

# SCALABLE REPRESENTATION LEARNING WITH INVARIANCES

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Changping Meng

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

**THE PURDUE UNIVERSITY GRADUATE SCHOOL**  
**STATEMENT OF DISSERTATION APPROVAL**

Dr. Jennifer Neville, Co-Chair

Department of Computer Science and Statistics

Dr. Bruno Ribeiro, Co-Chair

Department of Computer Science

Dr. Dan Goldwasser

Department of Computer Science

Dr. Yexiang Xue

Department of Computer Science

**Approved by:**

Dr. Clifton W Bingham

Head of the Computer Science Graduate Program

## ACKNOWLEDGMENTS

First and foremost, I really appreciate my major professors, Professor Jennifer Neville and Professor Bruno Ribeiro. With their help and advice, I have learned a lot and accomplished more than I could have imagined. Professor Neville has consistently instructed me to think independently and work dedicatedly. When encountering frustrations, she patiently guided me through the difficulties and filled me with courage. I have benefited immensely from Professor Bruno Ribeiro's intelligence, passion and enthusiasm for research.

I also want to thank my committee members, Professors Dan Goldwasser, Yexiang Xue. They lent me their expertise to provide context, feedback, and suggestions as I pursued this research.

I am also grateful to the members of Jen's lab and Bruno's lab. Since all of them provided generous help and support, I want to thank them alphabetically: Leonardo Cotta, Mahak Goindani, Guilherme Gomes, Jianfei Gao, Mengyue Hang, Mayank Kakodar, Ryan Murphy, Bala Srinivasan, Leonardo Teixeira, Yi-Yu Lai, Ying-Chun Lin, Chandra Mouli, Hogun Park, Jiasen Yang, Giselle Zeno, Yangze Zhou.

I am so grateful to my elder sister Lili Meng for her help and guidance. Her passion for research inspired me to start and go through this Phd journey. I also want to thank my parents Shilin Meng and Aichun Tian their unwavering support.

Last but not the least, I want to thank my wife, Ziyun Ding for her love and support. Marrying her is one of my best decisions made at Purdue University.

## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	vi
LIST OF FIGURES . . . . .	vii
ABSTRACT . . . . .	ix
1 INTRODUCTION . . . . .	1
1.1 Research Questions . . . . .	3
1.2 Main Hypothesis and Proposed Research . . . . .	4
1.3 Contributions . . . . .	5
1.4 Thesis Organization . . . . .	7
2 MODELS FOR GRAPH, SET AND SETS-OF-SETS . . . . .	8
2.1 Definitions . . . . .	8
2.2 Related work . . . . .	10
2.2.1 Graphs . . . . .	10
2.2.2 Sets . . . . .	12
2.2.3 Set of Sets . . . . .	13
3 SUBGRAPH PATTERN NEURAL NETWORK . . . . .	15
3.1 Introduction . . . . .	15
3.2 Heterogeneous Subgraph Prediction . . . . .	17
3.2.1 Relationship with Convolutional Neural Networks . . . . .	25
3.3 Results . . . . .	26
3.3.1 Empirical Results . . . . .	26
4 PERMUTATION INVARIANT FUNCTIONS FOR SET . . . . .	41
4.1 Introduction . . . . .	41
4.2 Representation Learning of Variable-size Sets . . . . .	43
4.2.1 Set Inputs . . . . .	43
4.2.2 Set Representation Functions . . . . .	43
4.3 Invariant Neural Network Architectures . . . . .	44
4.3.1 Ideal Representation Routing . . . . .	44
4.4 Better Set Representation Architectures . . . . .	49
4.4.1 Existing Graph Topologies . . . . .	50
4.4.2 Build Computational Graph with Self-attention . . . . .	50
4.4.3 Graph Node Structure . . . . .	51
4.4.4 Stochastic Optimization . . . . .	53

	Page
4.5 Experiments . . . . .	55
4.5.1 Arithmetic Tasks on Sequence of Integers . . . . .	56
4.5.2 Vertex Classification . . . . .	58
4.5.3 Natural Language Processing Tasks . . . . .	60
4.5.4 Point Cloud Classification . . . . .	60
4.5.5 Reasoning tasks . . . . .	62
5 PERMUTATION INVARIANT FUNCTIONS FOR SET-OF-SETS . . . . .	64
5.1 Introduction . . . . .	64
5.2 Inductive Embeddings of Set-of-Sets . . . . .	65
5.2.1 Inductive SoS Embeddings . . . . .	66
5.3 Learning Inductive SoS Embeddings . . . . .	69
5.3.1 The HATS Architecture . . . . .	70
5.3.2 Stochastic Optimization for HATS . . . . .	75
5.4 Experiments . . . . .	77
5.4.1 Simple Arithmetic Tasks . . . . .	80
5.4.2 Computing the Adamic/Adar Index . . . . .	82
5.4.3 Subgraph Hyperlink Prediction . . . . .	83
5.4.4 Point-Cloud SoS Classification . . . . .	84
6 SUMMARY AND FUTURE DIRECTIONS . . . . .	88
6.1 Conclusion . . . . .	88
6.2 Future Directions . . . . .	90
6.2.1 Subgraph Collective Inference . . . . .	90
6.2.2 Increase Subgraph Counting Efficiency . . . . .	90
6.2.3 Random Graph Model for Set . . . . .	91
6.2.4 Apply the proposed Set model on the Set-of-Sets tasks . . . . .	91
6.2.5 Temporal Graph Classification with Set of Sets . . . . .	91
BIBLIOGRAPHY . . . . .	91

## LIST OF TABLES

Table	Page
3.1 Max Area Under Curve (AUC) scores of SPNN against baselines. . . . .	37
3.2 Time to sample 1000 examples+training time. . . . .	38
3.3 Max Area Under Curve (AUC) scores of SPNN against baselines. . . . .	39
3.4 Time to sample 1000 examples + learning time. . . . .	40
4.1 Computational graph properties of methods for set with $n$ elements . . . .	46
4.2 Diameters and average path length under different prune rate for graph with 100 nodes. . . . .	58
4.3 Number of Ops and Parameters for Arithmetic Tasks <i>Unique Count</i> per minibatch in one iteration. . . . .	59
4.4 MicroF1 score (standard deviations) using different aggregation functions in a GNN – GRAPH SAGE. . . . .	59
4.5 Accuracy (standard deviations) for two GLUE classification tasks. . . . .	60
4.6 Accuracy (standard deviations) on Point Cloud label classification and label counting. . . . .	61
4.7 Accuracy (standard deviations) for subset sum problem: Given a set with $n$ integers from $[-200, 200]$ , decide whether it contains a subset which sums to 0. . . . .	61
5.1 Implementation details for various tasks. . . . .	80
5.2 Summary of network dataset statistics. . . . .	80
5.3 Prediction accuracies for <i>interactive</i> arithmetic tasks for different member- set size $m$ . . . . .	81
5.4 Accuracies for <i>aggregative</i> tasks with different member-set size $m$ . . . . .	82
5.5 Predicting Adamic/Adar-index on Cora. . . . .	83
5.6 Subgraph hyperlink prediction accuracies for different subgraph size $m$ . . .	84
5.7 Point-cloud classification results. . . . .	87

## LIST OF FIGURES

Figure	Page
1.1 Examples of datasets which are better modeled as graphs or sets . . . . .	2
1.2 Examples of variance: (a) Graph Isomorphism and (b) Set Permutations . .	2
1.3 Methods to handle invariance. (a) Pool and count subgraphs under same patterns. (b) Use Janossy Pooling [15] as a wrapper. . . . .	4
3.1 (1) Illustration of the training in a citation network with (A)uthors, (T)opics, (V)enues. At the top is the graph evolution $G_1$ to $G_2$ , whose induced subgraphs are used as training data to predict the evolution of subgraphs in $G_2$ to $G_3$ ; below $\mathcal{Y}^3$ shows 3-node subgraph patterns partitioned into two classes. (2) Labels for subgraph evolution. The appearance of two links are considered as label $y_1^3$ . All other subgraphs are assigned label $y_2^3$ . (3) Features for $U = V_2$ . $A_2, T_3$ under the patterns (4) SPNN model. . . . .	19
3.2 ROC curves (True Pos $\times$ False Pos): DBLP and Friendster tasks. . . . .	25
3.3 Learning curves (AUC $\times$ Training Size) w/shaded 95% confidence intervals. . . .	26
3.4 (DBLP task) Pattern layer weight difference between Class 1 (whether both dashed links appear at time $t+1$ ) and Class 2 (everything else) for pattern $F_j^\square$ . Pattern $F_4^\square$ , when the author has published in a topic related to the venue, strongly predicts the appearance of both links. Pattern $F_2^\square$ , when a co-author has published at the venue and topic of interest but not the author, strongly predicts the absence of the joint links. . . . .	27
3.5 Sequence Graph Learning curves (AUC $\times$ Training Size) compared to logistic regression and MLP (w/shaded 95% conf. intv.). . . . .	31
3.6 ROC curves (True Pos $\times$ False Pos): Facebook and WordNet tasks. . . . .	31
3.7 Facebook and WordNet Prediction tasks . . . . .	32
3.8 Learning curves (AUC $\times$ Training Size) w/shaded 95% confidence intervals for dynamic Facebook and WordNet. . . . .	33
3.9 Learning curves (AUC $\times$ Training Size) of SPNN against competing methods in Static Graph (w/shaded 95% conf. intv.). . . . .	34
3.10 Learning curves comparing SPNN to logistic regression and MLP in Static Graph. . . . .	35

Figure	Page
3.11 ROC curves (True Pos $\times$ False Pos): Facebook, WordNet tasks in Static Graph. . . . .	35
3.12 ROC curves (True Pos $\times$ False Pos): Facebook, WordNet tasks in manually generated dynamic graphs. . . . .	36
4.1 Computational graph examples. . . . .	45
4.2 Weighted DAG resulting from the self-attention adjacency matrix $\mathbf{G}$ . . . .	52
4.3 Different Types of Hybrid Node can be adopted in the computational graph.	52
4.4 Double Count: check duplicates for sequence length $n$ with $1.5 \times n$ vocabulary size. . . . .	58
4.5 Unique Count: count of unique elements for sequence length $n$ with $n$ vocabulary size. . . . .	58
4.6 Accuracy of <i>Self-Attention GRU</i> when randomly pruning edges of computational graph at different prune rate. . . . .	58
5.1 HATS architecture for SoS inputs. . . . .	75
5.2 Visualization of point-cloud tasks. . . . .	86
5.3 Anomaly detection accuracies for varying point-cloud size $m$ . . . . .	86
5.4 Anomaly detection accuracies for varying $k$ -ary dependency. . . . .	86



## ABSTRACT

Meng, Changping PhD, Purdue University, May 2020. Scalable Representation Learning with Invariances . Major Professor: Jennifer Neville, Bruno Ribeiro.

In many complex domains, the input data are often not suited for the typical vector representations used in deep learning models. For example, in knowledge representation, relational learning, and some computer vision tasks, the data are often better represented as graphs or sets. In these cases, a key challenge is to learn a representation function which is invariant to permutations of set or isomorphism of graphs.

In order to handle graph isomorphism, this thesis proposes a subgraph pattern neural network with invariance to graph isomorphisms and varying local neighborhood sizes. Our key insight is to incorporate the unavoidable dependencies in the training observations of induced subgraphs into both the input features and the model architecture itself via high-order dependencies, which are still able to take node/edge labels into account and facilitate inductive reasoning.

In order to learn permutation-invariant set functions, this thesis shows how the characteristics of an architecture’s computational graph impact its ability to learn in contexts with complex set dependencies, and demonstrate limitations of current methods with respect to one or more of these complexity dimensions. I also propose a new *Self-Attention GRU* architecture, with a computation graph that is built automatically via self-attention to minimize average interaction path lengths between set elements in the architecture’s computation graph, in order to effectively capture complex dependencies between set elements.

Besides the typical set problem, a new problem of representing sets-of-sets (SoS) is proposed. In this problem, multi-level dependence and multi-level permutation in-

variance need to be handled jointly. To address this, I propose a hierarchical sequence-attention framework (HATS) for inductive set-of-sets embeddings, and develop the stochastic optimization and inference methods required for efficient learning.

## 1 INTRODUCTION

Deep learning has been successfully applied to a myriad of applications, in which the input data typically involves fixed-length vectors. Examples include image recognition, video classification, sentiment analysis, among many others. A critical aspect of vector representations is that the position of elements matters. This ordering is needed for many classic deep learning tasks. For instance, in image or video recognition, the ordering in vectors needs to align with the spatial orientation. For NLP or speech recognition, the ordering in the vector data represents the sequential information from past to future.

However, some complex data are not well-suited for vector representations since the “ideal” ordering for use in a vector is either non-trivial to compute or does not exist at all. Examples include social networks, physical networks, LIDAR readings, and point clouds, as shown in Figure 1.1. These datasets are often better represented as *graphs* or *sets* for a wide range of applications such as recommendations [1, 2], object detection [3, 4], relational analysis [5, 6, 7, 8, 9], logical reasoning [10, 11], and scene understanding [8, 12]. As there is no natural ordering to the elements in a graph or set, any model must jointly learn functions over all the elements in order to capture relational dependencies.

Initial work on learning neural-network models for heterogeneous inputs often transformed the data into variable-length sequences [5, 7, 13]. However, these methods learn models (*i.e.*, *embedding functions*) that are *permutation-sensitive*. In other words, the output of the learned model (*i.e.*, embedding) depends on the order chosen for the input vector. For sets, the same elements permuted in different orders will be embedded to different points in space. The same issue also exists in graphs where isomorphic graphs are embedded in different positions as shown in Figure 1.2.

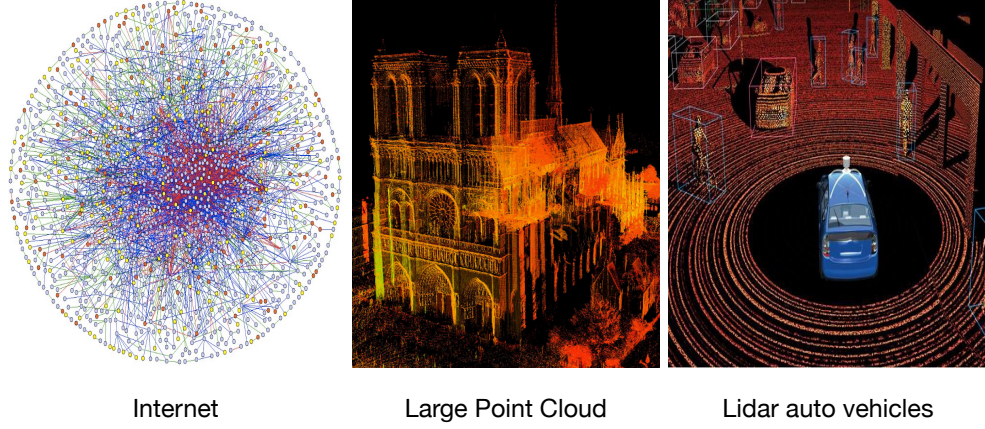


Figure 1.1.: Examples of datasets which are better modeled as graphs or sets

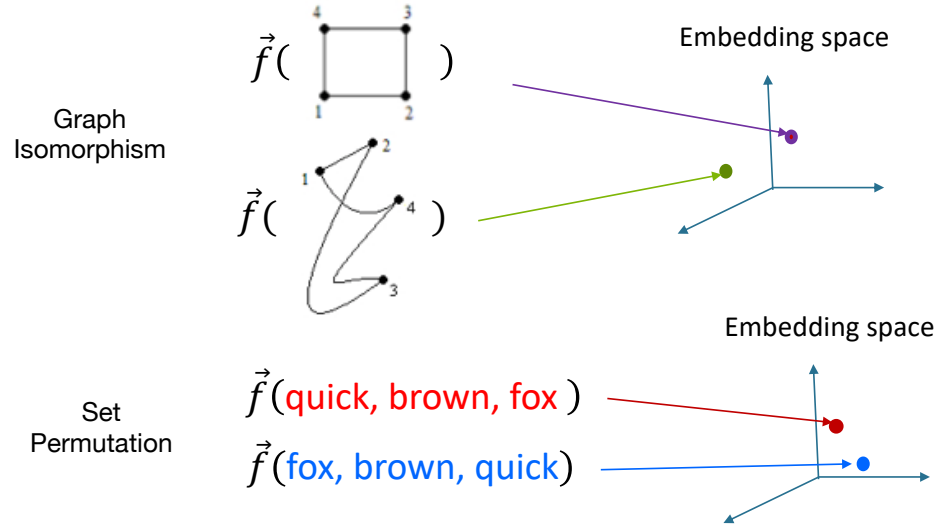


Figure 1.2.: Examples of variance: (a) Graph Isomorphism and (b) Set Permutations

An *invariant* (i.e. transformation insensitive) function is a function whose output remains unchanged when a certain transformation is applied to the input objects. The transformation can be relabelling of graphs (e.g. to produce isomorphic graphs) or permutations of sets. For the same examples in Figure 1.2, isomorphism-*invariant* functions will embed the isomorphic graphs into the *same* position in space. Sim-

ilarly, the embedding of a set under different permutations should be equal for a permutation-invariant function.

Recurrent neural networks, which are commonly used in sequence-to-sequence models, are not invariant to permutations in the input sequence (e.g.  $\vec{f}(\begin{smallmatrix} \text{red} & \text{blue} \\ \text{blue} & \text{red} \end{smallmatrix}) \neq \vec{f}(\begin{smallmatrix} \text{blue} & \text{red} \\ \text{red} & \text{blue} \end{smallmatrix})$ ). Indeed, [14] showed that the order of the input sequence could significantly affect the quality of the learned model.

A more principled approach to learning functions is to learn an embedding function that is invariant to *permutation* and which could be used to model the input themselves directly. More recent work has focused on developing principled approaches to learning these set representations [4, 14, 15, 16, 17]. The key contribution of these works has been to provide scalable methods that can learn inductive embeddings which are provably invariant to the ordering of the input. However, the models proposed in this thesis will provide more accurate predictions through the development of better architectures that capture high-order dependencies more effectively while maintaining permutation invariance. I include a more detailed comparison with this recent work later in the thesis.

Specifically, I propose several ways to handle invariances. For subgraph isomorphism, I propose an input pooling method based on Subgraph Patterns. As shown in Figure 1.3, different isomorphic graphs with the same Subgraph Pattern are counted together (see details in Chapter 3). For set permutations, I propose to use a Janossy Pooling [15] wrapper around a permutation-sensitive function, which uses permutation sampling to compute a permutation invariant function (see details in Chapter 4/5).

## 1.1 Research Questions

In this work, I propose neural network models with the following properties:

- The model should be invariant to *graph isomorphism* or *set permutation*.

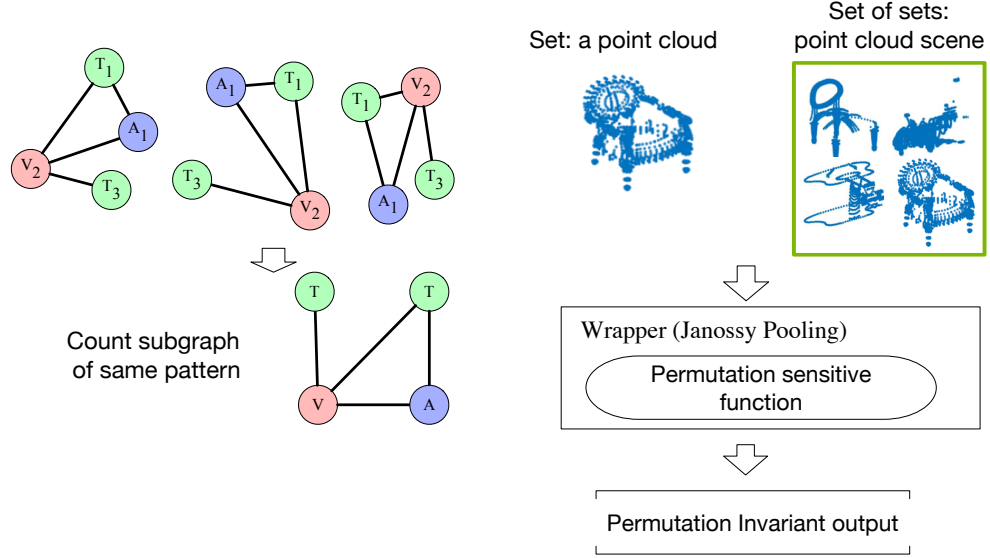


Figure 1.3.: Methods to handle invariance. (a) Pool and count subgraphs under same patterns. (b) Use Janossy Pooling [15] as a wrapper.

- High-order relationships among the elements within the input should be modeled in order to increase the prediction quality.
- The model should provide inductive embeddings. A model learned from training data should generalize to unseen test data.
- Optimization methods for the model should scale polynomially to the data size, in order to be applied to large datasets.

## 1.2 Main Hypothesis and Proposed Research

The goal of the present research is to verify the following hypothesis. Representation learning models with invariances to graph isomorphism or set permutations have better-expressive power than order-sensitive models, and thus will improve prediction accuracy. Moreover, for the factorial-scale possible isomorphic graphs and permuted sets, I hypothesize that I can develop polynomial-scale optimization methods to learn invariant models, through the use of stochastic sampling.

## Proposed Research

The present works is mainly divided into 3 parts.

- For graph applications, I focus on the task of subgraph evolution prediction. This thesis presents a subgraph pattern neural network with invariance to graph isomorphisms and varying local neighborhood sizes. It can take node/edge labels into account and facilitate inductive reasoning. A subgraph-based sampling method is used to enable the model scale to large datasets.
- For set tasks, I show how the characteristics of an architecture’s *computational graph* impact its ability to learn in contexts with complex set dependencies. Moreover, I analyze existing permutation-invariant functions and demonstrate limitations of current methods with respect to one or more of these complexity dimensions. To address this, I propose a new architecture, with a computation graph that is built automatically via self-attention to minimize average interaction path lengths between set elements in the architecture’s computation graph, in order to more effectively capture complex dependencies between set elements.
- To move beyond set problems, I define a new problem called sets-of-sets (SoS), which facilitates logical reasoning and multi-instance learning. In this task, intra-set and inter-set dependence need to be captured with invariance to intra-set and inter-set permutations. To address this, I propose a hierarchical sequence model with attention mechanisms named HATS. The model is invariant to two levels of permutations on the input data—within each set and among the sets. Higher-order relationships with each set and among the sets are captured.

### 1.3 Contributions

This dissertation develops scalable invariant models for graphs and sets. On the one hand, these models are invariant to permutations of sets or isomorphism of graphs.

On the other hand, these models are scalable to model large-scale datasets with polynomial runtimes even though there are a factorial number of permutations.

- In Chapter 3, I study subgraph prediction on dynamic graphs.
  - I propose induced subgraph patterns as features to model high-order dependence between nodes on the graph.
  - I develop a Subgraph Neural Network model *SPNN*, which is a first step in the development of more interpretable models, features, and classifiers that can encode the complex correlations between graph structure and labels.
  - I evaluate the problem of predicting induced subgraph evolution in heterogeneous graphs and show this generalizes a variety of existing tasks. Our results show *SPNN* to consistently achieve better performance than competing approaches.
- In Chapter 4, I study scalable permutation invariant functions for sets.
  - I show that the characteristics of an architecture’s computational graph impact its ability to learn in contexts with complex set dependencies, and demonstrate limitations of current methods with respect to one or more of these complexity dimensions.
  - I develop a neural network architecture *Self-Attention GRU* designed to better capture both long-range and high-order dependencies.
  - I demonstrate *Self-Attention GRU* achieves improved performance over a wide range of applications and against state-of-the-art baselines.
- In Chapter 5, I study the problem of sets-of-sets.
  - I outline how the properties of inter-set and intra-set dependencies of sets-of-sets problem can not be modeled by set models.
  - I propose a framework for learning permutation-invariant inductive SoS embeddings with neural networks, and introduce HATS, a hierarchical,



bi-directional LSTM with attention, which is designed to better capture intra-set and inter-set interactions in sets-of-sets while maintaining SoS permutation-invariance.

- I demonstrate our proposed model HATS achieves superior performance over a wide range of application tasks involving SoS inputs.

## 1.4 Thesis Organization

This dissertation is organized as follows:

- In Chapter 2, I provide an overview of the data input and problem definition. I also review the existing literature on the models for graphs and sets, and discuss the drawbacks of existing models, which form the basis of our methods to overcome these limitations.
- Chapter 3, 4, and 5 build the main contributions of this dissertation, which show how to handle invariance in a scalable way. In Chapter 3, I propose the subgraph neural network model, which overcomes the invariance of graph isomorphism and captures high-order relationships on the graph. In Chapter 4, I show how the characteristics of an architecture’s computational graph impact its ability to learn in contexts with complex set dependencies and propose a new model based on our findings. In Chapter 5, I extend the current set problem to sets-of-sets problems. Besides permutation invariance, I also model inter-set interactions and intra-set interactions.
- Finally, Chapter 6 concludes with a summary of our contributions and outlines the future directions.

## 2 MODELS FOR GRAPH, SET AND SETS-OF-SETS

Representation learning with invariances is a challenge that exists in a variety of problems, with the well-known examples of graph isomorphism and set permutations. In this chapter, I provide some key definitions which describe our problem and data input. I also review relevant literature on graph and set models, laying the foundations for our developments in subsequent chapters. Specifically, in Chapter 3, I study graph isomorphism. In Chapter 4 and 5, I develop models to handle permutation invariance for sets and sets of sets.

### 2.1 Definitions

For our graph task, the input is a *Graph Sequence*. In order to define a *Graph Sequence*, I first define a single graph.

**Definition 1 (Graph with labels)** *Graph  $G_n = (V, E_n, \Phi_n, \Psi_n)$  is simple (i.e., without loops or multiple edges) and heterogeneous (i.e., with labeled (typed) nodes/edges). I denote the node and edge set of  $G_n$  by  $V(G_n)$  and  $E(G_n)$ , respectively. Node and edge labels of  $G_n$  are defined by functions  $\Phi_n$  and  $\Psi_n$ , respectively, s.t.  $\Phi_n : V \rightarrow 2^{|A|}$ , for a set of node classes  $A$ , and  $\Psi_n : E \rightarrow 2^{|R|}$ , for a set of edge types  $R$ .*

**Definition 2 (Graph sequence)** *A graph sequence is a sequence of graphs with labels ordered by timestamp,  $\{G^t\}$  with  $t \in [1, T]$ . The labels mapping  $\Phi, \Psi$  is consistent across the sequence of graphs. Consecutive graphs have common nodes, i.e.,  $V^{i-1} \cap V^i \neq \emptyset$  for  $i \in [2, T]$ .*

**Definition 3 (Graph isomorphism)** *Graph  $G$  and  $H$  are isomorphic graphs if there is a bijection  $f$  between the vertex sets of  $G$  and  $H$  such that any two ver-*

tices  $u$  and  $v$  of  $G$  are adjacent in  $G$  if and only if  $f(u)$  and  $f(v)$  are adjacent in  $H$ .

**Definition 4** *Set Input* A set of  $n$  variables can be represented as a list of vectors. I denote a set as a matrix  $\mathbf{X}$ , and  $X_i \in \mathbb{R}^d$  represents the  $i$ -th row of the matrix, where  $i = 1, 2, \dots, n$  and  $d$  represents the number of dimensions of the variable. With a slight abuse of notation, I shall use  $n$  to denote the number of rows of matrix  $\mathbf{X}$ .

**Definition 5 (Permutation)** A permutation is a rearrangement of the elements of set into a one-to-one correspondence with itself [18]. More specifically, for a set with  $n$  elements, permutation can be denoted as  $\pi$  which is a bijection from integers  $[1, 2, \dots, n]$  which indicates the position of set element in the vector. A set  $\mathbf{X}$  under permutation  $\pi$  is denoted as  $\mathbf{X}_\pi$  where the order of elements are rearranged according to  $\pi$ . The number of permutations on a set of  $n$  elements is  $n!$ .

**Definition 6** *Set of Sets Input* A set-of-sets (SoS) is a multiset whose elements are themselves multisets (which will be referred to as member sets). The multisets belonging to an SoS can have varying sizes. An SoS can be naturally represented as a list of lists of vectors; in this work, I shall denote an SoS as a  $n \times m \times d$  tensor  $\mathbf{X}$  whose  $i$ -th row  $\mathbf{X}_{i**}$  corresponds to the  $i$ -th multiset, and  $X_{ij*}$  denotes the  $j$ -th element of the  $i$ -th multiset, which is itself a  $d$ -dimensional vector,  $d \geq 1$ . With a slight abuse of notation, I shall use  $n$  to denote the number of rows of tensor  $\mathbf{X}$  and  $m$  the number of columns of  $\mathbf{X}$ . In particular,  $m$  is equal to the cardinality of the largest constituent multiset; for the other multisets I pad their columns with a special null symbol (e.g., “#”). In order to simplify the notations, if the third dimension of tensor is “\*”, it is omitted. For instance,  $\mathbf{X}_{i*}$  indicates  $\mathbf{X}_{i**}$ , and  $X_{ij}$  indicates  $X_{ij*}$ .

**Definition 7 (Invariant Set Representation)** A function  $\bar{f}$  acting on set inputs (i.e., matrices of variable dimensions) is an inductive set embedding if its output is invariant under any permutation  $\pi$  of elements. Formally, for any  $n \times d$  matrix  $\mathbf{X}$ ,

$$\bar{f}(\mathbf{X}) = \bar{f}(\mathbf{X}_\pi), \quad (2.1)$$

where  $\mathbf{X}_\pi$  denotes the  $n \times d$  matrix with  $(i)$ -th entry  $\mathbf{X}_i$  equal to  $\mathbf{X}_{\pi(i)}$  and  $\pi$  is a permutation of the integers  $\{1, \dots, n\}$  (rows of  $\mathbf{X}$ ).  $\bar{f}$  is inductive if it is applicable to any set input  $\mathbf{X}$  without any constraints on its dimensions or values.

**Definition 8 (Invariant SoS Representation)** A function  $\bar{f}$  acting on SoS inputs (i.e., tensor) is an inductive SoS embedding if its output is invariant under any permutation  $\phi$  of the member sets, as well as permutations  $\pi_1, \dots, \pi_n$  of the elements in each member set. Formally, for any  $n \times m \times d$  tensor  $\mathbf{X}$ ,

$$\bar{f}(\mathbf{X}) = \bar{f}(\mathbf{X}_{\phi, \pi_\phi}), \quad (2.2)$$

where  $\mathbf{X}_{\phi, \pi_\phi}$  denotes the  $n \times m \times d$  tensor with  $(i, j)$ -th entry  $X_{i,j,*}$  equal to  $X_{\phi(i), \pi_\phi(i)(j),*}$  and  $\phi$  is a permutation of the integers  $\{1, \dots, n\}$  (rows of  $\mathbf{X}$ ), and  $\{\pi_i\}_{i=1}^n$  are permutations of the integers  $\{1, \dots, m\}$  (independent permutation of each column of  $\mathbf{X}$ ). I note that the permutation  $\pi_i$  of the columns of the  $i$ -row can be restricted to only permute the non-null elements in each member set  $X_{i*}$ .

## 2.2 Related work

### 2.2.1 Graphs

In what follows I classify the existing literature based on the main obstacles in designing supervised learning methods for dynamic subgraph heterogeneous tasks: **(a)** The varying sizes of the different node neighborhoods, **(b)** accounting for distinct nodes and edge labels in the neighborhood; **(c)** isomorphic-invariance of graph representations (permutations of nodes in the adjacency matrix should not affect the representation); **(d)** the graph evolution; **(e)** learns from a single graph and includes the dependence structure of induced subgraphs that share edges and non-edges.

There are no existing approaches that can address all the above challenges. Existing approaches can be classified in the following categories:

(1) *Compute canonical representations of the whole graph (e.g., kernel or embeddings).* These methods require multiple examples of a whole graph (rather than

induced subgraphs). Examples include GraphNN [19], diffusion-convolution neural networks [20], and graph kernels, such as Orsini et al.[21] and Yanardag and Vishwanathan [22, 23], which compare small graphs based on the existence or count of small substructures such as shortest paths, graphlets, etc.. These whole-graph methods, however, are designed to classify small independent graphs, and fail to account for the sample dependencies between multiple induced subgraphs that share edges and non-edges. These whole-graph classification methods, collectively, address challenges **(a)**, **(b)**, **(c)**, and **(d)**, but fail to account for **(e)**.

(2) *Compute canonical representations of small induced subgraphs of the original graph*, (e.g., PATCHY [24]) offers a convolutional neural network graph kernel that only addresses challenges **(b)** and **(c)**, but does not address **(d)** and **(e)**, and needs to pad features with zeros or arbitrarily cut neighbors from the feature vector, thus, not truly addressing **(a)**.

(3) *Compute isomorphism-invariant metrics over the graph*, such as most graph kernels methods, various diffusions (e.g., node2vec [25], PCRW [26], PC [27], deepwalk [28], LINE [29], DSSM [30], and deep convolutional networks for graph-structured data [31, 32]), which address problems **(a)** and **(c)** but not **(b)**, **(d)** (specially because of edge labels), and **(e)**.

(4) *Perform a tensor factorization*, (e.g., RESCAL [33] and extensions [34, 35]), which addresses problems **(a)** and **(b)** but not **(c)**, **(d)**, and **(e)**. These methods are tailored specifically for the task of link prediction in heterogeneous graphs and are widely used.

To the best of our knowledge, there does not exist supervised learning methods designed to predict subgraph evolution. In Chapter 3, I will design a method for this problem and it can address all the above challenges.

Other classical link prediction methods that can also be adapted to subgraph link prediction tasks. These methods [36, 37] use a wide variety of edge features, including pair-wise features such as Adamic-Adar score, or path counts, such as from PCRW.

Separately, collective inference procedures [38, 39, 40, 41], although traditionally evaluated at the node and edge level, can also include SPNN as a baseline predictor to be readily applied to dynamic subgraph tasks.

### 2.2.2 Sets

There exists plenty of work on learning set representations [4, 14, 15, 16, 17, 42, 43, 44, 45, 46, 47]. In particular, *DeepSets* [4] designs a deep neural network architecture which embeds each element of the set individually, and then performs a simple pooling operation such as sum/mean/max to aggregate these into an individual embedding, latter passing the result through upper feedforward layers. *Janossy pooling* [15] provides a generalized framework for pooling operations on set. The key idea of Janossy Pooling is that any permutation-invariant function could be expressed as the average of a permutation-sensitive function applied to all permutations of the input. Since any permutation-sensitive function can be applied in this framework, more complex functions such as LSTM can be used to capture high-order relationship. This framework shows promising results in a large variety of set applications, as well as more complicated sets-of-sets applications [47]. Set transformer [17] provides a set model based on self-attention which shows better performance. It aggregated pair-wise interactions calculated based on covariance calculated by self-attention. Later I will show theoretically and empirically that these models could capture long-range and high-order dependence at the same time.

Besides proposing new models, there also exist works on evaluating existing models theoretically. Analysis [48] showed representation for set inputs could only be achieved with a latent dimension at least the size of the maximum number of input elements. The number of activation patterns [49] can be used to quantify the expressive power of a model. [50] analyzed detecting the interaction between elements in neural networks. These works laid a solid foundation to help us evaluating general set

models. Furthermore, none of the theoretical work has tapped the connection between model performance and the computational graph of the neural network architecture.

Learning permutation-invariant functions also have direct impact on other research fields such as graph mining [5, 6, 7, 51], point-cloud modeling [3, 52], natural language processing [53, 54, 55] and problem reasoning [6]. The purpose of this work is to offer a general framework to model permutation invariant functions for set-related applications, instead of outperforming state-of-the-art approaches that are crafted for specific applications. Our generalized approach provides flexibility and can be tailored based on needs to adapt to various tasks.

### 2.2.3 Set of Sets

In many applications, the input examples are actually *sets-of-sets*, rather than (plain) sets, and I argue that their hierarchical nature deserves special treatment in modeling. This motivates us to move beyond (single-level) sequence models to hierarchical ones when designing the neural network architecture (see Section 5.3 for details), and our experiments demonstrate that the hierarchical models yield significant performance gains in practice. I do note that [56] have studied matrix-factorization models for learning interactions across two sets (users and movies) in the specific context of recommender systems. However, their Kronecker product-based approach is transductive rather than inductive and designed for a very specific application, whereas I am interested in general inductive embedding approaches for set-of-sets.

As with most recent works in the literature, I choose to parametrize the set-of-sets permutation-invariant function using deep neural networks, thanks to their expressiveness as universal function approximators. Regarding the choice of neural network architectures, I focus on recurrent neural networks (RNNs)—in particular, long short-term memory (LSTM) networks. The choice is in contrast to that made by *e.g.*, [43], which focused on convolution-based approaches. As with many other works in the literature [14, 15], I believe that sequence models are more appropriate for model-

ing variable-size inputs. In Section 5.4 I demonstrate empirically that the proposed LSTM-based models lead to improved performance over the CNN-based model of [43] across a variety of tasks and the less specialized use of LSTMs in [15]. I further investigate attention-based mechanisms for hierarchical LSTM models to enhance their capability of capturing long-range dependencies. I note that similar hierarchical-attention architectures have been considered for document classification [57] in the natural language processing literature, but I adapt it to modeling SoS functions in order to preserve permutation-invariance.

Methods for learning permutation-invariant functions on set structures have direct implications to relational learning and graph mining (*e.g.*, [5, 6, 7, 51]), point-cloud modeling (*e.g.*, [3, 52]) and scene understanding [8, 12] in computer vision, among other applications. While I have conducted experiments on subgraph hyperlink prediction and point-cloud classification tasks to evaluate the performance of our proposed approaches, I emphasize that (as in *e.g.*, [4, 15, 43]) the aim of our work is to provide a general characterization and framework for modeling functions with sets-of-sets inputs, rather than outperforming state-of-the-art approaches that are crafted for specific applications. Importantly, the generality of our proposed approach enables practitioners the flexibility of tailoring it to their specific tasks at hand.



### 3 SUBGRAPH PATTERN NEURAL NETWORK

#### 3.1 Introduction

Learning predictive models of heterogeneous relational and network data is a fundamental task in machine learning and data mining [25, 26, 35, 41, 58]. Much of the work in heterogeneous networks (graphs with node and edge labels) has focused on developing methods for label prediction or single link prediction. There has been relatively little development in methods that make joint predictions over larger substructures (e.g., induced  $k$ -node subgraphs). Recent research has shown rich higher-order organization of such networks [59, 60] and complex subgraph evolution patterns within larger graphs [61]. Applications range from predicting group activity on social networks (e.g., online social network ad revenues rely heavily on user activity), computational social science (e.g., predicting the dynamics of groups and their social relationships), relational learning (e.g., find missing and predicting future joint relationships in knowledge graphs).

The main challenge in learning a model to predict the evolution of labeled *subgraphs* is to jointly account for the induced subgraph dependencies that emerge from subgraphs sharing edges. Unlike node and edge prediction tasks, it is not clear how to describe an approximate model that can account for these dependencies. A variety of recent methods have developed heuristics to encode joint label and structure information into low dimensional node or edge embeddings, but it is unclear how these ad-hoc methods can properly address the induced subgraph dependencies [20, 25, 26, 33, 35, 36, 58, 62]. Our empirical results show that these methods tend to perform poorly in induced subgraph prediction tasks.

The task of predicting induced subgraph evolution requires an approach that can take into account higher-order dependencies between the induced subgraphs (due to

their shared edges and non-edges<sup>1</sup>). Our two main contributions are: (1) I target the evolution of larger graph structures than nodes and edges, which, to the best of our knowledge, has never been focused before. Traditional link prediction tasks are simpler special cases of our task.

(2) I incorporate the unavoidable dependencies within the training observations of induced subgraphs into both the input features and the model architecture itself via high-order dependencies. I denote our model architecture a Subgraph Pattern Neural Network (SPNN ) and show that its strength is due to a representation that is invariant to isomorphisms and varying local neighborhood sizes, can also take node/edge labels into account, and which facilitates inductive reasoning.

SPNN is a discriminative feedforward neural network with hidden layers that represent the *dependent* subgraph patterns observed in the training data. The input features of SPNN extend the definition of induced isomorphism density [63] to a local graph neighborhood in a way that accounts for joint edges and non-edges in the induced subgraphs. Moreover, SPNN is inductive (it can be applied to unseen portions of the graph), and is isomorphic-invariant, such the learned model is invariant to node permutations. I also show that SPNN learns to predict using an interpretable neural network structure.

SPNN finds a variety of major industrial and scientific applications:

1. Predicting group activity: Facebook’s \$9.6 billion-dollar ad revenue in 2017 Q2 depends entirely on user activity. In 4 years, MySpace went from a \$12B company with 300M active users to a \$35M price with mostly inactive users in 2011. Users on social media are active because of the activity of their friends and followers (known as the network effect in economic theory). Our SPNN model can be used to predict of group activity levels.

---

<sup>1</sup>A non-edge marks the absence of an edge

2. Computational social sciences: SPNN can also be used to predict the probability that a group of friends dissolves, and the factors that predict the dissolution.
3. Augment NLP/Vision methods: SPNN can help predict the type of posted image in social networks (node label), or the topic of a text (edge label) by looking at it as a group event, rather than each individual user in isolation.
4. Relational learning: SPNN can learn relationships between events. Improve, fix, and predict dynamics in knowledge graphs.

### 3.2 Heterogeneous Subgraph Prediction

In what follows I define the heterogeneous pattern prediction task and present a classification approach that uses a neural network classifier whose structure is based on *connected* induced subgraphs. In what follows, to avoid confusion with work on “learning low dimensional embeddings,” I *avoid* using the correct-graph theoretic term *graph embeddings* [64] in favor of the less standard term *induced subgraphs* of a smaller graph pattern into a larger graph.

#### Definition 3.2.1 (Induced Labeled Subgraphs)

*Let  $F$  and  $G$  be two arbitrary heterogeneous graphs such that  $|V(F)| \leq |V(G)|$ . An induced subgraph of  $F$  into  $G$  is an adjacency preserving injective map  $\gamma_F : V(F) \rightarrow V(G)$  s.t. for all pairs of vertices  $i, j \in V(F)$ , the pair  $(\gamma_F(i), \gamma_F(j)) \in E(G)$  iff  $(i, j) \in E(F)$ , and all the corresponding node and edge labels of  $i$  and  $j$  match, i.e.,  $\Phi(i) = \Phi(\gamma_F(i))$ ,  $\Phi(j) = \Phi(\gamma_F(j))$ , and, if  $(i, j) \in E(F) \implies \Psi((i, j)) = \Psi((\gamma_F(i), \gamma_F(j)))$ .*

In the remainder of the paper, I consider these “ $F$ ”s as small  $k$ -node graphs and refer to them as *subgraph patterns*.

#### Definition 3.2.2 (Task Definition)

**Subgraph Patterns of Interest:** *The  $k$ -node subgraph patterns of interest are*

$\mathcal{F}^k = \{F_1, \dots, F_c\}$ , where  $c \geq 1$ ,  $|V(F_i)| = k$ ,  $\forall i$ .

**Labels:** In order to simplify the classification task, I further partition these patterns into sets with  $r$  distinct “classes”, which I denote  $\mathcal{Y}_1^k, \dots, \mathcal{Y}_r^k$  (as shown in Figure 3.1).

**Training data:**  $\mathcal{T}_1^k$  and  $\mathcal{T}_2^k$  are the set of all  $k$ -node induced subgraphs of patterns  $\mathcal{F}^k$  in  $G_1$  and  $G_2$ , respectively, as described in Definition 3.2.1. For each induced subgraph  $U \in \mathcal{T}_1^k$ , I define its label  $y_2(U)$  by looking at the pattern these same nodes form in  $\mathcal{T}_2^k$ , where  $y_2(U) = r$ , if the nodes  $V(U)$  form an induced subgraph with pattern  $F \in \mathcal{Y}_r^k$ . Note that the patterns in  $\mathcal{F}^k$  must encompass all possible evolution of the induced subgraphs in  $\mathcal{T}_1^k$ . The training data is

$$\mathcal{D}_{\text{train}} = \{(U, y_2(U)) : U \in \mathcal{T}_1^k\}.$$

*Examples (Figure 3.1, best seen in color):* The induced subgraph  $U \in \mathcal{T}_1^3$  shown in the blue oval, with vertices  $V(U) = \{V_2, T_3, A_2\}$  (a venue, a topic, an author), has pattern  $F = \text{red, blue, green} \in \mathcal{F}^3$ . The label of  $U$  is  $y_2(U) = 1$  as the vertices  $V(U)$  form pattern  $F = \text{red, blue, green} \in \mathcal{Y}_1^3$  in  $G_2$ . The induced subgraph  $U' \in \mathcal{T}_1^3$  shown in the red oval,  $V(U') = \{V_1, T_1, A_1\}$ , has pattern  $\text{red, blue, green}$  in  $G_1$  and pattern  $\text{red, blue, green}$  in  $G_2$ , thus,  $y_2(U') = 2$ .

**Prediction Task:** Given the induced subgraphs in  $\mathcal{T}_2^k$ , our goal is to predict their corresponding pattern in  $G_3$ . These predicted patterns must be in  $\mathcal{F}^k$ .

Traditional link prediction tasks [65] can be seen as special instances of the task in Definition 3.2.2, where  $k = 2$  and the target set of patterns  $\mathcal{Y}_1^2$  consist of edges (i.e., 2-node connected induced subgraphs) and non-edges  $\mathcal{Y}_2^2$ . In the single link prediction case, the focus is on predicting individual links such as friendship links in Facebook, citation links in DBLP, or links in knowledge bases such as WordNet.

**Obtaining Training Data from Large Networks.** Let  $\mathcal{T}_t^k$ , be all  $k$ -node induced subgraphs with patterns  $\mathcal{F}^k$  over  $G_t$ . Our training data consists of  $\mathcal{T}_1^k$  and the future patterns of these induced subgraphs in  $\mathcal{T}_2^k$ , both which can be very large even for moderately small networks. I reduce *computational resources* needed to generate the **training data** by filtering the data of  $\mathcal{T}_1^k$  as follows.

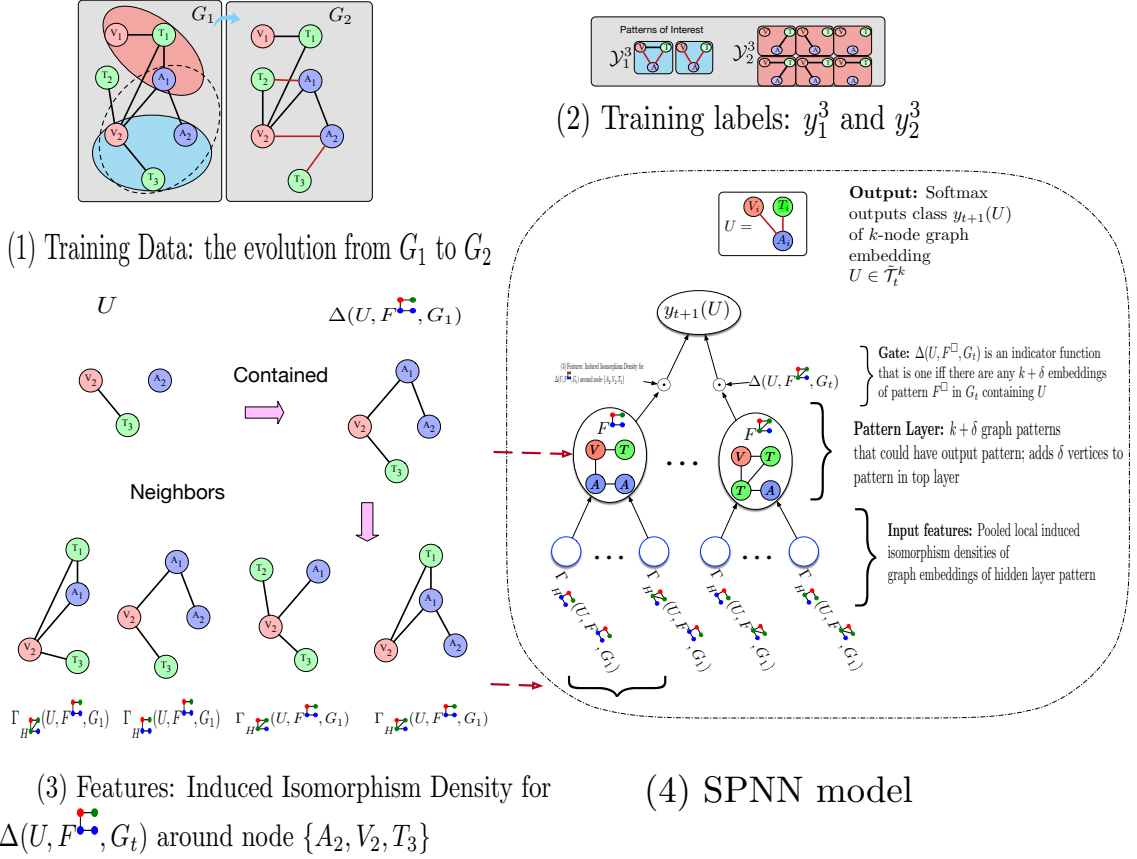


Figure 3.1.: (1) Illustration of the training in a citation network with (A)uthors, (T)opics, (V)enues. At the top is the graph evolution  $G_1$  to  $G_2$ , whose induced subgraphs are used as training data to predict the evolution of subgraphs in  $G_2$  to  $G_3$ ; below  $\mathcal{Y}^3$  shows 3-node subgraph patterns partitioned into two classes. (2) Labels for subgraph evolution. The appearance of two links are considered as label  $y_1^3$ . All other subgraphs are assigned label  $y_2^3$ . (3) Features for  $U = V_2$ .  $A_2, T_3$  under the patterns (4) SPNN model.


I construct a training dataset  $\tilde{\mathcal{T}}_1^k \subseteq \mathcal{T}_1^k$  such that a  $k$ -node induced subgraph  $U \in \tilde{\mathcal{T}}_1^k$  must belong to a larger  $(k + \delta)$ -node **connected** induced subgraph in  $G_1$ ,  $\delta \geq 1$ . This constraint facilitates the identification of more *relevant* disconnected subgraphs of size  $k$  without having to fully enumerate all the possibilities. By *relevant*, I mean that those  $k$ -node disconnected subgraphs are overwhelmingly more likely to

evolve into connected patterns because the  $k$  nodes have shortest paths of length up to  $(k + \delta - 1)$  hops in  $G_1$ . Thus, the choice of  $\delta$  is not arbitrary: I choose  $\delta$  such that most of training examples with the labels I are most interested in predicting (e.g., Class 1 in Figure 3.1) are still in  $\tilde{\mathcal{T}}_1^k$ .



This filtering procedure also helps us quickly sample the training data from  $G_1$  using a fast connected subgraph sampling method with known sampling bias (such that the bias can be removed) [66].

### (SPNN ) Subgraph-Pattern Neural Network

Subgraph-Pattern Neural Network (SPNN ) is our proposed classifier. SPNN is a 3-layer gated neural network with a sparse structure generated from the training data in a pre-processing step. The second neural network layer, which we call the Pattern Layer, is interpretable as it represents the  $(k + \delta)$ -node patterns in  $G_1$  that were found while collecting the training data, described next. The neural network also has gates to deactivate the backpropagation of errors to the hidden units as we will describe later.

**Pattern Layer.** In the example of Figure 3.1, the 3-node training example of induced subgraph  $U$  in  $G_1$ ,  $V(U) = \{A_2, V_2, T_3\}$  (blue oval), belongs to a connected 4-node subgraph ( $\{A_1, A_2, V_2, T_3\}$ , the dotted oval) that matches the pattern  $F$   represented in Figure 3.1(3). More generally, the set of all such patterns is

$$\begin{aligned} \mathcal{F}_{t=1}^{\square(k+\delta)}(\mathcal{T}_{t=1}^k) &= \{F^\square : \forall F^\square \in \mathcal{P}_{(conn)}^{(k+\delta)} \text{ s.t. } \exists U \in \mathcal{T}_1^k, \\ &\exists R \in \text{Ind}(F^\square, G_1), \text{ and } R \in T_1^{(k+\delta)}(U, F^\square)\}, \delta \geq 0 \end{aligned} \quad (3.1)$$

where the square  $\square$  indicates a connected subgraph pattern (e.g., ,  $\dots$ , ,  $\mathcal{P}_{(conn)}^{(k+\delta)}$  is the set of **all**  $(k + \delta)$ -node **connected** graph patterns containing all possible node

and edge labels,  $\text{Ind}(F, G)$  denotes the set of induced subgraphs of  $F$  into a graph  $G$ , and for  $U \in \tilde{\mathcal{T}}_1^k$  we define

$$T_1^{(k+\delta)}(U, F^\square) = \{(k+\delta)\text{-node induced connected subgraphs of } F^\square \text{ at } G_1 \text{ having all nodes of } U\}. \quad (3.2)$$

For instance,  $T_1^{(k+\delta)}(U, F^{\text{red, blue}})$  with  $V(U) = \{V_2, A_2, T_3\}$  in Figure 3.1 (3). In practice, we also mark the nodes of  $U \in \tilde{\mathcal{T}}_1^k$  that appear in the  $(k+\delta)$ -node patterns with unique special types, so we can distinguish their structural role in the larger  $(k+\delta)$ -node subgraph.

Figure 3.1 (4) illustrates the SPNN architecture using the task illustrated in Figure 3.1 (1) as an example. For instance, we want to jointly predict whether an author  $A$  will publish at a venue  $V$  and in topic  $T$  at  $G_2$  given such author did not publish at venue  $V$  or topic  $T$  at  $G_1$ .

**Pattern Layer & Gates.** The hidden layer of SPNN represents  $\mathcal{F}^{\square(k+\delta)} := \{F^{\square_1}, F^{\square_2}, \dots\}$ , **all observed**  $(k+\delta)$ -node patterns in the training data  $\tilde{\mathcal{T}}_1^k$ . This procedure *only eliminates patterns that are not observed in the training data*. For example, in the illustration of Figure 3.1,  $\delta = 1$ , there would be no 4-node patterns of a fully connected graph in  $\mathcal{F}^{\square(3+1)}$  as there are no fully connected 4-node graphs in  $G_1$ .

For the training example  $U \in \tilde{\mathcal{T}}_t^k$ , it may be the case that pattern  $F^\square \in \mathcal{F}_t^{\square(k+\delta)}$  has no induced subgraph on  $G_t$  that contains the example  $U$ , i.e.,  $T_t^{(k+\delta)}(U, F^\square) = \emptyset$ . If this happens, we should not backpropagate the error of the hidden unit associated with  $F^\square$ . For instance, for  $\delta = 1$  in the illustration of Figure 3.1, the training data induced subgraph  $U \in \tilde{\mathcal{T}}_t^3$  with vertices  $V(U) = \{A_1, T_1, V_1\}$  will only backpropagate the error to the hidden units matching the patterns of induced subgraphs  $\{A_1, T_1, V_1, V_2\}$  and  $\{A_1, T_1, V_1, A_2\}$ . We use a gate function

$$\Delta(U, F^\square, G_t) = \mathbb{1}\{T_t^{(k+\delta)}(U, F^\square) \neq \emptyset\}, \quad (3.3)$$

with  $T_t^{(k+\delta)}$  as defined in Eq.(3.2). The gate  $\Delta(U, F^\square, G_t)$  ensures we are only training the neural network unit of  $F^\square$  when the induced subgraph example  $U$  applies to that unit.

Our pattern layer has an interpretable definition: each pattern neuron represents a larger subgraph pattern containing the target subgraph. If a specific neuron has a significant impact activating the output, we know that its corresponding pattern is important in the predictions.

**Input Features.** In what follows we define the features given to the input layers of SPNN. Our features need the definition of a *local* induced isomorphism density around the induced subgraph of pattern  $F^\square$  on  $G_t$ , with  $F^\square \in \mathcal{F}_t^{\square(k+\delta)}$ .

**Definition 3.2.3 (Local induced isomorphism density)** *Let  $R$  be a induced subgraph of  $G$  and let  $F$  be a subgraph pattern s.t.  $|V(G)| > |V(F)| \geq |V(R)|$ . The local induced isomorphism density,  $t_{\text{local}}$ , rooted at  $R$  with subgraph pattern  $F$  is the proportion of induced subgraphs of  $F$  at  $G$  in a ball of radius  $d$  from the nodes of  $V(R)$ . More precisely,  $t_{\text{local}}(R, F, G, d) \propto |\text{LocInd}(R, F, G, d)|$ , where  $\text{LocInd}(R, F, G, d) = \{R' \in \text{Ind}(F, G) : |V(R') \cup V(R)| - |V(R') \cap V(R)| \leq d\}$ .*

The quantity  $t_{\text{local}}$  is the proportion of induced subgraphs of pattern  $F$  at  $G$  constrained to the set of vertices that are up to  $d$  hops away from the set of nodes  $V(R)$ . If  $G$  has a small diameter,  $d$  should be small.

We now use  $t_{\text{local}}$  to define the input features for an example  $U \in \tilde{\mathcal{T}}_t^k$ . For each  $F^\square \in \mathcal{F}_t^{\square(k+\delta)}$ , there will be a vector  $\phi(U, F^\square, G_t)$  of dimension  $m_{F^\square}$  (to be defined below), where

$$\begin{aligned} (\phi(U, F^\square, G_t))_i &:= \Gamma_{H_i}(U, F^\square, G_t) \\ &= \sum_{R \in T_t^{(k+\delta)}(U, F^\square)} t_{\text{local}}(R, H_i, G_t, d), \quad H_i \in \mathcal{P}^{(k+\delta)}, \end{aligned} \tag{3.4}$$

where, as before,  $\mathcal{P}_{(\text{conn})}^{(k+\delta)}$  is the set of all possible  $(k+\delta)$ -node connected patterns. Each input feature  $\Gamma_H$  is a pooled value of  $t_{\text{local}}$  that counts the density of induced subgraphs of a  $(k+\delta)$ -node pattern  $H$  around a ball of radius  $d$  from the vertices  $V(R)$ , where  $R$  is a  $(k+\delta)$ -node connected induced subgraph that contains the example  $U$ . Thus,  $\Gamma_H$  sums the densities of induced subgraphs that *can have* up to  $d+\delta$  nodes different from



$U$ . We only include  $\Gamma_H$  in the vector  $\phi(U, F^\square, G_t)$  if  $\exists U \in \tilde{\mathcal{T}}_t^k$  s.t.  $\Gamma_H(U, F^\square, G_t) > 0$ . As  $m_{F^\square}$  is the number of non-zero values of  $\Gamma$ , then  $m_{F^\square} \leq |\mathcal{P}_{(conn)}^{(k+\delta)}|$ .

To illustrate the  $\Gamma$  metric, consider pattern  $F$  illustrated in Figure 3.1 (1) and the training example  $U$  as the induced subgraph  $\{A_2, V_2, T_3\}$  in  $G_1$  in Figure 3.1 (3).  $U$  is contained in the connected 4-node subgraph with  $V(R) = \{A_2, V_2, T_3, A_1\}$ . The pattern  $H$  has  $\Gamma_H(U, F, G_t) = 1/4$  as there is only one induced subgraph  $\{(T_2, V_2), (V_2, A_1), (A_1, T_1), (T_1, V_2)\}$  with pattern  $H$  out of the 4 induced 4-node subgraphs that are within a radius of  $d = 1$  of the nodes  $V(R)$ .

**The SPNN Classifier.** We now put all the different components together for a  $r$ -class classification task. Consider the class  $y_{t+1}(U)$  as a one-of- $K$  encoding vector. For a  $k$ -node induced subgraph  $U$  of  $G_t$ , the probability nodes  $V(U)$  form an induced subgraph in  $G_{t+1}$  with a pattern of class  $i$ , for  $1 \leq i \leq r$ , is

$$\begin{aligned} p(y_{t+1}(U); \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)})_i \\ = \text{softmax}((\mathbf{W}^{(1)} h_t(U; \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) + \mathbf{b}^{(1)})_i), \end{aligned}$$

where  $\mathbf{b}^{(1)} \in \mathbb{R}^d$  is the bias of the output layer and  $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times |\mathcal{F}_t^{\square(k+\delta)}|}$  are the linear weights of the pattern layer. The input to the pattern layer is

$$\begin{aligned} h_t(U; \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) &= (\Delta(U, F_1^\square, G_t) \cdot \sigma( \\ &(\mathbf{W}_1^{(2)})^\top \phi(U, F_1^\square, G_t)), \Delta(U, F_2^\square, G_t) \cdot \sigma( \\ &(\mathbf{W}_2^{(2)})^\top \phi(U, F_2^\square, G_t)), \dots) + \mathbf{b}^{(2)}, \end{aligned}$$

where for each unit associated with  $F_j^\square$ ,  $j = 1, 2, \dots$ , we have  $\mathbf{b}_j^{(2)} \in \mathbb{R}$  as the bias and  $\mathbf{W}_j^{(2)}$  as the classifier weights, and  $\sigma$  is an activation function (our empirical results use tanh), the feature vector  $\phi(U, F_j^\square, G_t)$  is as defined in Eq.(3.4), and  $\Delta$  is the 0–1 gate function defined in Eq. (3.3). Our optimization objective is maximizing the log-likelihood

$$\begin{aligned} \arg \max_{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}} \sum_{U \in \tilde{\mathcal{T}}_t^k} (y_{t+1}(U))^\top \log p(y_{t+1}(U); \\ \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}). \end{aligned} \quad (3.5)$$

The parameters  $\mathbf{W}^{(1)}$ ,  $\mathbf{W}^{(2)}$ ,  $\mathbf{b}^{(1)}$ , and  $\mathbf{b}^{(2)}$  are learned from Eq.(3.5) via stochastic gradient descent with early stopping. In what follows we show SPNN learns the same parameters irrespective of graph isomorphisms (see Supplemental Material for proof).

**Theorem 3.2.1** *SPNN is isomorphic invariant. That is, given two graph sequences  $G_1, G_2$  and  $G'_1, G'_2$ , where  $G_n$  is isomorphic to  $G'_n$ , then the learned parameters  $\hat{\mathbf{W}}^{(1)}$ ,  $\hat{\mathbf{W}}^{(2)}$ ,  $\mathbf{b}^{(1)}$ ,  $\mathbf{b}^{(2)}$  are exactly the same for the graph sequences  $(G_1, G_2)$  and  $(G'_1, G'_2)$  (assuming the same random seed).*

**Proof** [Proof of Theorem 3.2.1] In this proof I show that SPNN's input features, the training data, and the convolutional architecture all have a canonical representation invariant to isomorphisms. To this end, I show that: (a) the features of each training example  $U \in \tilde{\mathcal{T}}_t^k$  have a canonical representation invariant to isomorphisms; (b) the training data  $\tilde{\mathcal{T}}_t^k$  used in our stochastic gradient descent algorithm also has a canonical representation; and finally, (c) the neural network structure also has a canonical representation invariant to isomorphisms.

(a) The features of each training example  $U \in \tilde{\mathcal{T}}_t^k$  are the vectors  $\phi(U, F^\square, G_t)$  introduced in Eq.(3.4) for different patterns  $F^\square \in \mathcal{P}_{(conn)}^{(k+\delta)}$  that appear in the training data. All I need to show is that vector  $\phi$  has a canonical order invariant to graph isomorphisms. Observing Eq.(3.4), the  $i$ -th element of  $\phi$ ,  $(\phi(U, F^\square, G_t))_i$ , has a canonical order as I can impose a canonical order on  $\mathcal{P}_{(conn)}^{(k+\delta)}$  (e.g., lexicographic on the edges [67]). The value inside  $(\phi(U, F^\square, G_t))_i$  is also clearly invariant to isomorphisms as it is the isomorphism density.

(b) The training data  $\tilde{\mathcal{T}}_t^k$  are subgraphs of  $G_t$  and, thus, also have a canonical representation via lexicographical ordering [67].

(c) As  $\mathcal{P}_{(conn)}^{(k+\delta)}$  has a canonical order, so does the hidden layer of SPNN. Moreover, the  $\Gamma$ 's are similarly ordered.

The induced subgraphs of the training examples of the two isomorphic graphs  $G_1$  and  $G'_1$  have the same class labels, as the class labels are by definition isomorphic invariant. As there are canonical orderings of the data, features, class labels, and

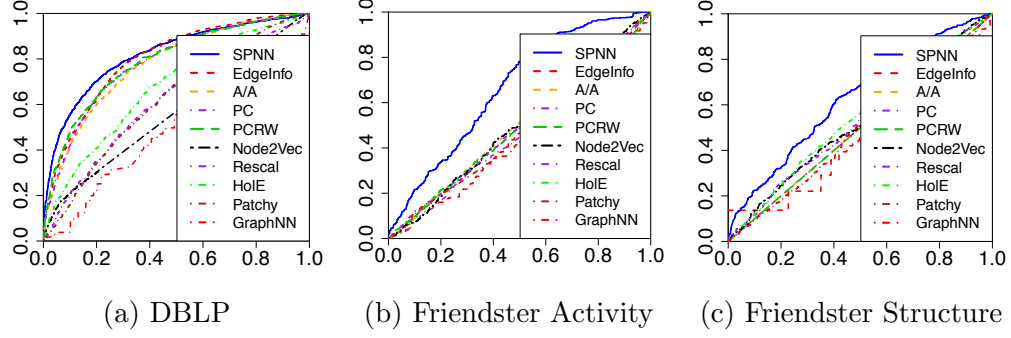


Figure 3.2.: ROC curves (True Pos  $\times$  False Pos): DBLP and Friendster tasks.

model structure that are invariant to isomorphic transformations of the graphs, and the stochastic gradient descent has the same random seed for all graphs, I conclude that SPNN must learn the same parameters. ■

### 3.2.1 Relationship with Convolutional Neural Networks

Images are lattices, trivial topologies, while general graphs are complex. Fundamentally, a CNN computes the output of various filters over local neighborhoods. In SPNN, the filter is the pattern, which maps the local neighborhood (within  $d + \delta$  hops away from the target subgraph) into a single value. The distinct patterns act on overlapping regions of the neighborhood, but the amount of overlap is nontrivial for non-lattices. At CNNs, pooling at the upper layers often act as a rotation-invariance heuristic. SPNN upper layers are isomorphic-invariant by construction and SPNN performs pooling at the inputs. Moreover, similar to CNNs, SPNN can be augmented by multiple layers of fully connected units between the pattern layer and the predicted target.

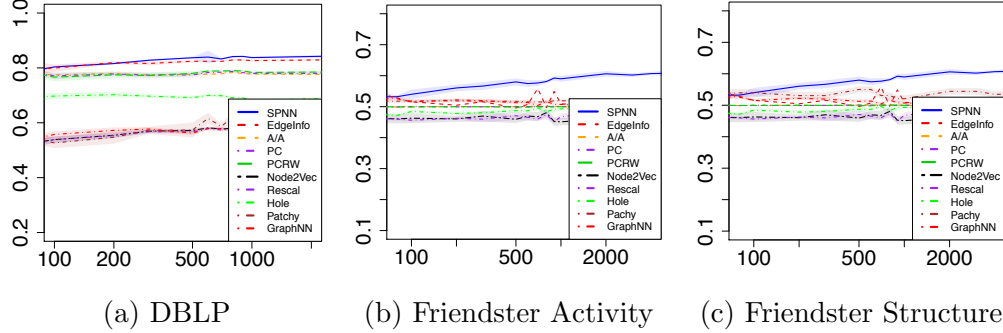


Figure 3.3.: Learning curves (AUC $\times$ Training Size) w/shaded 95% confidence intervals.

### 3.3 Results

In this section we test the efficacy of SPNN , comparing it to other existing methods in the literature. We adapt these competing methods to the induced dynamic subgraph prediction task, as they are not designed for such tasks.

Our evaluation shows SPNN outperforms nine state-of-the-art methods in three real-world dynamic tasks. We show that the learned SPNN model weights can be used to draw insights into the predictions. We also evaluate SPNN across a variety of other synthetic dynamic tasks using static graphs (**Facebook** and **WordNet**), all reported in the appendix.

In the appendix, we also show that the architecture of SPNN also outperforms fully connected neural network layers for small training samples (both using the unique induced subgraph input features designed for SPNN , which explicitly model the subgraph dependencies). SPNN and fully connected layers have the same performance over larger training datasets.

#### 3.3.1 Empirical Results

**Datasets.** We use two representative heterogeneous graph datasets with temporal information. **DBLP** [68] contains scientific papers in four related areas (AI, DB, DM, IR) with 14,376 papers, 14,475 authors, 8,920 topics, and 20 venues. We organize

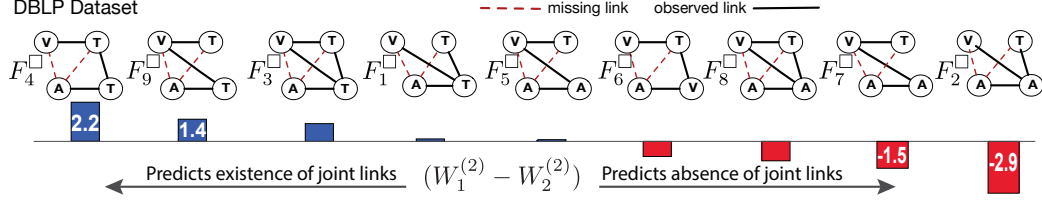


Figure 3.4.: (DBLP task) Pattern layer weight difference between Class 1 (whether both dashed links appear at time  $t + 1$ ) and Class 2 (everything else) for pattern  $F_j^\square$ . Pattern  $F_4^\square$ , when the author has published in a topic related to the venue, strongly predicts the appearance of both links. Pattern  $F_2^\square$ , when a co-author has published at the venue and topic of interest but not the author, strongly predicts the absence of the joint links.

the dataset into authors, venues, and topics. Published papers represent links, for instance, two authors have a link at  $G_n$  if they have co-authored a paper at time step  $n$ .

**Friendster** is a social network where user can post messages on each other's homepages. This dataset contains 14 millions of nodes and 75 million messages. Directed edges in this dynamic graph mark users writing on each other's message walls. The heterogeneous graph includes hometown, current locations, college, interests, and messages sent between users.

### Subgraph Pattern Prediction Tasks.

- (a) **DBLP** task is to predict the evolution of 3-node subgraphs: whether an author will publish in a venue and a topic that the author did not publish in the previous timestamp.
- (b) **Friendster Activity** task predicts the increase in activity in weighted 4-node subgraphs: whether the total number of messages sent between four users, which are connected in the current time interval ( $G_2$ ), increases in the next time interval ( $G_3$ ).

- (c) **Friendster Structure** task predicts the evolution of 4-node subgraphs: whether four friends who were weakly connected by three edges in the previous timestamp ( $G_2$ ) will not send any messages in the next time stamp (i.e., be disconnected in  $G_3$ ).

To learn a predictive model of subgraph evolution, we divide the data into three temporal graphs  $G_1, G_2, G_3$ . The training set  $\mathcal{T}_1^3$  comprises 3-node or 4-node subgraphs from  $G_1$  with class labels  $\mathbf{y}$  determined from  $G_2$ , and the test set  $\mathcal{T}_2^3$  comprises subgraphs from  $G_2$  with class labels from  $G_3$ . Since DBLP is a dynamic network with timestamps, we construct  $G_1$  from the data in 2003–2004,  $G_2$  from 2005–2006, and  $G_3$  from 2007–2008. For Friendster, we construct  $G_1$  from data in Jan 2007–April 2007,  $G_2$  from May 2007–Aug 2007, and  $G_3$  from Sep 2007–Dec 2007. We selected year 2007 because it is the most active time period for Friendster.

**Baselines.** We compare our approach to the nine methods discussed in Related Work. Five methods use isomorphic-invariant measures over the graph: (i) **AA**: Adamic–Adar score only [69]; (ii) **EdgeInfo**: Uses all edge features listed in [37]; (iii) **PC**: Path counts (a.k.a. metapaths) [27]; (iv) **PCRW**: Path constrained random walk [26]; (v) **Node2Vec**: Node embedding [25]. Two methods perform tensor factorizations: (vi) **Rescal**: Rescal embedding [33]; (vii) **HolE**: Holographic embedding [35]. One method computes canonical representations of small induced subgraphs of the original graph; (viii) **Patchy**: Patchy CNN graph kernel [24]; (ix) **GraphNN**: Embedding Mean-Field Inference [19].

The above baselines, except **Patchy** and **GraphNN**, are originally intended to predict single missing links rather than make joint link predictions. We consider two different variants of the methods to apply the baselines to our joint link prediction tasks. The **Independent** approach trains separate classifiers, one for each link independently, and then combines the independent predictions into a joint prediction. The **Joint** approach concatenates the features of the multiple links into a single subgraph feature, then uses a classifier over the subgraph feature to make joint link predictions.

Moreover, these baselines, which are not developed for subgraph evolution tasks, generally achieve very poor predictive performance in a real temporal task that uses graphs  $G_1$  and  $G_2$  to predict  $G_3$ . Consider, for instance, the two distinct embeddings that **Node2Vec**, **Rescal**, and **HolE** assign to same nodes in  $G_1$  and  $G_2$  due to changes in the graph topology between  $G_1$  and  $G_2$ . In order to use **Node2Vec**, **Rescal**, and **HolE** to predict links in dynamic graphs, we first learn node embeddings over  $G_1$  and train a Multilayer Perceptron to predict links in  $G_2$ . Using this trained classifier, we again use the node embeddings of  $G_1$  to predict the new links in  $G_3$ , and this improves their classification performance.

**Implementation.** We implement SPNN in Theano. The loss function is the negative log likelihood plus L1 and L2 regularization penalties over the parameters, both with regularization penalty 0.001. We train SPNN using stochastic gradient descent over a maximum of 30000 epochs and learning rate 0.01. 20% of the training examples are separated as validation for early stopping. All the data has the same amount of positive and negative examples. Source code is available at <https://github.com/PurdueMINDS/SPNN>.

**Comparison to Baselines.** Figure 3.2a-c shows the ROC curves of SPNN and baselines to predict balanced classes. We use 1000 induced subgraphs for training and 2000 induced subgraphs for testing (in all DBLP, Friendster Activity and Friendster Structure tasks). Since the testing sets have the same number of positive and negative examples, AUC scores are meaningful metrics to compare the models. SPNN outperforms all baselines in all tasks. Figure 3.3 shows the learning curves where training set sizes vary from 100 to 2000 subgraphs. Note that SPNN consistently achieves the best AUC scores. We summarize our results in Table 3.1, where we see that SPNN has significantly better AUC scores than the baselines over all tasks and datasets.

Table 3.1 also compares the performance of the **Independent** and **Joint** prediction approaches. Most methods show similar performance in both their **Independent**

and **Joint** variants. This is likely due to the fact that the pair-wise similarity methods model link formation independently. Thus, the joint representation makes no difference in the two approaches. For low-rank decomposition methods (such as Rescal and HolE), we speculate that this is because edges are conditionally independent given the model, and, thus, they are unable to learn good low-dimensional embeddings for subgraph tasks where missing edges are dependent given the model.

Finally, Table 3.2 shows the wall-clock execution times of SPNN against the baselines HolE, Rescal, and Node2Vec. The server is an Intel E5 2.60GHz CPU with 512 GB of memory. SPNN is orders of magnitude faster than HolE and Rescal and one order of magnitude faster than Node2Vec in the three tasks. Training SPNN takes around 90 seconds to sample and construct features for four-node subgraphs in DBLP, and 9 minutes for five-node subgraphs in Friendster. The significant difference in execution time is rooted in how long it takes to collect the induced subgraphs to train our model. For the relatively small two-year-sliced of DBLP, we enumerate all possible subgraphs and sample 1000 from them. For Friendster Activity and Friendster Structure tasks, we use the connected induced subgraph sampling method of Wang et al. [66] with an added bias to sample induced subgraphs of interest. In the worst case, learning SPNN takes  $O(h|\mathcal{Y}||A|^k|R|^{k^2})$  time per iteration per training example, where  $h = |\cup_n \mathcal{P}_n^{k+\delta}(\mathcal{T}_n^{(\text{sample})})|$  is the number of subgraph patterns in the pattern layer,  $|\mathcal{Y}|$  is the number of distinct patterns in subgraph classes,  $|A|$  is the number of node classes, and  $|R|$  is the number of edge classes.

**Interpreting SPNN results.** Unlike most link prediction methods, SPNN’s parameters are interpretable so that we can easily make sense of the predictions. Figure 3.4 shows the weight difference  $W_1^{(2)}(j) - W_2^{(2)}(j)$  in SPNN’s pattern layer between Class 1 and Class 2 for patterns  $F_j^\square$  in the DBLP task. Large positive values indicate subgraph patterns that encourage the appearance of both dotted links while large negative values indicate patterns that discourage the appearance of both dotted links. Figure 3.4 caption details the examples of patterns  $F_4^\square$  and  $F_2^\square$ .



**Link prediction on synthetic datasets** Besides the datasets with sequential information like DBLP and Friendster, I also test our proposed method on other famous heterogeneous datasets.

**Facebook** is a sample of the Facebook users from one university. The dataset contains 75,000 nodes and 8 million links. The heterogeneous graph includes friendship connections, user groups, political and religious views. **WordNet** is a knowledge

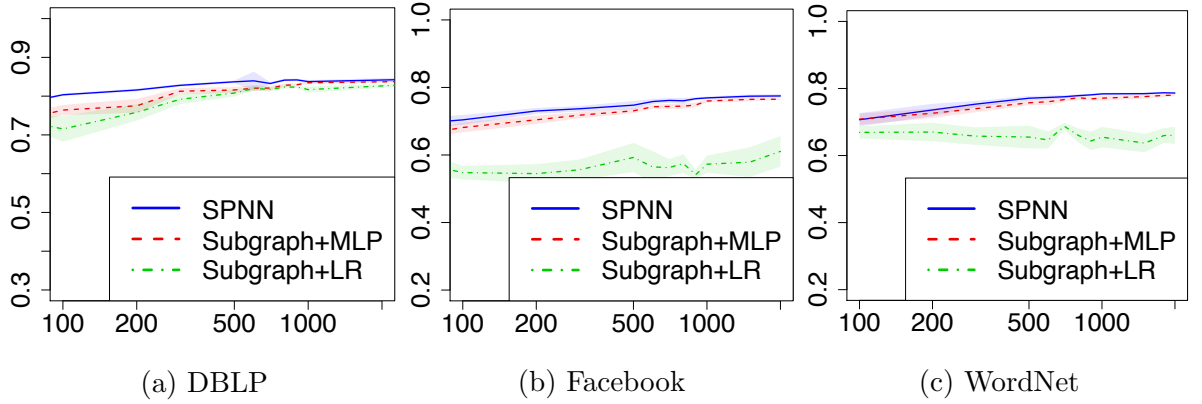


Figure 3.5.: Sequence Graph Learning curves ( $\text{AUC} \times \text{Training Size}$ ) compared to logistic regression and MLP (w/shaded 95% conf. intv.).

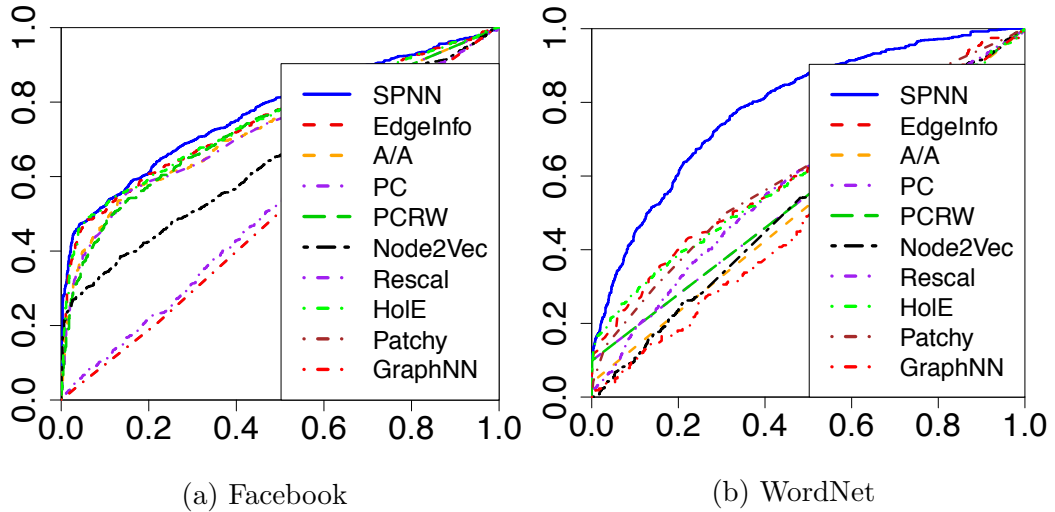


Figure 3.6.: ROC curves ( $\text{True Pos} \times \text{False Pos}$ ): Facebook and WordNet tasks.

graph that groups words into synonyms and provides lexical relationships between words. The WN18 dataset is a subset of WordNet, containing 40,943 entities, 18 relation types, and 151,442 triplets.

As discussed in Results Section, in order to learn a predictive model of subgraph evolution, I divide the data into three temporal graphs  $G_1, G_2, G_3$ . The Facebook and WordNet graphs are not dynamic, so I set  $G_3$  to be the full network, and then randomly remove the links from 10% of the subgraphs in Figure 3.7 (1)-(2) to construct  $G_2$ . Another 10% are removed from  $G_2$  to construct  $G_1$ .

Figure 3.6a-b shows the ROC curves of SPNN and the baselines with 1000 training induced subgraphs and 2000 test induced subgraphs for Facebook and WordNet. SPNN outperforms all baselines in all tasks. Figure 3.8 shows the learning curves where training set sizes vary from 100 to 2000 subgraphs. Note that SPNN consistently achieves the best AUC scores. I summarize our results in first two rows of Table 3.3, where I see that SPNN has significantly better AUC scores than the baselines over all tasks and datasets.

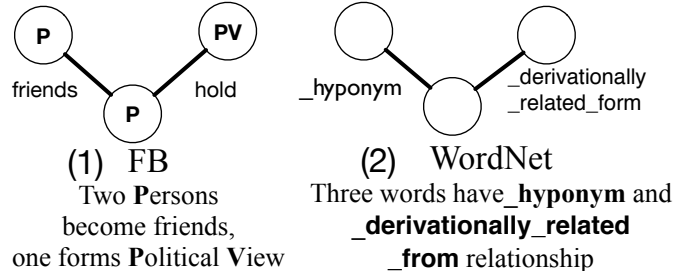


Figure 3.7.: Facebook and WordNet Prediction tasks

**Understanding Performance Gains.** To measure both the effect of (a) our induced isomorphism density features and (b) our sparse neural network architecture I compare SPNN against a logistic regression with the same input features as SPNN. The L2 regularized logistic regression verifies two things: (a) whether the deep architecture of SPNN is useful for our prediction task and (b) whether the induced isomorphism density features are more informative for our tasks than the Node2Vec,

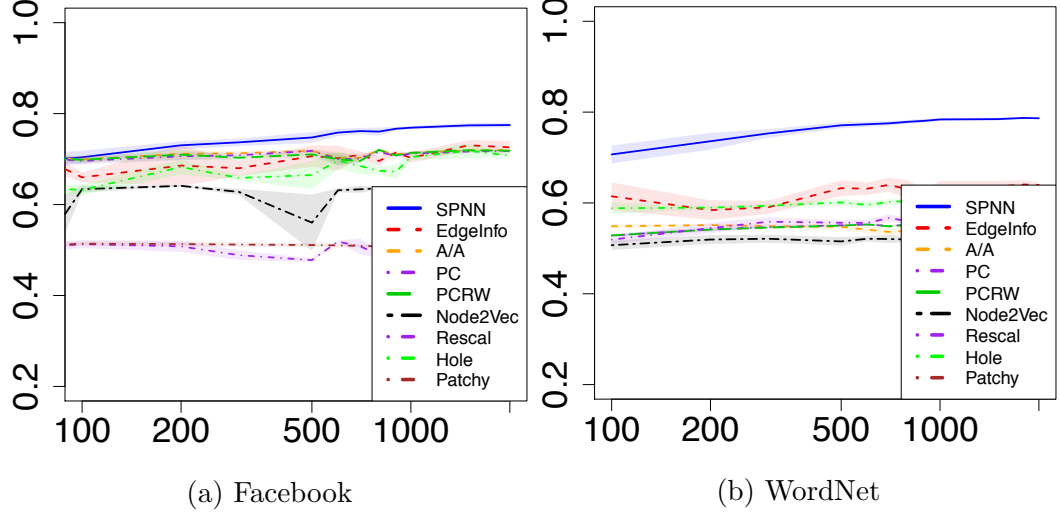


Figure 3.8.: Learning curves ( $\text{AUC} \times \text{Training Size}$ ) w/shaded 95% confidence intervals for dynamic Facebook and WordNet.

PCRW, Path Counts, and Edge features. The learning curves in Figure 3.5 show both (i) the benefit of one extra layer in the neural network and (ii) the gain in our features by contrasting the logistic regression against the learning curves of Figure 3.8.

The multi-layer perceptron (MLP) and SPNN differ in that MLP’s input layer and hidden layer are fully connected. The MLP will help us test whether SPNN’s sparse architecture is a good regularizer. The learning curves in Figure 3.5 show that SPNN outperforms MLP in majority cases with rare cases which have similar but not worse performance. This shows that the SPNN sparse architecture is indeed a good regularizer for the joint link prediction problem.

**Subgraph prediction in static graphs.** The experiments in Results Section has showed that our proposed method outperforms the state of the art in subgraph prediction on dynamic graphs. Our method can also predict missing links in static graphs such as Facebook and WordNet datasets without timestamps. 50% of the edges which belong to the two specified edge types in subgraph tasks shown in Figure 3.7 are removed randomly. To obtain positive examples, I sample or enumerate 4-node induced

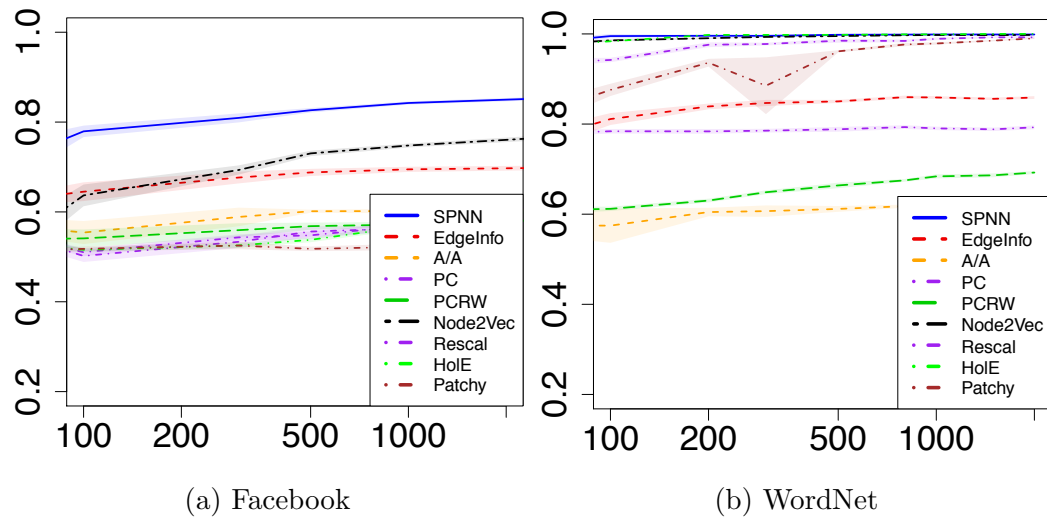


Figure 3.9.: Learning curves (AUC  $\times$  Training Size) of SPNN against competing methods in Static Graph (w/shaded 95% conf. intv.).

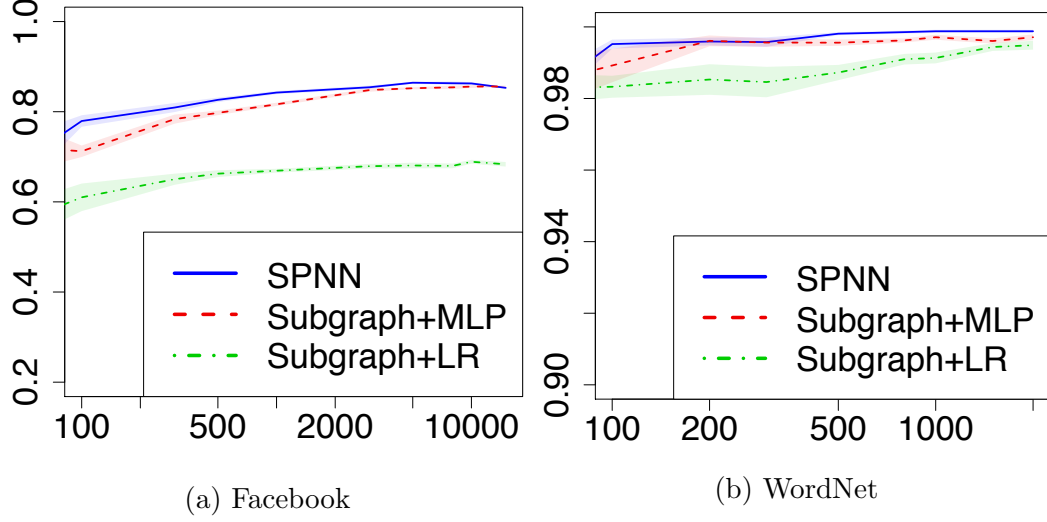


Figure 3.10.: Learning curves comparing SPNN to logistic regression and MLP in Static Graph.

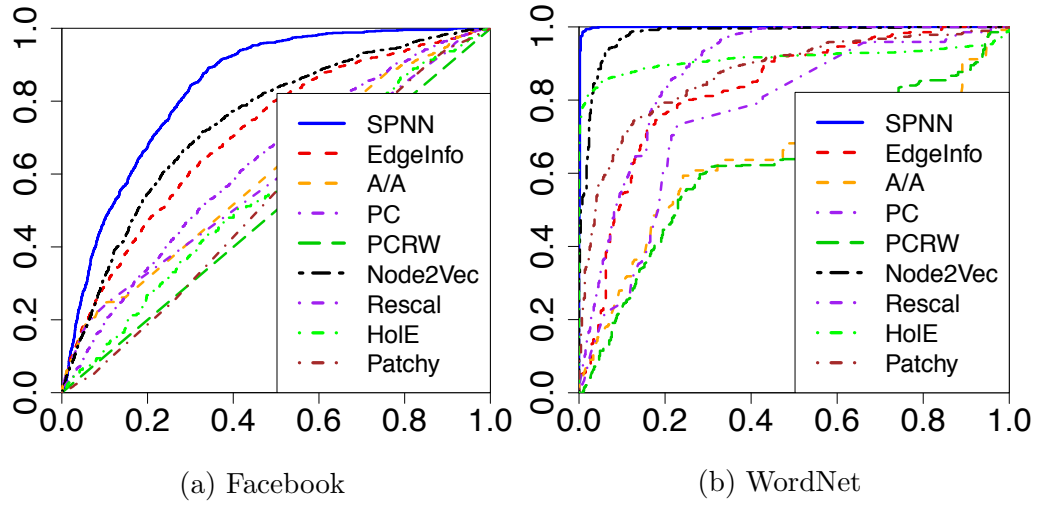


Figure 3.11.: ROC curves (True Pos  $\times$  False Pos): Facebook, WordNet tasks in Static Graph.

subgraphs  $\mathcal{T}_1^{\square(4)}(3)$  which contains the removed subgraph. Randomly sample same amount of 4-node subgraphs which do not contain the removed structure as negative examples. Last two rows of Table 3.3 shows the performance against competing

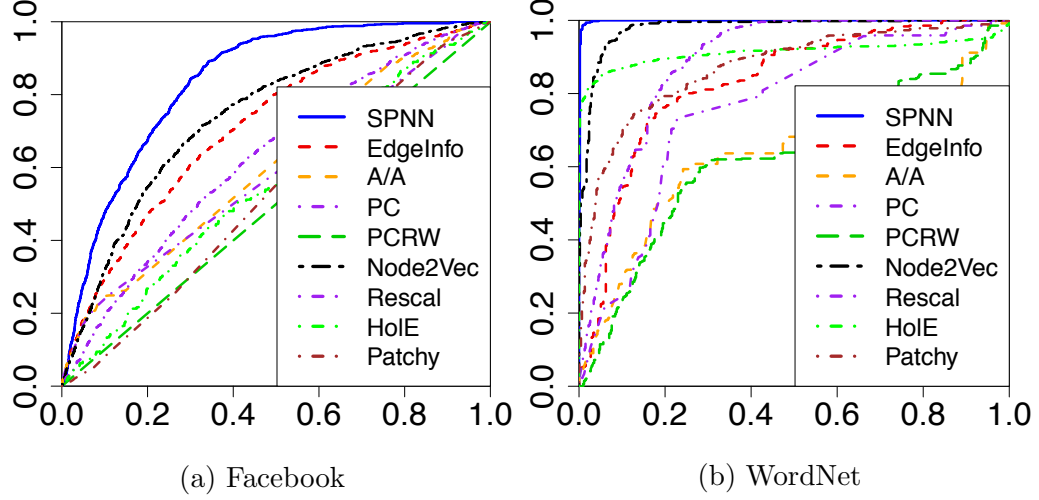


Figure 3.12.: ROC curves (True Pos  $\times$  False Pos): Facebook, WordNet tasks in manually generated dynamic graphs.

methods to predict subgraphs in static graphs. Figure 3.9 shows the learning curves. Both of these figures show that our proposed SPNN consistently achieves the best performance, compared to all other methods. Figure 3.10 shows that our proposed SPNN sparse architecture is indeed a good regularizer for the joint link prediction problem in static graphs.

Table 3.1.: Max Area Under Curve (AUC) scores of SPNN against baselines.

Independently Trained (Single Link Predictions)						Jointly Trained Multi-Link Task									
	EdgeInfo	PCRW	PC	N2V	Rescal	HolE	EdgeInfo	PCRW	PC	N2V	Rescal	HolE	Patchy	GraphNN	SPNN
DBLP	0.811	0.786	0.783	0.567	0.611	0.681	0.830	0.782	0.788	0.582	0.611	0.690	0.627	0.571	<b>0.846</b>
	$\pm 0.012$	$\pm 0.007$	$\pm 0.012$	$\pm 0.008$	$\pm 0.025$	$\pm 0.024$	$\pm 0.007$	$\pm 0.007$	$\pm 0.014$	$\pm 0.007$	$\pm 0.025$	$\pm 0.024$	$\pm 0.003$	$\pm 0.021$	$\pm 0.011$
Friendster	0.530	0.516	0.509	0.512	0.521	0.513	0.502	0.516	0.515	0.524	0.502	0.506	0.519	0.521	<b>0.690</b>
(Activity)	$\pm 0.088$	$\pm 0.007$	$\pm 0.006$	$\pm 0.011$	$\pm 0.031$	$\pm 0.006$	$\pm 0.007$	$\pm 0.012$	$\pm 0.012$	$\pm 0.018$	$\pm 0.012$	$\pm 0.013$	$\pm 0.010$	$\pm 0.023$	$\pm 0.008$
Friendster	0.568	0.501	0.501	0.501	0.558	0.501	0.501	0.502	0.552	0.540	0.521	0.530	0.547	0.523	<b>0.607</b>
(Structure)	$\pm 0.011$	$\pm 0.002$	$\pm 0.002$	$\pm 0.003$	$\pm 0.009$	$\pm 0.002$	$\pm 0.004$	$\pm 0.002$	$\pm 0.019$	$\pm 0.017$	$\pm 0.017$	$\pm 0.021$	$\pm 0.025$	$\pm 0.019$	$\pm 0.017$

Table 3.2.: Time to sample 1000 examples+training time.

	DBLP	Friendster	Friendster
		Activity	Structure
Rescal	47.3min	28h32min	28h33min
HolE	43.5min	26h21min	26h22min
node2vec	2.9min	3h51min	3h51min
SPNN	3.6min	9min	9min



Table 3.3.: Max Area Under Curve (AUC) scores of SPNN against baselines.

Independently Trained (Two Single Link Predictions)						Jointly Trained Multi-Link Task							
	EdgeInfo	PCRW	PC	N2V	Rescal	HolE	EdgeInfo	PCRW	PC	N2V	Rescal	Hole	Patchy SPNN
Facebook Dynamic	0.748	0.738	0.725	0.526	0.523	0.750	0.747	0.723	0.725	0.527	0.632	0.746	0.510
	( $\pm 0.014$ )	( $\pm 0.009$ )	( $\pm 0.011$ )	( $\pm 0.011$ )	( $\pm 0.017$ )	( $\pm 0.007$ )	( $\pm 0.003$ )	( $\pm 0.009$ )	( $\pm 0.012$ )	( $\pm 0.011$ )	( $\pm 0.007$ )	( $\pm 0.006$ )	( $\pm 0.015$ )
WordNet Dynamic	0.606	0.553	0.551	0.528	0.586	0.618	0.639	0.551	0.553	0.524	0.586	0.611	0.574
	( $\pm 0.028$ )	( $\pm 0.003$ )	( $\pm 0.004$ )	( $\pm 0.018$ )	( $\pm 0.009$ )	( $\pm 0.009$ )	( $\pm 0.023$ )	( $\pm 0.003$ )	( $\pm 0.003$ )	( $\pm 0.018$ )	( $\pm 0.009$ )	( $\pm 0.009$ )	( $\pm 0.038$ )
Facebook Static	0.578	0.543	0.568	0.781	0.665	0.674	0.703	0.592	0.521	0.781	0.664	0.672	0.522
	( $\pm 0.014$ )	( $\pm 0.020$ )	( $\pm 0.011$ )	( $\pm 0.021$ )	( $\pm 0.017$ )	( $\pm 0.016$ )	( $\pm 0.011$ )	( $\pm 0.028$ )	( $\pm 0.018$ )	( $\pm 0.011$ )	( $\pm 0.021$ )	( $\pm 0.017$ )	( $\pm 0.006$ )
WordNet Static	0.936	0.695	0.798	0.997	0.997	0.996	0.861	0.816	0.803	0.996	0.992	0.996	0.990
	( $\pm 0.006$ )	( $\pm 0.007$ )	( $\pm 0.005$ )	( $\pm 0.002$ )	( $\pm 0.001$ )	( $\pm 0.001$ )	( $\pm 0.007$ )	( $\pm 0.003$ )	( $\pm 0.003$ )	( $\pm 0.001$ )	( $\pm 0.001$ )	( $\pm 0.001$ )	( $\pm 0.001$ )

Table 3.4.: Time to sample 1000 examples + learning time.

	DBLP	Facebook	WordNet
Rescal	47.3min	55h43.2min	2h58.1min
HolE	43.5min	58h32.4min	2h53.3min
node2vec	2.9min	74.0min	10.0min
SPNN	3.6min	3.0min	14.3min

## 4 PERMUTATION INVARIANT FUNCTIONS FOR SET

### 4.1 Introduction

The increase of data volume and hardware computation power, have unleashed large neural network models into real-world applications. Input data is usually represented as fixed-length vectors. The key aspect of vector representations is that the elements need to have a specific (*i.e.*, canonical) order. However, in key applications—such as logical reasoning [10, 11], relational learning [5, 6, 7, 8, 9], scene understanding [8, 12], and object detection from LiDAR readings [3, 4]—, the inputs are better represented as *sets*. Since sets do not have a natural input order, neural network models must learn permutation-invariant representations of the input (which is described as a vector) that can capture complex relational dependencies within set elements.

The effect of input order can have a significant effect on the learned model’s quality as shown in [14], if the neural network does not take steps to learn representations that are invariant to input permutations. A more principled approach is, hence, to directly learn representations that are invariant to permutations of the input. Recently, there have been several efforts focused on directly learning permutation-invariant representations [4, 14, 15, 16, 17]. The key contribution of these works has been to provide methods that can learn representation functions which are provably invariant to input permutations.

A neural network architecture for domains with set inputs should be permutation invariant and able to process sets of any size [17]. Unfortunately, as I argue in this work, these two properties are necessary but not sufficient for large sets. In particular, the effectiveness of set-based neural network architectures is impacted by their ability to capture *long-range* and *high-order* dependence among the set elements. *Long-range dependence* refers to the model’s ability to learn dependencies between elements of

the set that are farther apart in its vectorized input. This is influenced by the architecture’s *computation graph*, through the interaction paths between input elements. *High-order dependence* refers to the model’s ability to learn complex relationships involving multiple set elements. This is influenced by choice of local functions used in the architecture, because information needs to be propagated and remembered in order to be combined into higher-order patterns. Since relational patterns are typically higher-order, this is also a critical concern for real-world applications.

Existing permutation-invariant neural network architectures have not been designed to capture both *high-order* and *long-range dependencies*. Simple pooling methods such as sum/min/max used in DeepSets [4] are limited in their ability to capture *high-order dependencies* [48]. Set transformer models [17] capture *long-range dependence* through self-attention but only capture pairwise relationships, which is not enough to capture *high-order dependencies*. Janossy Pooling [15] uses recurrent neural networks (*e.g.*, GRUs) to model high-order dependencies. However, recurrent sequence models cannot easily capture *long-range dependencies* due to the limitation of the model’s local function, which has  $O(1)$  memory.

In this work, I describe how characteristics of the architecture’s *computation graph* impact the method’s ability to model *high-order dependence* and *long-range dependence*. Specifically, the ability to capture dependence between elements can be analyzed using the length of the *interaction path* connecting the involved elements in the architecture. Shorter interaction path lengths indicate there are fewer steps in information propagation and gradient computation, which leads to better modeling interactions between set elements. The ability to model high-order dependencies mainly depends on the order of functions used to aggregate information from different input elements and intermediate results.

## 4.2 Representation Learning of Variable-size Sets

This section provides a formal problem definition of inductive set embeddings. An *inductive embedding* is a function that takes any set as input and outputs an embedding that must remain unchanged for any input that represents the same set (*i.e.*, all permutations). As sets are special cases of multisets, we will use the term multisets to define set in the next section.

### 4.2.1 Set Inputs

*Set input* is defined at Definition 4 in Chapter 2.

In this work, we are interested in functions that yield inductive set representation functions for large set inputs, and which should satisfy permutation-invariance of elements. Over  $\mathbf{X}$ , these representation functions are invariant to permutations of the rows (not columns, since those are the features). By inductive representation, we mean that the representation function is learned on a dataset but can be applied to a different test dataset.

### 4.2.2 Set Representation Functions

Let  $\Pi_n$  denote the set of all permutations on the integers  $\{1, \dots, n\}$ . We shall adopt the notation in [15] and use a double-bar (as in  $\overline{\overline{f}}$ ) to indicate that a function is invariant to permutations of the input in the sense of Definition 7. We shall use an arrow (as in  $\vec{f}$ ) to denote arbitrary (possibly row-permutation-sensitive) functions over matrices of variable dimensions. Functions over scalars, vectors, or “simple” sets whose elements are scalars or vectors, will be denoted without such annotations.

We begin by defining functions that give set representation functions (inductive embeddings).

**Invariant Set Representation Function** is defined in Definition 7 in Chapter 2.

### 4.3 Invariant Neural Network Architectures

#### 4.3.1 Ideal Representation Routing

**Definition 9 (Computational Graphs)** *A computational graph  $G = (V, E)$  is a graph with the following properties in nodes  $V$  and edges  $E$ . where:*

1.  *$V$  is a set of nodes in the computational graph. I divide the nodes into three types based on the data source and whether processing it. (1) The input node accepts the input dataset without processing. (2) The process node accepts the intermediate output from other nodes and processes it. (3) The hybrid node accepts data and processes it. The data must include the input dataset. The rest of the data is from the intermediate output from other nodes. Each process node or hybrid node has a local function to process the input and intermediate result.*
2.  *$E$  the edges in the computational graph. For each  $(j, i) \in E$ ,  $j < i$  in the topology order.*

The computational graphs of some widely-used neural network models are shown in Figure 4.1. For the DeepSets [4], 2-layer Perceptron, and Self-attention, each input node accepts one element from the set. The process node accepts output from other nodes and processes it. The number of input nodes equals the number of elements in the set. The number of process nodes depends on the hyperparameters of these architectures. For the LSTM model, it only contains hybrid nodes. Each hybrid node accepts one element from the input set and processes it. Even though a hybrid node can be decomposed into one input node and one process node, I prefer to keep it as one united node which made the number of nodes equals the number of elements in the set.

**Definition 10 (Interaction Path)** *Given a computational graph, I define  $P(j, i)$  be the shortest path between node  $j$  and node  $i$ , while ignoring the directions of edges*

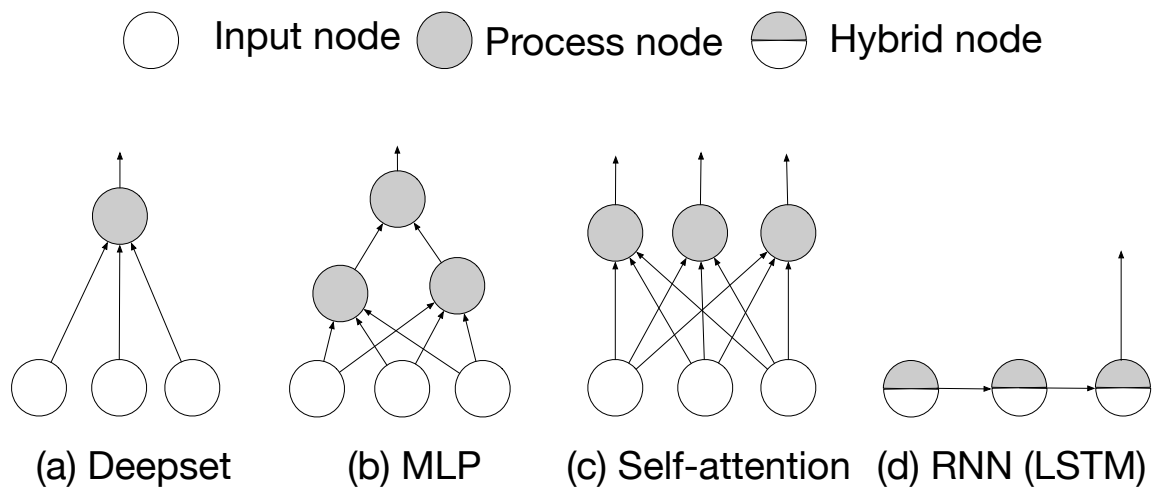


Figure 4.1.: Computational graph examples.

Table 4.1.: Computational graph properties of methods for set with  $n$  elements

	DeepSets([4])	LSTM	Set Transformer([17])	GNN	Target model
Average interaction path $p$	1	$n/2$	1	1	1
Order of dependence $k$	1	$n$	2	2	$n$

in the computational graph.  $P(j, i) = P(i, j)$ . *Interaction Path Length* is the number of nodes that has process functions executed along the interaction path.

An interaction path has the following two forms. (a)  $i$  and  $j$  are connected by a directed path.  $P(j, i)$  can be  $j \rightarrow \dots \rightarrow i$ . (b)  $i$  and  $j$  are connected to the same ancestor.  $j \rightarrow \dots \rightarrow m \leftarrow \dots \leftarrow i$ .

In the reminder of this section, I show the connections between learning long-range/high-order dependence and the computational graph.

**Definition 11 ( $l$ -range dependence)** For a set  $\mathbf{X}$  under a fixed permutation  $\pi$  denoted as  $\mathbf{X}_\pi$ , suppose  $X_i, X_j$  are two arbitrarily selected elements from  $\mathbf{X}_\pi$ . If the dependence between  $X_i$  and  $X_j$  can be captured by function  $\bar{f}(\mathbf{X})$  when  $|i - j| \leq l$ . While interaction between  $X_i$  and  $X_j$  can not be perfectly captured when  $|i - j| > l$ . The function  $\bar{f}(\mathbf{h})$  can capture at most  $l$ -range dependence.

Example of *long-range dependence*: Given a sequence containing  $n$  pictures detect whether there exists a pair of duplicate pictures. Permuting the sequence in arbitrary order, the distance between two duplicated pictures is in the range  $[1, n - 1]$ . Considering the worst-case scenario, where the duplicate elements are at the head and tail of this sequence, this task needs a model capable of capturing  $(n - 1)$ -range dependencies. If using sequence models such as LSTM to encode these  $n$  elements, the embedded state vector  $\mathbf{h}_{n-1}$  at step  $n - 1$  contains little information of the first element due to the limitation of size  $\mathbf{h}$ . At  $n$ -th step, LSTM could hardly identify whether there is duplicate based on  $X_n$  and  $\mathbf{h}_{n-1}$ .



**Remark 1** Assume  $\bar{f} : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^h$ , for some  $h \geq 1$ . Shorter interaction path between two input variables  $X_i$  and  $X_j$  has lower probability to cause errors when modeling long range dependence.

Remark 1 has the following justification: To simplify the notation, define  $\bar{g}(X_i, X_j) = \bar{f}(\dots, X_i, \dots, X_j, \dots)$ . To understand how a long computation path can be an issue when learning  $\bar{g}$ , consider  $\bar{g}(X_i, X_j) = \vec{g}_0(g_{i,1} \circ \dots \circ g_{i,n}, g_{j,1} \circ \dots \circ g_{j,m})$ , with a total path length of  $n + m$  functions connecting the inputs  $X_i$  and  $X_j$ . Assume each  $g_{\cdot,\cdot}$  is Lipschitz continuous with Lipschitz bound  $|g_{\cdot,\cdot}(X_1) - g_{\cdot,\cdot}(X_2)| \leq K|X_1 - X_2|$ . Then, the composition has Lipschitz bound  $K^{n+m}$ , that is, a small change to  $X_i$  can require an enormous effect in  $g_{j,m}$ —the function that takes  $X_j$  as input—causing instability in gradient descent, which in turn leads to optimization (learning) issues. Clearly, since the interaction path length is  $n + m$ , there are  $n + m$  gradient computations needed to backpropagate from the *interaction node* to the respective inputs  $X_i$  and  $X_j$ . More gradient computations can lead to more numerical instability, which again leads to more optimization issues [70].

Table 4.1 shows a comparison of the computational graph path length of existing set representation functions. Long short-term memory (LSTM) networks have an average path length of  $n/2$ , which is consistent with the findings [71] that LSTM suffers when modeling long sequences. DeepSets [4], Set Transformer [17] and Graph Neural Networks [5] contain average path lengths of one. The ideal model should have an interaction path length of one since it is easier to model long-range dependencies in the input.

Taking our *duplicate checking* task as an example again, a model needs to predict whether a set contains a pair of duplicate elements. For models with an interaction length of one, any two elements can be sent to a process function to detect just 1-hop step. While for LSTM, the information needs to be forwarded through multiple hops, which may lead to information loss.

Long-range dependence defines the ability to capture the interaction between elements far away from each other in the input sequence. But long-range dependence

is not the only factor determining the quality of a permutation-invariant neural network architecture. For instance, both DeepSets [4] and Set Transformer [17] have interaction path lengths of one, which means every two elements can interact within 1-hop of functions. However, [17] show that Set Transformer has a better ability to model complex dependencies in a set than DeepSets. Theoretically, this ability to model  $k$ -order dependencies has been described by Janossy Pooling [15], with  $k$ -order dependence is defined as follows:

**Definition 12 ( $k$ -order dependence [15])** *For a set  $\mathbf{X} \in \mathbb{R}^{n \times d}$  under one certain permutation  $\pi$  denoted as  $\mathbf{X}_\pi$ , define  $\downarrow_k(\mathbf{X}_\pi)$  as the first  $k$  elements in  $\mathbf{X}_\pi$  and  $k \leq n$ . The function  $\vec{f}$  accepts input of length  $k$ . The function  $\overline{\vec{f}}(\mathbf{X})$  can be decomposed into the following form  $\overline{\vec{f}}(\mathbf{X}) = \sum_{\pi \in \Pi_n} \vec{f}(\downarrow_k(\mathbf{X}_\pi))$ . The smallest  $k$  in the function is defined as the order of the interaction modeled by function  $\overline{\vec{f}}(\mathbf{X})$ .*

Example of *high-order dependence*: Taking a simple example where a set  $\mathbf{X}$  contains 3 elements,  $\overline{\vec{f}}(\mathbf{X}) = X_1 \cdot X_2 \cdot X_3$ . This 3 – order dependence cannot be decomposed into a summation of multiple 2-order dependencies  $\alpha_1(X_1 \cdot X_2) + \alpha_2(X_1 \cdot X_3) + \alpha_3(X_2 \cdot X_3)$ , with  $\alpha_i \in \mathbb{R}$ ,  $i = 1, 2, 3$ .

**Remark 2** *For any  $k \in \mathcal{N}$ , define  $\mathcal{F}_k$  as the functions can be represented by a function with  $k$ -order dependence. Then  $\mathcal{F}_{k-1}$  is a proper subset of  $\mathcal{F}_k$ . Thus, a model that can capture higher-order dependence is preferred.*

As shown in Murphy et al. [15], a model which can capture  $k$  order dependence could also capture  $k - 1$  order dependence, however, there exists  $k$ -order dependencies that cannot be captured by an architecture designed to capture  $k - 1$  order dependencies.

Table 4.1 compares the dependencies of existing methods: DeepSets [4] models 1-order dependencies, while SetTransformer [17] models 2-order dependencies. Recurrent Neural Network with Janossy Pooling [15] can model  $n$ -order dependencies. In this work, since the target is to model higher-order dependence with  $n$  as the opti-

mal goal, I will also adopt Recurrent Neural Network (GRU [72]) under the Janossy Pooling framework as our local function in *Self-Attention GRU*.

*To sum up Remarks 1 and 2, a better permutation-invariant architecture should have short interaction path lengths while being able to capture high-order dependencies.*

#### 4.4 Better Set Representation Architectures

I aim to design a neural network architecture that has a computational graph with the following properties: Property 1 that is related with long-range dependence of Remark 1. Property 2 that is related to high-order dependence of Remark 2:

1. Short maximum interaction path lengths. As shown in the previous section, shorter interaction path lengths indicate easiness of capturing long-range dependence.
2. Use local functions that can model high-order dependencies, including functions that aggregate the information from different computation nodes in the computation graph.

Besides these targeted properties, of course, the set representation function should also be permutation invariant. An arbitrary function  $\vec{f}$  could hardly satisfy this requirement. The inference and back-propagation of  $\vec{f}$  is based on the topology sorted order of the computational graph. Changing the orders of local functions in the computational graph can result in different results and gradients.

Fortunately, Janossy Pooling [15] shows how to make an arbitrary  $\vec{f}$  be permutation-invariant by summing all possible permutations of the input, as shown in Equation 4.1.

$$\overline{\vec{f}}(\mathbf{X}) = \sum_{\pi \in \Pi} \vec{f}(\mathbf{X}_{\pi}) \quad (4.1)$$

In order to make the resulting model tractable, Janossy Pooling proposes  $\pi$ -SGD to optimize the model. I use the same approach for our GRU local function.

#### 4.4.1 Existing Graph Topologies

In this section, I introduce and compare some famous graph models in graph theory. Generalized De Bruijn graph (GDBG) is a type of small-world network with the shortest path length besides the Moore graph, which could not be generalized to every  $n$ . Balanced Tree is one of the most popular topologies.

**Definition 13 (Generalized De Bruijn graph(GDBG))** *A  $(n, r)$  Generalized De Bruijn Graph is a graph with  $n$  nodes numbered from 0 to  $n - 1$  and the edges are connected as follows. The  $r$  outgoing links of node  $i$  are connected to nodes  $(i \times r) \bmod n$ ,  $(i \times r + 1) \bmod n$ , ...,  $(i \times r + r - 1) \bmod n$ , where  $\bmod$  is the modulo operator.*

**Definition 14 (Balanced Tree)** *A  $(n, r)$  balanced tree is a tree with  $n$  nodes numbered from 0 to  $n - 1$  and the edges are connected as follows. The  $r$  outgoing links of node  $i$  are connected to nodes  $(i \times r + 1)$ ,  $(i \times r + 2)$ , ...,  $(i \times r + r)$*

*From Graph to the Computational Graph:* The graph topology models introduced in the previous section contain cycles, and some of them are undirected. The computational graph needs to be a Directed Acyclic Graph (DAG). It is preferred that there is only one output node in which case the output vector of this node can be represented as the embedding for the entire computational graph.

In order to guarantee the graph does not have cycles, one simple heuristic is to change the edge directions of the original Generalized De Bruijn Graph. The edge only directs from the low-index node to the high-index node. Since no edge is removed from the original graph, the maximum computational path length is the same as the original diameter of the undirected graph.

#### 4.4.2 Build Computational Graph with Self-attention

Instead of a pre-defined graph, the graph can also be automatically learned. This approach builds a graph based on self-attention [73]. Compared to existing

approaches, the number of parameters is constant and unrelated to the number of nodes in the graph. The graph  $\mathbf{G}$  is then computed by

$$\mathbf{G} = \text{softmax}((\mathbf{X}\mathbf{W}_1) \times (\mathbf{X}\mathbf{W}_2)^T), \quad (4.2)$$

where  $\mathbf{X} \in \mathbb{R}^{n \times d}$  is a feature matrix, and  $\mathbf{W}_1, \mathbf{W}_2 \in \mathbb{R}^{d \times h}$  are two weight matrices used to learn and encode the features matrix  $\mathbf{X}$ . By computing  $(\mathbf{X}\mathbf{W}_1) \times (\mathbf{X}\mathbf{W}_2)^T$ , the covariance between encoded elements are computed, and this learned adjacency matrix  $\mathbf{G} \in \mathbb{R}^{n \times n}$  represents a weighted clique.  $\mathbf{G} = \mathbf{G}^T$ . The upper triangular of the adjacency matrix  $\mathbf{G}$  can be used to construct the computational graph. In this graph, there are only edges pointing from lower indexed elements to higher indexed elements. Hence, this computational graph is a Directed Acyclic Graph(DAG) which satisfies the requirement of a computational graph, see the example illustrated in Figure 4.2.

The number of parameters required to build the adjacency matrix  $\mathbf{G}$  is  $O(d \cdot h)$ . Even though the computational cost is  $O(n^2)$ , it can be computed efficiently since it is a parallel matrix computation. Comparing to other  $O(n^2)$  GNN methods [74], since each pair of nodes needs to be computed individually, the number of computational operations is much lower for our self-attention graph  $\mathbf{G}$ , which will show tremendous advantages in our experiment section.

#### 4.4.3 Graph Node Structure

Since I adopted the hybrid node in our computational graph, an arbitrary node  $i$  accepts one element  $X_i$  from the original input, as well as several intermediate results. These data are processed by the local function  $f_i$ . The process function can be sum model or GRU, as shown in Figure 4.3. For pre-defined unweighted graph such as Generalized De Bruijn graph or Balanced Tree,  $X'_i = X_i + \sum_{j \in \text{Child}(i)} X_j$ .

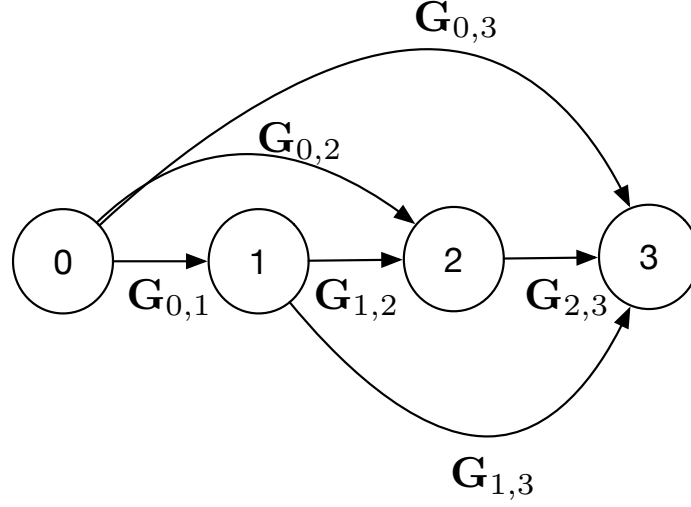


Figure 4.2.: Weighted DAG resulting from the self-attention adjacency matrix  $\mathbf{G}$ .

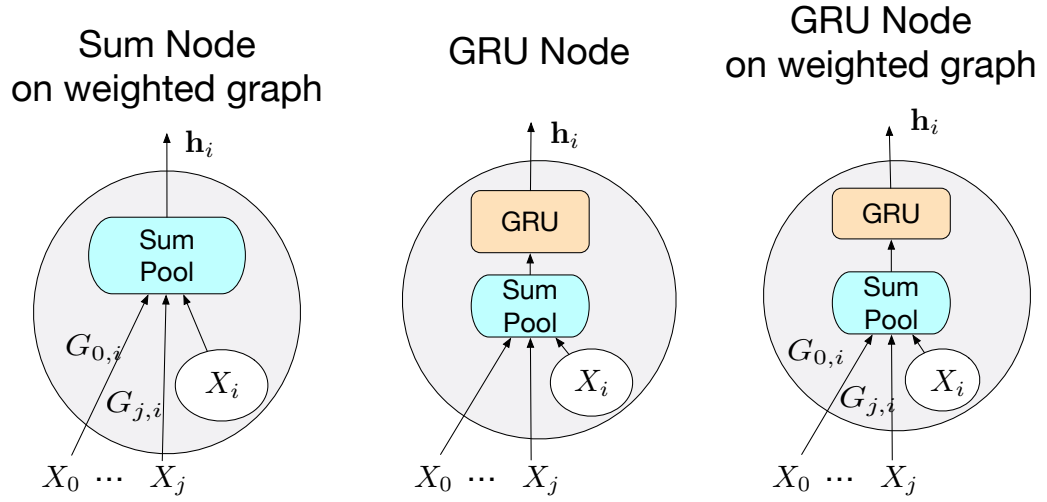


Figure 4.3.: Different Types of Hybrid Node can be adopted in the computational graph.

On the other hand, for the computational graph generated by self attention, the input to process node  $i$  is  $X'_i = X_i + \sum_{j \in \text{Child}(i)} G_{i,j} X_j$ . After the sum pooling, Gated Recurrent Unit is adopted to model the higher order dependence.

$$\mathbf{z}_i = \sigma_g(\mathbf{W}^{(z)} X'_i + \mathbf{U}^{(z)} \mathbf{h}_{i-1} + \mathbf{b}^{(z)})$$

$$\mathbf{r}_i = \sigma_g(\mathbf{W}^{(r)} X'_i + \mathbf{U}^{(r)} \mathbf{h}_{i-1} + \mathbf{b}^{(r)})$$

$$\mathbf{h}_i = \mathbf{z}_i \odot \mathbf{h}_{i-1} + (1 - \mathbf{z}_i) \odot$$

$$\phi_h(\mathbf{W}^{(h)} X'_i + \mathbf{U}^{(h)} (\mathbf{r}_i \odot \mathbf{h}_{i-1}) + \mathbf{b}^{(h)})$$

The value stored in  $X'_i$  is the aggregated input feature to the node  $i$  on the computational graph.  $\mathbf{h}_i$  is the output vector from node  $i$  on the computational graph.  $\mathbf{z}_i$  is update gate vector, while  $\mathbf{r}_i$  is the reset gate vector.  $\mathbf{W}^{(\cdot)}$ ,  $\mathbf{U}^{(\cdot)}$  are parameter matrices and  $\mathbf{b}^{(\cdot)}$  is parameter vector, while  $\sigma_g$  is a sigmoid activation function, with  $\phi_h$  as the hyperbolic tangent activation function.

An attention layer is then used to integrate  $\mathbf{h}_i, i \in [1, n]$

$$\mathbf{v}_i = \tanh(\mathbf{W} \mathbf{h}_i + b),$$

where  $\mathbf{W}$  and  $b$  are the weights and bias of another feedforward layer and  $\beta_i$  are importance weights defined as

$$\beta_i = \frac{\exp(\mathbf{v}_i^\top \mathbf{c})}{\sum_{i'} \exp(\mathbf{v}_{i'}^\top \mathbf{c})},$$

where  $\mathbf{c}$  is a parameter of the model (the context vector). Finally,

$$\vec{f}(\mathbf{X}) = \sum_{i=1}^n \beta_i \mathbf{v}_i,$$

where both  $\mathbf{v}_i$  and  $\beta_i$  depend on the input  $\mathbf{X}$ .

Since the above  $\vec{f}(\mathbf{X})$  is sensitive to permutations of  $\mathbf{X}$ , in order to achieve the permutation invariance promised by Janossy Pooling [15], I use Equation (4.1) to compute  $\bar{\bar{f}}(\mathbf{X})$  from  $\vec{f}(\mathbf{X}_\pi)$ , over all permutations  $\pi \in \Pi_n$ .

#### 4.4.4 Stochastic Optimization

Optimizing the model in Equation (4.1) is computationally prohibitive since it contains the summation of  $n!$  different permutations. To learn our model, I adopt the stochastic optimization procedure  $\pi$ -SGD of Murphy et al. [15]. The optimization goal is to learn an function  $\bar{\bar{f}}(\cdot; \theta)$  with parameters  $\theta$  which minimizes loss on the training data.

$$\theta^* = \arg \min_{\theta} \sum_{s=1}^n L(y_s, \bar{\bar{f}}(\mathbf{X}^{(s)}; \theta)). \quad (4.3)$$

Instead of summing over all possible permutations, I sample a single permutation uniformly at random from the space of permutations:  $\tilde{\pi} \sim \text{Uniform}(\Pi_n)$ , and compute the Monte Carlo estimate

$$\widehat{\widehat{f}}(\mathbf{X}) = \vec{f}(\mathbf{X}_{\tilde{\pi}}). \quad (4.4)$$

The estimate in Equation (4.4) is unbiased, since:

$$\mathbb{E}_{\tilde{\pi}}[\widehat{\widehat{f}}(\mathbf{X}; \boldsymbol{\theta})] = \vec{f}(\mathbf{X}; \boldsymbol{\theta}).$$

The details of the optimization are shown in Algorithm 1. The Monte Carlo estimate can be used to effectively infer Equation (4.1) by performing a forward pass over the Self-Attention GRU with randomly sampled permutations. The complexity of gradient computation at each optimization step is the same as if I gave  $\vec{f}$  the original input  $\mathbf{X}$  rather than  $\mathbf{X}_{\tilde{\pi}}$ .

---

**Algorithm 1:** Stochastic optimization for learning Self-Attention GRU.

---

**Input:** Labeled set training examples  $\{(\mathbf{X}^{(s)}, y_s)\}_{s=1}^n$ ;

**Input:** Self-Attention GRU model  $\vec{f}(\mathbf{X}; \boldsymbol{\theta})$  with unknown parameters  $\boldsymbol{\theta}$ ;

**Input:** Loss function  $L(y, \hat{y})$ ;

**Input:** Number of optimization epochs  $T$ ;

**Input:** Mini-batch size  $B$ ; learning-rate schedule  $\{\eta_t\}_{t=1}^T$ ;

**Output:** Learned parameters  $\boldsymbol{\theta}$  for the model  $\vec{f}(\mathbf{X}; \boldsymbol{\theta})$ .

```

1 Initialize parameters  $\boldsymbol{\theta}^{(0)}$  ;
2 for  $t = 1, \dots, T$  do
3    $\mathbf{g}_t \leftarrow \mathbf{0}$  ;
4   for  $s$  in mini-batch-indices do
5      $\widetilde{\mathbf{X}}^{(s)} \leftarrow$  Permute the rows of  $\mathbf{X}^{(s)}$  ;
6      $\mathbf{g}_t \leftarrow \mathbf{g}_t + \frac{1}{B} \nabla_{\boldsymbol{\theta}} L(y_s, \vec{f}(\widetilde{\mathbf{X}}^{(s)}; \boldsymbol{\theta}))$  ;
7    $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta_t \mathbf{g}_t$  ;
8 return  $\boldsymbol{\theta}^{(T)}$ 
```

---



## 4.5 Experiments

In this section, I demonstrate the utility of our proposed model by experimenting on a variety of set tasks ranging from arithmetic tasks, to vertex classification on real-world graphs, to expanding a bag of words in natural language processing, to predicting complex point cloud tasks.

**Baselines:** I evaluate our proposed model against existing set models summarized below:

DEEPSETS [4]: A feed-forward neural network model with sum/max pooling to achieve permutation-invariance.

J-GRU [15]: The vanilla Gated Recurrent Unit with attention layer uses Janossy pooling [15] to model permutation invariant functions.

SETTRANSFORMER [17]: A transformer for set with self-attention.

GNN [74]: Graph Neural Network which models a set as a complete graph and elements as node on the graph.

TREEGRU: Use the balanced tree to build a computational graph, use GRU as the aggregation function.

GRAPHGRU: Use Generalized De Bruijn Graph to build a computational graph, use GRU as the aggregation function.

SELF-ATTENTION [73]: Use 1 layer of Scaled Dot-Product Attention.

SELF-ATTENTION GRU: Our proposed model which uses computational graph generate by self-attention.

**Computational complexity:** For a set with  $n$  elements, the time complexity of DEEPSETS, J-GRU, TREEGRU, GRAPHGRU are proportional to the size of the input data:  $O(n)$ . For SETTRANSFORMER, SELF-ATTENTION and *Self-Attention GRU* are  $O(n^2)$ . GNN has the complexity of  $O(n^2k)$  where  $k$  is the number of GNN iterations needed. I use  $k = 6$  layers following the experimental setting of the original Xu et al. [74] paper that proposed the approach.

For DEEPSET [4], J-GRU [15], SETTRANSFORMER [17], I adopt the authors’ implementations.<sup>1</sup> Details on the models and training procedures are provided in the Appendix. Our source code to reproduce all of our experiments will be released after acceptance.

#### 4.5.1 Arithmetic Tasks on Sequence of Integers

Following prior work [4, 15], I start the evaluation of our proposed model on simple arithmetic tasks. Since an ideal set model should represent both *long-range dependence* and *high-order dependence*, I designed the following two tasks to have an emphasis on these two requirements.

1. *Double count*: given a sequence of  $n$  integers ranging from 0 to  $\lceil 1.5 \times n \rceil - 1$ , check whether there is a duplicate integer. This is a binary classification task with half positive and half negative labels.
2. *Unique count*: given a sequence of  $n$  integers ranging from 0 to  $n - 1$ , count the number of unique elements. This is a multi-class classification task with the class label as the number of unique integers.

The *double count* task emphasizes long-range dependencies, since the distance between the duplicate integers can be as long as the sequence length. The order of dependence for *double count* task is two (order-2) since every possible pair of elements should be compared in order to detect whether there is a duplicate. Figure 4.4 shows that SELF-ATTENTION and *Self-Attention GRU* get almost 100 percent accuracy from set sizes from 10 to 100, since they can capture both order-2 and long-range dependencies. The accuracy of J-GRU drops significantly after length 30 since the  $O(1)$  memory does not have enough capacity to model long-range dependence. TREE-

---

<sup>1</sup><https://github.com/manzilzaheer/DeepSets>  
[https://github.com/juho-lee/set\\_transformer](https://github.com/juho-lee/set_transformer)  
<https://github.com/PurdueMINDS/JanossyPooling>

GRU and GRAPHGRU have less significant decay with the increase of set size than J-GRU, since the shorter interaction path length helps keep longer-range dependence.

The *unique count* task results are shown in Figure 4.5. The *unique count* is a high-order dependence version of the *double count* task, since I can apply *double count* function  $n$  times to perform the *unique count* task. The decay in performance of J-GRU, TREEGRU and GRAPHGRU is consistent with the analysis of *double count*. In this task, *Self-Attention GRU* works better than SELF-ATTENTION after set size 50 mainly because the GRU of *Self-Attention GRU* can better capture higher-order relationships than sum pooling of SELF-ATTENTION.

In order to further evaluate the effects of the interaction path length of our proposed model *Self-Attention GRU*, I randomly prune the edges in the weighted graph generated by self-attention. The statistics of diameters and average path length under different prune rates are shown in Table 4.2. As the result shown in Figure 4.6, larger prune rates, which indicate longer interaction path length will lead to lower accuracy in the task.

In order to evaluate the efficiency of different models, I measure the number of parameters of the model and the number of operations in training calculated by CHOP library<sup>2</sup>. Taking *unique count* with  $n = 100$  as an example, the result is shown in Table 4.3. The number of ops is the number of operations needed to train a 64-size minibatch in one iteration. As shown in Table 4.3, DEEPSET has the smallest number of parameters and ops since each element is encoded separately and aggregation method is simple. For J-GRU, TREE-GRU and GRAPH-GRU, these three models have the same number of parameters. Since the computational graph is predefined and no additional parameters are needed. The number of ops is higher in TREE-GRU and GRAPH-GRU since they need to aggregate the intermediate results from multiple incoming nodes. Regarding SETTRANSFORMER, since multiple layers of Self-Attention are stacked together to model higher-order relationships, the number of parameters and ops are much higher than SELF-ATTENTION. For GNN,

<sup>2</sup><https://github.com/Lyken17/pytorch-OpCounter>

the number of ops is enormous, mainly because each pair of elements need to be calculated separately instead of using matrix multiplication to compute in parallel. The number of parameters of our proposed model *Self-Attention GRU* is moderate considering the consistently better accuracy than the baselines.

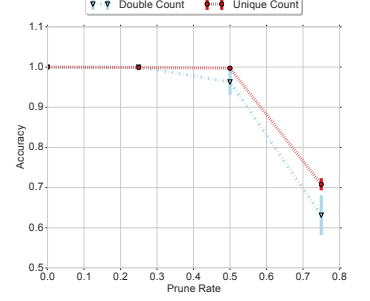
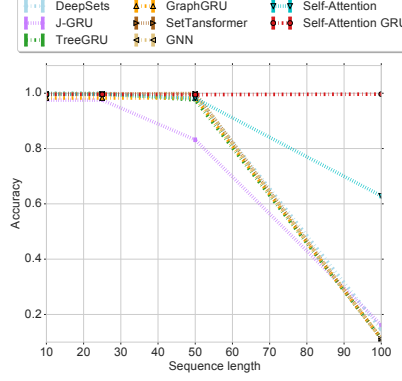
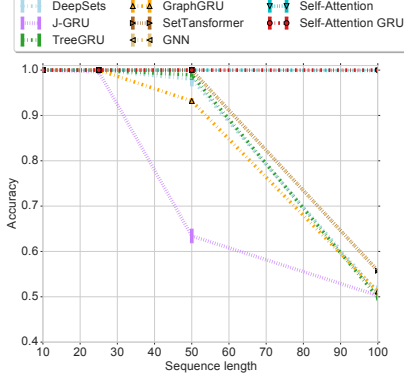


Figure 4.4.: Double Count: Figure 4.5.: Unique Count: *Self-Attention GRU* when check duplicates for sequence count of unique elements for randomly pruning edges of length  $n$  with  $1.5 \times n$  vocabulary size. sequence length  $n$  with  $n$  vocabulary size. computational graph at different prune rate.

Table 4.2.: Diameters and average path length under different prune rate for graph with 100 nodes.

Prune Rate	0	0.25	0.5	0.75
Diameter	1	2	2	3
Average Path	1	1.25	1.5	1.75

#### 4.5.2 Vertex Classification

I evaluate our proposed model in the task of vertex classification using supervised graph vertex embedding. I adopt the same experiment setting as [15] which used

GRAPHSAGE [5] framework to perform vertex classification. GRAPHSAGE contains an operation to aggregate neighbor information. Since the orders of neighbor nodes should not affect the representation learning, they can be considered as a permutation-invariant set of features. Different set models are applied to test their performance. The statistics of the used dataset *Cora* and *Pubmed* are shown in the Appendix.

As the result shown in Table 4.4, The accuracy of *Cora* stays roughly the same for different methods mainly since one thousand training examples are not sufficient to well-train a deep neural model. For *Pubmed* dataset, J-GRU, TREEGRU and GRAPHGRU shows slightly better performance than DEEPSET since they can model higher-order dependence than DEEPSET. Our proposed *Self-Attention GRU* shows better performance than the rest since it can model higher-order and long-range dependence at the same time. The advantage of *Self-Attention GRU* against J-GRU is moderate since the size of the neighbor set is small, which is within the range dependence of J-GRU.

Table 4.3.: Number of Ops and Parameters for Arithmetic Tasks *Unique Count* per minibatch in one iteration.

	DeepSet	J-GRU	SetTransformer	GNN	TreeGRU	GraphGRU	Self-Attention	Self-Attention GRU
<b>Parameter(Million)</b>	0.08	0.09	0.65	0.08	0.09	0.09	0.03	0.59
<b>Total ops(Billion)</b>	0.42	1.55	2659.57	311428.8	4.0	4.1	49.73	621.45

Table 4.4.: MicroF1 score (standard deviations) using different aggregation functions in a GNN – GRAPHSAGE.

	DeepSet	J-GRU	SetTransformer	GNN	TreeGRU	GraphGRU	Self-Attention	Self-Attention GRU
<b>Cora</b>	0.860 (0.008)	0.860 (0.008)	0.850 (0.009)	0.850 (0.010)	0.853 (0.010)	0.850 (0.009)	0.855 (0.008)	0.860 (0.008)
<b>Pubmed</b>	0.881 (0.011)	0.889 (0.010)	0.872 (0.010)	0.883 (0.010)	0.892 (0.012)	0.889 (0.011)	0.891(0.011)	<b>0.911</b> (0.010)

### 4.5.3 Natural Language Processing Tasks

Even though Natural Language Processing tasks mainly focus on sequences instead of sets, texts can be processed as a bag of words, which are sets. In this experiment, I only use text as a set of words to evaluate the performance of different set models instead of competing state-of-the-art NLP models. The widely-used General Language Understanding Evaluation (GLUE) is adopted as benchmark [53]. I select the sentence classification tasks *SST-2* and *CoLA* from GLUE, since others are question-answer tasks. SST-2 is a movie review dataset with 67k training and 1.8k testing examples. CoLA is a miscellaneous dataset with 8.5k training and 1k testing examples. Table 4.5 shows our results: The difference in accuracy in CoLA dataset is not significant, since the dataset is not large enough to train a deep neural network model. For the SST-2 dataset, our proposed model achieved significantly better performance than the baselines. It is mainly because high-order dependence is needed to understand the complex interactions between multiple words; moreover, long-range dependence is needed to propagate the information to words in different positions of the randomly-ordered input sequence.

Table 4.5.: Accuracy (standard deviations) for two GLUE classification tasks.

	DeepSet	J-GRU	SetTransformer	GNN	TreeGRU	GraphGRU	Self-Attention	Self-Attention GRU
<b>CoLA</b>	0.682 (0.007)	0.688 (0.006)	0.689 (0.008)	0.662 (0.005)	0.686 (0.006)	0.678 (0.008)	0.688 (0.008)	0.689 (0.007)
<b>SST-2</b>	0.761 (0.004)	0.765 (0.005)	0.769 (0.007)	0.746 (0.008)	0.771 (0.010)	0.761 (0.007)	0.766 (0.007)	<b>0.782 (0.008)</b>

### 4.5.4 Point Cloud Classification

*Point-clouds* (*i.e.*, sets of low-dimensional vectors in Euclidean space) gathers much more attention with the rise of autonomous driving using LIDAR data [3].

Table 4.6.: Accuracy (standard deviations) on Point Cloud label classification and label counting.

Architecture	Classification			Count unique label		
	100 pts	1000 pts	5000 pts	100 pts	1000 pts	5000 pts
DeepSet(Max pool)	0.82 (0.02)	0.87 (0.01)	<b>0.90 (0.003)</b>	0.79 (0.01)	0.82 (0.01)	0.82 (0.01)
Set Transformer(Max pool)	0.82 (0.01)	<b>0.89 (0.01)</b>	<b>0.90 (0.010)</b>	0.58 (0.03)	0.61 (0.02)	0.63 (0.03)
Set Transformer(PMA)	<b>0.84 (0.01)</b>	0.86 (0.01)	0.87 (0.01)	0.55 (0.03)	0.56 (0.03)	0.57 (0.03)
J-GRU	0.82 (0.01)	0.85 (0.01)	0.86 (0.01)	0.75 (0.01)	0.82 (0.01)	0.82 (0.02)
GNN	0.82 (0.01)	0.84 (0.01)	0.87 (0.01)	0.78 (0.01)	0.84 (0.01)	0.84 (0.02)
TreeGRU	0.82 (0.01)	0.84 (0.01)	0.86 (0.01)	0.76 (0.01)	0.83 (0.01)	0.84 (0.02)
GraphGRU	0.81 (0.01)	0.83 (0.01)	0.85 (0.01)	0.77 (0.01)	0.83 (0.01)	0.84 (0.02)
SelfAttention	0.76 (0.02)	0.83 (0.02)	0.84 (0.01)	0.74 (0.01)	0.77 (0.01)	0.82 (0.02)
Self-Attention GRU	<b>0.84 (0.01)</b>	0.88 (0.01)	0.88 (0.01)	<b>0.81 (0.01)</b>	<b>0.86 (0.01)</b>	<b>0.87 (0.01)</b>

Table 4.7.: Accuracy (standard deviations) for subset sum problem: Given a set with  $n$  integers from  $[-200, 200]$ , decide whether it contains a subset which sums to 0.

Model	DeepSet	SetTransformer	J-GRU	GNN	TreeGRU	GraphGRU	Self-Attention	Self-Attention GRU
$n = 6$	0.662 (0.002)	0.710(0.002)	0.721(0.002)	0.742(0.002)	0.731(0.002)	0.733(0.002)	0.723(0.002)	<b>0.752(0.001)</b>
$n = 10$	0.710 (0.002)	0.732(0.002)	0.742(0.002)	<b>0.762(0.003)</b>	0.741(0.002)	0.743(0.002)	0.731(0.002)	<b>0.771(0.003)</b>

I evaluated our model on the **ModelNet40** [75] dataset. It contains three-dimensional objects in 40 classes. Each set is a set of  $n$  3-dimensional vectors in  $\mathbb{R}^3$ . The experiments are evaluated with input set size  $n \in \{100, 1000, 5000\}$ .

I construct two tasks in order to evaluate the performance of the different neural network architectures:

*Classification:* This is the classic ModelNet classification task. Each set is one point cloud with one among 40 different labels.

*Unique-label counting:* Each set consists of 10 randomly selected point-clouds from the database. And mix the point from these 10 point clouds and save in an aggregated set. The task is to predict the number of unique object types (labels) in the aggre-

gated set. This is a harder task that can be applied in self-driving when detecting multiple mixed objects simultaneously.

The first three columns of Table 4.6 shown the *classification* accuracy. MAX pooling shows better performance than pooling in multi-head attention and GRU, which is the suggested approach for Point Clouds by [3]. For point clouds with a larger number of points, the accuracy is higher since more points provide more information to predict the class label.

For *Unique-label counting* task, the results are shown in the last three columns of Table 4.6. *Self-Attention GRU* has significantly better results than competitors. Compared with DEEPSET and SETTRANSFORMER, *Self-Attention GRU* can model higher-order relationships and thus got higher accuracy. Compared with J-GRU, TREEGRU, GRAPHGRU, *Self-Attention GRU* also got better accuracy because it can model longer-range dependence since the interaction path length is much shorter. The accuracy improvement from 100 points to 1000 points is more significant than the improvement from 1000 points to 5000 points which implies that 1000 elements in a set provide enough information to count the label.

#### 4.5.5 Reasoning tasks

Besides previous classic set problems, our proposed model can also be used in reasoning tasks. For instance, the subset-sum is a critical decision problem in complexity theory and cryptography with NP-complete complexity [76]. This experiment uses the same problem setting as [74]. Given a set with  $n$  integers uniformly sampled from integers from between -200 and 200, check whether there exists a subset that sums to zero. This is also a set task, since the task is input order-invariant, and the target is a binary classification label. Table 4.7 shows the results. Models with  $n$ -order dependence capability such as J-GRU, TREEGRU, GRAPHGRU and *Self-Attention GRU* perform better than 1-order DEEPSET and 2-order SETTRANSFORMER. By iterating 6 times, GNN combined 2-order dependence into higher-order dependence



and also achieved compelling performance. But it is not practical for large sets since  $\log_2(n)$  iterations are needed to capture  $n$ -order dependence. Hence, *Self-Attention GRU* also has an advantage in this task.

## 5 PERMUTATION INVARIANT FUNCTIONS FOR SET-OF-SETS

### 5.1 Introduction

While these recent works provide principled approaches to learning over sets, they are not directly applicable to tasks where the data comprise *sets-of-sets* (SoS). For example, subgraph prediction tasks in relational data involve a *set of nodes*, each of which has a *set of neighbors*. In LIDAR scene classification, the data consist of sets of LIDAR readings, with each reading being a *point-cloud*. Set-of-sets also arise in logical reasoning, multi-instance learning, among other applications (see Section 5.4 for a few examples). In these tasks, the embedding function to be learned needs to be invariant to two levels of permutations on the input data—within each set, and among the sets. Effective neural network architectures that learn inductive SoS embeddings need to efficiently take into account both levels of permutation-invariance.

In this work, I formalize the problem of learning *inductive embedding* functions over SoS inputs, and explore neural network architectures for learning inductive set-of-sets embeddings. I shall use the terms *set* and *multisets* (sets with repeated elements) interchangeably, as our techniques work on both scenarios.

I begin by proving that inductive embeddings for sets [4, 15] are not powerful enough to be used for SoS tasks. I then propose a general framework for learning inductive SoS embeddings that preserve SoS permutation-invariance by extending the characterization of sets in [15] to sets-of-sets.

Our proposed framework allows us to apply sequence models to learn inductive SoS embeddings. Under our framework, I propose *HATS*, a *hierarchical*, bidirectional long short-term memory (LSTM) network with *attention* mechanisms. A first bidirectional LSTM takes as input a sequence given by a random ordering of the elements

in a set. This LSTM is applied to each member set of an SoS (in random order), and the outputs are fed into a second bidirectional LSTM. The attention mechanism helps make both LSTMs more robust to input orderings. The HATS architecture uses the concept of *hierarchical attention* for sequence models, first introduced in document classification by [57]. However, unlike the model in [57] (which is permutation-sensitive), HATS is based on our SoS embedding framework and uses *permutation sampling* to perform stochastic optimization. This allows HATS to learn inductive SoS embeddings that are provably invariant to SoS permutations. At inference time, I adopt an efficient Monte Carlo procedure that approximately preserves this invariance.

In experiments, I show that HATS significantly outperforms existing approaches on various predictive tasks involving SoS inputs. While the SoS permutation-invariance of HATS is only approximate due to our reliance on Monte Carlo sampling for inference, our experiment results demonstrate that in practice HATS achieves significant performance-gains over state-of-the-art approaches.

## 5.2 Inductive Embeddings of Set-of-Sets

In this section, I provide a formal definition of inductive SoS embeddings. An *inductive embedding* is a function that takes any set-of-sets as input (including set-of-sets not observed in the training data) and outputs an embedding that must remain unchanged for any input that represents the same set-of-sets. This is in contrast to transductive embedding methods such as matrix and tensor factorizations, which cannot be directly applied to new data and do not directly consider set-of-sets inputs. For convenience, henceforth I shall use the term *sets* to refer to both sets and *multisets*. Multisets are sets that may contain duplicate elements [77]. As sets are special cases of multisets, I will use the term multisets to define set-of-sets in the next section.

*Sets-of-Sets input* is defined at Definition 6 in Chapter 2.

In this work, I shall be interested in functions that yield inductive SoS embeddings for set-of-sets inputs, and which should satisfy permutation-invariance on two different levels: (i) *element-level*: the function should be invariant to permutations of elements within each member set; and (ii) *member-set-level*: the function should be invariant to permutations of the member sets within an SoS. I formalize these notions in the next section.

### 5.2.1 Inductive SoS Embeddings

Notation.

Let  $\Pi_n$  denote the set of all permutations on the integers  $\{1, \dots, n\}$ . I shall adapt the notation in [15] and use a double-bar (as in  $\overline{\overline{f}}$ ) to indicate that a function taking SoS inputs is invariant to permutations in the sense of Definition 8. I shall use an arrow (as in  $\vec{f}$ ) to denote arbitrary (possibly permutation-sensitive) functions over matrices of variable dimensions. Functions over scalars, vectors, or “simple” sets whose elements are scalars or vectors, will be denoted without such annotations.

#### Definition and Characterization

I begin by defining functions that give inductive sets-of-sets (SoS) embeddings.

The **Inductive Set-of-Sets Embeddings** is defined in Definition 8 in Chapter 2. In what follows I show that set-of-sets inductive embeddings cannot be achieved with inductive set embeddings proposed in Deep Sets [4] or Janossy pooling [15], which are the two most general representation learning approaches for sets.

The following proposition shows that the inductive SoS embeddings cannot be represented simply as a inductive set embedding by any clever transformation of the input.

**Proposition 1** *Consider an SoS tensor  $\mathbf{X}$  and an inductive SoS embedding  $\overline{\overline{f}}$  of  $\mathbf{X}$  satisfying Definition 8. Consider a valid encoding of  $\mathbf{X}$  into a vector  $\mathbf{a}$ . Then, there*

exists a tensor  $\mathbf{X}$  such that no permutation invariant function  $\bar{g}$  over  $\mathbf{a}$  gives the same output as  $\bar{f}$ .

**Proof** Consider  $\mathbf{X}$  a tensor with the same  $d$ -dimensional elements over all sets. Let  $\bar{f}$  be a function that returns the maximum number of non-null elements in a row of  $\mathbf{X}$  (i.e., the maximum set size). Let  $\mathbf{a}$  be a vectorization of  $\mathbf{X}$ . If  $\mathbf{a}$  does not have null markings  $\#$  denoting the *end-of-a-set*, clearly the number of elements in each set is unrecoverably lost. If  $\mathbf{a}$  has null  $\#$  markings, then the embedding  $\bar{g}$  that returns the maximum number of non-null elements in a row of  $\mathbf{X}$  cannot be a set embedding, because it would need to act equally over all permutations of  $\mathbf{a}$ , which, again, would lose information about the set sizes. ■

Now that I have shown that set embeddings are not powerful enough to represent set-of-sets, I will propose an alternative representation function motivated by the work of [15] and the concept of *Janossy densities* in the theory of point processes [78], I can characterize any scalar- or vector-valued SoS permutation-invariant function as the average of another SoS function sensitive to SoS-type permutations over all possible member-set-level and element-level permutations:

**Theorem 1** *Given an universal approximator neural network  $\vec{f}$  (as that described by [79]) which is sensitive to permutations in the input, and an  $n \times m \times d$  tensor  $\mathbf{X}$  representing a set-of-sets as defined in Definition 6, consider the function*

$$\bar{f}(\mathbf{X}) = \frac{1}{n! \cdot (m!)^n} \sum_{\phi \in \Pi_n} \left[ \sum_{\pi_1 \in \Pi_m} \cdots \sum_{\pi_n \in \Pi_m} \vec{f}(\mathbf{X}_{\phi, \pi(\phi)}) \right], \quad (5.1)$$

where  $\mathbf{X}_{\phi, \pi(\phi)}$  denotes the tensor with  $(i, j)$ -th entry  $A_{\phi(i), \pi_{\phi(i)}(j)}$ , as given in Definition 8. Then,  $\bar{f}(\mathbf{X})$  can approximate the most powerful SoS embedding representation of  $\mathbf{X}$  arbitrarily well.

**Proof** It is straightforward to verify that the function  $\bar{f}$  as defined in Equation (5.1) satisfies the requirement of Definition 8. I show that  $\bar{f}(\mathbf{X})$  can arbitrarily approximate the most powerful SoS embedding representation of  $\mathbf{X}$  by contradiction. Assume

there is an inductive SoS embedding function  $\overline{\overline{f}}'$  that is not universally approximated by  $\overline{\overline{f}}(\mathbf{X})$ . Then, there exists a permutation-sensitive function  $\vec{f}'$  that adds a term to  $\overline{\overline{f}}'(\mathbf{X}_{\phi, \pi(\phi)})$  that is permutation-sensitive, which vanishes when averaged over all such permutations. Thus, the universal approximator  $\vec{f}$  of [79] that vectorizes the input  $\mathbf{X}_{\phi, \pi(\phi)}$  cannot approximate a permutation-sensitive function  $\overline{\overline{f}}'$ , which contradicts the fact that  $\vec{f}$  is an universal approximator of permutation-sensitive functions, concluding the proof. ■

Theorem 1 provides critical insight into how an inductive SoS embedding  $\overline{\overline{f}}$  could be modeled. In particular, while it is intractable to directly model  $\overline{\overline{f}}$ , one must instead focus on tractable approaches to learn  $\vec{f}$ . And because modeling  $\vec{f}$  does not have to obey any permutation-invariance constraints, I am now open to various permutation-sensitive models. Thus, as long as one has a sufficiently flexible model for  $\vec{f}$ , one could *in principle* learn any SoS representation  $\overline{\overline{f}} : \mathbf{X} \mapsto y$  mapping an SoS to a target value  $y$  (e.g., class label in classification tasks or real-valued response in regression tasks). Before investigating how to apply Equation (5.1) to learn inductive SoS embeddings in practice, I first demonstrate the utility of the characterization in Equation (5.1) with a few examples.

### Examples of SoS Embedding Tasks

I discuss several representative examples of inductive SoS embeddings  $\overline{\overline{f}}$ . In Section 5.4, I will demonstrate that these functions could be effectively learned in practice by exploiting Equation (5.1).

*Basic set/multiset operations.* Possibly the simplest examples of SoS representation functions on sets-of-sets are those that involve basic set/multiset operations, such as set union and intersection. These allow one to compute various population statistics for an SoS, such as counting or summing-up the (unique) elements in the union or intersection across all the member sets in an SoS.

*Adamic/Adar index.* In social network analysis, the *Adamic/Adar index* [80] is a simple and popular measure of the similarity between any two nodes in a network, which could be used to predict unseen links between nodes. For a node  $v$  in the network, denote its set of neighbors by  $N_v$ , then the Adamic/Adar index between any two nodes  $u$  and  $v$  is defined as

$$g(u, v) = \sum_{x \in N_u \cap N_v} \frac{1}{\log |N_x|}, \quad (5.2)$$

where  $|N_x|$  gives the *degree* of node  $x$ . Compared to other similarity measures such as the Jaccard coefficient, the Adamic/Adar index down-weights the importance of shared neighbors with very large neighborhoods. To see how Equation (5.2) could be cast in the form of Equation (5.1), let  $\mathbf{X}^{(u,v)}$  be an SoS consisting of two sets  $X_u$  and  $X_v$ , corresponding to the neighborhoods of nodes  $u$  and  $v$ , respectively. Specifically, I set  $X_u = \{(x, |N_x|) : x \in N_u\}$  (and similarly for  $A_v$ )—that is,  $X_u$  contains both the identifier and the degree of each neighboring node. Then, it is clear that  $g(u, v)$  can be expressed as an inductive SoS embedding  $\bar{g}(\mathbf{X}^{(u,v)})$ .

*Multi-instance learning.* In multi-instance learning, the training data consists of not a set of instances, but a set of labeled *bags*, each containing many instances. Thus, the learner seeks to learn a function  $f$  mapping a set to a class label. In many applications, such as predicting hyper-links between subgraphs and anomaly detection with collections of point-clouds (see Section 5.4 for more details on both tasks), each bag is also a set, and the function to be learned is an inductive SoS embedding that is invariant to permutations both across bags, and across the instances within each bag.

### 5.3 Learning Inductive SoS Embeddings

In this section, I explore approaches to learning inductive SoS embeddings  $\bar{f}$ , as described in Definition 8, that maps an input set-of-sets to an embedding, a class label (in classification tasks), or real-valued response (in regression tasks). In particular, I shall exploit the characterization provided by Equation (5.1): rather than directly

modeling  $\overline{\overline{f}}$ , I seek to learn the function  $\vec{f}$ , which gives us the freedom to apply any flexible family of models without being subject to the constraint of the function being invariant to inputs that represent the same set-of-sets. Thanks to their expressiveness and flexibility as universal function approximators, I shall model  $\vec{f}$  using deep neural networks. By applying Equation (5.1) as well as the insights discussed in Section 5.3.2, I will be able to learn an inductive SoS embedding  $\overline{\overline{f}}$  for set-of-sets inputs.

Next, I describe the details of our proposed architecture HATS for learning  $\overline{\overline{f}}$ , and then discuss other related neural-network architectures for modeling  $\vec{f}$ .

### 5.3.1 The HATS Architecture

In this section I describe HATS as an architecture to learn flexible functions  $\vec{f}(\mathbf{X}; \boldsymbol{\theta})$  in Equation (5.1), where  $\boldsymbol{\theta}$  is a set of learnable parameters. Later I show how advances in stochastic optimization, combined with Monte Carlo sampling, can be used as a tractable approach to transform a HATS  $\vec{f}$  into a practical inductive SoS embedding. The HATS architecture uses the concept of hierarchical attention for sequence models, first used in document classification [57] and adds two permutation layers, which are key ingredients in the tractable optimization and inference of HATS.

Background.

Recurrent neural networks (RNNs) have been shown to be very well-suited for modeling functions over variable-length sequences. In particular, the use of parameter-sharing allows RNNs to simultaneously achieve flexibility and expressiveness in capturing complex interactions over variable-length sequences with only a fixed-number of parameters. In fact, [81] showed that (with exact computations) RNNs are *universal* functions in that any function computable by a Turing machine can be computed by an RNN of finite size. To alleviate the problem of vanishing or exploding gradients associated with capturing long-range dependencies, gated RNNs such as *long short-term memory networks* (LSTMs) and *gated recurrent units* (GRUs) have been



proposed, and both have achieved great success in practical applications. I shall focus on LSTM-based architectures in this work.

I begin the description of HATS by describing a vanilla LSTM model for modeling SoS functions, and then propose more sophisticated designs that are tailored to specific aspects of inductive sets-of-sets embeddings.

#### LSTM model for sets-of-sets

In our sets-of-sets problem, the input  $\mathbf{X}$  is a collection of sequences arranged in some specific (unknown) order. Then,  $\vec{f}$  of Equation (5.1) can be a single LSTM where I can make  $\mathbf{X}$  into a sequence by concatenating all rows  $1, \dots, n$  into a single row, collapsing all consecutive null symbols, like  $\#\#$ ,  $\#\#\#$ , into a single null symbol  $\#$ . The last long-term memory state (or output state) of the LSTM is then fed into a multi-layer perceptron to obtain the final embedding value.

#### H-LSTM: Hierarchical LSTM model for sets-of-sets

By simply concatenating the constituent sequences within each set, the vanilla LSTM model discussed previously does not take into account the hierarchical nature of the sets-of-sets problem. In this section, I propose to use a two-level hierarchical LSTM (H-LSTM) model to capture the structure of sets-of-sets. Hierarchical LSTM models have been studied in the natural language processing literature (*e.g.* [82]; see Section 2.2.3 for a detailed discussion).

Given an input SoS tensor  $\mathbf{X}$  consisting of  $n$  sets (viewed as sequences)  $\mathbf{X}_{i,*}$ ,  $i = 1, \dots, n$ , with maximum cardinality  $m$ , the first layer of the H-LSTM model applies a bidirectional LSTM to each sequence  $\mathbf{X}_{i,*}$ . Specifically, let

$$\overrightarrow{\mathbf{h}}_{ij} = \overrightarrow{\text{LSTM}}_1(\mathbf{X}_{ij}), \quad \overleftarrow{\mathbf{h}}_{ij} = \overleftarrow{\text{LSTM}}_1(\mathbf{X}_{ij}), \quad j = 1, \dots, m \quad (5.3)$$

denote the forward and backward hidden states for the  $j$ -th element in  $\mathbf{X}_{i,*}$  obtained from the forward and backward LSTMs, respectively. I obtain an *annotation* for  $\mathbf{X}_{ij}$

by concatenating the forward and backward hidden states:  $\mathbf{h}_{ij} = [\overrightarrow{\mathbf{h}_{ij}}, \overleftarrow{\mathbf{h}_{ij}}]$ , which summarizes the information regarding  $X_{ij}$  in the set  $\mathbf{X}_{i,*}$ . The last hidden state of the trained H-LSTM model then provides an *embedding* of the whole set  $\mathbf{X}_{i,*}$ , which I denote as  $\mathbf{h}_i$ .

Existing approaches to modeling permutation-invariant functions use various forms of pooling operations (*e.g.*, max-pooling [3, 4] or Janossy pooling [15]) to aggregate the embeddings obtained for each element in a set. While these simple approaches are guaranteed to be invariant to permutations in the input, they do not allow flexibility to model complex interactions among the elements. Instead, I propose to concatenate the embeddings  $\mathbf{h}_i$  obtained for each set  $\mathbf{X}_{i,*}$ , and then apply another bidirectional LSTM to model the dependencies among the set embeddings:

$$\overrightarrow{\mathbf{y}}_i = \overrightarrow{\text{LSTM}_2}(\mathbf{h}_i), \overleftarrow{\mathbf{y}}_i = \overleftarrow{\text{LSTM}_2}(\mathbf{h}_i), \mathbf{y}_i = [\overrightarrow{\mathbf{y}}_i, \overleftarrow{\mathbf{y}}_i], i = 1, \dots, n. \quad (5.4)$$

The last hidden state of this upper-layer LSTM then provides an overall embedding  $\mathbf{y}$  of the SoS  $\mathbf{X}$  that takes into account its hierarchical structure. Finally, the target output (*e.g.*, class label in classification tasks) can be modeled with a fully connected layer using a softmax function.

## HATS architecture

Capturing long-range dependencies is especially important in modeling functions over sets. Unlike language models, where adjacent words in a sentence typically (but not always) provide more information than words that are farther apart, the elements in a set are typically arranged in random order within a sequence,<sup>1</sup> and elements that appear in the early parts of the sequence contain information that is equally relevant to the final output embedding as those that are near the end.

Thus, when using a sequence model, such as an LSTM, to model a function over sets, it is essential to ensure that the model is able to capture both long-range and

---

<sup>1</sup>With the exception of domain-specific scenarios where a *canonical* ordering could be imposed on the elements in the set; which is typically unavailable in general settings.

short-term dependencies. The same argument also applies to the set-level: since there is typically no canonical ordering for sets within an SoS, the top-level LSTM used in the H-LSTM model of the previous section should also be able to preserve long-range information in its final output embedding  $\mathbf{y}$ .

While LSTMs hypothetically should be able to capture long-range dependencies in sequences, in practice their performance are often less than ideal. Intuitively, requiring the last hidden state of an LSTM to encode information from a long input sequence into a single fixed-length vector also seems too much to ask for. Such inability to capture long-range dependencies has aroused much concern in the natural language processing and machine translation communities, and many clever tricks (such as reversing the order of the input sequence) have been devised to improve their practical performance. However, when modeling functions over sets, these tricks are typically ineffective as the elements are arranged in random order in the input sequence.

Rather than attempting to encode a whole input sequence into a single fixed-bit vector (*i.e.*, the last hidden state), attention mechanisms [83] adaptively compute a weighted combination of all the hidden states during the decoding phase. By learning the weights in the attention mechanism, the decoder could then decide on which parts of the input sequence to focus on. This relieves the burden of having to preserve all information in the input sequence from the encoder, and allows the RNN to capture long-range dependencies.

In our context, different elements of a set may possess varying degrees of importance to the task at hand. For instance, when predicting the unique number of elements in a set (see Section 5.4.1 for more details), elements that occur very frequently may be regarded as less important than rare elements. Similarly, inside an SoS, smaller sets may contain less information than larger sets (or vice versa). To capture long-range dependencies on both element-level and set-level, I propose to adopt a *hierarchical attention* mechanism in a hierarchical bidirectional LSTM.

*Element-level attention.* Given an input SoS  $\mathbf{X}$  comprising the sets  $\mathbf{X}_{i,*}$ ,  $i = 1, \dots, n$ , let  $\mathbf{h}_{ij}$  denote the annotation for the element  $X_{ij}$  in set  $\mathbf{X}_{i,*}$  obtained by

concatenating the forward and backward hidden states of Equation (5.3). I first pass  $\mathbf{h}_{ij}$  through a feedforward layer with weights  $\mathbf{W}_1$  and bias term  $b_1$  to obtain a hidden representation of  $\mathbf{h}_{ij}$ :

$$\mathbf{u}_{ij} = \tanh(\mathbf{W}_1 \mathbf{h}_{ij} + b_1),$$

then compute the (normalized) similarity of between  $\mathbf{u}_{ij}$  and an element-level context vector  $\mathbf{c}_1$  via

$$\alpha_{ij} = \frac{\exp(\mathbf{u}_{ij}^\top \mathbf{c}_1)}{\sum_j \exp(\mathbf{u}_{ij}^\top \mathbf{c}_1)},$$

which I use as importance weights to obtain the final embedding of the set  $\mathbf{X}_{i,*}$ :

$$\mathbf{h}_i = \sum_j \alpha_{ij} \mathbf{h}_{ij}. \quad (5.5)$$

*Member-set-level attention.* I feed the embeddings  $\mathbf{h}_1, \dots, \mathbf{h}_n$  obtained from Equation (5.5) into the upper-level bidirectional LSTM and obtain the set annotations  $\mathbf{y}_1, \dots, \mathbf{y}_n$  via Equation (5.4). Following a similar manner as in element-wise attention, I compute

$$\begin{aligned} \mathbf{v}_i &= \tanh(\mathbf{W}_2 \mathbf{y}_i + b_2), \\ \beta_i &= \frac{\exp(\mathbf{v}_i^\top \mathbf{c}_2)}{\sum_j \exp(\mathbf{v}_i^\top \mathbf{c}_2)}, \\ \mathbf{y} &= \sum_j \beta_i \mathbf{y}_i. \end{aligned}$$

where  $\mathbf{W}_2$  and  $b_2$  are weights and bias of another feedforward layer,  $\mathbf{c}$  is a set-level context vector,  $\beta_i$  are importance weights, and  $\mathbf{y}$  is the final embedding for SoS  $\mathbf{X}$ .

The overall *hierarchical attention* (HATS) model is illustrated in Figure 5.1. The LSTM with element-level attention encodes each set into a permutation-invariant embedding; the LSTM with member-set-level attention then computes a final permutation-invariant embedding for the SoS. The main difference with existing architectures [57] lies in the two permutation layers. The lower permutation layer performs an intra-set permutation for each set, while the upper layer performs an inter-set permutation. These two layers combine to guarantee that the model is SoS permutation-invariant.

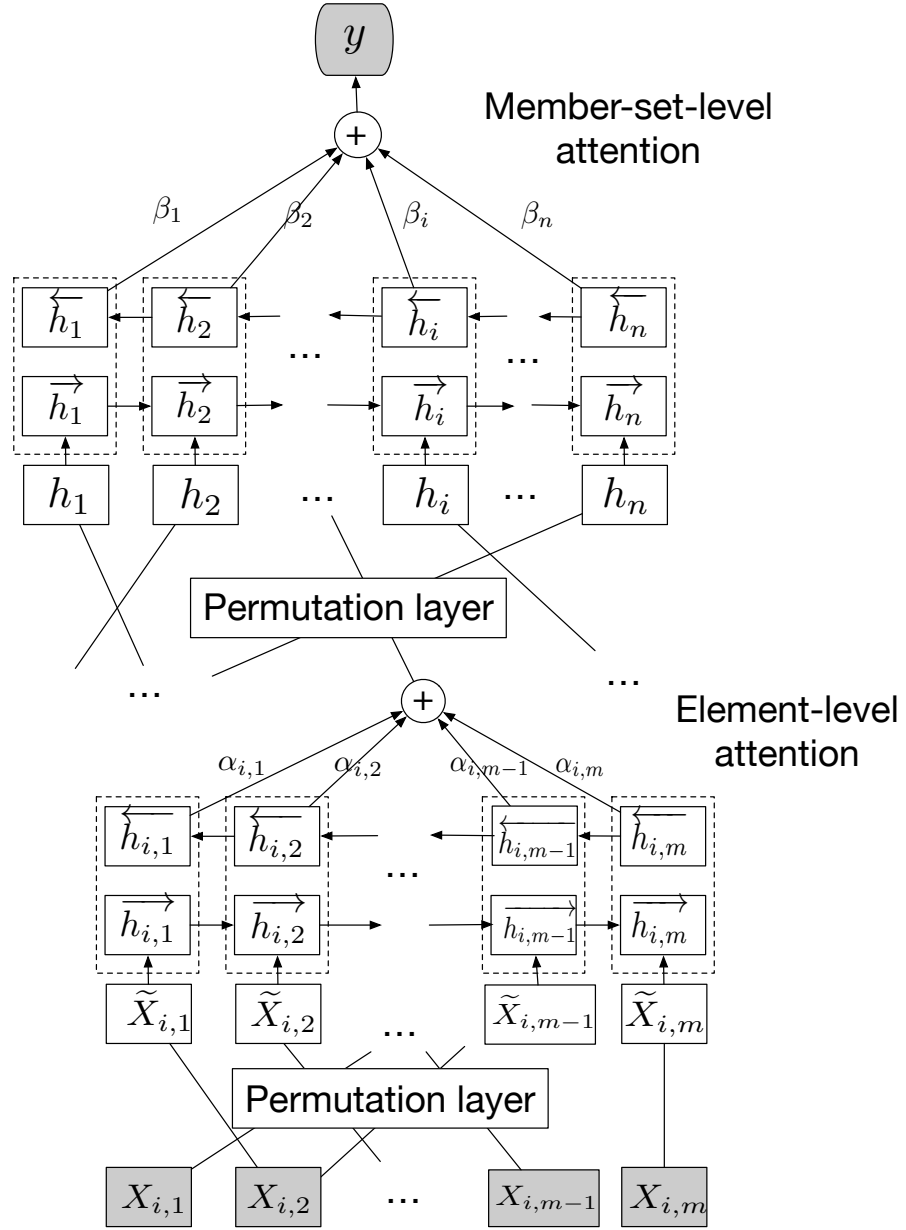


Figure 5.1.: HATS architecture for SoS inputs.

### 5.3.2 Stochastic Optimization for HATS

The obvious caveat in applying Equation (5.1) to learn  $\overline{\overline{f}}$  in practice is that the  $(n + 1)$  summations involved would be computationally prohibitive in all but the simplest scenarios, especially since each summation is over  $n!$  (for  $\phi$ ) or  $m!$  (for  $\pi_i$ ,  $i = 1, \dots, n$ ) possible permutations and thus, is already intractable for moderate  $n$  and  $m$ .

To learn HATS, I adapt the stochastic optimization procedure ( $\pi$ -SGD) of [15] to set-of-sets inputs.

For simplicity, I consider a supervised learning task. It is easy to apply these results to regression tasks. Consider  $N$  labeled SoS training examples  $\{(\mathbf{X}^{(s)}, y_s)\}_{s=1}^N$ , where  $\mathbf{X}^{(s)}$  is a set-of-sets and  $y_s$  is its label (class label in classification tasks or real-valued response for regression tasks). Let  $\hat{y}$  be the predicted label with respect to SoS input  $\mathbf{X}$ . Consider a loss function  $L(y, \hat{y})$  such as squared-loss or cross-entropy loss. In general,  $L$  only needs to be convex in  $\hat{y}$ , but it does *not* need to be convex with respect to the neural network parameters  $\boldsymbol{\theta} = (\mathbf{W}_1, b_1, \mathbf{W}_2, b_2, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ , where  $\boldsymbol{\theta}_1$  and  $\boldsymbol{\theta}_2$  are the parameters of the two bidirectional-LSTMs of HATS.

I wish to learn a function mapping  $\bar{f}(\cdot; \boldsymbol{\theta}) : \mathbf{X} \mapsto y$  with parameters  $\boldsymbol{\theta}$ , that minimizes the empirical risk on the training data:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{s=1}^N L\left(y_s, \bar{f}(\mathbf{X}^{(s)}; \boldsymbol{\theta})\right). \quad (5.6)$$

Naturally,  $\bar{f}$  should satisfy Definition 8, since the input set-of-sets are invariant under SoS permutations. Rather than optimizing over the sum of all permutations in Equation (5.1), I will not minimize Equation (5.6). Rather, I will minimize an upper bound of the loss  $\sum_{s=1}^N L\left(y_s, \bar{f}(\mathbf{X}^{(s)}; \boldsymbol{\theta})\right)$  without explicitly evaluating  $\bar{f}(\mathbf{X}; \boldsymbol{\theta})$ , which is computationally intractable as it involves summing over all possible permutations.

Our optimization procedure is described in Algorithm 2. Rather than summing over all possible permutations, I sample all orderings uniformly at random from the space of all valid permutations. Specifically, I sample  $\tilde{\phi} \sim \text{Uniform}(\Pi_n)$  and  $\tilde{\pi}_1, \dots, \tilde{\pi}_n \sim \text{Uniform}(\Pi_m)$ , and compute

$$\hat{\hat{f}}(\mathbf{X}) = \bar{f}(\mathbf{X}_{\tilde{\phi}, \tilde{\pi}(\tilde{\phi})}). \quad (5.7)$$

It is easy to see that Equation (5.7) provides an unbiased estimator:

$$\mathbb{E}_{\tilde{\phi}, \tilde{\pi}_1, \dots, \tilde{\pi}_n} [\hat{\hat{f}}(\mathbf{X}; \boldsymbol{\theta})] = \bar{f}(\mathbf{X}; \boldsymbol{\theta}).$$

For the  $s$ -th training example, using the sampled permutations, Algorithm 2 can be shown to optimize

$$\mathbb{E}_{\tilde{\phi}, \tilde{\pi}_1, \dots, \tilde{\pi}_n} [L(y_s, \hat{\hat{f}}(\mathbf{X}^{(s)}; \boldsymbol{\theta}))],$$

Since  $L(y_s, \cdot)$  is a convex function, by Jensen’s inequality,

$$\mathbb{E}_{\tilde{\phi}, \tilde{\pi}_1, \dots, \tilde{\pi}_n} [L(y_s, \hat{\hat{f}}(\mathbf{X}^{(s)}; \boldsymbol{\theta}))] \geq L(y_s, \bar{\bar{f}}(\mathbf{X}^{(s)}; \boldsymbol{\theta})).$$

Hence, the optimization in Algorithm 2 is a proper surrogate to optimize the original loss Equation (5.6). In practice, one could also sample multiple permutations and average over them in Equation (5.7) to reduce the variance of the estimator.

The computational cost of optimizing HATS lies in backpropagating the gradients through the neural network architecture (*cf.* Figure 5.1) of the HATS model  $\vec{f}$ . Thus, the overall time complexity of Algorithm 2 is equal to  $\mathcal{O}(mndTB)$ . At inference time, I perform a Monte Carlo estimate of Equation (5.1) by sampling a few permutations and performing a forward pass over the HATS neural network. Our experiments show that in practice five to twenty Monte Carlo samples are sufficient for estimating Equation (5.1).

## 5.4 Experiments

I demonstrate the utility of HATS and its approximate stochastic optimization and Monte Carlo inference by conducting experiments on a variety of SoS tasks spanning a number of applications—ranging from arithmetic tasks and computing similarity measures between sets, to predicting hyperlinks across subgraphs in large networks, to detecting anomalous point-clouds in computer vision.

### Models

In the experiments, I evaluate the performance of our proposed models, and compare with several existing approaches in the literature (*cf.* Section 2.2.3). I summarize all the models below:

---

**Algorithm 2:** Stochastic optimization for learning HATS.

---

**Input:** Labeled SoS training examples  $\{(\mathbf{X}^{(s)}, y_s)\}_{s=1}^N$ ;  
**Input:** HATS model  $\vec{f}(\mathbf{X}; \boldsymbol{\theta})$  with unknown parameters  $\boldsymbol{\theta}$ ;  
**Input:** Loss function  $L(y, \hat{y})$ ;  
**Input:** Number of optimization epochs  $T$ ;  
**Input:** Mini-batch size  $B$ ; learning-rate schedule  $\{\eta_t\}_{t=1}^T$ ;  
**Output:** Learned parameters  $\boldsymbol{\theta}$  for the model  $\vec{f}(\mathbf{X}; \boldsymbol{\theta})$ .

```

1 Initialize parameters  $\boldsymbol{\theta}^{(0)}$  ;
2 for  $t = 1, \dots, T$  do
3    $\mathbf{g}_t \leftarrow \mathbf{0}$  ;
4   for  $s$  in mini-batch-indices do
5      $\tilde{\mathbf{X}}^{(s)} \leftarrow$  Permute the rows of  $\mathbf{X}^{(s)}$  ;
6     for  $i = 1, \dots, |\tilde{\mathbf{X}}^{(s)}|$  do
7       Permute the entries of  $\tilde{\mathbf{X}}_{i*}^{(s)}$  ;
8      $\mathbf{g}_t \leftarrow \mathbf{g}_t + \frac{1}{B} \nabla_{\boldsymbol{\theta}} L(y_s, \vec{f}(\tilde{\mathbf{X}}^{(s)}; \boldsymbol{\theta}))$  ;
9    $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta_t \mathbf{g}_t$  ;
10 return  $\boldsymbol{\theta}^{(T)}$ 

```

---

DEEPSET [4]: A feedward neural network model with sum-pooling to achieve permutation-invariance.

MI-CNN [43]: Convolutional network with gated attention mechanisms for permutation-invariant multi-instance learning.

J-LSTM [15]: The vanilla LSTM model of Section 5.3.1; equivalent to applying the Janossy pooling method [15] using LSTMs.

H-LSTM: The hierarchical LSTM of Section 5.3.1.

HATS: The hierarchical attention network of Section 5.3.1.



Since DEEPSET [4], MI-CNN [43], and J-LSTM [15] were originally designed for set (rather than SoS) inputs, I flatten each SoS by concatenating its member sets into a single sequence as a preprocessing step. The J-LSTM, H-LSTM, and HATS models are all trained using the framework described in Algorithm 2. For DEEPSET [4], MI-CNN [43], and J-LSTM [15] I adopt the authors’ implementations.<sup>2</sup> Details on the models and training procedures are provided in next subsection; I will also release source codes for reproducing all experiment results at [github.com/anonymous](https://github.com/anonymous).

### Training Details

All the models are implemented using Python 3.6 with PyTorch 1.0. The LSTM cells used in J-LSTM [15], H-LSTM, and HATS all have 20-dimensional hidden states. The mini-batch-size in Algorithm 2 is set to 32 for point-cloud classification tasks, and 128 for all other tasks. DEEPSET [4] was originally proposed to handle (plain) set-inputs rather than SoS inputs. I use the **Adder** function in the authors’ code to aggregate the embedding of all the member sets within an SoS into a single embedding.

For all the LSTM-based models, I use the ADAM optimizer [84] with initial learning-rate 0.001. For each method/task, I use the validation set to select the best model as follows: during the training process, I retain the model that achieves the best validation metrics on the validation set and use it for testing. The validation metrics, as well as hyper-parameter values and loss functions are summarized in Table 5.1. Following [4, 15], I treat the simple arithmetic task as a regression task using  $L_1$ -loss. Since the true values of the arithmetic tasks are integers, I round the regression outputs to the nearest integers before computing the prediction accuracy.

Table 5.2 summarizes the statistics of the network datasets used in Section 5.4.3.

---

<sup>2</sup><https://github.com/manzilzaheer/DeepSets>  
<https://github.com/AMLab-Amsterdam/AttentionDeepMIL>  
<https://github.com/PurdueMINDS/JanossyPooling>

Table 5.1.: Implementation details for various tasks.

	Simple arithmetic	Adamic/Adar index	Subgraph hyperlink predict	Point-cloud classify
Task type	Regression	Regression	Classification	Classification
Loss function	$L_1$	$L_1$	Cross-entropy	Cross-entropy
Max. num. epochs	4000	4000	4000	2000
Validation metric	Accuracy	$L_1$ -loss	Accuracy	Accuracy
Num. training examples	10,000	1,000	10,000	2,000
Num. validation examples	10,000	1,000	10,000	1,000
Num. test examples	10,000	1,000	10,000	2,000

Table 5.2.: Summary of network dataset statistics.

Dataset	$ V $	$ E $	#Classes
Cora [85]	2,708	5,429	7
Wiki-vote [86]	7,115	103,689	1
PPI [5]	3,890	76,584	50

#### 5.4.1 Simple Arithmetic Tasks

Similar to [4, 15], I begin by considering simple arithmetic tasks that involve predicting summary statistics for sets-of-sets containing integers. In our experiments, each SoS contains  $n = 4$  member multisets, created by drawing  $m$  integers from  $\{0, 1, \dots, 9\}$  with replacement. Given  $N = 10,000$  SoS training examples, I predict:

$\cap$  *Binary*: Whether the intersection of all member sets is empty.

$\cap$  *Sum*: Sum of all elements in the intersection of all member sets.

$\cup$  *Sum*: Sum of all elements in the union of all member sets.

*Unique count*: Number of unique elements across all member sets.

*Unique sum*: Sum of all unique elements across all member sets.

Table 5.3.: Prediction accuracies for *interactive* arithmetic tasks for different member-set size  $m$ .

Methods	$m = 5$			$m = 10$		
	$\cap$ <i>Binary</i>	$\cap$ <i>Sum</i>	$\cup$ <i>Sum</i>	$\cap$ <i>Binary</i>	$\cap$ <i>Sum</i>	$\cup$ <i>Sum</i>
DeepSet [4]	<b>0.742 (0.005)</b>	<b>0.765 (0.003)</b>	0.078 (0.003)	0.900 (0.003)	0.069 (0.003)	0.873 (0.003)
MI-CNN [43]	0.741 (0.007)	0.739 (0.006)	0.425 (0.111)	<b>0.904 (0.001)</b>	0.071 (0.0026)	0.873 (0.071)
J-LSTM [15]	0.729 (0.005)	<b>0.762 (0.003)</b>	0.061 (0.002)	0.271 (0.001)	0.599 (0.006)	0.123 (0.002)
H-LSTM	0.736 (0.003)	<b>0.763 (0.002)</b>	0.963 (0.061)	0.903 (0.002)	0.967 (0.009)	0.893 (0.012)
HATS	<b>0.740 (0.007)</b>	<b>0.765 (0.002)</b>	<b>0.996 (0.005)</b>	<b>0.904 (0.003)</b>	<b>0.998 (0.001)</b>	<b>0.925 (0.012)</b>

Note that the first three tasks ( $\cap$  *Binary*,  $\cap$  *Sum*, and  $\cup$  *um*) require learning *interactions* across the member sets of an SoS, while the last two tasks (*Unique count* and *Unique sum*) are *aggregate* tasks in that their results would remain unchanged if one simply flattens all the member sets into a single set and computed the unique count/sum of its elements.

For each model, I use a validation set containing another 10,000 examples and evaluate its predictions on a held-out test set with 10,000 examples. For each task, I also experiment with two different member-set sizes  $m$ . For each method, I repeat for five random trials and report the mean and standard deviations of their test-set prediction accuracies. The results are shown in Tables 5.3 and 5.4; for each task, I indicate the highest accuracy values (within two standard errors) in boldface.

I observe that the sequence models (J-LSTM, H-LSTM, HATS) significantly outperforms MI-CNN and DEEPSET on most tasks. This shows that RNNs (learned with the approximate stochastic optimization) are more suitable for modeling variable length inputs than CNNs or sum-pooling (DEEPSET). I also note that H-LSTM substantially outperforms J-LSTM, which shows that modeling the hierarchical structure of set-of-sets can better capture and decouple the inter-set and intra-set dependencies. Furthermore, I observe that HATS performs on par with or superior to H-LSTM, demonstrating the effectiveness of the element-level and set-level attention mechanisms in capturing higher-range dependencies within and across member sets.

SoS improvement over aggregative tasks.

*Aggregative* tasks are designed as set tasks disguised as SoS tasks, as concatenates all the member sets into a single set input works for these tasks. Thus, DEEPSET, MI-CNN, and J-LSTM should perform well on these tasks since the hierarchical structure of SoS’s do not play a role in how the true label was generated. However, from Table 5.4 I still observe that by modeling what is essentially a set function as an SoS function, H-LSTM and HATS are still able to produce significant gains over the other approaches. I believe that this is due to the fact that sequence models (even LSTMs) still have trouble capturing long-range dependencies—by segmenting a single long sequence (*i.e.*, the flattened SoS) into a collection of short sequences (*i.e.*, modeling a set as an SoS), one could improve the models’ capability of capturing dependencies across elements.

#### 5.4.2 Computing the Adamic/Adar Index

In Section 5.2.1, I showed that the Adamic/Adar (A/A) index [80] between two nodes in a network could be cast as an SoS function. I perform experiments on the

Table 5.4.: Accuracies for *aggregative* tasks with different member-set size  $m$ .

Methods	$m = 20$		$m = 40$	
	<i>Unique count</i>	<i>Unique sum</i>	<i>Unique count</i>	<i>Unique sum</i>
DeepSet [4]	0.432 (0.009)	0.080 (0.002)	0.858 (0.002)	0.872(0.002)
MI-CNN [43]	0.071 (0.003)	0.873 (0.071)	0.860 (0.003)	0.876 (0.002)
J-LSTM [15]	0.431 (0.007)	0.081 (0.001)	0.858 (0.001)	0.872 (0.004)
H-LSTM	0.988 (0.007)	0.991 (0.002)	0.892 (0.007)	0.948 (0.069)
HATS	<b>0.996 (0.006)</b>	<b>0.996 (0.007)</b>	<b>0.938 (0.03)</b>	<b>0.998 (0.002)</b>

Table 5.5.: Predicting Adamic/Adar-index on Cora.

Models	MAE	MSE
LSTM	0.0026 (0.0002)	0.0022 (0.0001)
H-LSTM	0.0023 (0.0002)	0.0013 (0.0008)
HATS	<b>0.0021 (0.0001)</b>	<b>0.0008 (0.0001)</b>

Cora [85] dataset,<sup>3</sup> in which I evaluate J-LSTM, H-LSTM, and HATS approaches for predicting the Adamic/Adar index between pairs of nodes using their neighbor sets. More specifically, as described in Section 5.2.1, each input SoS contains two neighbor sets; an element in each set is a tuple containing the unique identifier of neighboring node and its degree. Since computing the A/A index requires both node identifier and degree information, it is not straightforward to transform the SoS inputs into a single set. Thus, I only evaluate the A/A task over methods that can operate with native SoS inputs. An added difficulty is the variable number of neighbors of nodes.

I use  $N = 1,000$  training SoS examples (*i.e.*, node-pairs) and another 1,000 examples for the validation set. Evaluation computes the predicted A/A values for 1,000 held-out test examples and measure the mean-absolute error (MAE) and mean-squared error (MSE) between the model predictions and the true A/A index. The results are shown in Table 5.5. I observe that HATS attains the lowest errors, followed by H-LSTM and with J-LSTMlast. This may be partly due to the fact that A/A implicitly requires computing the intersection of two sets, which as our previous task on Table 5.3 has shown, both HATS and H-LSTM perform well.

#### 5.4.3 Subgraph Hyperlink Prediction

Modeling higher-order structures within a network have recently attracted attention in relational learning and graph mining (*e.g.*, [87, 88]). Here, our task is to learn

<sup>3</sup>Table 5.2 summarizes the dataset statistics.

SoS embeddings that can help predict the existence of *hyperlinks* between subgraphs of a large network. Specifically, a hyperlink exists between two subgraphs if there is at least one link connecting two nodes from different  $m$ -node induced subgraphs in a larger graph.

I perform experiments on three widely used network datasets: the citation network Cora [85], the Wikipedia voting network Wiki-vote in [86] and the protein-protein interaction network PPI in [5]. Table 5.2 in Appendix 5.2 provides a summary of the network statistics. For each network, I first obtain  $d = 256$ -dimensional feature representations for each node using unsupervised GRAPH-SAGE [5]. Given an SoS example consisting of the features of every node in each subgraph, the task is to predict a binary label indicating whether a hyperlink exists between the two subgraphs in that SoS. For each dataset, I also vary the subgraph size  $m$ .

Table 5.6 shows the hyperlink prediction accuracies for each network dataset. I observe that HATS outperforms the other approaches in almost all tasks, with H-LSTM only besting HATS once.

Table 5.6.: Subgraph hyperlink prediction accuracies for different subgraph size  $m$ .

Methods	$m = 4$			$m = 10$		
	Wiki-vote	Cora	PPI	Wiki-vote	Cora	PPI
DEEPSET [4]	0.657 (0.028)	0.676 (0.018)	0.731 (0.031)	0.638 (0.017)	0.661 (0.021)	0.431 (0.027)
MI-CNN [43]	0.832 (0.010)	0.937 (0.002)	0.894 (0.002)	0.675 (0.006)	0.938 (0.001)	0.853 (0.011)
J-LSTM [15]	0.715 (0.045)	0.834 (0.003)	0.782 (0.013)	0.493 (0.034)	0.936 (0.002)	0.5118 (0.031)
H-LSTM	0.932 (0.009)	0.940 (0.004)	0.889 (0.008)	<b>0.783 (0.006)</b>	0.948 (0.004)	<b>0.873 (0.013)</b>
HATS	<b>0.948 (0.008)</b>	<b>0.960 (0.028)</b>	<b>0.901 (0.009)</b>	0.773 (0.006)	<b>0.961 (0.003)</b>	<b>0.872 (0.012)</b>

#### 5.4.4 Point-Cloud SoS Classification

*Point-clouds* (*i.e.*, sets of low-dimensional vectors in Euclidean space) arise naturally in many computer vision applications using LIDAR data such as self-driving cars [3]. I perform experiments on the ModelNet40 [75] point-cloud database which

contains more than 12,311 point-clouds, each labeled as one of 40 classes (such as *desk*, *chair*, or *plane*; see Figure 5.3 for some example visualizations) .

In our experiments, each SoS example, representing a point-cloud scene, contains  $m = 10$  point-clouds, and each point-cloud comprises of 2,000 points. I construct SoS examples in two different ways to perform two prediction tasks:

*Anomaly detection*: Among the 10 point-clouds in each SoS, at least 9 of them have the same class label, but there is a 50% chance that the remaining one has a different label (*i.e.*, is an anomaly). Given such an SoS, the task is to predict whether this SoS contains an anomalous point-cloud.

*Unique-label counting*: Each SoS consists of 10 randomly selected point-clouds from the database. Given such an SoS, the task is to predict the number of unique object types (labels) in the SoS.

For each task, I use  $N = 2,000$  SoS examples for training, 1,000 for validation, and 2,000 ones held-out for testing. Table 5.7 shows the prediction accuracies for each method on both tasks. Once again, I observe that HATS performs best among all approaches.

To further gauge the relative performance of the sequence models, Figure 5.3 varies the size  $m$  of each point-cloud (*i.e.*, the number of points it contains), and plot their accuracies (along with standard errors) for the anomaly detection task in Figure 5.3. I observe that HATS consistently outperforms H-LSTM and J-LSTM, thanks to its attention mechanism for capturing long-range dependencies even as  $m$  gets to 1,500 elements in the member sets.

For SoS's whose member sets are rather large (for instance, each point-cloud instance contains  $m = 2,000$  points), one could further speed up the training procedure of our proposed models by retaining only the first  $k$  number of columns of the permuted SoS  $\tilde{\mathbf{A}}$  from  $m$  to a smaller number  $k$  before performing the stochastic gradient update on line 8 in Algorithm 2. This approach can be viewed as an example of imposing *k-ary dependency restrictions* [15] as a means of promoting computational

efficiency. For the point-cloud anomaly detection task, Figure 5.3 investigates how such  $k$ -ary restrictions affect prediction performance. I observe that even with small values of  $k$ , the loss in prediction accuracy remains tolerable, even as  $k$  decreases from the original  $m = 2,000$  points (Table 5.7) to 50 points.

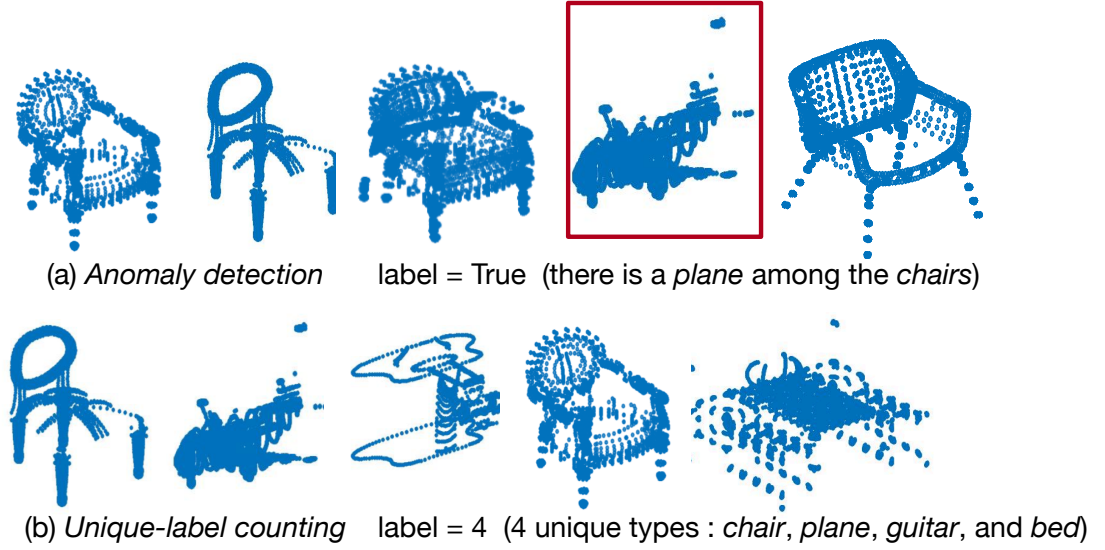


Figure 5.2.: Visualization of point-cloud tasks.

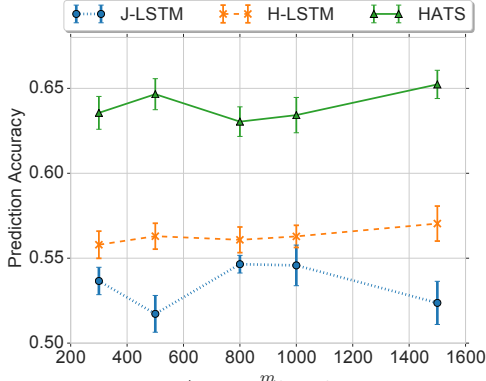


Figure 5.3.: Anomaly detection accuracies for varying point-cloud size  $m$ .

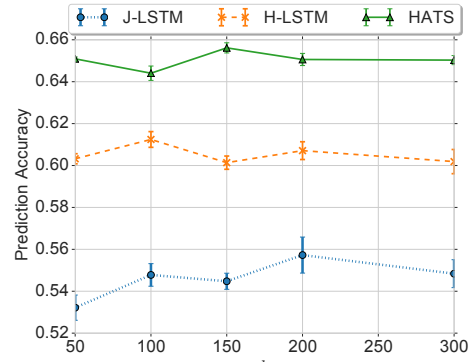


Figure 5.4.: Anomaly detection accuracies for varying  $k$ -ary dependency.



Table 5.7.: Point-cloud classification results.

Methods	<i>Anomaly detection</i>	<i>Unique-label counting</i>
DEEPSET [4]	0.553 (0.021)	0.231(0.021)
MI-CNN [43]	0.513 (0.003)	0.356 (0.011)
J-LSTM [15]	0.525 (0.018)	0.369 (0.009)
H-LSTM	0.596 (0.004)	0.357 (0.008)
HATS	<b>0.663 (0.016)</b>	<b>0.388 (0.011)</b>

## 6 SUMMARY AND FUTURE DIRECTIONS

### 6.1 Conclusion

In this dissertation, I developed scalable invariant methods to learn representations from graphs, sets, and sets of sets. On the one hand, I proposed methods which are invariant to graph isomorphism or set permutations. On the other hand, our proposed method can model high-order relationships which can increase the prediction quality. In this chapter, I summarize the contributions of this dissertation, and outline several avenues for future research.

The contributions of this dissertation fall into the following aspects:

#### Theoretical

- In Chapter 3, I proposed induced subgraph pattern as features to model high-order dependence between nodes on graph.
- In Chapter 4, I showed the characteristics of an architecture’s computational graph impact its ability to learn in contexts with complex set dependencies, and demonstrate limitations of current methods with respect to one or more of these complexity dimensions.
- In Chapter 5, I proposed the properties of inter-set and intra-set dependencies of Set-of-Sets problem can not be modeled by set models.

#### Modeling

- In Chapter 3, I developed a Subgraph Neural Network model *SPNN* which is a first step in the development of more interpretable models, features, and

classifiers that can encode the complex correlations between graph structure and labels.

- In Chapter 4, I developed a neural network architecture *Self-Attention GRU* designed to better capture both long-range and high-order dependencies.
- In Chapter 5, I proposed a framework for learning permutation-invariant inductive SoS embeddings with neural networks, and introduced HATS, a hierarchical, bi-directional LSTM with attention, that is designed to better capture intra-set and inter-set interactions in sets-of-sets while maintaining SoS permutation-invariance.

## Empirical

- In Chapter 3, I evaluated the problem of predicting induced subgraph evolution in heterogeneous graphs and generalizes a variety of existing tasks. Our results show SPNN to consistently achieve better performance than competing approaches.
- In Chapter 4, I demonstrated *Self-Attention GRU* achieved improved performance over a wide range of applications and against state-of-the-art baselines.
- In Chapter 5, I demonstrated our proposed model HATS achieved superior performance over a wide range of application tasks involving SoS inputs.

Besides the above mentioned theoretical, modeling and empirical contributions, I also learned the following four properties are essential to representation learning with invariances: (1) Model should be invariant to graph isomorphism or set permutations. Subgraph Pattern Pooling and Janossy Pooling [15] can handle the invariances in representation learning. (2) The learned model should be inductive. In order to learn more inductive models, subgraph tasks can adopt explicit subgraph counting, while set and set-of-sets tasks can use a better architecture and use attention mechanism. (3) The model should capture high-order relationships. Subgraph tasks can

use subgraphs to model multi-node interactions. Set tasks can use high-order process functions. Set-of-sets tasks can adopt the hierarchical structure to model both intra-set and inter-set dependence. (4) Models should scale polynomial to the data size even though possible isomorphic graphs and permutated sets are in factorial-scale. Subgraph tasks can adopt subgraph random walk [89] to sample subgraphs. Set and set-of-sets tasks can adopt  $\pi$ -SGD [15] to optimize and inference efficiently.

## 6.2 Future Directions

### 6.2.1 Subgraph Collective Inference

Collective inference can be used to further improve the accuracy of subgraph prediction. The common nodes between neighbouring subgraphs can help to collective inference the labels of subgraphs. Existing works [7] mainly focus on single node collective inference. No attention has been paid on high-order subgraph collective inference. In this case, I can apply collective inference to enhance current subgraph classification models. Since neighboring subgraph may share common nodes, it is reasonable that the labels of neighboring subgraphs are correlated. I can develop a collective classification method to create a joint classifier (e.g., [38, 39]) for SPNN , as our approach can be used as a local conditional model for joint prediction.

### 6.2.2 Increase Subgraph Counting Efficiency

The proposed Subgraph Pattern Neural Network needs to count the induced neighboring subgraphs. Enumerating all the neighbors is time consuming especially for hub nodes. Facing this issue, the proposed model constrains the search space by considering 1-hop subgraph neighbors with maximum of 4 nodes. Even though there are some follow-up works [90] to increase the efficiency of larger subgraph counting, this problem still worth further attention since it will benefits a larger range of models besides my proposed model.

### 6.2.3 Random Graph Model for Set

Since adding randomness to the order of input elements by Janossy Pooling [15] can solve the invariance issues in set models, adding randomness to the model’s architecture may also achieve the same goal. Using a random graph as the model’s computational graph and re-generate graph randomly at each epoch may help us learn permutation-invariant functions. Since adding randomness to models can improve the prediction quality in image recognition [91], exploring less constrained search spaces in set problems may achieve competitive prediction results. Among these random graph models, the random regular graph can provide balanced memory usage for each local process functions.

### 6.2.4 Apply the proposed Set model on the Set-of-Sets tasks

The SoS work is finished earlier than the Set architecture work. Studying the Set-of-sets problem in Chapter 5 provides the intuitions for our proposed model in Chapter 5. Due to time constraint, I have not applied the model in Chapter 4 in set-of-sets problems in Chapter 5 yet. Since the proposed model can better model high-order dependence and long-range dependence, applying it to set-of-sets tasks should have better performance.

### 6.2.5 Temporal Graph Classification with Set of Sets

Recent works on node classification using temporal interaction information showed randomized time steps achieved most of the accuracy gains [92]. This finding implies temporal graph information may also be permutation-invariant. Since the neighboring graph nodes can be represented as a set [7], a set of graphs in different timestamps can be modeled as a set of sets. In this case, applying our SoS models HATSmay achieve promising results.

## BIBLIOGRAPHY

- [1] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *J. Assoc. Inf. Sci. Technol.*, 58(7), 2007.
- [2] Daniel Mauricio Romero and Jon Kleinberg. The directed closure process in hybrid social-information networks, with an analysis of link formation on twitter. In *Fourth International AAAI Conference on Weblogs and Social Media*, 2010.
- [3] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 Conference on Computer Vision and Pattern Recognition*, 2017.
- [4] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. Deep sets. In *NeurIPS*, 2017.
- [5] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.
- [6] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICML*, 2019.
- [7] John Moore and Jennifer Neville. Deep collective inference. In *aaai conference on artificial intelligence*, pages 2364–2372, 2017.
- [8] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. In *NeurIPS*, 2017.
- [9] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *2016 ICML*, 2016.

- [10] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. 2018.
- [11] Gil Lederman, Markus N Rabe, Edward A Lee, and Sanjit A Seshia. Learning heuristics for automated reasoning through reinforcement learning. 2018.
- [12] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. 2018.
- [13] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.
- [14] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.
- [15] Ryan L. Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *ICML*, 2019.
- [16] Seungil You, David Ding, Kevin Canini, Jan Pfeifer, and Maya Gupta. Deep lattice networks and partial monotonic functions. In *NeurIPS*, 2017.
- [17] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer. *arXiv:1810.00825*, 2018.
- [18] James Victor Uspensky. Introduction to mathematical probability. 1937.
- [19] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *ICML*, pages 2702–2711, 2016.

- [20] James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *NIPS*, 2016.
- [21] Francesco Orsini, Paolo Frasconi, and Luc De Raedt. Graph Invariant Kernels. In *IJCAI*, 2015.
- [22] Pinar Yanardag and S.V.N. Vishwanathan. A Structural Smoothing Framework For Robust Graph Comparison. In *NIPS*, 2015.
- [23] Pinar Yanardag and S.V.N. Vishwanathan. Deep Graph Kernels. In *KDD*, 2015.
- [24] M. Niepert, N. Mohamed, and N. Konstantin. Learning Convolutional Neural Networks for Graphs. In *ICML*, 2016.
- [25] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, 2016.
- [26] Ni Lao and William W. Cohen. Relational retrieval using a combination of path-constrained random walks. *Mach. Learn.*, 81(1), 2010.
- [27] Y. Sun, R. Barber, M. Gupt, C. Aggarwal, and J. Han. Co-author relationship prediction in heterogeneous bibliographic networks. In *ASONAM*, 2011.
- [28] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *KDD. ACM*, 2014.
- [29] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *WWW. ACM*, 2015.
- [30] Larry Heck and Hongzhao Huang. Deep learning of knowledge graph embeddings for semantic parsing of twitter dialogs. In *GlobalSIP. IEEE*, December 2014.
- [31] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral Networks and Locally Connected Networks on Graphs. In *ICLR*, dec 2013.



- [32] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep Convolutional Networks on Graph-Structured Data. *arXiv:1506.05163v1*, jun 2015.
- [33] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-Way Model for Collective Learning on Multi-Relational Data. *ICML*, 2011.
- [34] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *IEEE*, mar 2015.
- [35] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *AAAI*. AAAI Press, 2016.
- [36] Xin Dong, Evgeniy Gabrilovich, Jeremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. *KDD*, 2014.
- [37] Ryan N. Lichtenwalter, Jake T. Lussier, and Nitesh V. Chawla. New perspectives and methods in link prediction. In *KDD*, 2010.
- [38] Matthew Richardson and Pedro Domingos. Markov logic networks. *Mach. Learn.*, 2006.
- [39] J. Neville and D. Jensen. Relational dependency networks. *JMLR*, 2007.
- [40] Cristina Manfredotti. Modeling and inference with relational dynamic bayesian networks. In *Advances in artificial intelligence*. Springer, 2009.
- [41] Lise Getoor and Lilyana Mihalkova. Learning statistical models from relational data. In *SIGMOD*, page 1195. ACM Press, 2011.
- [42] Andrew Cotter, Maya Gupta, Heinrich Jiang, James Muller, Taman Narayan, Serena Wang, and Tao Zhu. Interpretable set functions. 2018.
- [43] Maximilian Ilse, Jakub M Tomczak, and Max Welling. Attention-based deep multiple instance learning. In *2016 ICML*, 2018.

- [44] Siamak Ravanbakhsh, Jeff Schneider, and Barnabas Poczos. Deep learning with sets and point clouds. *arxiv:1611.04500*, 2016.
- [45] Benjamin Bloem-Reddy and Yee Whye Teh. Probabilistic symmetry and invariant neural networks. 2019.
- [46] Yan Zhang, Jonathon Hare, and Adam Prugel-Bennett. Deep set prediction networks. In *Advances in Neural Information Processing Systems*, pages 3207–3217, 2019.
- [47] Changping Meng, Jiasen Yang, Bruno Ribeiro, and Jennifer Neville. Hats: A hierarchical sequence-attention framework for inductive set-of-sets embeddings. In *SIGKDD*, 2019.
- [48] Edward Wagstaff, Fabian B Fuchs, Martin Engelcke, Ingmar Posner, and Michael Osborne. On the limitations of representing functions on sets. *arXiv preprint arXiv:1901.09006*, 2019.
- [49] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In *ICML*, 2017.
- [50] Michael Tsang, Dehua Cheng, and Yan Liu. Detecting statistical interactions from neural network weights. *arXiv preprint arXiv:1705.04977*, 2017.
- [51] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. volume 34, pages 18–42, 2017.
- [52] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *NeurIPS*, 2017.
- [53] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*, 2018.

- [54] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [55] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.
- [56] Jason Hartford, Devon R Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. Deep models of interactions across sets. In *2016 ICML*, 2018.
- [57] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. Hierarchical attention networks for document classification. In *NAACL-HLT*, 2016.
- [58] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning Entity and Relation Embeddings for Knowledge Graph Completion. *AAAI*, 2015.
- [59] Austin R Benson, David F Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295), 2016.
- [60] Jian Xu, Thanuka L. Wickramaratne, and Nitesh V. Chawla. Representing higher-order dependencies in networks. *Science Advances*, 2(5), 2016.
- [61] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in Temporal Networks. In *WDSM*, dec 2017.
- [62] Mahmudur Rahman and Mohammad Al Hasan. Link prediction in dynamic networks using graphlet. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 394–409. Springer, 2016.
- [63] László Lovász and Balázs Szegedy. Limits of dense graph sequences. *J. Comb. Theory, Ser. B*, 2006.

- [64] C Borgs, J T Chayes, L Lovász, V T Sós, and K Vesztergombi. Convergent Sequences of Dense Graphs I: Subgraph Frequencies, Metric Properties and Testing. *Advances in Mathematics*, 2008.
- [65] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7), May 2007.
- [66] Pinghui Wang, John C. S. Lui, Bruno Ribeiro, Don Towsley, Junzhou Zhao, and Xiaohong Guan. Efficiently Estimating Motif Statistics of Large Networks. *ACM TKDD*, 9(2), sep 2014.
- [67] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*. IEEE Comput. Soc, 2003.
- [68] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. PathSim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, 4(11), 2011.
- [69] Lada Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3), July 2003.
- [70] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.
- [71] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased lstm: Accelerating recurrent network training for long or event-based sequences. In *Advances in neural information processing systems*, pages 3882–3890, 2016.
- [72] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

- [73] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*. 2017.
- [74] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *ICLR*, 2020.
- [75] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3D Shapenets: A deep representation for volumetric shapes. In *Conference on Computer Vision and Pattern Recognition*, 2015.
- [76] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.
- [77] Wayne D. Blizard. Multiset theory. volume 30, 1988.
- [78] D. J. Daley and D. Vere-Jones. *An Introduction to the Theory of Point Processes (Vol. II)*. Springer, second edition, 2008.
- [79] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. volume 2, pages 359–366, 1989.
- [80] Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. 2001.
- [81] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Appl. Math. Lett.*, 4:77–80, 1991.
- [82] Duyu Tang, Bing Qin, and Ting Liu. Document modeling with gated recurrent neural network for sentiment classification. In *2015 Conference on Empirical Methods on Natural Language Processing*, 2015.
- [83] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. 2014.
- [84] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

- [85] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. 2008.
- [86] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *CHI*, 2010.
- [87] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. Simplicial closure and higher-order link prediction. 2018.
- [88] Dong Li, Zhiming Xu, Sheng Li, and Xin Sun. Link prediction in social networks based on hypergraph. In *WWW*, 2013.
- [89] Pinghui Wang, John Lui, Bruno Ribeiro, Don Towsley, Junzhou Zhao, and Xiaohong Guan. Efficiently estimating motif statistics of large networks. *ACM TKDD*, 9(2), 2014.
- [90] Carlos HC Teixeira, Leornado Cotta, Bruno Ribeiro, and Wagner Meira. Graph pattern mining and learning through user-defined relations. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 1266–1271. IEEE, 2018.
- [91] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1284–1293, 2019.
- [92] Hogun Park and Jennifer Neville. Exploiting interaction links for node classification with deep graph neural networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 3223–3230. AAAI Press, 2019.