriting images [14].

We consider the setting in which we want to compare graphs without necessarily finding any node correspondence between them. Existing graph comparison methods can be categorized into three main (not necessarily disjoint) classes: *feature extraction*, *graph kernels*, and *graph matching*.

*Feature extraction* methods compare graphs across a set of features, such as specific subgraphs or numerical properties that capture the topology of the graphs [4, 30]. The efficiency and performance of such methods is highly dependent on the feature selection process. Most *graph kernels* [44] are based on the idea of R-convolutional kernels [19], a way of defining kernels on structured objects by decomposing the objects into substructures and comparing pairs in the decompositions. For graphs, the substructures include graphlets [39], shortest paths [6], random walks [16], and subtrees [38]. While graph kernel methods are effective, their time complexity is quadratic in the number of graphs. *Graph matching* algorithms use the topology of the graphs, their nodes and edges directly, counting matches and mismatches [8, 36]. These approaches do not consider the global structure of the graphs and are sensitive to noise.

In this paper, we propose an unsupervised approach for learning graph representations using long short-term memory (LSTM) recurrent neural networks [22]. An unsupervised graph representation approach can be used not only in processing labeled data, such as in graph classification in bioinformatics, but can be also applied in many practical applications, such as anomaly detection in social networks or streaming data, as well as in exploratory analysis and scientific hypothesis generation. An unsupervised method for learning graph representations provides a fundamental capability to analyze graphs based on their intrinsic properties.


*KDD'18 Deep Learning Day, August 2018, London, UK*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We leverage the LSTM sequence-to-sequence learning framework of [40], which uses one LSTM to encode the input sequence into a vector and another LSTM to generate the output sequence from that vector. We use the same sequence as both input and output, making this a sequence-to-sequence LSTM autoencoder [27]. We consider several ways to generate sequences from the graphs, including random walks, breadth-first search, and shortest paths. After training the autoencoder, we represent a graph by averaging the encodings of several of its graph sequences. The graph representations are then used for graph classification.

Figure 1 shows the result of using our approach to embed the graphs in the Proteins dataset [5] into a 100-dimensional space, visualized with t-SNE [29]. Each point is one of the 1113 graphs in the dataset. The two class labels were not used when learning the graph representations, but we still generally find that graphs of the same class are clustered in the learned space.

We demonstrate the efficacy of our approach by using our learned graph representations in classification tasks for both labeled and unlabeled graphs. We outperform the state-of-the-art on nearly all datasets considered, showing performance gains over prior graph kernel methods while also being asymptotically more efficient. Indeed, the time complexity of our approach is linear in the number of graphs, while kernel methods are quadratic.

## 2 RELATED WORK

While we briefly discussed prior work in Section 1, we now provide some additional remarks to provide more perspective on other work in this area.

Recently, some new graph kernels such as Deep Graph Kernel (DGK) [47] and optimal-assignment Weisfeiler-Lehman (WL-OA) [23] have been proposed and evaluated for graph classification tasks. The DGK [47] uses methods from unsupervised learning of word embeddings [31] to augment the kernel with substructure similarity. WL-OA [23] is an assignment kernel that finds an optimal bijection between different parts of the graph. While graph kernel methods are effective, their time complexity is quadratic in the number of graphs, and there is no opportunity to customize their representations for supervised tasks.

There has been other work in using neural methods to learn graph representations. The representations obtained by these approaches are tailored for a supervised task and are not determined solely based on the graph structure. Niepert et al. [34] developed a framework (PSCN) to learn graph representations using convolutional neural networks (CNNs). Deep Graph Convolutional Neural Network (DGCNN) [48] is another model that applies CNNs for graph classification. The main difference between DGCNN and PSCN is the way they deal with the vertex-ordering problem. We compare to them in our experiments, outperforming them on all datasets.

Message passing neural networks [17] are another group of supervised approaches that have been recently used for graph structured data, such as molecular property prediction in chemistry [13, 17, 28]. Duvenaud et al. [13] introduced a CNN to create "fingerprints" (vectors that encode molecule structure) for graphs derived by molecules. The information about each atom and its neighbors are fed to the neural network, and neural fingerprints are used to predict new features for the graphs. Bruna et al. [7] proposed spectral networks, generalizations of CNNs on low-dimensional graphs via graph Laplacians. Henaff et al. [20] and Defferrard et al. [11] extended spectral networks to high-dimensional graphs. Scarselli et al. [37] proposed graph neural networks (GNN) and find node representations using random walks. Li et al. [28] extended GNNs with gating recurrent neural networks to predict sequences from graphs. In general, neural message passing approaches can suffer from high computational and memory costs, since they perform multiple iterations of updating hidden node states in graph representations. However, our unsupervised approach obtains strong performance without the requirement of passing messages between vertices for multiple iterations.

Graph2vec [32] is an unsupervised method inspired by document embedding models [25]. This approach finds a representation for a graph by maximizing the likelihood of graph subtrees given the graph embedding. Our approach outperforms this method by a large margin. Graph2vec suffers from its lack of capturing global information in the graph structure by only considering subtrees as graph representatives. Other methods have been developed to learn representations for individual nodes in graphs, such as DeepWalk [35], LINE [41], node2vec [18], and many others. These methods are not directly related to graph comparison because they require aggregating node representations to represent entire graphs. However, we compared to one representative method in our experiments to show that the aggregation of node embeddings is not informative enough to represent the structure of a graph.

## 3 BACKGROUND

We briefly discuss the required background about graphs and LSTM recurrent neural networks.

**Graphs.** For a graph $G = (V, E)$, $V$ denotes its vertex set and $E \subseteq V \times V$ denotes its edge set. $G$ is called a *labeled graph* if there is a labeling function $label : V \rightarrow L$ that assigns a label from a set of labels $L$ to each vertex. The graph $G$ is called an *unlabeled graph* if no label has been assigned to each vertex in the graph.

**Long short-term memory (LSTM).** An LSTM [22] is a recurrent neural network (RNN) designed to model long-distance dependencies in sequential data. We denote the input vector at time $t$ by $x_t$ and we denote the hidden vector computed at time $t$ by $h_t$. At each time step, an LSTM computes a memory cell vector $c_t$, an input gate vector $i_t$, a forget gate vector $f_t$, and an output gate vector $o_t$:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + K_i c_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + K_f c_{t-1} + b_f) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + K_o c_{t-1} + b_o) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{1}$$

where $\odot$ denotes elementwise multiplication, $\sigma$ is the logistic sigmoid function, each $W$ is a weight matrix connecting inputs to particular gates (denoted by subscripts), each $U$ is an analogous matrix connecting hidden vectors to gates, each $K$ is a diagonal matrix connecting cell vectors to gates, and each $b$ is a bias. We refer to this as an "LSTM encoder" because it converts an input sequence into a sequence of hidden vectors $h_t$. We will also use a type of LSTM that predicts the next item $\overline{x}$ in the sequence from

$h_t$. This architecture, which we refer to as an "LSTM decoder," adds the following:

$$\overline{x} = g(h_t) \tag{2}$$

where $g$ is a function that takes a hidden vector and outputs a predicted observation $\overline{x}$. With symbolic data, $g$ typically computes a softmax distribution over symbols and returns the max-probability symbol. When using continuous inputs, $g$ could be an affine transform of $h_t$ followed by a nonlinearity.

## 4 APPROACH OVERVIEW

We propose an unsupervised approach for learning representations of labeled and unlabeled graphs. Our goal is to learn graph embeddings such that graphs with similar structure lie close to one another in the embedding space. We seek to learn a mapping function $\Phi : G \rightarrow \mathbb{R}^d$ that embeds a graph $G$ into a $d$-dimensional space. We are interested in methods that scale linearly in the number of graphs in a dataset, as opposed to graph kernel methods that require quadratic time.

Our approach uses autoencoders [21], which form one of the principal frameworks of unsupervised learning. Autoencoders are typically trained to reconstruct their input in such a way that they learn useful properties of the data. There are two parts to an autoencoder: an encoder that maps the input to some intermediate representation, and a decoder that attempts to reconstruct the input from this intermediate representation. For graphs, we need to decide how to represent graphs in a form that can be encoded and then reconstructed. We do this by extracting sequences of symbols from the graphs that we want to represent. Section 5 describes several methods for doing this, including the use of random walks, shortest paths between all pairs of nodes, and paths derived from breadth-first search. These sequences can also be generated for unlabeled graphs.

Given sequences from a graph, we then need an autoencoding framework that can handle variable-length sequences. We choose LSTMs for both the encoder and decoder, forming an LSTM autoencoder [27, 40]. LSTM autoencoders use one LSTM to read the input sequence and encode it to a fixed dimensional vector, and then use another LSTM to decode the output sequence from the vector. We consider several variations for the encoder and decoder, described in Section 6. We train our LSTM autoencoders on sequences pooled from a set of graphs. We experiment with two training objectives, described in Section 7.

Given the trained encoder $LSTM_{enc}$, we define the graph embedding function $\Phi(G)$ as the mean of the vectors output by the encoder for graph sequences extracted from $G$:

$$\Phi(G) = \frac{1}{|Seq(G)|} \sum_{s \in Seq(G)} LSTM_{enc}(s) \tag{3}$$

where $Seq(G)$ is the set of sequences extracted from $G$.[1] We use $\Phi$ to represent graphs in our experiments in Section 8, demonstrating state-of-the-art performance for several graph classification tasks.

[1]Using the mean outperformed max pooling in our experiments so we only report results using the mean in this paper.

## 5 GENERATING GRAPH SEQUENCES

We use *Random Walks*, *Shortest Paths*, and *Breadth-First Search* to generate sequences. These sequences are then used to train our LSTM autoencoders.

**Random Walks (RW)**: Given a source node $u$, we generate a random walk $w_u$ with fixed length $m$. Let $v_i$ denote the $i$th node in $w_u$, starting with $v_0 = u$. $v_{t+1}$ is a node from the neighbors of $v_t$ that is selected with probability $1/d(v_t)$, where $d(v_t)$ is the degree of $v_t$.

**Shortest Paths (SP)**: We generate all the shortest paths between each pair of nodes in the graph using the Floyd-Warshall algorithm [15].

**Breadth-First Search (BFS)**: We run the BFS algorithm at each node to generate graph sequences for that node. The graph sequences for the graph include the BFS sequences starting at each node in the graph, limited to a maximum number of edges from the starting node. We give details on the maximum used in our experiments below.

### 5.1 Embedding vertices

When creating graph sequences for training our autoencoders, we need to represent individual vertices as vectors. We use $Emb(v)$ to denote the embedding of vertex $v$. For labeled graphs, we can use the vertex labels to define $Emb(v)$, but for unlabeled graphs it is less clear how to do this. Our solution, which we use for both labeled and unlabeled graphs, is to use the *Weisfeiler-Lehman* (WL) algorithm [46]. This algorithm is used as a graph isomorphism test that determines whether two graphs are isomorphic. It is known as an iterative vertex classification or vertex refinement procedure [38].

The WL algorithm uses multiset labels to encode the local structure of the graphs. The idea is to create a multiset label for each node using the sorted list of its neighbors' labels. Then, the sorted list is compressed into a new value. This labeling process continues until the new multiset labels of the two graphs are different or the number of iteration reaches a specified limit.

After running the WL algorithm on a graph, we obtain an integer label for each vertex $v$. We assign a parameter vector $Emb(v)$ to each unique integer label. $Emb(v)$ is a vector in a $d$-dimensional space where each entry is initialized with a draw from a uniform distribution with range $[-1, 1]$. These embeddings are updated during training. We note that we can apply the Weisfeiler-Lehman algorithm on both labeled and unlabeled graphs. For unlabeled graphs, we start the algorithm by assigning the same label to all vertices. However, for labeled graphs, we start by using the provided vertex labels in the graph. So, for labeled graphs, WL can produce a richer, more informative vertex labeling because it starts with the provided labels for the dataset and then enriches the labels using the neighbors' labels (thereby capturing the local structure of each node).

## 6 SEQ-TO-SEQ AUTOENCODERS

We discussed three ways of extracting vertex sequences from graphs and our approach for embedding vertices. We now describe how we will learn our graph embedding function.

We formulate graph representation learning as training an autoencoder on vertex sequences generated from graphs. The most common type of autoencoder is a feed-forward deep neural network, but they suffer from the limitation of requiring fixed-length inputs and an inability to model sequential data. Therefore, we focus in this paper on *sequence-to-sequence autoencoders* which can support arbitrary-length sequences.

These autoencoders are based on the sequence-to-sequence learning framework of Sutskever *et al.* [40], an LSTM-based architecture in which both the inputs and outputs are sequences of variable length. The architecture uses one LSTM as the encoder $LSTM_{enc}$ and another LSTM as the decoder $LSTM_{dec}$. An input sequence $s$ with length $m$ is given to $LSTM_{enc}$ and its elements are processed one per time step. The hidden vector $h_m$ at the last time step $m$ is the fixed-length representation of the input sequence. This vector is provided as the initial vector to $LSTM_{dec}$ to generate the output sequence.

Li *et al.* [27] adapted the sequence-to-sequence learning framework for autoencoding by using the same sequence for both the input and output. They trained the autoencoder such that the decoder $LSTM_{dec}$ reconstructs the input using the final hidden vector from $LSTM_{enc}$. We use this same approach, using our graph sequences in place of their textual sequences. We develop several modifications to this model and evaluate their impact in our experiments.

In our experiments, we use several graph datasets. We train a single autoencoder for each graph dataset. The autoencoder is trained on a training set of graph sequences pooled across all graphs in the dataset. After training the autoencoder, we obtain the representation $\Phi(G)$ for a single graph $G$ by encoding its sequences $s \in Seq(G)$ using $LSTM_{enc}$, then averaging its encoding vectors, as in Eq. (3).

**S2S-AE**: This is the standard sequence to sequence autoencoder inspired by [27] that we customize for embedding graphs. We use $h_t^{enc}$ to denote the hidden vector at time step $t$ in $LSTM_{enc}$ and $h_t^{dec}$ to denote the hidden vector at time step $t$ in $LSTM_{dec}$. We define shorthand for Eq. 1 as follows:

$$h_t^{enc} = LSTM_{enc}(Emb(v_t), h_{t-1}^{enc}) \tag{4}$$

where $Emb(v_t)$ takes the role of $x_t$ in Eq. 1. The hidden vector at the last time step $h_{last}^{enc} \in \mathbb{R}^d$ denotes the representation of the input sequence, and is used as the hidden vector of the decoder at its first time step:

$$h_0^{dec} = h_{last}^{enc} \tag{5}$$

The last cell vector of the encoder is copied over in an analogous way. Then each decoder hidden vector $h_t^{dec}$ is computed based on the hidden vector and vertex embedding from the previous time step:

$$h_t^{dec} = LSTM_{dec}(Emb(v_{t-1}), h_{t-1}^{dec}) \tag{6}$$

The decoder uses $h_t^{dec}$ to predict the next vertex embedding $\overline{Emb(v_t)}$ as in Eq. 2. We have two different loss functions to test with this model. First, we consider the node embeddings fixed and compute a loss based on the difference between the predicted vertex embedding $\overline{Emb(v_t)}$ and the true one $Emb(v_t)$. Second, we update the node embeddings in addition to the model parameters using a cross entropy function. We discuss training in Section 7. For $Emb(v_0)$, we use a vector of all zeroes.

**S2S-AE-PP**: In the previous model, $LSTM_{dec}$ predicts the embedding of the vertex at time step $t$ using $h_{t-1}^{dec}$ and the true vertex embedding at time step $t-1$, $Emb(v_{t-1})$. However, this may enable the decoder to rely too heavily on the previous true vertex in the path, thereby making it easier to reconstruct the input and reducing the need for the encoder to learn an effective representation of the input sequence. We consider a variation in which we use the previous predicted (**PP**) vertex $Emb(\overline{v_{t-1}})$ instead of the previous true one:

$$h_t^{dec} = LSTM_{dec}(Emb(\overline{v_{t-1}}), h_{t-1}^{dec}) \tag{7}$$

This forces the encoder and decoder to work harder to respectively encode and decode the graph sequences. This variation is related to scheduled sampling [3], in which the training process is changed gradually from using true previous symbols to using predicted previous symbols more and more during training.

**S2S-AE-PP-WL1,2**: This model is similar to S2S-AE-PP except that, for each node in the sequence, we use two embeddings corresponding to different stopping iterations of the WL algorithm. We use $x_{1_t}$ to denote the embedding of the label produced by one iteration of WL and $x_{2_t}$ for that produced by two iterations of WL. Eq. 1 is modified to receive both as inputs; e.g., the first line of Eq. 1 becomes:

$$i_t = \sigma(W_{1_t} x_{1_t} + W_{2_t} x_{2_t} + U_i h_{t-1} + K_i c_{t-1} + b_i)$$

The other equations are changed analogously. The embeddings for both the WL1 and WL2 labels are learned.

**S2S-N2N-PP**: This model is a "neighbors-to-node" (N2N) prediction model and uses random walks as graph sequences. That is, each item in the input sequence is the set of neighbors (their embeddings are averaged) for the corresponding node in the output sequence:

$$h_t^{enc} = LSTM_{enc}(\underset{v_i \in nbrs(v_t)}{Avg}(Emb(v_i)), h_{t-1}^{enc}) \tag{8}$$

where $nbrs(v)$ returns the set of neighbors of $v$ and we predict the nodes in the random walk via the decoder as in Eq. 7. Unlike the other models, this model is not an autoencoder because the input and output sequences are not the same.

## 7 TRAINING

Let $S$ be a sequence training set generated from a set of graphs. The representations of the sequences $s \in S$ are computed using the encoders described in the previous section. We use two different loss functions to train our models: *squared error* and *categorical cross-entropy*. The goal is to minimize the following loss functions, summed over all examples $s \in S$, where $s : v_1, \ldots, v_{|s|}$.

### 7.1 Squared Error

We used the squared error (SE) loss function for the WL embeddings that are fixed and are not considered as trainable parameters of the model. We include a nonlinear transformation to estimate the embedding of the $t$th vertex in $s$ using the hidden vector of the decoder at time step $t$:

$$\overline{Emb(v_t)} = \text{ReLU}(h_t^{dec} W + b) \tag{9}$$

where ReLU is the rectified linear unit activation function and $W$ and $b$ are additional parameters.

Given the predicted vertex embeddings for the sequence $s$, the squared error loss function computes the average of the element-wise squared differences between the input and output sequences:

$$loss_{SE}(s) = \frac{1}{|s|}\sum_{t=1}^{|s|}\left\|\overline{Emb(v_t)} - Emb(v_t)\right\|_2^2 \quad (10)$$

## 7.2 Categorical Cross Entropy

We use the categorical cross entropy (CE) loss function for experiments in which we update the vertex embeddings during training. We predict the $t$th vertex as follows:

$$\overline{v_t} = \underset{l \in L}{\text{argmax}}(h_t^{dec}.Emb(l)) \quad (11)$$

where $L$ is the set of labels and . denotes dot product. The loss computes the categorical cross entropy between the input embeddings and the predicted output embeddings:

$$loss_{CE}(s) = -\sum_{t=1}^{|s|}\log p(\overline{v_t} = l) \quad (12)$$

where $l$ denotes the true label of vertex $v_t$, and the predicted probability of the true label is computed as follows:

$$p(\overline{v_t} = l) = \frac{e^{h_t^{dec}.Emb(l)}}{\sum_{l' \in L} e^{h_t^{dec}.Emb(l')}} \quad (13)$$

# 8 EXPERIMENTS

In this section, we evaluate our representation learning procedure on both labeled and unlabeled graphs. We use our learned representations for the task of graph classification using several benchmark datasets and compare the accuracy of our models to state-of-the-art approaches.

## 8.1 Datasets

For our classification experiments with labeled graphs, we use six datasets of bioinformatics graphs. For unlabeled graphs, we use six datasets of social network graphs.

**Labeled graphs**: The bioinformatics benchmarks include several well known datasets of labeled graphs. MUTAG [10] is a dataset of mutagenic aromatic and heteroaromatic nitro compounds. PTC [42] contains several compounds classified in terms of carcinogenicity for female and male rats. Enzymes [5] includes 100 proteins from each of the 6 Enzyme Commission top level enzymes classes. Proteins [5] consists of graphs classified into enzymes and non-enzymes groups. Nci1 and Nci109 [45] are two balanced subsets of chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines respectively.

**Unlabeled graphs**: We use several datasets developed by [47] for unlabeled graph classification. COLLAB is a collaboration dataset where each network is generated from ego-networks of researchers in three research fields. Networks are classified based on research field. IMDB-BINARY and IMDB-MULTI include ego-networks for film actors/actresses from various genres on IMDB, and networks are classified by genre. The Reddit datasets include graphs that show the relations between users extracted from different subreddits. The task is to identify the community each graph belongs to.

## 8.2 Baselines

We compare to several well known graph kernels: shortest path kernel (SPK) [6], random walk kernel (RWK) [16], graphlet kernels (GK) [39], and Weisfeiler-Lehman subtree kernel (WLSK) [38]. We report the results obtained by [47] using these methods on all datasets. We also compare to five recent approaches: Deep Graph Kernels (DGK) [47], the convolutional neural network method (PSCN) of [34], Deep Graph Convolutional Neural Network (DGCNN) [48], the optimal-assignment Weisfeiler-Lehman (WL-OA) kernel [23] and Graph2vec [32]. We also compare to a graph representation method based on node2vec [18]; we use it to learn node embeddings and average them for all nodes in a graph. We report the best results from prior work on each dataset, choosing the best from multiple configurations of their methods.

## 8.3 Experimental setup

For each dataset, we perform 10 fold cross-validation on its graph representations using a C-SVM classifier from LIBSVM [9] with a radial basis kernel. Each 10 fold cross-validation experiment is repeated 10 times (with different random splits) and we report average accuracies and standard deviations. We use nested cross-validation for tuning the regularization and kernel hyperparameters.

## 8.4 Hyperparameter selection

We treat three labeled bioinformatics graph datasets (MUTAG, PTC, Enzymes) and two unlabeled social network datasets (IMDB-BINARY and REDDIT-BINARY) as development datasets for tuning certain high-level decisions and hyperparameters of our approach, though we generally found results to be robust across most values. Figure 5 shows the effect of dimensionality of the graph representation, showing robustness across values larger than 50; we use 100 in all experiments below. The dashed lines in the figure show accuracy when the vertex embeddings are fixed and the solid lines when the vertex embeddings are updated during training. We use SE (Sec. 7.1) when the vertex embeddings are fixed and CE (Sec. 7.2) when we update the vertex embeddings during training. CE consistently outperforms SE and we use CE for all remaining experiments. We use AdaGrad [12] with learning rate 0.01 and mini-batch size 100.

In the experiments that used BFS for sequence generation, we only consider vertices that are at most 1 edge from the starting node. In some cases, this still leads to extremely long sequences for nodes with many neighbors. We convert these to multiple sequences such that each has maximum length 10. When doing so, we still prepend the initial starting vertex to each of the truncated partial sequences.

When using random walks, we generate multiple random walks from each vertex. We compared random walk lengths among $\{3, 5, 10, 15, 20\}$. Figure 6 shows robust performance with length 5 across datasets and we use this length below.

## 8.5 Comparing models, embeddings, and sequences

Figures 2, 3 and 4 show the results of the classification task on the development labeled graphs for our models, with varying types of graph sequences (RW, BFS, SP) and label embeddings (Original labels, WL1, WL2).

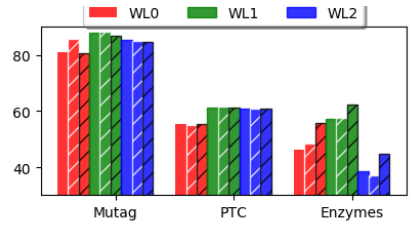


**Figure 2: WL embedding (solid: RW, white hatch: BFS, black hatch:SP)**

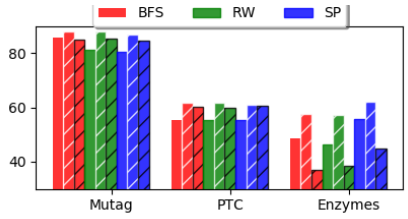


**Figure 3: Type of sequences (solid: WL0, white hatch: WL1, black hatch:WL2)**

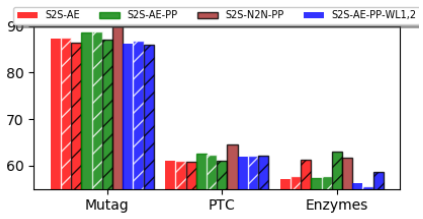


**Figure 4: Models (solid: RW, white hatch: BFS, black hatch:SP)**

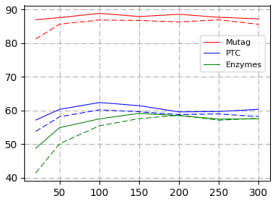


**Figure 5: Representation dimensionality ($x$ axis) vs classification accuracy ($y$ axis).**

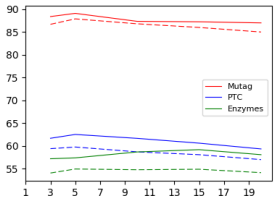


**Figure 6: Random walk length ($x$ axis) vs graph classification accuracy ($y$ axis).**

Figure 2 shows that using WL labels improves accuracy substantially compared to using the original labels. We show the average of accuracies of the two autoencoders S2S-AE and S2S-AE-PP (rather than all four, since S2S-N2N-PP does not have the full range of substructures and S2S-AE-PP-WL1,2, uses both WL labels together, thus, not comparable). There is a large gap between accuracies of the labels generated by the first iteration of the WL algorithm and original labels across all models and sequence types. This provides strong evidence that WL can enrich the provided vertex labels in a way that improves graph representations by capturing additional local node structure.

The number of unique labels increases with more iterations of WL. Although distinctive labels provide more information about the local structure of nodes, using WL labels with larger iterations does not necessarily lead to better graph representations. For example, the accuracy of Enzymes shows a significant drop when using the second iteration of the WL algorithm. We believe that the reason of such a sharp drop in this dataset can be explained by the graphs' label entropy [26]. Given a graph $G$ and a set of labels $L$, the label entropy of $G$ is defined as $H(G) = -\sum_{l \in L} p(l) \log p(l)$. The average label entropy in each dataset is shown in Table 1. The entropy of the Enzymes dataset increases more than MUTAG and PTC from the first iteration to the second iteration of WL algorithm.

Moreover, Figure 3 shows that the accuracies of classification using different types of sequences are very close to each other while using Weisfeiler-Lehman labels. The difference between types of sequences is more evident with the original labels. In Enzymes, shortest paths clearly obtain better accuracies than random walks and BFS, regardless of the type of embeddings that are used. For the same type of graphs, Borgwardt and Kriegel [6] similarly observed that their shortest path kernel was better than walk-based kernels. We suspect that the reason is related to the clustering coefficient, a popular metric in network analysis. The clustering coefficient is the number of triangles connected to node $v$ over the number of connected triples centered on node $v$. Having many triangles in the ego-network of node $v$ may cause the tottering problem in a walk traversing node $v$ and may generate less discriminative BFS sequences from that ego-network. Shortest paths prevent tottering and capture global graph structure. BFS sequences mainly consider the local topological features of a graph, and random walks collect a representative sample of nodes rather than of topology [24]. Fortunately, using the WL labels we can reduce the effect of sequence type in most settings.

Figure 4 shows the comparison between different unsupervised models using WL1. Model S2S-AE-PP is better than Model S2S-AE in nearly all cases. As we conjectured above, Model S2S-AE-PP may force the encoder representation to capture the entire sequence since the decoder has less assistance during reconstruction. Model S2S-N2N-PP obtains higher accuracy in almost all datasets, showing the benefit of capturing both local neighborhoods and longer paths (via the use of random walks). With S2S-AE-PP-WL1,2, we only observe improvements over the other S2S-AE models on the PTC dataset. With S2S-AE-PP-WL1,2, we only observe improvements over the other S2S-AE models on the PTC dataset. This suggests that adding WL labels with different iterations could not provide more informative graph representations, regardless of the type of sequences. The reason could be due to the fact that in this model we add too many parameters for WL labels and our model is not able to learn meaningful embeddings for these labels, which leads to poor graph representation.

## 8.6 Comparison to state-of-the-art

We compare S2S-N2N-PP to the state-of-the-art in Table 1. While other methods outperform ours in certain datasets, we exceed all prior results on PTC, Enzymes, and Proteins. Considering that none of the previous work can outperform the others in all datasets, we use an average ranking measure to compare the performance of all approaches to one another. Our approach shows robustness, achieving the first rank among all methods under this measure.

Table 2 compares our method to prior work on the unlabeled graph datasets. Our approach establishes new state-of-the-art accuracies on all dataset except REDDIT-BINARY.

While our approach performs best when using Weisfeiler-Lehman, this is not solely due to the Weisfeiler-Lehman algorithm itself.

**Table 1: Classification accuracies for labeled graph datasets.**

| Datasets | MUTAG | PTC | Enzymes | Proteins | Nci1 | Nci109 | |
|---|---|---|---|---|---|---|---|
| # Graphs | 188 | 344 | 600 | 1113 | 4110 | 4127 | |
| EntropyWL1 | 2.72 | 4.03 | 5.45 | 5.54 | 4.21 | 4.22 | |
| EntropyWL2 | 4.65 | 7.09 | 12.60 | 13.26 | 8.23 | 8.25 | |
| ClusteringCoef | 0 | 0.0025 | 0.453 | 0.51 | 0.003 | 0.003 | |
| # Labels | 7 | 19 | 2 | 3 | 37 | 38 | |
| # Classes | 2 | 2 | 6 | 2 | 2 | 2 | Avg Ranking |
| SPK (BK05) | 85.2±2.4 | 58.2±2.4 | 40.1±1.5 | 75.1±0.5 | 73.0±0.2 | 73.0±0.2 | 4.5 |
| RWK (G+03) | 83.7±1.5 | 57.8±1.3 | 24.2±1.6 | 74.2±0.4 | — | — | — |
| GK (S+09) | 81.7±2.1 | 57.2±1.4 | 26.6±0.9 | 71.7±0.5 | 62.3±0.2 | 62.6±0.1 | 5.5 |
| WLSK (S+11) | 80.7±3.0 | 57.0±2.0 | 53.1±1.1 | 72.9±0.5 | 80.1±0.5 | 80.2±0.3 | 5.5 |
| node2vec (G+16) | 82.01±1.0 | 55.60±1.4 | 19.42±2.3 | 70.76±1.2 | 61.91±0.3 | 61.53±0.9 | 6.5 |
| DGK (YW15) | 87.4±2.7 | 60.1±2.5 | 53.4±0.9 | 75.7±0.5 | 80.3±0.4 | 80.3±0.3 | 2.8 |
| PSCN (N+16) | **92.6** | 62.3 | — | 75.9 | 78.6 | — | — |
| WL-OA (K+16) | 86.0±1.7 | 63.6±1.5 | 59.9±1.1 | 76.4±0.4 | **86.1±0.2** | **86.3±0.2** | 1.8 |
| graph2vec (N+17) | 83.15±9.2 | 60.17±6.9 | — | 73.30±2.0 | 73.22±1.9 | 74.26±1.5 | — |
| DGCNN (Z+18) | 85.83±1.6 | 58.59±2.4 | — | 75.54±0.9 | 74.44±0.4 | — | — |
| S2S-N2N-PP | 89.86±1.1 | **64.54±1.1** | **63.96±0.6** | **76.61**±0.5 | 83.72±0.4 | 83.64±0.3 | **1.3** |

**Table 2: Classification accuracies for unlabeled graph datasets.**

| Dataset | COLLAB | IMDB-BINARY | IMDB-MULTI | REDDIT-BINARY | REDDIT-MULTI-5k | REDDIT-MULTI-12k | |
|---|---|---|---|---|---|---|---|
| # Graphs | 5000 | 1000 | 1500 | 2000 | 5000 | 11929 | |
| # Classes | 3 | 2 | 3 | 2 | 5 | 11 | Avg Ranking |
| node2vec (G+16) | 56.06±0.2 | 50.17±0.9 | 36.02±0.7 | 71.31±2.2 | 33.11±1.7 | 23.62±0.3 | 4.0 |
| DGK (YW15) | 73.0±0.2 | 66.9±0.5 | 44.5±0.5 | 78.0±0.3 | 41.2±0.1 | 32.2±0.1 | 2.8 |
| PSCN (N+16) | 72.6±2.1 | 71.0±2.2 | 45.2±2.8 | 86.3±1.5 | 49.1±0.7 | 41.3±0.4 | 2.1 |
| WL-OA (K+16) | 80.7±0.1 | — | — | 89.3±0.3 | — | — | — |
| DGCNN (Z+18) | 73.76±0.49 | 70.03±0.86 | 47.83±0.85 | — | — | — | — |
| S2S-N2N-PP | **81.75±0.8** | **73.8±0.7** | **51.19±0.5** | 86.50±0.8 | **52.28±0.5** | **42.47±0.1** | **1.0** |

Some methods shown in Tables 1 and 2 also use Weisfeiler-Lehman (graph2vec, WLSK, PSCN, and WL-OA) yet our approach outperforms them on average.

## 9 CONCLUSIONS

We proposed an unsupervised approach for learning representations of graphs using sequence-to-sequence LSTM architectures. We trained using sequences generated by random walks, shortest paths, and breadth-first search. Our experiments demonstrate that our graph representations can increase the accuracy of graph classification tasks on both labeled and unlabeled graphs, achieving to our knowledge the best results on several datasets considered.

## REFERENCES


[1] Leman Akoglu, Mary McGlohon, and Christos Faloutsos. 2010. Oddball: Spotting anomalies in weighted graphs. In *PAKDD*.

[2] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. 2009. Gephi: An Open Source Software for Exploring and Manipulating Networks. In *AAAI ICWSM*.

[3] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *NIPS*.

[4] Michele Berlingerio, Danai Koutra, Tina Eliassi-Rad, and Christos Faloutsos. 2012. NetSimile: a scalable approach to size-independent network similarity. *arXiv* (2012).

[5] K. Borgwardt, C. Ong, S. Schönauer, S. Vishwanathan, A. Smola, and H. Kriegel. 2005. Protein function prediction via graph kernels. *Bioinformatics* 21 (2005).

[6] Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *ICDM*.

[7] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2013. Spectral Networks and Locally Connected Networks on Graphs. *CoRR* (2013).

[8] Horst Bunke. 2000. Graph matching: Theoretical foundations, algorithms, and applications. In *Vision Interface*.

[9] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM TIST* 2 (2011).

[10] A. Debnath, R. Lopez de Compadre, G. Debnath, A. Shusterman, and C. Hansch. 1991. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. *Journal of medicinal chemistry* 34 (1991).

[11] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. *arXiv* (2016).

[12] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR* (2011).

[13] David Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*.

[14] Andreas Fischer, Ching Y. Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. 2013. A Fast Matching Algorithm for Graph-Based Handwriting Recognition. In *GbRPR*.

[15] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5 (1962).

[16] Thomas Gärtner, Peter Flach, and Stefan Wrobel. 2003. On graph kernels: Hardness results and efficient alternatives. In *COLT*.

[17] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *CoRR* (2017).

[18] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*.

[19] David Haussler. 1999. *Convolution kernels on discrete structures.* Technical Report. Citeseer.

[20] M. Henaff, J. Bruna, and Y. LeCun. 2015. Deep convolutional networks on graph-structured data. *arXiv* (2015).

[21] Geoffrey E Hinton and Richard S Zemel. 1993. Autoencoders, minimum description length, and Helmholtz free energy. In *NIPS*.

[22] S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural computation* 9 (1997).
[23] Nils M. Kriege, Pierre-Louis Giscard, and Richard Wilson. 2016. On Valid Optimal Assignment Kernels and Applications to Graph Classification. In *NIPS*.
[24] Maciej Kurant, Athina Markopoulou, and Patrick Thiran. 2011. Towards unbiased BFS sampling. *IEEE Journal on Selected Areas in Communications* 29 (2011).
[25] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *ICML*.
[26] Geng Li, Murat Semerci, Bulent Yener, and Mohammed J Zaki. 2011. Graph classification via topological and label attributes. In *MLG*.
[27] J. Li, M. Luong, and D. Jurafsky. 2015. A hierarchical neural autoencoder for paragraphs and documents. In *ACL*.
[28] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. 2015. Gated Graph Sequence Neural Networks. *arXiv* (2015).
[29] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *JMLR* (2008).
[30] Owen Macindoe and Whitman Richards. 2010. Graph comparison using fine structure analysis. In *SocialCom*.
[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[32] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning Distributed Representations of Graphs. In *MLG*.
[33] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* (2003).
[34] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. In *ICML*.
[35] B. Perozzi, R. Al-Rfou, and S Skiena. 2014. DeepWalk: Online learning of social representations. In *KDD*.
[36] Kaspar Riesen, Xiaoyi Jiang, and Horst Bunke. 2010. Exact and inexact graph matching: Methodology and applications. In *Managing and Mining Graph Data*.
[37] F. Scarselli, M. Gori, C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20 (2009).
[38] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-Lehman graph kernels. *JMLR* (2011).
[39] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M Borgwardt. 2009. Efficient graphlet kernels for large graph comparison.. In *AISTATS*, Vol. 5.
[40] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *NIPS*.
[41] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. In *WWW*.
[42] H. Toivonen, A. Srinivasan, R. King, S. Kramer, and C. Helma. 2003. Statistical evaluation of the predictive toxicology challenge. *Bioinformatics* 19 (2003).
[43] Bernadette CM Van Wijk, Cornelis J Stam, and A. Daffertshofer. 2010. Comparing brain networks of different size and connectivity density using graph theory. *PloS one* 5 (2010).
[44] S. Vishwanathan, N. Schraudolph, R. Kondor, and K. Borgwardt. 2010. Graph kernels. *JMLR* 11 (2010).
[45] Nikil Wale, Ian A. Watson, and George Karypis. 2008. Comparison of descriptor spaces for chemical compound retrieval and classification. *KAIS* 14 (2008).
[46] B. Weisfeiler and A. Lehman. 1968. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia* (1968).
[47] Pinar Yanardag and SVN Vishwanathan. 2015. Deep graph kernels. In *KDD*.
[48] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *AAAI*.