LEARNING NEURAL ALGORITHMS WITH GRAPH STRUCTURES

A Dissertation
Presented to
The Academic Faculty

By

Hanjun Dai

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

May 2020

Copyright © Hanjun Dai 2020

LEARNING NEURAL ALGORITHMS WITH GRAPH STRUCTURES

Approved by:

Dr. Le Song, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Duen Horng (Polo) Chau
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Pushmeet Kohli
DeepMind
Alphabet Inc.

Dr. John Schulman
OpenAI
OpenAI Inc.

Date Approved: December 03, 2019

To my beloved Mom, Dad, Yiting, and Luna.

ACKNOWLEDGEMENTS

I am so fortunate to have Le Song as my Ph.D. thesis advisor. I still remember five years ago when I first began my graduate study, I had no idea what kind of topics I should work on. I worked hard during my first year, but I wasn't getting anywhere. While I was stuck in the pit of self-doubt, my advisor pointed the way forward. Not only was it novel and challenging, but it also fit my personal interests and strengths. What's more, my advisor inspires me greatly with his sharp mind and rich experience. His high standards on the algorithmic contribution and state-of-the-art outcome will always guide me on how to be a good researcher.

Throughout my internships during my graduate studies, I am extremely grateful to have many great mentors and colleagues. I will never forget my first internship here in Atlanta advised by Huiming Qu, who is both kind and thoughtful. She had been a great help and support to me when I was going through a tough time back then. In addition, I really cherish the unique opportunity to work with Zornitsa Kozareva and Alexander Smola. They showed me how the industrial-scale research looks like. Moreover, I highly enjoy the close collaboration with John Schulman, whose talent in algorithm and engineering will always be my pursuit. Last but not least, Pushmeet Kohli demonstrates to me what a thriving team should look like with his invaluable insights and strong interpersonal skills.

I would like to express my deepest appreciation to Richard Vuduc and Duen Horng (Polo) Chau for being my thesis committee members. The completion of my thesis would not have been possible without their unparalleled support and ingenious suggestions.

Particularly helpful to me during this time were my labmates as well as friends, namely Bo Dai, Nan Du, Weiyang Liu, Zhen Liu, Yichen Wang, Bo Xie, Yuyu Zhang. It is one of my most precious memories to burn the midnight oil with them from one paper submission deadline to another. Besides every inspiring discussion we had, I also had a great pleasure to have casual chats or to hang out with them. Thanks for adding so much color to my

Ph.D. studies.

I also wish to thank my direct collaborators. They are Ahmet Cecen, Binghong Chen, Xinshi Chen, Connor Coley, Xin Gao, Po-Sen Huang, Elias Khalil, Thomas Kipf, Chengtao Li, Shuang Li, Yujia Li, Mayur Naik, Xujie Si, Rishabh Singh, Harsh Shrivastava, Yingtao Tian, Rakshit Trivedi, Ramzan Umarov, Chenglong Wang, Yuan Yang, Yan-Ming Zhang. None of the thesis projects can be made without them.

Thanks should also go to other friends I met at Georgia Tech, namely Joyce Gao, Lina Hu, Xiaojia Jia, Yaoyao Jia, Mengfan Jiang, Jiasen Lu, Zhaoyang Lv, Jingwei Qi, Hang Su, Fei Wang, Ke Wang, Hang Wu, Zhaoming Wu, Jianwei Yang, Li Yi, and Yiyang Zhu. It was great meeting them at Georgia Tech. I will always remember the happiness we shared and the help they provided me with. I would also like to specially thank other senior Gatech students Zhengyi Hu, Zhongtian Jiang, Yongchao Liu, Yijie Wang, Zheng Yong for providing various kinds of help to help me settle down during first year.

Also a special thank should give to my host family Allen McDonald and Ann Carter, who offered great help and warm throughout the past 5 years. They make Atlanta feel like a second hometown to me.

Finally, I cannot leave Georgia Tech without mentioning my parents and my wife. They've provided relentless support in every way I could imagine with encouragement and patience. They help me learn what's the most important things in life, and they never wavered in their support for me. Special thanks to my family members.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Graph structures, like syntax trees, social networks, and programs, are ubiquitous in many real world applications including knowledge graph inference, chemistry and social network analysis. Over the past several decades, many expert-designed algorithms on graphs have been proposed with nice theoretical properties. However most of them are not data-driven, and will not benefit from the growing scale of available data. Recent advances in deep learning have shown strong empirical performances for images, texts and signals, typically with little domain knowledge. However the combinatorial and discrete nature of the graph data makes it non-trivial to apply neural networks in this domain. Based on the pros and cons of these two, this thesis will discuss several aspects on how to build a tight connection between neural networks and the classical algorithms for graphs. Specifically:

- **Algorithm inspired deep graph learning** The existing algorithms provide an inspiration of deep architecture design, for both the discriminative learning and generative modeling of graphs. Regarding the discriminative representation learning, we show how the graphical model inference algorithms can inspire the design of graph neural networks for chemistry and bioinformatics applications, and how to scale it up with the idea borrowed from steady states algorithms like PageRank; for generative modeling, we build an HSMM inspired neural segmental generative modeling for signal sequences; and for a class of graphs, we leverage the idea of attribute grammar for syntax trees to help regulate the deep networks.

- **Deep learning enhanced graph algorithms** the algorithm framework has procedures that can be enhanced by learnable deep network components. We demonstrate by learning the heuristic function in greedy algorithms with reinforcement learning for combinatorial optimization problems over graphs, such as vertex cover and max cut, and optimal touring problem for real world applications like fuzzing.

- **Towards Inductive reasoning with graph structures** As the algorithm structure generally provides a good inductive bias for the problem, we take an initial step towards inductive reasoning for such structure, where we make attempts to reason about the loop invariant for program verification and the reaction templates for retrosynthesis structured prediction.

Keywords: Deep learning, Graph representation learning, Structured generative modeling, Structured prediction, Reinforcement learning, Chemistry/Bioinformatics applications, Program understanding.

CHAPTER 1
INTRODUCTION

Graphs are ubiquitous in many real world applications. In chemical engineering, the molecules can be represented by graphs with atoms as nodes and bonds as edges; in knowledge graph, entities and corresponding relationships form a graph; in social network, users and their interactions can also be characterized as graphs. Despite the convenience of such representation, the discrete and combinatorial nature of the graphs also brings many difficulties into various machine learning problems.

Difficulties in learning with graphs Here we briefly mention some difficulties people will generally meet when dealing with graphs.

- **Representation Learning:** learning to represent the graphs into fixed dimensional vector is nontrivial. Unlike images where we can represent them as fixed dimensional tensors, different graphs have different sizes and structures. Also the real-world graphs consist of millions to billions of nodes, which also challenges the scalability and efficiency of the learning algorithm.

- **Generative modeling:** Generating the graphs is also tricky. While continuous data like images can be noise tolerant, discrete graphs typically have to be accurate. For example, typically a Carbon node in a molecule can have degree of at most 4. Generating such structures with minor perturbations that violate the constraints will completely fail the graphs obtained. Also, the combinatorial and discrete nature of graphs makes it hard to apply gradient based adjustments during the generation procedure.

- **Combinatorial Optimization** Real-world problems like ads targeting, package delivering can be formulated as combinatorial optimization on graphs, such as Minimum Vertex

Cover, Traveling Salesman Problem, etc. Unfortunately they are all NP-complete, which means there's no known polynomial time algorithm.

Fortunately, over the past decades, researchers have designed algorithms with nice theoretical properties for many of these problems. To list a few:

Exemplar human designed algorithms for graphs

- **Weisfeiler-Lehman (WL) graph isomorphism test:** the WL algorithm [238] was designed to check whether the two graphs are isomorphic. The algorithm works in an iterative fashion, where in each iteration, each node updates its label with the concatenation of neighboring labels. In the end the set of labels are used to distinguish between the graphs. Such idea have been applied into a family of graph kernels called WL graph kernel [207], which hashes the output of WL algorithm to get the explicit feature maps.
- **Greedy Algorithm for Minimum Vertex Cover (MVC):** The MVC problem asks to pick a minimum set of nodes in a graph, such that all the edges have at least one end in this set. To construct a solution for MVC, the greedy algorithm works by sequentially picking the two ends of uncovered edges with maximum degree, and remove the two nodes and associated edges. Such algorithm guarantees the maximum approximation ratio of 2 (i.e., at most pick twice as many as number of nodes in optimal solution).

Such algorithms are very efficient and also effective in many cases. However they have their own limitations: WL kernel requires very high dimensional feature maps, and may not be informative enough as the features are unsupervised; greedy algorithms for MVC may perform worse than branch and bound even in the time limited setting. Also both of these methods are not data driven, which cannot fully exploit the benefit from the big data.

Deep learning has shown superior performances in many domains, including computer vision, natural language, and speech. Though the theoretical aspect of its representation

power is not fully understood, typically the deep learning methods are considered to be flexible and having rich representation power.

Based on above discussion, it is natural to ask the question: can we tightly integrate such human designed algorithms and deep learning to tackle the difficult problems in graph learning? In this thesis work, we are trying to answer this question.

1.1 Connecting deep learning with discrete algorithms

In this document, we focus on three aspects of such connection:

1.1.1 Part I: Algorithm inspired Deep Learning

According to the universal approximation theorem [52], the neural network with one hidden layer can fit continuous functions, but it didn't state the learnability problem. Thus it is often important to design neural network with right inductive bias [17]. The human designed discrete algorithms encode the expert knowledge into the computation procedures, which provide us the inspiration of such design. In this part, we will how the human designed algorithms can inspire us to tackle both the representation learning and generative modeling in the graph structured domain.

Representation Learning We connect both WL kernel and graphical model inference algorithms with new design of graph neural networks named `structure2vec`. With the inspiration from PageRank [171], we present `SSE`, which scale up this discriminative representation into graphs with hundreds millions of nodes.

Generative Modeling We study the generation problem of both sequences and tree/graphs. For sequence generation, we borrow the idea of Hidden-Semi Markov Model (HSMM) and present our `r-HSMM` which models the generation of segments using Recurrent Neural Networks (RNN). For tree/graph generation, we focus on a set of problems which have explicit syntax and semantics, such as ASTs for representing programs, or SMILES [1] for represent-

[1] https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html

ing molecules. Our method SD-VAE borrows the idea of syntax-directed translation [1] in compiler theory, which regulates the semantics in deep generative models.

1.1.2 Part II: Deep Learning enhanced algorithms

The simplicity of some algorithms often comes with low capacity. With the integration of deep learning into submodules of these algorithms, one can typically get boosted performance. In this part, we present a method to learn the heuristic functions inside greedy algorithms. Specifically, we tackle a set of NP-complete problems on graphs, where the algorithmic steps are defined by greedy algorithm. We learn the heuristics using graph neural networks jointly with reinforcement learning (RL), which we name our method as S2V-DQN. Such generic algorithm has a range of applications in the combinatorial optimization domain. We extended our approach into security domain, namely the adversarial attack [173] problem in combinatorial space, and program/App testing problem [58].

1.1.3 Part III: Towards Reasoning with Graphs

The above two sections have discussed the situations where we have an algorithm at hand, and the remaining is how to apply the algorithm better with deep learning on structural data. With such good inductive bias, we can typically obtain better generalization ability with smaller sample complexity. This procedure can be viewed as deductive reasoning. On the contrary, the inductive reasoning requires the ability to summarize from the observations and conclude the generic knowledge or rules. Such ability of 'creativity' is one of the key differences between human intelligence and artificial intelligence.

In this part, we extend the scope by viewing the algorithms as combinatorial structures. In mathematics or computer science view, the algorithms are instructions with dependency. Such dependency can be defined with computation graphs, and we can thus represent the algorithms with graphs. There could be other structural views of the algorithms. If we express the algorithm as programs, then one example of this inductive reasoning is program synthesis. The problem of structured prediction is also relevant. The classical methods

like structured SVM [227] typically has a candidate set of outputs (typically obtained using dynamic programming), which reduces the decision space. However such dynamic programming structure may not exist in more generic settings.

Here we take an initial step to tackle this problem, where we treat the logic rules as graphs and focus on the problem of inferring loop invariants for program synthesis. In program verification, the loops are typically unrolled with maximum number of steps. With the help of loop invariant, one can provide more efficient and stronger analysis of the program. We name it as `code2inv`, as it takes a piece of source code and directly infer the loop invariant. Such initial attempt uses RL with specially designed reward function.

The above work composes logic expressions from scratch. Typically in real world applications, it would be more feasible to select set of inductive rules from a candidate pool. We demonstrate this idea for the retrosynthesis prediction problem in chemistry. The retrosynthesis is the reverse problem of reaction prediction. In reaction prediction, we are asked to predict the outcome compounds given the reactants. In retrosynthesis, the desired outcome is given and we are asked to predict the reactants. This is abductive reasoning, as the desired outcome typically doesn't have the side products, which are necessary in chemical reaction. Our work brings the chemical knowledge into play, where the chemical reaction rules server as the templates for reducing the decision space. This work combines such logic reasoning with deep learning on molecule structures.

1.2 Organization of the thesis

The following of the document will be organized in this way: in Chapter 2, we present the existing works in the related areas, such as representation learning, generative learning, and combinatorial optimization on graph. Then in Part I and II, we will talk about how the human designed algorithms and deep learning can benefit from each other. In the third part, we will present our initial attempts to inductive reasoning with structures. Finally in Chapter 11, we conclude and discuss about future research directions.

CHAPTER 2

LITERATURE SURVEY

In this chapter, we will discuss the related works in different graph learning problems.

2.1 Graph representation learning

Graphs and networks are commonly used data structure for representing the real-world relationships, *e.g.*, molecular 3D structure networks, knowledge graphs, publication networks, social and communication networks, *etc.*. Analysis based on the network structured data, including prediction over nodes, edges, and the whole graph, becomes more and more attractive and has been applied to many problems. Most of these tasks rely on embedding the graph/network information into vectors, which is essential to the success of these applications. This topic is typically referred as graph or network embedding, *i.e.*, the representation graphs into vector space.

A typical solution for the representation of nodes and edges in graph relies on the hand-crafted domain-specific features. Regardless of the tedious effort and requirement for expert knowledge for feature engineering, such hand-crafted features may not be effective to capture the properties of the graph for the specific target tasks. In literature, various spectral methods [51, 226, 19] have been proposed. Since these algorithms relies on matrix decomposition of the adjacent matrix, both their memory and computational cost become prohibitive for large-scale networks, which may contains millions or billions of nodes. Meanwhile, as explained in [224, 90], these algorithms only preserves the *first-order* neighborhood information in the resulted embeddings, where the higher-order structural information is abandoned.

Recently, the deep learning techniques open a new view for graph embedding. One promising direction is preserving some desired properties of the graph through approximat-

ing the corresponding statistics with embeddings. Specifically, the DeepWalk [177] extends the word2vec [154] in NLP to graph embeddings, which tries to preserve the co-occurrence between nodes in the local structure obtained by random walks. Different heuristics, such as biased random walk model [90] or first and second order proximity [224] have been proposed later. However, the capacity of these models might be limited due to: **i)**, the long range information might not be easily incorporated in a computational or statistical efficient way (via longer random walks or high-order proximity); **ii)**, these models are featureless and trained unsupervisedly, *i.e.*, independent to the down-streaming tasks, therefore, the obtained embeddings may be inferior in capturing the relevant information to the labels.

Another set of work relies on the convolution operator over graphs, which has been showing their effectiveness in capturing the long-range information in graph in real-world applications. By treating the convolution in spatial domain, the graph neural networks (GNNs) [193] proposed a general framework of performing neural network operators over structured data. However it utilizes Almeida-Pineda algorithm for optimization, which limits the scalability of the model. Following this, several work has been proposed to scale up the training procedure, by unrolling the convolution operator with several predefined convolution layers [71, 145, 168, 14, 55, 83, 140, 99, 229]. Such expansion could be understood as an approximation to the Almeida-Pineda algorithm with finite-depth neural network, therefore, alleviating the computational burden. The graph convolution has also been raised in spectrum domain [34, 130, 63].

However, the above mentioned algorithms still cannot achieve the speed requirement for graphs with hundreds of millions of nodes.

2.2 Graph generative modeling

Generating naturally looking images have been a hot topic in computer vision, especially since the invention of generative models like Variational AutoEncoder (VAE [129]) and Generative Adversarial Network (GAN [88]). However, unlike images which can be repre-

sented with tensors in continuous space, the graphs are discrete and combinatorial, which makes it tricky to apply GAN (though there have been several attemps, *e.g.*, NetGAN [26] which is based on SeqGAN [246]).

When the graphs have language representation with explicit syntax and semantics, one can typically reduce it to the sequence generation problem. This has been well studied under the seq2seq [219] framework that models the generation of sequence as a series of token choices parameterized by recurrent neural networks (RNNs). From there, CVAE [86] is a representative work of such paradigm for the chemical molecule generation, using the SMILES line notation [237] for representing molecules. Several improvements have been proposed, including an extra validator model [109], data augmentation [25], active learning [108] and reinforcement learning [93]. However, because of the lack of formalization of syntax and semantics serving as the restriction of the *particular* structured data, underfitted *general-purpose* string generative models will often lead to invalid outputs [22]. For the considerations of computational cost and model generality, context-free grammars (CFG) have been taken into account in the decoder parametrization. For instance, in molecule generation tasks, [137] proposes a grammar variational autoencoder (GVAE) in which the CFG of SMILES notation is embedded into the decoder. With the advances of tree based decoder [253, 174, 7, 68], the model generates the parse trees directly in a top-down direction, by repeatedly expanding any nonterminal with its production rules. Note that such VAE based frameworks not only allow sequence modeling, but can also be used to generate the junction trees [111] and adjacency matrix in graphs like GraphVAE [131].

The other line of works following the idea of auto-regressive distribution modeling. These models generate the graph nodes (and edges) one by one, following the auto-regressive parameterization. Representative works include GraphRNN [245], GCPN [244], Deep Graph Generator [146], *etc.*. However, such parameterization may not be scalable for generating large graphs, as the computation cost is typically $O(E(V+E))$ which is quadratic to the number of edges.

2.3 Combinatorial optimization over graphs

Combinatorial optimization problems over graphs arising from numerous application domains, such as social networks, transportation, communications and scheduling, are NP-hard, and have thus attracted considerable interest from the theory and algorithm design communities over the years. Traditional approaches to tackling an NP-hard graph optimization problem have three main flavors: exact algorithms, approximation algorithms and heuristics. Exact algorithms are based on enumeration or branch-and-bound with an integer programming formulation, but are generally prohibitive for large instances. On the other hand, polynomial-time approximation algorithms are desirable, but may suffer from weak optimality guarantees or empirical performance, or may not even exist for inapproximable problems. Heuristics are often fast, effective algorithms that lack theoretical guarantees, and may also require substantial, problem-specific research and trial-and-error on the part of algorithm designers.

Machine learning for combinatorial optimization. Recently, there has been some seminal work on using deep architectures to learn heuristics for combinatorial problems, including the Traveling Salesman Problem [230, 20, 89]. However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems. As we show later, these architectures often require a huge number of instances in order to learn to generalize to new ones. Furthermore, existing works typically use the policy gradient for training [20], a method that is not particularly sample-efficient. While methods [230, 20] can be used on graphs with different sizes – a desirable trait – they require manual, ad-hoc input/output engineering to do so (e.g. padding with zeros).

Reinforcement learning is used to solve a job-shop flow scheduling problem in [252]. Boyan and Moore [31] use regression to learn good restart rules for local search algorithms. Both of these methods require hand-designed, problem-specific features, a limitation with the learned graph embedding.

Machine learning for branch-and-bound. *Learning to search* in branch-and-bound is another related research thread. This thread includes machine learning methods for branching [138, 126], tree node selection [100, 190], and heuristic selection [191, 125]. In comparison, our work promotes an even tighter integration of learning and optimization.

Deep learning for continuous optimization. In continuous optimization, methods have been proposed for learning an update rule for gradient descent [10, 144] and solving black-box optimization problems [41]; these are very interesting ideas that highlight the possibilities for better algorithm design through learning.

2.4 Reasoning with graphs

In this section, we focus on the literature of inductive reasoning with graph structures. There has been a long history in the logic reasoning and format method literature. Due to the limited space, what we will cover here mainly include the recent advances in the area of program synthesis and program learning. More specific related works will be covered in the later sections where we will introduce our works in detail.

Automatically synthesizing a program from its specification has been a key challenge problem since Manna and Waldinger's work [153]. In this context, syntax-guided synthesis (SyGuS) [5] was proposed as a common format to express these problems. Besides several implementations of SyGuS solvers [94, 6, 194, 5], a number of probabilistic techniques have been proposed to model syntactic aspects of programs and to accelerate synthesis[24, 152, 167]. While logical program synthesis approaches guarantee semantic correctness, they are chiefly limited by their scalability and requirement of rigorous specifications.

There have been several attempts to learn general programs using neural networks. One large class of projects includes those attempting to use neural networks to accelerate the discovery of *conventional programs* [16, 163, 64, 174]. Most existing works only consider specifications which are in the form input-output examples, where weak supervision [147, 40, 35] or more fine grained trace information is provided to help training. In our setting,

there is no supervision for the ground truth loop invariant, and the agent needs to be able to compose a loop invariant purely from trial-and-error. Drawing inspiration from both programming languages and embedding methods, we build up an efficient learning agent that can perform end-to-end reasoning, in a way that mimics human experts.

PART I: Algorithm inspired deep learning for graphs

In this part, we will cover several of our works that are inspired by the human designed algorithm patterns. In Chapter 3 and 4, we present the discriminative feature learning over graphs, which take the inspiration from graphical model inference algorithms and steady state algorithms like PageRank; in Chapter 5 and 6, we study the generative modeling of structured data, for both sequences and tree/graphs, in which we built upon classical models like Hidden-Semi Markov Models and classical algorithms like syntax directed translation.

CHAPTER 3

DISCRIMINATIVE GRAPH REPRESENTATION LEARNING

Kernel classifiers and regressors designed for structured data, such as sequences, trees and graphs, have significantly advanced a number of interdisciplinary areas such as computational biology and drug design. Typically, kernels are designed beforehand for a data type which either exploit statistics of the structures or make use of probabilistic generative models, and then a discriminative classifier is learned based on the kernels via convex optimization. However, such an elegant two-stage approach also limited kernel methods from scaling up to millions of data points, and exploiting discriminative information to learn feature representations.

We propose, `structure2vec`, an effective and scalable approach for structured data representation based on the idea of embedding latent variable models into feature spaces, and learning such feature spaces using discriminative information. Interestingly, `structure2vec` extracts features by performing a sequence of function mappings in a way similar to graphical model inference procedures, such as mean field and belief propagation. In applications involving millions of data points, we showed that `structure2vec` runs 2 times faster, produces models which are $10,000$ times smaller, while at the same time achieving the state-of-the-art predictive performance.

3.1 Introduction

Structured data, such as sequences, trees and graphs, are prevalent in a number of interdisciplinary areas such as protein design, genomic sequence analysis, and drug design [195]. To learn from such complex data, we have to first transform such data explicitly or implicitly into some vectorial representations, and then apply machine learning algorithms in the resulting vector space. So far kernel methods have emerged as one of the most effective

tools for dealing with structured data, and have achieved the state-of-the-art classification and regression results in many sequence [142, 232] and graph datasets [80, 28].

The success of kernel methods on structured data relies crucially on the design of kernel functions — positive semidefinite similarity measures between pairs of data points [196]. By designing a kernel function, we have implicitly chosen a corresponding feature representation for each data point which can potentially has infinite dimensions. Later learning algorithms for various tasks and with potentially very different nature can then work exclusively on these pairwise kernel values without the need to access the original data points. Such modular structure of kernel methods has been very powerful, making them the most elegant and convenient methods to deal with structured data. Thus designing kernel for different structured objects, such as strings, trees and graphs, has always been an important subject in the kernel community. However, in the big data era, this modular framework has also limited kernel methods in terms of their ability to scale up to millions of data points, and exploit discriminative information to learn feature representations.

For instance, a class of kernels are designed based on the idea of "bag of structures" (BOS), where each structured data point is represented as a vector of counts for elementary structures. The spectrum kernel and variants for strings [142], subtree kernel [183], graphlet kernel [208] and Weisfeiler-lehman graph kernel [207] all follow this design principle. In other words, the feature representations of these kernels are fixed before learning, with each dimension corresponding to a substructure, independent of the supervised learning tasks at hand. Since there are many unique substructures which may or may not be useful for the learning tasks, the explicit feature space of such kernels typically has very high dimensions. Subsequently algorithms dealing with the pairwise kernel values have to work with a big kernel matrix squared in the number of data points. The square dependency on the number of data points largely limits these BOS kernels to datasets of size just thousands.

A second class of kernels are based on the ingenious idea of exploiting the ability of

probabilistic graphical models (GM) in describing noisy and structured data to design kernels. For instance, one can use hidden Markov models for sequence data, and use pairwise Markov random fields for graph data. The Fisher kernel [107] and probability product kernel [110] are two representative instances within the family. The former method first fits a common generative model to the entire dataset, and then uses the empirical Fisher information matrix and the Fisher score of each data point to define the kernel; The latter method instead fits a different generative model for each data point, and then uses inner products between distributions to define the kernel. Typically the parameterization of these GM kernels are chosen before hand. Although the process of fitting generative models allow the kernels to adapt to the geometry of the input data, the resulting feature representations are still independent of the discriminative task at hand. Furthermore, the extra step of fitting generative models to data can be a challenging computation and estimation task by itself, especially in the presence of latent variables. Very often in practice, one finds that BOS kernels are easier to deploy than GM kernels, although the latter is supposed to capture the additional geometry and uncertainty information of data.

In this chapter, we wish to revisit the idea of using graphical models for kernel or feature space design, with the goal of scaling up kernel methods for structured data to millions of data points, and allowing the kernel to learn the feature representation from label information. Our idea is to model each structured data point as a latent variable model, then embed the graphical model into feature spaces [211, 214], and use inner product in the embedding space to define kernels. Instead of fixing a feature or embedding space beforehand, we will also learn the feature space by directly minimizing the empirical loss defined by the label information. The resulting embedding algorithm, `structure2vec`, runs in a scheme similar to graphical model inference procedures, such as mean field and belief propagation. Instead of performing probabilistic operations (such as sum, product and renormalization), the algorithm performs nonlinear function mappings in each step, inspired by kernel message passing algorithm in [213, 212]. Furthermore, `structure2vec` is also different

from the kernel message passing algorithm in several aspects. First, structure2vec deals with a different scenario, *i.e.*, learning similarity measure for structured data. Second, structure2vec learns the nonlinear mappings using the discriminative information. And third, a variant of structure2vec can run in a mean field update fashion, different from message passing algorithms.

Besides the above novel aspects, structure2vec is also very scalable in terms of both memory and computation requirements. First, it uses a small and explicit feature map for the nonlinear feature space, and avoids the need for keeping the kernel matrix. This makes the subsequent classifiers or regressors order of magnitude smaller compared to other methods. Second, the nonlinear function mapping in structure2vec can be learned using stochastic gradient descent, allowing it to handle extremely large scale datasets. Finally in experiments, we show that structure2vec compares favorably to other kernel methods in terms of classification accuracy in medium scale sequence and graph benchmark datasets including SCOP and NCI. Furthermore, structure2vec can handle extremely large data set, such as the 2.3 million molecule dataset from Harvard Clean Energy Project, run 2 times faster, produce model $10,000$ times smaller and achieve state-of-the-art accuracy. These strong empirical results suggest that the graphical models, theoretically well-grounded methods for capturing structure in data, combined with embedding techniques and discriminative training can significantly improve the performance in many large scale real-world structured data classification and regression problems.

3.2 Backgrounds

We denote by X a random variable with domain $\mathcal{X}$, and refer to instantiations of X by the lower case character, x. We denote a density on $\mathcal{X}$ by $p(X)$, and denote the space of all such densities by $\mathcal{P}$. We will also deal with multiple random variables, $X_1, X_2, \ldots, X_\ell$, with joint density $p(X_1, X_2, \ldots, X_\ell)$. For simplicity of notation, we assume that the domains of all $X_t, t \in [\ell]$ are the same, but the methodology applies to the cases where they

have different domains. In the case when $\mathcal{X}$ is a discrete domain, the density notation should be interpreted as probability, and integral should be interpreted as summation instead. Furthermore, we denote by H a hidden variable with domain $\mathcal{H}$ and distribution $p(H)$. We use similar notation convention for variable H and X.

Kernel Methods. Suppose the structured data is represented by $\chi \in \mathcal{G}$. Kernel methods owe the name to the use of kernel functions, $k(\chi, \chi') : \mathcal{G} \times \mathcal{G} \mapsto \mathbb{R}$, which are symmetric positive semidefinite (PSD), meaning that for all $n > 1$, and $\chi_1, \ldots, \chi_n \in \mathcal{G}$, and $c_1, \ldots, c_n \in \mathbb{R}$, we have $\sum_{i,j=1}^{n} c_i c_j k(\chi_i, \chi_j) \geqslant 0$. A signature of kernel methods is that learning algorithms for various tasks and with potentially very different nature can work exclusively on these pairwise kernel values without the need to access the original data points.

Kernels for Structured Data. Each kernel function will correspond to some feature map $\phi(\chi)$, where the kernel function can be expressed as the inner product between feature maps, *i.e.*, $k(\chi, \chi') = \langle \phi(\chi), \phi(\chi') \rangle$. For structured input domain, one can design kernels using counts on substructures. For instance, the spectrum kernel for two sequences χ and χ' is defined as [142]

$$k(\chi, \chi') = \sum\nolimits_{s \in \mathcal{S}} \#(s \in \chi) \#(s \in \chi') \tag{3.1}$$

where $\mathcal{S}$ is the set of possible subsequences, $\#(s \in x)$ counts the number occurrence of subsequence s in x. In this case, the feature map $\phi(\chi) = (\#(s_1 \in \chi), \#(s_2 \in \chi), ...)^\top$ corresponds to a vector of dimension $|\mathcal{S}|$. Similarly, the graphlet kernel [208] for two graphs χ and χ' can also be defined as (3.1), but $\mathcal{S}$ is now the set of possible subgraphs, and $\#(s \in \chi)$ counts the number occurrence of subgraphs. We refer to this class of kernels as "bag of structures" (BOS) kernel.

Kernels can also be defined by leveraging the power of probabilistic graphical models. For instance, the Fisher kernel [107] is defined using a parametric model $p(\chi|\theta^*)$

around its maximum likelihood estimate θ^*, *i.e.*, $k(\chi, \chi') = U_\chi^\top I^{-1} U_{\chi'}$, where $U_\chi := \nabla_{\theta=\theta^*} \log p(\chi|\theta)$ and $I = \mathbb{E}_\mathcal{G}[U_\mathcal{G} U_\mathcal{G}^\top]$ is the Fisher information matrix. Another classical example along the line is the probability product kernel [110]. Different from the Fisher kernel based on generative model fitted with the whole dataset, the probability product kernel is calculated based on the models $p(\chi|\theta)$ fitted to individual data point, *i.e.*, $k(\chi, \chi') = \int_\mathcal{G} p(\tau|\theta_\chi)^\rho p(\tau|\theta_{\chi'})^\rho d\tau$ where θ_χ and $\theta_{\chi'}$ are the maximum likelihood parameters for data point χ and χ' respectively. We refer to this class of kernels as the "graphical model" (GM) kernels.

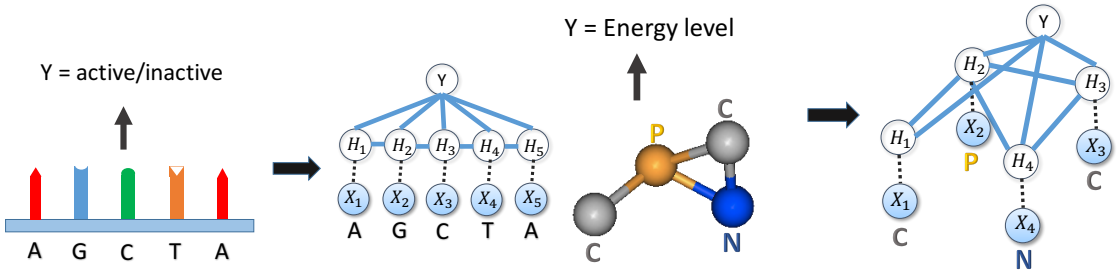
Hilbert Space Embedding of Distributions. Hilbert space embeddings of distributions are mappings of distributions into potentially *infinite* dimensional feature spaces [211],

$$\mu_X := \mathbb{E}_X [\phi(X)] = \int_\mathcal{X} \phi(x) p(x) dx \ : \ \mathcal{P} \mapsto \mathcal{F} \tag{3.2}$$

where the distribution is mapped to its expected feature map, *i.e.*, to a point in a feature space. Kernel embedding of distributions has rich representational power. Some feature map can make the mapping injective [216], meaning that if two distributions, $p(X)$ and $q(X)$, are different, they are mapped to two distinct points in the feature space. For instance, when $\mathcal{X} = \mathbb{R}^d$, the feature spaces of many commonly used kernels, such as the Gaussian RBF kernel $\exp(-\|x - x'\|_2^2)$, can make the embedding injective.

Alternatively, one can treat an injective embedding μ_X of a density $p(X)$ as a sufficient statistic of the density. Any information we need from the density is preserved in μ_X: with μ_X one can uniquely recover $p(X)$, and any operation on $p(X)$ can be carried out via a corresponding operation on μ_X with the same result. For instance, this property will allow us to compute a functional $f : \mathcal{P} \mapsto \mathbb{R}$ of the density using the embedding only, *i.e.*,

$$f(p(x)) = \tilde{f}(\mu_X) \tag{3.3}$$



(a) LVM for string data (b) LVM for graph data

Figure 3.1: Building latent variable models (LVM) from structured string and general graph data. Y is the supervised information, which can be real number (for regression) or discrete integer (for classification).

where $\tilde{f} : \mathcal{F} \mapsto \mathbb{R}$ is a corresponding function applied on μ_X. Similarly the property can also be generalized to operators. For instance, applying an operator $\mathcal{T} : \mathcal{P} \mapsto \mathbb{R}^d$ to a density can also be equivalently carried out using its embedding, *i.e.*,

$$\mathcal{T} \circ p(x) = \widetilde{\mathcal{T}} \circ \mu_X, \tag{3.4}$$

where $\widetilde{\mathcal{T}} : \mathcal{F} \mapsto \mathbb{R}^d$ is the alternative operator working on the embedding. In our later sections, we will extensively exploit this property of injective embeddings, by assuming that there exists a feature space such that the embeddings are injective.

3.3 Model for a Structured Data Point

Without loss of generality, we assume each structured data point χ is a graph, with a set of nodes $\mathcal{V} = \{1, \ldots, V\}$ and a set of edges $\mathcal{E}$. We will use x_i to denote the value of the attribute for node i. We note the node attributes are different from the label of the entire data point. For instance, each atom in a molecule will correspond to a node in the graph, and the node attribute will be the atomic number, while the label for the entire molecule can be whether the molecule is a good drug or not. Other structures, such as sequences and trees, can be viewed as special cases of general graphs.

We will model the structured data point χ as an instance drawn from a graphical model.

More specifically, we will model the label of each node in the graph with a variable X_i, and furthermore, associate an additional hidden variable H_i with it. Then we will define a pairwise Markov random field on these collection of random variables

$$p(\{H_i\}, \{X_i\}) \propto \prod_{i \in \mathcal{V}} \Phi(H_i, X_i) \prod_{(i,j) \in \mathcal{E}} \Psi(H_i, H_j) \tag{3.5}$$

where Ψ and Φ are nonnegative node and edge potentials respectively. In this model, the variables are connected according to the graph structure of the input data point. That is to say, we use the graph structure of the input data directly as the conditional independence structure of an undirected graphical model. Figure 3.1 illustrates two concrete examples in constructing the graphical models for strings and graphs. One can design more complicated graphical models which go beyond pairwise Markov random fields, and consider longer range interactions with potentials involving more variables. We will focus on pairwise Markov random fields for simplicity of representation.

We note that such a graphical model is built for each individual data point, and the conditional independence structures of two graphical models can be different if the two data points χ and χ' are different. Furthermore, we do not observe the value for the hidden variables $\{H_i\}$, which makes the learning of the graphical model potentials Φ and Ψ even more difficult. Thus, we will not pursue the standard route of maximum likelihood estimation, and rather we will consider the sequence of computations needed when we try to embed the posterior of $\{H_i\}$ into a feature space.

3.4 Embedding Latent Variable Models

We embed the posterior marginal $p(H_i | \{x_i\})$ of a hidden variable using $\phi(H_i)$, *i.e.*,

$$\mu_i = \int_{\mathcal{H}} \phi(h_i) p(h_i | \{x_i\}) dh_i. \tag{3.6}$$

The exact form of $\phi(H_i)$ and the parameters in MRF $p(H_i|\{x_i\})$ is not fixed at the moment, and we will learn them later using supervision signals for the ultimate discriminative target. For now, we will assume that $\phi(H_i) \in \mathbb{R}^d$ is a finite dimensional feature space, and the exact value of d will determined by cross-validation in later experiments. However, compute the embedding is a very challenging task for general graphs: it involves performing an inference in graphical model where we need to integrate out all variables expect H_i, *i.e.*,

$$p(H_i|\{x_i\}) = \int_{\mathcal{H}^{V-1}} p(H_i, \{h_j\} \,|\, \{x_j\}) \prod_{j \in \mathcal{V} \setminus i} dh_j. \tag{3.7}$$

Only when the graph structure is a tree, exact computation can be carried out efficiently via message passing [175]. Thus in the general case, approximate inference algorithms, *e.g.*, mean field inference and loopy belief propagation (BP), are developed. In many applications, however, these variational inference algorithms exhibit excellent empirical performance [166]. Several theoretical studies have also provided insight into the approximations made by loopy BP, partially justifying its application to graphs with cycles [234, 242].

In the following subsection, we will explain the embedding of mean field and loopy BP. More detailed mathematical derivations can be found in Appendix A. We show that the iterative update steps in these algorithms, which are essentially minimizing approximations to the exact free energy, can be simply viewed as function mappings of the embedded marginals using the alternative view in (3.3) and (3.4).

3.4.1 Embedding Mean-Field Inference

The vanilla mean-field inference tries to approximate $p(\{H_i\} \,|\, \{x_i\})$ with a product of *independent* density components $p(\{H_i\} \,|\, \{x_i\}) \approx \prod_{i \in \mathcal{V}} q_i(h_i)$ where each $q_i(h_i) \geq 0$ is a valid density, such that $\int_{\mathcal{H}} q_i(h_i) dh_i = 1$. Furthermore, these density components are found

by minimizing the following variational free energy [234],

$$\min_{q_1,\ldots,q_d} \int_{\mathcal{H}^d} \prod_{i\in\mathcal{V}} q_i(h_i) \log \frac{\prod_{i\in\mathcal{V}} q_i(h_i)}{p(\{h_i\} \mid \{x_i\})} \prod_{i\in\mathcal{V}} dh_i.$$

One can show that the solution to the above optimization problem needs to satisfy the following fixed point equations for all $i \in \mathcal{V}$

$$\begin{aligned}\log q_i(h_i) =& c_i + \log(\Phi(h_i, x_i)) + \sum_{j\in\mathcal{N}(i)} \int_{\mathcal{H}} q_j(h_j) \log(\Psi(h_i, h_j)\Phi(h_j, x_j))dh_j \\ =& c_i' + \log \Phi(h_i, x_i) + \sum_{j\in\mathcal{N}(i)} \int_{\mathcal{H}} q_j(h_j) \log \Psi(h_i, h_j)dh_j\end{aligned}$$

where $c_i' = c_i + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \Phi(h_j, x_j)dh_j$. Here $\mathcal{N}(i)$ are the set of neighbors of variable H_i in the graphical model, and c_i is a constant. The fixed point equations in (3.8) imply that $q_i(h_i)$ is a functional of a set of neighboring marginals $\{q_j\}_{j\in\mathcal{N}(i)}$, *i.e.*,

$$q_i(h_i) = f\left(h_i, x_i, \{q_j\}_{j\in\mathcal{N}(i)}\right). \tag{3.8}$$

If for each marginal q_i, we have an injective embedding $\widetilde{\mu}_i = \int_{\mathcal{H}} \phi(h_i)q_i(h_i)dh_i$. Then, using similar reasoning as in (3.3), we can equivalently express the fixed point equation from an embedding point of view, *i.e.*, $q_i(h_i) = \tilde{f}(h_i, x_i, \{\widetilde{\mu}_j\}_{j\in\mathcal{N}(i)})$, and consequently using the operator view from (3.4), we have

$$\widetilde{\mu}_i = \widetilde{\mathcal{T}} \circ \left(x_i, \{\widetilde{\mu}_j\}_{j\in\mathcal{N}(i)}\right). \tag{3.9}$$

For the embedded mean field (3.9), the function $\tilde{f}$ and operator $\widetilde{\mathcal{T}}$ have complicated non-linear dependencies on the potential functions Ψ, Φ, and the feature mapping ϕ which is unknown and need to be learned from data. Instead of first learning the Ψ and Φ, and then working out $\widetilde{\mathcal{T}}$, we will pursue a different route where we directly parameterize $\widetilde{\mathcal{T}}$ and later

Algorithm 1 Embedded Mean Field

Input: parameter $\mathbf{W}$ in $\widetilde{\mathcal{T}}$
Initialize $\widetilde{\mu}_i^{(0)} = \mathbf{0}$, for all $i \in \mathcal{V}$
for $t = 1$ **to** T **do**
  for $i \in \mathcal{V}$ **do**
    $l_i = \sum_{j \in \mathcal{N}(i)} \widetilde{\mu}_i^{(t-1)}$
    $\widetilde{\mu}_i^{(t)} = \sigma(W_1 x_i + W_2 l_i)$
  end for
end for ▷ fixed point equation update
return $\{\widetilde{\mu}_i^T\}_{i \in \mathcal{V}}$

Algorithm 2 Embedding Loopy BP

Input: parameter $\mathbf{W}$ in $\widetilde{\mathcal{T}}_1$ and $\widetilde{\mathcal{T}}_2$
Initialize $\widetilde{\nu}_{ij}^{(0)} = \mathbf{0}$, for all $(i,j) \in \mathcal{E}$
for $t = 1$ **to** T **do**
  for $(i,j) \in \mathcal{E}$ **do**
    $\widetilde{\nu}_{ij}^{t} = \sigma(W_1 x_i + W_2 \sum_{k \in \mathcal{N}(i) \backslash j} \widetilde{\nu}_{ki}^{(t-1)})$
  end for
end for
for $i \in \mathcal{V}$ **do**
  $\widetilde{\mu}_i = \sigma(W_3 x_i + W_4 \sum_{k \in \mathcal{N}(i) \backslash j} \widetilde{\nu}_{ki}^{(T)})$
end for
return $\{\widetilde{\mu}_i\}_{i \in \mathcal{V}}$

learn it with supervision signals.

In terms of the parameterization, we will assume $\widetilde{\mu}_i \in \mathbb{R}^d$ where d is a hyperparameter chosen using cross-validation. For $\widetilde{\mathcal{T}}$, one can use any nonlinear function mappings. For instance, we can parameterize it as a neural network

$$\widetilde{\mu}_i = \sigma\Big(W_1 x_i + W_2 \sum_{j \in \mathcal{N}(i)} \widetilde{\mu}_j\Big) \tag{3.10}$$

where $\sigma(\cdot) := \max\{0, \cdot\}$ is a rectified linear unit applied elementwisely to its argument, and $\mathbf{W} = \{W_1, W_2\}$. The number of the rows in $\mathbf{W}$ equals to d. With such parameterization, the mean field iterative update in the embedding space can be carried out as Algorithm 1. We could also multiply $\widetilde{\mu}_i$ with V to rescale the range of message embeddings if needed. In fact, with or without V, the functions will be the same in terms of the representation power. Specifically, for any $(\mathbf{W}, V)$, we can always find another 'equivalent' parameters $(\mathbf{W}', I)$ where $\mathbf{W}' = \{W_1, W_2 V\}$.

3.4.2 Embedding Loopy Belief Propagation

Loopy belief propagation is another variational inference method, which essentially optimizes the Bethe free energy taking *pairwise* interactions into account [243],

$$\begin{aligned}\min_{\{q_{ij}\}_{(i,j)\in\mathcal{E}}} \quad & -\sum_i (|\mathcal{N}(i)|-1)\int_{\mathcal{H}} q_i(h_i)\log\frac{q_i(h_i)}{\Phi(h_i,x_i)}dh_i \\ & +\sum_{i,j}\int_{\mathcal{H}^2} q_{ij}(h_i,h_j)\log\frac{q_{ij}(h_i,h_j)}{\Psi(h_i,h_j)\Phi(h_i,x_i)\Phi(h_j,x_j)}dh_i dh_j\end{aligned}$$

subject to pairwise marginal consistency constraints: $\int_{\mathcal{H}} q_{ij}(h_i,h_j)dh_j = q_i(h_i)$, $\int_{\mathcal{H}} q_{ij}(h_i, h_j)dh_j = q_i(h_i)$, and $\int_{\mathcal{H}} q_i(h_i)dh_i = 1$. One can obtain the fixed point condition for the above optimization for all $(i,j)\in\mathcal{E}$,

$$\begin{aligned} m_{ij}(h_j) &\propto \int_{\mathcal{H}} \prod_{k\in\mathcal{N}(i)\setminus j} m_{ki}(h_i)\Phi_i(h_i,x_i)\Psi_{ij}(h_i,h_j)dh_i, \\ q_i(h_i) &\propto \Phi(h_i,x_i)\prod_{j\in\mathcal{N}(i)} m_{ji}(h_i). \end{aligned} \tag{3.11}$$

where $m_{ij}(h_j)$ is the intermediate result called the message from node i to j. Furthermore, $m_{ij}(h_j)$ is a nonnegative function which can be normalized to a density, and hence can also be embedded.

Similar to the reasoning in the mean field case, the (3.11) implies the messages $m_{ij}(h_j)$ and marginals $q_i(h_i)$ are functionals of messages from neighbors, *i.e.*,

$$m_{ij}(h_j) = f\left(h_j, x_i, \{m_{ki}\}_{k\in\mathcal{N}(i)\setminus j}\right), \quad q_i(h_i) = g\left(h_i, x_i, \{m_{ki}\}_{k\in\mathcal{N}(i)}\right).$$

With the assumption that there is an injective embedding for each message $\widetilde{\nu}_{ij} = \int \phi(h_j)m_{ij}(h_j)dh_j$ and for each marginal $\widetilde{\mu}_i = \int \phi(h_i)q_i(h_i)dh_i$, we can apply the reasoning from (3.3) and (3.4), and express the messages and marginals from the embedding view,

$$\widetilde{\nu}_{ij} \;=\; \widetilde{\mathcal{T}}_1 \circ \left(x_i, \{\widetilde{\nu}_{ki}\}_{k\in\mathcal{N}(i)\setminus j}\right), \quad \widetilde{\mu}_i = \widetilde{\mathcal{T}}_2 \circ \left(x_i, \{\widetilde{\nu}_{ki}\}_{k\in\mathcal{N}(i)}\right). \tag{3.12}$$

We will also use parametrization for loopy BP embedding similar to the mean field case, *i.e.*, neural network with rectified linear unit σ. Specifically, assume $\widetilde{\nu}_{ij} \in \mathbb{R}^d$, $\widetilde{\mu}_i \in \mathbb{R}^d$

$$\widetilde{\nu}_{ij} = \sigma\Big(W_1 x_i + W_2 \sum_{k \in \mathcal{N}(i) \setminus j} \widetilde{\nu}_{ki}\Big), \quad \widetilde{\mu}_i = \sigma\Big(W_3 x_i + W_4 \sum_{k \in \mathcal{N}(i)} \widetilde{\nu}_{ki}\Big) \tag{3.13}$$

where $\mathbf{W} = \{W_1, W_2, W_3, W_4\}$ are matrices with appropriate sizes. Note that one can use other nonlinear function mappings to parameterize $\widetilde{\mathcal{T}}_1$ and $\widetilde{\mathcal{T}}_2$ as well. Overall, the loopy BP embedding updates is summarized in Algorithm 2.

With similar strategy as in mean field case, we will learn the parameters in $\widetilde{\mathcal{T}}_1$ and $\widetilde{\mathcal{T}}_2$ later with supervision signals from the discriminative task.

3.4.3 Embedding Other Variational Inference

In fact, there are many other variational inference methods, with different forms of free energies or different optimization algorithms, resulting different message update forms, *e.g.*, double-loop BP [249], damped BP [156], tree-reweightd BP [233], and generalized BP [243]. The proposed embedding method is a general technique which can be tailored to these algorithms. The major difference is the dependences in the messages. For the details of embedding of these algorithms, please refer to the Appendix section of original paper [55].

3.5 Discriminative Training

Similar to kernel BP [213, 212] and kernel EP [113], our current work exploits feature space embedding to reformulate graphical model inference procedures. However, different from the kernel BP and kernel EP, in which the feature spaces are chosen beforehand and the conditional embedding operators are learned locally, our approach will learn both the feature spaces, the transformation $\widetilde{\mathcal{T}}$, as well as the regressor or classifier for the target values end-to-end using label information.

Specifically, we are provided with a training dataset $\mathcal{D} = \{\chi_n, y_n\}_{n=1}^N$, where χ_n is a structured data point and $y_n \in \mathcal{Y}$, where $\mathcal{Y} = \mathbb{R}$ for regression or $\mathcal{Y} = \{1, \ldots, K\}$ for classification problem, respectively. With the feature embedding procedure introduced in Section 3.4, each data point will be represented as a set of embeddings $\{\tilde{\mu}_i^n\}_{i \in V_n} \in \mathcal{F}$. Now the goal is to learn a regression or classification function f linking $\{\tilde{\mu}_i^n\}_{i \in V_n}$ to y_n.

More specifically, in the case of regression problem, we will parametrize function $f(\chi_n)$ as $u^\top \sigma(\sum_{i=1}^{V_n} \tilde{\mu}_i^n)$, where $u \in \mathbb{R}^d$ is the final mapping from summed (or pooled) embeddings to output. The parameters u and those $\mathbf{W}$ involved in the embeddings are learned by minimizing the empirical square loss

$$\min_{u, \mathbf{W}} \sum\nolimits_{n=1}^{N} \left(y_n - u^\top \sigma \left(\sum\nolimits_{i=1}^{V_n} \widetilde{\mu}_i^n \right) \right)^2 . \tag{3.14}$$

Note that each data point will have its own graphical model and embedded features due to its individual structure, but the parameters u and $\mathbf{W}$, are shared across these graphical models.

In the case of K-class classification problem, we denote z is the 1-of-K representation of y, *i.e.*, $z \in \{0, 1\}^K$, $z^k = 1$ if $y = k$, and $z^i = 0, \forall i \neq k$. By adopt the softmax loss, we obtain the optimization for embedding parameters and discriminative classifier estimation:

$$\min_{\mathbf{u} = \{u^k\}_{k=1}^K, \mathbf{W}} \sum_{n}^{N} \sum_{k=1}^{K} -z_n^k \log u^k \sigma \left(\sum_{i=1}^{V_n} \widetilde{\mu}_i^n \right), \tag{3.15}$$

where $\mathbf{u} = \{u^k\}_{k=1}^K$, $u^k \in \mathbb{R}^d$ are the parameters for mapping embedding to output.

The same idea can also be generalized to other discriminative tasks with different loss functions. As we can see from the optimization problems (3.14) and (3.15), the objective functions are directly related to the corresponding discriminative tasks, and so as to $\mathbf{W}$ and $\mathbf{u}$. Conceptually, the procedure starts with representing each datum by a graphical model constructed corresponding to its *individual* structure with *sharing* potential functions, and then, we embed these graphical models with the *same* feature mappings. Finally the em-

Algorithm 3 Discriminative Embedding

Input: Dataset $\mathcal{D} = \{\chi_n, y_n\}_{n=1}^N$, loss function $l(f(\chi), y)$.
Initialize $\mathbf{U}^0 = \{\mathbf{W}^0, \mathbf{u}^0\}$ randomly.
for $t = 1$ **to** T **do**
  Sample $\{\chi_t, y_t\}$ uniform randomly from $\mathcal{D}$.
  Construct latent variable model $p(\{H_i^t\}|\chi_n)$ as (3.5).
  Embed $p(\{H_i^t\}|\chi_n)$ as $\{\widetilde{\mu}_i^n\}_{i \in \mathcal{V}_n}$ by Algorithm 1 or 2 with $\mathbf{W}^{t-1}$.
  Update $\mathbf{U}^t = \mathbf{U}^{t-1} + \lambda_t \nabla_{\mathbf{U}^{t-1}} l(f(\widetilde{\mu}^n; \mathbf{U}^{t-1}), y_n)$.
end for
return $\mathbf{U}^T = \{\mathbf{W}^T, \mathbf{u}^T\}$

bedded marginals are aggregated with a prediction function for a discriminative task. The shared potential functions, feature mappings and final prediction functions are all learned together for the ultimate task with supervision signals.

We optimize the objective (3.14) or (3.15) with stochastic gradient descent for scalability consideration. However, other optimization algorithms are also applicable, and our method does not depend on this particular choice. The gradients of the parameters $\mathbf{W}$ are calculated recursively similar to recurrent neural network for sequence models. In our case, the recursive structure will correspond the message passing structure. The overall framework is illustrated in Algorithm 3.

3.6 Experiments

Below we first compare our method with algorithms using prefixed kernel on string and graph benchmark datasets. Then we focus on Harvard Clean Energy Project dataset which contains 2.3 million samples. We demonstrate that while getting comparable performance on medium sized datasets, we are able to handle millions of samples, and getting much better when more training data are given. The two variants of `structure2vec` are denoted as DE-MF and DE-LBP, which stands for discriminative embedding using mean field or loopy belief propagation, respectively.

Our algorithms are implemented with C++ and CUDA, and experiments are carried out on clusters equipped with NVIDIA Tesla K20. The original code is available on

https://github.com/Hanjun-Dai/graphnn. A new reimplementation using PyTorch is also available at https://github.com/Hanjun-Dai/pytorch_structure2vec.

3.6.1 Benchmark structure datasets

We compare our algorithm on string benchmark datasets with the kernel method with existing sequence kernels, *i.e.*, the spectrum string kernel [142], mismatch string kernel [143] and fisher kernel with HMM generative models [107]. On graph benchmark datasets, we compare with subtree kernel [183] (R&G, for short), random walk kernel[80, 231], shortest path kernel [29], graphlet kernel[208] and the family of Weisfeiler-Lehman kernels (WL kernel) [207]. After getting the kernel matrix, we train SVM classifier or regressor on top.

We tune all the methods via cross validation, and report the average performance. Specifically, for structured kernel methods, we tune the degree in $\{1, 2, \ldots, 10\}$ (for mismatch kernel, we also tune the maximum mismatch length in $\{1, 2, 3\}$) and train SVM classifier [37] on top, where the trade-off parameter C is also chosen in $\{0.01, 0.1, 1, 10\}$ by cross validation. For fisher kernel that using HMM as generative model, we also tune the number of hidden states assigned to HMM in $\{2, \ldots, 20\}$.

For our methods, we simply use one-hot vector (the vector representation of discrete node attribute) as the embedding for observed nodes, and use a two-layer neural network for the embedding (prediction) of target value. The hidden layer size $b \in \{16, 32, 64\}$ of neural network, the embedding dimension $d \in \{16, 32, 64\}$ of hidden variables and the number of iterations $t \in \{1, 2, 3, 4\}$ are tuned via cross validation. We keep the number of parameters small, and use early stopping [82] to avoid overfitting in these small datasets.

String Dataset

Here we do experiments on two string binary classification benchmark datasets. The first one (denoted as SCOP) contains 7329 sequences obtained from SCOP (Structural Classification of Proteins) 1.59 database [9]. Methods are evaluated on the ability to detect

members of a target SCOP family (positive test set) belonging to the same SCOP superfamily as the positive training sequences, and no members of the target family are available during training. We use the same 54 target families and the same training/test splits as in remote homology detection [136]. The second one is FC and RES dataset (denoted as FC_RES) provided by CRISPR/Cas9 system, on which the task it to tell whether the guide RNA will direct Cas9 to target DNA. There are 5310 guides included in the dataset. Details of this dataset can be found in [67, 76]. We use two variants for spectrum string kernel: 1) kmer-single, where the constructed kernel matrix $K_k^{(s)}$ only consider patterns of length k; 2) kmer-concat, where kernel matrix $K^{(c)} = \sum_{i=1}^{k} K_k^{(s)}$. We also find the normalized kernel matrix $K_k^{Norm}(x, y) = \frac{K_k(x,y)}{\sqrt{K_k(x,x)K_k(y,y)}}$ helps.

Table 3.1: Mean AUC on string classification datasets

	FC_RES	SCOP
kmer-single	0.7606±0.0187	0.7097±0.0504
kmer-concat	0.7576±0.0235	0.8467±0.0489
mismatch	0.7690±0.0197	0.8637±0.1192
fisher	0.7332±0.0314	0.8662±0.0879
DE-MF	**0.7713±0.0208**	0.9068±0.0685
DE-LBP	0.7701±0.0225	**0.9167±0.0639**

Table 3.1 reports the mean AUC of different algorithms. We found two variants of `structure2vec` are consistently better than the string kernels. Also, the improvement in SCOP is more significant than in FC_RES. This is because SCOP is a protein dataset and its alphabet size $|\Sigma|$ is much larger than that of FC_RES, an RNA dataset. Furthermore, the dimension of the explicit features for a k-mer kernel is $O(|\Sigma|^k)$, which can make the off-diagonal entries of kernel matrix very small (or even zero) with large alphabet size and k. That's also the reason why kmer-concat performs better than kmer-single. `structure2vec` learns a discriminative feature space, rather than prefix it beforehand, and hence does not have this problem.

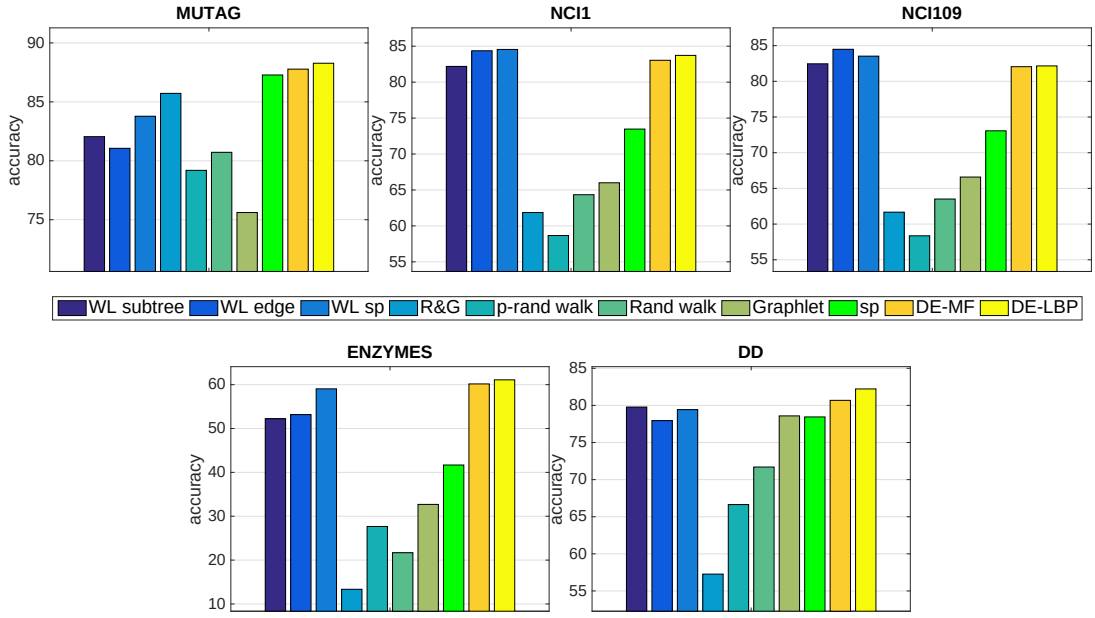


Figure 3.2: 10-fold cross-validation accuracies on graph classification benchmark datasets. The 'sp' in the figure stands for shortest-path.

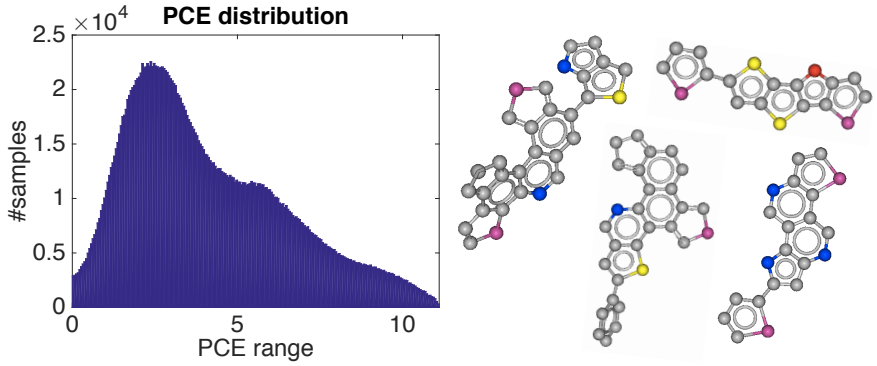
Table 3.2: Statistics [217] of graph benchmark datasets. $|V|$ is the # nodes while $|E|$ is the # edges in a graph. #labels equals to the number of different types of nodes.

	size	avg $\|V\|$	avg $\|E\|$	#labels
MUTAG	188	17.93	19.79	7
NCI1	4110	29.87	32.3	37
NCI109	4127	29.68	32.13	38
ENZYMES	600	32.63	62.14	3
D&D	1178	284.32	715.66	82

Graph Dataset

We test the algorithms on five benchmark datasets for graph kernel: MUTAG, NCI1, NCI109, ENZYMES and D&D. MUTAG [62]. NCI1 and NCI109 [235] are chemical compounds dataset, while ENZYMES [29] and D&D [66] are of proteins. The task is to do multi-class or binary classification. We show the statistics of these datasets in Table 3.2.

The results of baseline algorithms are taken from [207] since we use exactly the same setting here. From the accuracy comparison shown in Figure 3.2, we can see the proposed embedding methods are comparable to the alternative graph kernels, on different graphs



(a) PCE distribution (b) Sample molecules

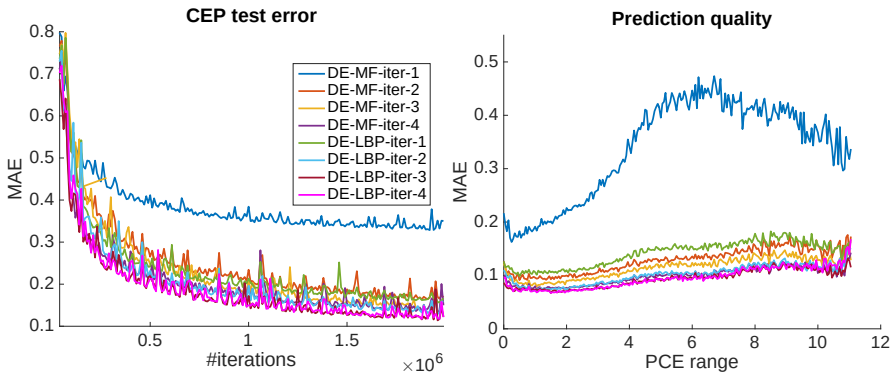
Figure 3.3: PCE value distribution and sample molecules from CEP dataset. Hydrogens are not displayed.

with different number of labels, nodes and edges. Also, in dataset D&D which consists of 82 different types of labels, our algorithm performs much better. As reported in [207], the time required for constructing dictionary for the graph kernel can take up to more than a year of CPU time in this dataset, while our algorithm can learn the discriminative embedding efficiently from structured data directly without the construction of the handcraft dictionary.

3.6.2 Harvard Clean Energy Project(CEP) dataset

The Harvard Clean Energy Project [97] is a theory-driven search for the next generation of organic solar cell materials. One of the most important properties of molecule for this task is the overall efficiency of the energy conversion process in a solar cell, which is determined by the power conversion efficiency (PCE). The Clean Energy Project (CEP) performed expensive simulations for the 2.3 million candidate molecules on IBM's World Community Grid, in order to get this property value. So using machine learning approach to accurately predict the PCE values is a promising direction for the high throughput screening and discovering new materials.

In this experiment, we randomly select 90% of the data for training, and the rest 10% for testing. This setting is similar to [178], except that we use the entire 2.3m dataset here.



(a) Test error vs iterations (b) Prediction quality

Figure 3.4: Details of training and prediction results for DE-MF and DE-LBP with different number of fixed point iterations.

Since the data is distributed unevenly (see Figure 3.3), we resampled the training data (but not the test data) to make the algorithm put more emphasis on molecules with higher PCE values, in order to make accurate prediction for promising candidate molecules. Since the traditional kernel methods are not scalable, we make the explicit feature maps for WL subtree kernel by collecting all the molecules and creating dictionary for the feature space. The other graph kernels, like edge kernel and shortest path kernel, are having too large feature dictionary to work with. We use RDKit [139] to extract features for atoms (nodes) and bonds (edges).

The mean absolute error (MAE) and root mean square error (RMSE) are reported in Table 3.3. We found utilizing graph information can accurately predict PCE values. Also, our proposed two methods are working equally well. Although WL tree kernel with degree 6 is also working well, it requires $10,000$ times more parameters than `structure2vec` and runs 2 times slower. The preprocessing needed for WL tree kernel also makes it difficult to use in large datasets.

To understand the effect of the inference embedding in the proposed algorithm framework, we further compare our methods with different number of fixed point iterations in Figure 3.4. It can see that, higher number of fixed point iterations will lead to faster convergence, though the number of parameters of the model in different settings are the same.

Table 3.3: Test prediction performance on CEP dataset. WL lv-k stands for Weisfeiler-lehman with degree k.

	test MAE	test RMSE	# params
Mean Predictor	1.9864	2.4062	1
WL lv-3	0.1431	0.2040	1.6m
WL lv-6	0.0962	0.1367	1378m
DE-MF	0.0914	0.1250	0.1m
DE-LBP	**0.0850**	**0.1174**	0.1m

The mean field embedding will get much worse result if only one iteration is executed. Compare to the loopy BP case with same setting, the latter one will always have one more round message passing since we need to aggregate the messages from edge to node in the last step. And also, from the quality of prediction we find that, though making slightly higher prediction error for molecules with high PCE values due to insufficient data, our algorithms are not overfitting the 'easy' (*i.e.*, the most popular) range of PCE values.

3.7 Summary

We propose, `structure2vec`, an effective and scalable approach for structured data representation based on the idea of embedding latent variable models into feature spaces, and learning such feature spaces using discriminative information. Interestingly, our method extracts features by performing a sequence of function mappings in a way similar to graphical model inference procedures, such as mean field and belief propagation. Our method provides a nice example for the general strategy of combining the strength of existing algorithms for graphical models with the deep learning approach, which we believe will become common in many other learning tasks.

However, the current method is not scalable for large graphs, as the runtime for each update is $O(L(V + E))$ with a L-layer `structure2vec` for graph of V nodes and E. In the next chapter, we will discuss how to leverage fixed point algorithms to scale up the embedding method.

CHAPTER 4

STOCHASTIC LARGE SCALE GRAPH EMBEDDING

Many graph analytics problems can be solved via iterative algorithms where the solutions are often characterized by a set of steady-state conditions. Different algorithms respect to different set of fixed point constraints, so instead of using these traditional algorithms, can we learn an algorithm which can obtain the same steady-state solutions automatically from examples, in an effective and scalable way? How to represent the meta learner for such algorithm and how to carry out the learning? In this chapter, we propose an embedding representation for iterative algorithms over graphs, and design a learning method which alternates between updating the embeddings and projecting them onto the steady-state constraints. We demonstrate the effectiveness of our framework using a few commonly used graph algorithms, and show that in some cases, the learned algorithm can handle graphs with more than 100,000,000 nodes in a single machine.

4.1 Introduction

Graphs and networks arise in various real-world applications and machine learning problems, such as social network analysis [99] and molecule screening [97, 71, 140]. Many graph analytics problems can be solved via iterative algorithms according to the graph structure, and the solutions of the algorithms are often characterized by a set of steady-state conditions. For instance, the PageRank [171] score of a node in a graph can be computed iteratively by averaging the scores of its neighbors, until the node score and this neighbor averaging are approximately equal. Mean field inference for the posterior distribution of a variable in a graphical model can be updated iteratively by aggregating the messages from its neighbors until the posterior is approximately equal to the results of the aggregation operator. More generally, the intermediate representation h_v for each node v in the node

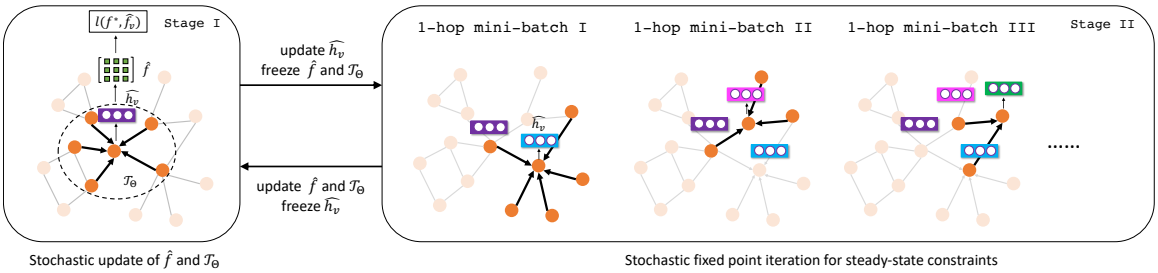


Figure 4.1: Overview of proposed graph steady-state learning algorithm. In stage I, we update the classifier $\widehat{f}_v$ and steady-state operator $\mathcal{T}_\Theta$ with 1-hop neighborhood of stochastic samples; in stage II, the embeddings $\widehat{h}_v$ are updated by stochastic fixed point iterations.

set $\mathcal{V}$ is updated iteratively according to an operator $\mathcal{T}$ as

$$\begin{aligned} h_v^{(t+1)} &\leftarrow \mathcal{T}\left(\{h_u^{(t)}\}_{u\in\mathcal{N}(v)}\right), \quad \forall t \geqslant 1, \quad \text{and} \\ h_v^{(0)} &\leftarrow \text{constant}, \quad \forall v \in \mathcal{V} \end{aligned} \tag{4.1}$$

until the steady-state conditions are met

$$h_v^* = \mathcal{T}\left(\{h_u^*\}_{u\in\mathcal{N}(v)}\right), \quad \forall v \in \mathcal{V}. \tag{4.2}$$

Variants of graph neural network (GNN) [193], like GCN [130], neural message passing network [83], GATs [229] *etc.*, perform fixed T rounds of updates to Eq (4.1) without respecting the steady state. Thus for learning algorithms like PageRank or mean field inference, a large T is required. In such case, both the computational cost and gradient updates will become problematic. Also note that due to the batch-update nature of GNN family models, multiple rounds of update over all nodes are needed. These two limitations make them not scalable and effective enough, regarding the computational cost and convergence.

In this chapter, instead of designing algorithms for each individual graph problem, we take a different perspective, and ask the question: *Can we design a learning framework for a diverse range of graph problems that learns the algorithm over large graphs achieving the steady-state solutions efficiently and effectively?* Furthermore, how to represent the

meta learner for such algorithm and how to carry out the learning of these algorithms? In this chapter we propose a stochastic learning framework of algorithm design based on the idea of embedding the intermediate representation of an iterative algorithm over graphs into vector spaces, and then learn such algorithms using example outputs from the desired algorithms to be learned.

More specifically, in our framework, each node in the graph will maintain an embedding vector, and these embedding vectors will be updated using a parameterized operator $\mathcal{T}_\theta$ where the parameters θ will be learned. Furthermore, following each embedding update step, the embedding will also be projected towards the steady state constraint space, gradually enforcing the steady-state conditions. As illustrated in Figure 4.1, both of the two steps are stochastic, which only requires 1-hop neighborhood for the update. We argue that such 1-hop stochasticity is key to the efficiency and effectiveness. Most of the GNN variants (e.g., [145]) need $O(T(|\mathcal{V}| + |\mathcal{E}|))$ computational cost and memory consumption per each round of parameter update. For large graphs, this would be quite expensive. [99] attempts the mini-batch update using T-hops neighborhood of sampled mini-batch of nodes. However, the neighborhood size grows exponentially with respect to T. As in the idea of six degrees of separation, $T = 6$ would already include all the nodes in the social network.

We note that this new algorithm is significantly different from the traditional graph embedding settings where the goal is to learn representations (or features) for nodes in a graph for classification. In contrast, our goal is to efficiently learn an *algorithm* which can run in a large graph and can respect specific condition with physical meaning. The successive stochastic projection of the embeddings onto the steady-state condition, which is not present in previous graph embedding methods, is a crucial step in our algorithm, and creates an important inductive bias which allows us to generalize the learned steady-state algorithm output to the entire network and even to a different network.

We showed that our framework can be adapted to learn the steady-state of a few commonly used graph algorithms, namely the detection of connected components, PageRank

scores, mean field inference, and node labeling problem over graphs. We conducted systematical comparison between the learned algorithms and several existing algorithms to demonstrate the benefits in terms of both effectiveness and scalability on both randomly generated graphs and real-world graphs. In particular, in the PageRank problem, the learned algorithm can easily handle graphs with more than 100,000,000 nodes in a single machine.

4.2 Iterative Algorithms over Graphs

Many iterative algorithms over graphs can be formulated into the form of Eq (4.1) and the solutions satisfy a requirement of the form of Eq (4.2). More specifically, for a graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with node set $\mathcal{V}$ and edge set $\mathcal{E}$, the iterative algorithm framework can be instantiated as follows

- **Graph component detection problem.** We want to find all nodes within the same connected component as source node $s \in \mathcal{V}$. This task can be solved by iteratively propagating the label at node s to other nodes

$$y_v^{(t+1)} = \max_{u \in \mathcal{N}(v)} y_u^{(t)}, \;\; y_s^{(0)} = 1, \; y_v^{(0)} = 0, \; \forall v \in \mathcal{V}$$

  where $\mathcal{N}(v)$ denotes the set of neighbors of v. At algorithm step $t = 0$, the label $y_s^{(0)}$ at node s are set to 1 (infected) and 0 for all other nodes. The steady state is achieved when nodes in the same connected component as s are infected. That is $y_v^* = \max_{u \in \mathcal{N}(v)} y_u^*$.

- **PageRank scores for node importance.** We want to estimate the importance of each node in a graph. The scores can be initialized to 0 ($r_v^{(0)} \leftarrow 0, \forall v \in \mathcal{V}$) and updated iteratively as

$$r_v^{(t+1)} \leftarrow \frac{(1-\lambda)}{|\mathcal{V}|} + \frac{\lambda}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} r_u^{(t)}, \;\; \forall v \in \mathcal{V}.$$

The steady-state scores r_v^* will satisfy the relation $r_v^* = \frac{(1-\lambda)}{|\mathcal{V}|} + \frac{\lambda}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} r_u^*$.

- **Mean field inference in graphical model.** We want to approximate the marginal distributions of a set of variables x_v in a graph model defined on $\mathcal{G}$. That is $p(\{x_v\}_{v\in\mathcal{V}}) \propto \prod_{v\in\mathcal{V}} \phi(x_v) \prod_{(u,v)\in\mathcal{E}} \phi(x_u, x_v)$ where $\phi(x_v)$ and $\phi(x_u, x_v)$ are the node and edge potential respectively. The marginal approximation $q(x_v)$ can be obtained in an iterative fashion by the following mean field update

$$q^{(t+1)}(x_v) \leftarrow \phi(x_v) \prod_{u\in\mathcal{N}(v)} \exp\left(\int_u q^{(t)}(x_u) \log \phi(x_u, x_v) \mathrm{d}u\right),$$

  with the steady-state $q^*(x_v) = \phi(x_v) \prod_{u\in\mathcal{N}(v)} \exp\left(\int_u q^*(x_u) \log \phi(x_u, x_v) \mathrm{d}u\right)$.

- **Compute long range graph convolution features.** We want to extract long range features from graph and use that figure to capture the relation between graph topology and external labels. One possible parametrization of graph convolution features h_v can be updated from zeros initialization as

$$h_v^{(t+1)} \leftarrow \sigma\left(W_1 x_v + W_2 \sum_{u\in\mathcal{N}(v)} h_u^{(t)}\right)$$

  where σ is a nonlinear elementwise operation, and W_1, W_2 are the parameters of the operator. The steady state is characterized as $h_v^* \leftarrow \sigma\left(W_1 x_v + W_2 \sum_{u\in\mathcal{N}(v)} h_u^*\right)$. Then the labeling function $f(h_v^*)$ for each node is determined by the feature h_v^*.

Typically, to learn these iterative algorithms with GNN family models, we need to run many iterations in order for them to converge to the steady-state solutions. Especially when the graph scale gets large, a large number of iterations are needed, making the GNNs very computationally intensive and slow. In the following, we will formulate a generic learning problem for designing a faster algorithms for these scenarios.

4.3 The Algorithm Learning Problem

In this section we propose a framework of algorithm design based on the idea of embedding the intermediate representation of an iterative algorithm over graphs into vector spaces, and then learn such algorithms using example outputs from the desired algorithms to be learned.

More specially, we assume that we have collected the output of an iterative algorithm $\mathcal{T}$ over a single large graph[1]. The training dataset consists of the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the output of the algorithm for a subset of nodes, $\mathcal{V}^{(y)} \subseteq \mathcal{V}$ from the graph:

$$\mathcal{D} = \left\{ f_v^* := f(h_v^*) \,|h_v^* = \mathcal{T}\left[\{h_u^*\}_{u \in \mathcal{N}(v)}\right],\ v \in \mathcal{V}^{(y)} \right\}. \tag{4.3}$$

In the dataset, h_v^* is the quantity in the algorithm which satisfies the steady-state conditions, and $f(\cdot)$ is an additional labeling function which takes the steady-state quantity and produces the final label for each node. In the case where h_v^* is the output of an algorithm, we can think of $f(\cdot)$ is the identity function.

Given the above dataset $\mathcal{D}$ from previous run of the algorithm, the goal is to learn a parameterized algorithm $\mathcal{A}_\Theta$ such that the output of the algorithm can mimic the output of the original algorithm $\mathcal{T}$. That is the learned algorithm $\mathcal{A}_\Theta$ produces $\{\ \widehat{f_v}\ \}_{v \in \mathcal{V}^{(y)}} = \mathcal{A}_\Theta[\mathcal{G}]$, which are close to f_v^* according to some loss function $\ell(f_v^*, \widehat{f_v})$.

Overall, the algorithm learning problem for $\mathcal{A}_\Theta$ can be formulated into the following optimization problem

$$\begin{aligned} \min_{\Theta} \quad & \sum_{v \in \mathcal{V}^{(y)}} \ell(f_v^*, \widehat{f_v}) \\ \text{s.t.} \quad & \{\ \widehat{f_v}\ \}_{v \in \mathcal{V}^{(y)}} = \mathcal{A}_\Theta[\mathcal{G}] \end{aligned} \tag{4.4}$$

In the above general statement of the learning problem, we have not specified the actual

[1] Our method can also be used for the cases where data are collected from multiple graphs. In this case, we can view multiple graphs as a single big graph with a collection of connected components.

form of the algorithm and the parametrization of the algorithm step. In the following section we will explain our design of fast iterative algorithm which can be learned.

The design goal of our model will focus on two key aspects: respect steady-state conditions and learn fast. Thus the core of our model is a steady-state operator $\mathcal{T}_\Theta$ between vector embedding representation of nodes, and a link function mapping the embedding to the algorithm output. Furthermore, the embeddings are obtained by solving the steady-state operator stochastically, making it very efficient for large scale graph problems.

4.3.1 Steady-state operator and linking function

We will associate each node in the graph with an unknown vector embedding representation $\widehat{h}_v \in \mathbb{R}^d$, and the core of our algorithm is a parameterized operator, $\mathcal{T}_\Theta$, for enforcing steady-state relations between these embeddings. Given a link function $\widehat{f}(h_v)$, our model makes predictions on the algorithm outputs by the following operations

$$\begin{aligned} \text{output}: \ & \left\{ \widehat{f}_v := \widehat{f}(\widehat{h}_v) \right\}_{v \in \mathcal{V}} \\ \text{s.t.} \ & \widehat{h}_v = \mathcal{T}_\Theta \left[\{\widehat{h}_u\}_{u \in \mathcal{N}(v)} \right] \end{aligned} \tag{4.5}$$

In our model, the steady-state operator $\mathcal{T}_\Theta$ and the linking function $\widehat{f}$ is not fixed before hand, and their parameters will be learned from dataset $\mathcal{D}$ in Eq (4.3). Furthermore, the vector embeddings $\widehat{h}_v$ need to be found from Eq (4.5), after which the embeddings are used for making predictions about the algorithm outputs via $\widehat{f}$. Thus, we need an algorithm for finding the (approximate) steady-state of Eq (4.5).

4.3.2 Finding steady-state

Here we use an iterative algorithm to find the steady-state of Eq (4.5). The algorithm will execute in a similar fashion as randomized Gauss-Seidel method which updates one unknown variable at the time according to the steady-state equation. Adapting the scheme

to our case, we will start all $\{\widehat{h}_v\}_{v\in\mathcal{V}}$ from some constant, and then update the embedding one at at time. That is

$$\widehat{h}_v \leftarrow \text{constant for all } v \in \mathcal{V}$$
$$\text{for } v \text{ sampled from } \mathcal{V}: \qquad \widehat{h}_v \leftarrow \mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v)}\right]$$

We note that in this randomized scheme, the embeddings $\{\widehat{h}_v\}_{v\in\mathcal{V}}$ are updated in an asynchronous fashion. Furthermore, each time the update is also carried out only one hop for the sampled node v. This makes it very efficient compared to synchronous update over the entire graph for T hops. For comparison, the synchronous update will amount to a computational complexity of $O(T(|\mathcal{V}| + |\mathcal{E}|))$ which quickly becomes prohibitive for large graphs. Instead, our steady-state finding algorithm is carried out using mini-batches.

4.3.3 Specific parameterization for $\mathcal{T}_\Theta$ and g

The operator $\mathcal{T}_\Theta$ and link function g can come from general nonlinear function class. The operator $\mathcal{T}_\Theta$ enforces the steady-state condition of node embeddings based on 1-hop local neighborhood information. Due to the variety of graph structures, this function should be able to handle different number of inputs (*i.e.*, different number of neighbor nodes) and be invariant to the ordering of these neighbors. In our work, we use the following parameterization:

$$\mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v)}\right] = W_1\sigma\left(W_2\Big[x_v, \sum_{u\in\mathcal{N}(v)}[\widehat{h}_u, x_u]\Big]\right) \tag{4.6}$$

where $\sigma(\cdot)$ is element-wise activation function, such as commonly used Sigmoid or ReLU. W_1 and W_2 are the weight matrices. x_v is the optional feature representation of nodes, such as observations in Markov Random Field (MRF). In general, a two-layer neural network formulation as above would be enough for most cases. But one can also use problem-specific parameterization for better performance.

For prediction function g, it takes the node embeddings as inputs, and predicts the

corresponding algorithm outputs. We also adopt a two-layer neural network, *i.e.*,

$$g(\widehat{h}_v) = \sigma\big(V_2^\top \text{ReLU}(V_1^\top \widehat{h}_v)\big), \tag{4.7}$$

where V_1, V_2 are parameters of $g(\cdot)$. $\sigma(\cdot)$ is a task-specific activation function. For linear regression this is the identity function $\sigma(x) = x$. For multi-class classification problem, $\sigma(\cdot)$ is softmax which would output a probabilistic simplex.

4.3.4 The optimization problem

Thus the overall optimization problem for learning our model can be formulated as

$$\begin{aligned} \min_{\{W_i, V_i\}_{i=1}^2} \mathcal{L}\left(\{W_i, V_i\}_{i=1}^2\right) &:= \frac{1}{|\mathcal{V}^y|} \sum_{v \in \mathcal{V}^{(y)}} \ell(f_v^*, g(\widehat{h}_v)) \\ \text{s.t. } \widehat{h}_v &= \mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u \in \mathcal{N}(v)}\right], \forall v \in \mathcal{V}. \end{aligned} \tag{4.8}$$

In the next section, we will introduce an alternating algorithm to solve the above optimization problem. The algorithm will alternate between using most current model to find the embeddings and make prediction, and using the gradient of the loss with respect to $\{W_1, W_2, V_1, V_2\}$ for update these parameters.

4.4 Learning Algorithm

It should be emphasized that directly applying the vanilla stochastic gradient descent requires visiting *all* the nodes in the graph many times due to the constraints in Eq. (4.8), making the reduction of the cost via stochastic gradient computation in vain. As we discussed in Section 4.3.2, this step is actually the computation bottleneck. In this section, we present a scalable algorithm which exploits the stochasticity in both equilibrium constraints and the objective in Eq. (4.8) to learn the parameters. Then, we provide the analysis of the computational and memory complexity in detail to show how our proposed approach could

save the computation in Section 4.4.2.

4.4.1 Stochastic Fixed-Point Gradient Descent

In fact, the optimization Eq. (4.8) can be understood as improving the policy which minimizing the cost that is proportional to f^*. The fix-point equation characterizes the dynamic programming whose solution is steady state $\widehat{h}_v$ for each node. Comparing to the reinforcement learning (RL), it plays a similar role as "value function". With these estimations of the steady states, we minimize the cost by updating the parameters in $\mathcal{T}_\Theta$ and g, which can be understood as a similar role as "policy" in RL. Based on such understanding, we design our algorithm inspired by the policy iteration in reinforcement learning [220]. Furthermore, to reduce the complexity in the first stage for estimating, we introduce an extra randomness over the constraints and solve it approximately through *stochastic fixed point iteration*.

Stochastic gradient descent for "policy" improvement. Specifically, at k-th round in the stochastic optimization, once we have $\left\{\widehat{h}_v^k\right\}_{v\in\mathcal{V}}$ satisfying the steady-state equation, *i.e.*, $\widehat{h}_v^k = \mathcal{T}_\Theta\left[\{\widehat{h}_u^k\}_{u\in\mathcal{N}(v)}\right], \forall v \in \mathcal{V}$, we have the gradient estimators as

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_i} &= \widehat{\mathbb{E}}\left[\frac{\partial \ell\left(f_v^*, g\left(\widehat{h}_v^k\right)\right)}{\partial g\left(\widehat{h}_v^k\right)}\frac{\partial g\left(\widehat{h}_v^k\right)}{\partial W_i}\right], \\
\frac{\partial \mathcal{L}}{\partial V_i} &= \widehat{\mathbb{E}}\left[\frac{\partial \ell\left(f_v^*, g\left(\widehat{h}_v^k\right)\right)}{\partial \widehat{h}_v^k}\frac{\partial \mathcal{T}_\Theta\left[\{\widehat{h^k}_u\}_{u\in\mathcal{N}(v)}\right]}{\partial V_i}\right],
\end{aligned}$$

where the expectation $\widehat{\mathbb{E}}[\cdot]$ is taken w.r.t. uniform distribution over labeled nodes $\mathcal{V}^{(y)}$. With such treatment, we can update the parameters, *i.e.*, $\{W_1, W_2, V_1, V_2\}$, as vanilla stochastic gradient descent.

Stochastic fixed-point iteration for "value" estimation. However, it is prohibitive to solve the steady-state equation exactly in large-scale graph with millions of vertices since it requires visiting all the nodes in the graph. Therefore, we introduce the extra randomness on the constraints for sampling the constraints to tackle the groups of equations approxi-

Algorithm 4 Learning with Stochastic Fixed Point Iteration

Initialize $W_1, W_2, V_1, V_2, \{\widehat{h}_v\}_{v\in\mathcal{V}}$ randomly
for $k = 1$ **to** K **do**
    for $t_h = 1$ **to** n_h **do**
        Sample $\widetilde{\mathcal{V}} = \{v_1, v_2, \ldots, v_N\} \in \mathcal{V}$
        Use Eq. (4.9) to update embedding $\widehat{h}_{v_i}, \forall v_i \in \widetilde{\mathcal{V}}$
    end for
    for $t_f = 1$ **to** n_f **do**
        Sample $\widetilde{\mathcal{V}}^{(y)} = \{v_1, v_2, \ldots, v_M\} \in \mathcal{V}^{(y)}$
        $\{W_i \leftarrow W_i - \eta\frac{\partial\mathcal{L}}{\partial W_i}\}_{i=1}^2$, $\{V_i \leftarrow V_i - \eta\frac{\partial\mathcal{L}}{\partial V_i}\}_{i=1}^2$
    end for
end for

mately. This technique is very effective in dealing with infinite constraints in approximately solving MDP [60, 61].

Specifically, in k-th step, we first sample a set of nodes $\widetilde{\mathcal{V}} = \{v_1, v_2, \ldots, v_N\} \in \mathcal{V}$ from the entire node set rather of the labeled set. For stability, we update the new embedding of v_i by moving average in following form:

$$\widehat{h}_{v_i} \leftarrow (1-\alpha)\widehat{h}_{v_i} + \alpha\mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v_i)}\right], \forall v_i \in \widetilde{\mathcal{V}}. \tag{4.9}$$

The overall algorithm is summarized in Algorithm 4. The whole iterative process will run K steps or untill convergence. During each macro iteration, the two stages can also have multiple inner loops. Specifically, let n_f be the number of inner loops for "policy" improvement, and n_h be the number of inner loops in "value" estimation.During the experiment we found that, having more fixed point iterations, *i.e.*, $n_h > n_f$ helps the model converge faster and achieve better generalization.

We name our algorithm Stochastic Steady-state Embedding (SSE), due to its stochasticity nature and steady-state enforcement.

4.4.2 Complexity analysis

In this section, we briefly analyze the computation and memory complexity of Algorithm 4.

In "policy" improvement stage, assume the labeled set $\mathcal{V}^{(y)}$ is an unbiased sample from $\mathcal{V}$, then the computational cost is $\Theta(M\frac{|\mathcal{E}|}{|\mathcal{V}|})$, since we only need 1-hop nodes to update. Here we use the average node degree in graph to calculate the expected number of edges in each mini-batch. Similarly, in "value" estimation stage, we have $\Theta(N\frac{|\mathcal{E}|}{|\mathcal{V}|})$. So in summary, the computational cost in each iteration is just proportional to the number of edges in each mini-batch.

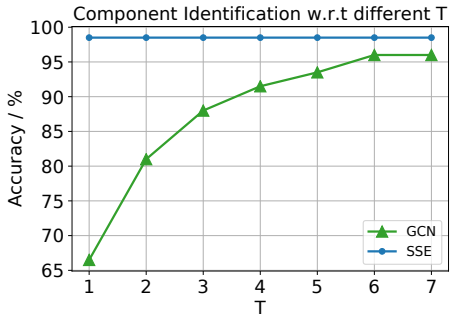
The memory cost of our algorithm is also smaller compared to the existing graph neural networks. Regardless of necessary memory held by parameters $W_{1,2}, V_{1,2}$ and node/edge features, the dominating part is the persistent node embedding matrix $\{\widehat{h}_v\}_{v\in\mathcal{V}}$ which takes $O(|\mathcal{V}|)$ space. This is also much cheaper than most GNN-family models which take $O(T|\mathcal{V}|)$ space, due to the requirement of storing intermediate embeddings for back-propagation use.

4.5 Experiments

In this section, we experimentally demonstrate the effectiveness and efficiency of learning various graph algorithms. We compare our proposed algorithm with some of the GNN variants who have the fixed finite number of propagations T, using experiments with both transductive and inductive settings. In transductive setting, we compare with GCN [130], a localized first-order approximation of spectral graph convolutions and structure2vec [55] which mimics the graphical model inference algorithms to obtain feature representation. The number of propagation steps is tuned in $T \in \{1, \ldots, 7\}$ for them. In inductive setting, we compare with GraphSage [99] and its variants. For our proposed algorithm, We tune the number of inner loops for SGD and fixed point iterations $n_f, n_h \in \{1, 5, 8\}$, to balance the parameter learning and fixed point constraint satisfaction.

We demonstrate the effectiveness of the proposed algorithm in capturing steady-state information with learning graph algorithms, *i.e.*, the graph connectivity detection, PageRank, and Mean Field Inference on graphical model, where the global-range steady information

(a) Graph consists of two disjoint chains.



(b) Accuracy *w.r.t.* different T.

Figure 4.2: Graph connectivity experiment.

is the key for success, in Section 4.5.1, 4.5.2 and 4.5.3, respectively. We also show the comparison on benchmark datasets in Section 4.5.4, where we can achieve comparable or better accuracy. Finally we show our advantage in terms of scalability in Section 4.5.5.

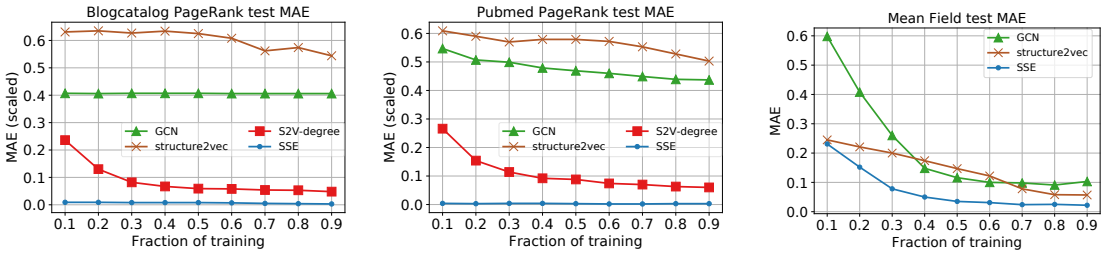
Table 4.1: Multi-class node classification Dataset statistics as reported in [130].

Dataset	# Nodes	# Edges	# Classes	# Features	Label rate
Citeseer	3,327	4,732	6	3,703	3.6%
Cora	2,708	5,429	7	1,433	5.2%
Pubmed	19,717	44,338	3	500	0.3%

Table 4.2: Multi-label node classification Dataset statistics

Dataset	# Nodes	# Edges	# Labels	Label type	Graph type
BlogCatalog	10,312	333,983	39	membership	social network
PPI(transductive)	3,890	76,584	50	Bio-states	protein
Wikipedia	4,777	184,812	40	POS-tag	word-net
Amazon	334,863	925,872	58	product type	co-purchasing
PPI(inductive)	56,944	818,716	121	Bio-states	protein

The real-world dataset used are shown in Table 4.1 and Table 4.2. The multi-class classification datasets are from [130], where the multi-label classification datasets are from [90], [99] and SNAP website. Datasets in Table 4.1 and also the inductive PPI dataset have extra node features. When available, we use the same train/valid/test split as in original paper.



(a) PageRank on BlogCatalog (b) PageRank on Pubmed (c) Mean-Field Inference for MRF

Figure 4.3: Algorithm learning for PageRank and Mean Field Inference. Error is measured using Mean Absolute Error (MAE).

4.5.1 Algorithm-learning: connectivity

The graph we constructed contains 2 disjoint chains. Each chain is a connected component which contains 1,000 nodes and 999 edges. Figure 4.2a illustrates the graph we created. The algorithm needs to know the multi-hop structure, in order to identify the component ID for a certain node. In this transductive setting, we use 10% nodes with labels for training, and the rest for testing. With proper parameter tuning, the GCN and structure2vec can achieve 96% accuracy in distinguishing two components, while our SSE gets 99%.

In Figure 4.2b, we vary the T of GCN, and report its test performance. Since our proposed algorithm doesn't have the dependency over T, we simply include it as a reference. We can see as T gets larger, the GCN model converges to better solution by taking longer range of information, while the computational cost increases linearly with T. Also through this experiment we find it is not only computationally more efficient, but also experimentally more effective in learning the steady-states.

4.5.2 Algorithm Learning: PageRank

In this task, we learn to predict the PageRank scores for each node in the network graph. In our experiment we use the default value (which is 0.85) for the damping factor.

Real-world graphs: We take the Blogcatalog and Pubmed graphs for evaluation. For each dataset, we first run the PageRank algorithm using networkx [98]. Since the raw PageRank

scores are normalized to a probabilistic simplex, we rescale it by multiplying the total number of nodes. This avoids some precision issue of the float numbers. In transductive setting, we reserve 10% nodes for held-out evaluation, and vary the training set size from 10% to 90% of the total nodes. We also modify the vanilla structure2vec model to use degree-weighted message aggregation, denoted as S2V-degree, for better performance in PageRank prediction task.

The quantitative results are shown in Figure 4.3a and 4.3b, respectively. We can see from the figure that, our proposed algorithm can achieve almost perfect fitting results on all the two datasets, even with only 10% nodes for training. However, although we've shown that with larger T the GCN can match our performance in Section 4.5.1, it is not effective in current experiment. Simply making T larger will cause problem for both gradient propagation and memory consumption, and thus it is not effective. The modified baseline S2V-degree performs the second best, so we compare with it in detail on Barabasi-Albert random graphs in next part.

Barabasi-Albert random graphs: To evaluate how the performance varies as graph size grows, we further carry out experiments on Barabasi-Albert (BA) graphs. We vary the number of nodes $n \in \{1k, 10k, 100k, 1m, 10m\}$, and use two different parameters $m = 1$ and $m = 4$ for BA model. It is known that when $m = 1$ the graph has diameter of $O(\log n)$ and for $m \geq 2$ it is $O(\log n / \log \log n)$ [27]. Thus for $m = 1$, it is more challenging since the number of hops of information need is larger.

In transductive setting, we split the nodes equally into training and test set; in inductive setting, the training is performed in a single graph, while the algorithm is asked to generalize to new graphs from the same distribution. For S2V-degree we set $T = 5$ due to the consideration of feasibility. The transductive and inductive results are shown in Table 4.3 and 4.4, respectively. As is expected, the MAE in $m = 4$ setting is lower than that in $m = 1$ setting. Our proposed algorithm achieves almost perfect MAE and increases slightly when the prediction task becomes more and more challenging as the size of graph increasing to

Table 4.3: Transductive learning of PageRank on Barabasi-Albert graphs with different sizes and hyperparameters ($m = 1, 4$). We report MAE on 50% held-out nodes.

	# nodes	1k	10k	100k	1m	10m
m=1	S2V-degree	0.0652	0.0843	0.1444	0.4012	0.4954
	SSE	0.0041	0.0054	0.0075	0.0088	0.0162

	# nodes	1k	10k	100k	1m	10m
m=4	S2V-degree	0.0138	0.0165	0.0347	0.0944	0.1223
	SSE	0.0043	0.0051	0.0056	0.0065	0.0083

Table 4.4: Inductive learning of PageRank on Barabasi-Albert graphs, trained on graph with same hyper-parameters.

	# nodes	1k	10k	100k	1m	10m
m=1	S2V-degree	0.0783	0.0956	0.1931	0.4532	0.5254
	SSE	0.0062	0.0074	0.0073	0.0097	0.0202

	# nodes	1k	10k	100k	1m	10m
m=4	S2V-degree	0.0172	0.0193	0.0394	0.1243	0.1527
	SSE	0.0057	0.0063	0.0066	0.0079	0.0101

10m nodes. In comparison, the performance of S2V-degree is significantly worse, especially when graph size grows. This is because $T = 5$ propagations cannot capture enough long range information. We emphasize that for large graphs with 10m nodes, it is also hard for batch algorithm like S2V-degree to converge and generalize well.

4.5.3 Algorithm Learning: mean-field inference

To further evaluate the ability of our proposed algorithm in capturing the steady-state information, we design a task to fit the posteriors from the mean-field (MF) inference algorithm. Here we define a lattice graph over a 128×128 grid. Specifically, we focus on the pair-wise Markov Random Field graphical model:

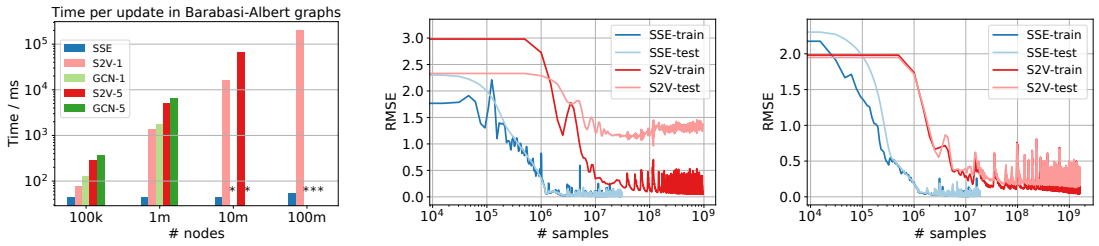
$$P(\{\mathbf{H}_v\}, \{\mathbf{x}_v\}) \propto \prod_{v \in \mathcal{V}} \Phi(\mathbf{H}_v, \mathbf{x}_v) \prod_{(u,v) \in \mathcal{E}} \Psi(\mathbf{H}_u, \mathbf{H}_v) \tag{4.10}$$

where $\mathbf{x}_v$ is the observation and $\mathbf{H}_v$ is the latent variable. The mean-field score for each $\mathbf{H}_v$ is a vector calculated using the UGM toolset [2]. The task is to learn the mean-field score $q(\mathbf{H}_v)$ for each node over a 128×128 lattice with $\mathbf{x}_v$ set to be binary with a Gaussian perturbation. The posterior in this case can be understood as steady-state that is expressed by nonlinear fixed point equation. We test the learned mean-field scores on the 10% of the vertices and vary the size of training set sampled from the remaining vertices.

[2] https://www.cs.ubc.ca/ schmidtm/Software/UGM.html

Table 4.5: Multi-label classification in Amazon product dataset. We report both Micro-F1 and Macro-F1 on held-out test set.

Amazon	Micro-F1/%									Macro-F1/%								
Methods	1%	2%	3%	4%	5%	6%	7%	8%	9%	1%	2%	3%	4%	5%	6%	7%	8%	9%
structure2vec	70.27	74.54	77.18	79.95	80.97	81.58	82.71	83.27	83.55	66.62	70.07	74.74	76.43	77.62	78.65	79.92	80.13	80.11
GCN	70.39	73.58	77.61	80.34	82.03	83.23	84.25	85.1	85.68	66.16	71.01	74.56	77.11	78.97	80.5	81.36	82.15	82.75
SSE	**78.36**	**81.06**	**82.61**	**83.79**	**84.59**	**85.08**	**85.68**	**86.57**	**87.13**	**75.07**	**77.67**	**79.03**	**79.86**	**81.14**	**81.59**	**82.39**	**83.13**	**84.03**



(a) Wall-clock time per round of update. The (*) in the figure denotes the out-of-memory error.

(b) Convergence on BA graphs with number of nodes=1,000,000 and m=1.

(c) Convergence on BA graphs with number of nodes=1,000,000 and m=4.

Figure 4.4: Results on scalability experiments. We compare both the time needed per update, as well as number of samples required for convergence in PageRank experiments with large Barabasi-Albert random graphs.

From Figure 4.3c we can see, our proposed algorithm still works best regarding the MAE metric, and can achieve better results with fewer labeled vertices. Here the fixed point equations are nonlinear, which is different from the PageRank experiment. The baseline algorithms can also achieve good performance with more supervision.

4.5.4 Application: node classification

Transductive setting:

To demonstrate the effectiveness of addressing steady-state information, we conduct experiments on a large graph dataset, namely the Amazon product co-purchasing network dataset [241][3]. Among the 75,149 product types, we select those with at least 5,000 products. This results in 58 labels finally.

From Table 4.5 we can see the SSE outperforms the baselines by a large margin. We also observed that in Amazon dataset, GNN-family models benefit more from more super-

[3] http://snap.stanford.edu/data/com-Amazon.html.

vision, due to the larger model capacity. Our proposed method achieves the best performance, regardless of the amount of supervision available. This suggests that our algorithm can effectively utilize the global-range information of graph structure.

To make the comparison comprehensive, we also conduct experiments on small benchmark datasets that are commonly used in the literature. We show that in the graphs where the diameter is small, existing algorithms can do pretty good, since local information is almost equivalent to global-range information. Nonetheless, our SSE can still achieve comparable performance in this scenario.

Table 4.6: Multi-label classification in small datasets. We report both Micro-F1 and Macro-F1 on held-out test set.

Blogcatalog	Micro-F1/%									Macro-F1/%								
Methods	10%	20%	30%	40%	50%	60%	70%	80%	90%	10%	20%	30%	40%	50%	60%	70%	80%	90%
structure2vec	**35.05**	**36.65**	**38.43**	**39.35**	**40.48**	**40.89**	**42.56**	**42.58**	**42.61**	19.78	22.39	**23**	**25.16**	**25.89**	**26.96**	**26.86**	**27.46**	**27.69**
GCN	36.80	38.42	39.47	40.88	40.88	41.69	42.06	42.43	42.50	19.31	20.96	20.43	22.3	21.86	22.14	23.06	23.2	23.43
SSE	33.90	36.42	36.80	37.39	37.91	37.92	38.58	39.10	40.28	**19.88**	**22.68**	22.88	23.89	23.89	24.08	24.38	25.12	24.99
PPI	Micro-F1/%									Macro-F1/%								
Methods	10%	20%	30%	40%	50%	60%	70%	80%	90%	10%	20%	30%	40%	50%	60%	70%	80%	90%
structure2vec	**19.86**	**23.19**	24.73	25.46	25.29	**27.79**	27.75	**28.32**	**28.99**	15.14	15.94	18.32	18.41	19.04	20.41	20.56	22.01	23.83
GCN	18.85	22.52	**25.40**	**26.36**	**26.52**	**27.80**	**27.96**	**28.28**	28.44	**16.03**	17.09	**19.01**	**20.45**	**21.01**	**21.62**	23.50	**23.29**	**24.13**
SSE	19.17	22.04	23.64	23.64	25.24	24.44	26.36	26.20	27.16	15.58	**17.79**	18.36	19.30	**20.99**	20.16	22.64	22.80	22.63
Wikipedia	Micro-F1/%									Macro-F1/%								
Methods	10%	20%	30%	40%	50%	60%	70%	80%	90%	10%	20%	30%	40%	50%	60%	70%	80%	90%
structure2vec	**47.94**	**50.24**	49.98	50.76	**52.45**	**53.54**	53.21	54.07	54.95	11.53	11.63	12.38	13.05	14.12	**16.65**	16.80	**17.37**	17.27
GCN	46.94	49.14	49.61	48.82	49.61	49.92	49.61	51.02	50.55	11.30	11.64	12.41	12.32	13.11	12.98	13.47	13.87	14.34
SSE	46.94	49.76	**51.33**	**51.44**	51.18	52.91	**54.32**	**54.33**	**55.26**	**13.63**	**13.70**	**16.00**	**16.26**	**16.33**	16.41	**17.00**	**17.33**	**17.42**

For multi-label classification, the evaluation metric we used here is Micro-F1 and Macro-F1 score. We tuned the hyperparameters for all the algorithms on 10% of training nodes, and then trained the model on full training set. The dimension of the embedding is set to 128. The results are shown in Table 4.6. We achieve the best results in Wikipedia, while getting comparable performances on the other two. In dataset like Blogcatalog, a small local neighborhood would be enough to infer the group membership of users in this friendship network, thus our approach would not benefit from taking global-range of information into account. However, in the Wikipedia dataset where we achieves the best Micro-F1 and Macro-F1 scores, it is important to know long range information to get a consensus among POS-tag labeling.

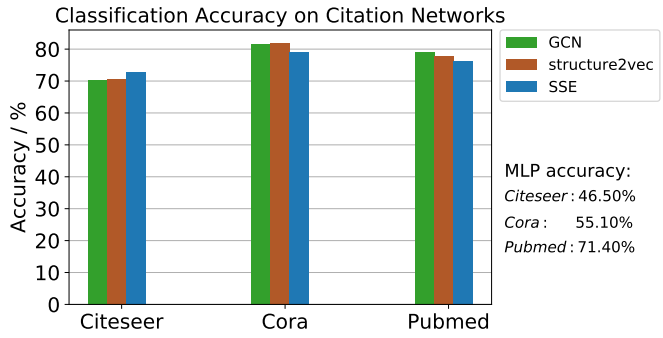


Figure 4.5: The document classification accuracy on benchmark citation networks.

Table 4.7: Inductive node classification using PPI dataset.

Method	Micro-F1
GraphSAGE-GCN	0.500
GraphSAGE-mean	0.598
GraphSAGE-LSTM	0.612
GraphSAGE-pool	0.600
SSE	**0.836**

For document classification, each node in the citation graph represents the corresponding document. Different from the above multi-label classification, here the documents have auxiliary bag-of-words features. Since the document classes are mutually exclusive, we train all the models with Cross Entropy loss. The edges (undirected) are formed by the citation relationship between articles. We use the same training/validation/test splits as in [130]. During training, only 20 instances per class are provided with corresponding labels. We report the test classification accuracy in Figure 4.5. When possible, we include the baselines' performances directly from previously published results [130]. From the figure we can see, the proposed SSE performs the best in Citeseer dataset, while being slightly worse than other GNN models in Cora and Pubmed dataset, respectively. We've also include the results that using the node feature with multi-layer perceptron (MLP). The MLP doesn't consider any graph structure into consideration, which serves a sanity check.

Inductive setting:

In this setting, we use the PPI dataset from GraphSage [99], which contains 56,944 nodes (for proteins) and 818,716 edges (for their interactions). It is a multi-label classification tasks, where each protein can have at most 121 labels. Each protein is associated with additional 50-dimensional features. We use the same train/valid/test split as in [99].

Table 4.7 shows the results. The GraphSage results are taken from the original paper,

since we are using the exactly same setting. We can see regarding the Micro-F1 metric, our proposed SSE achieves much better performance. We show that, since GraphSage is trained using mini-batch of nodes within T-hops, it is not effective enough to capture the steady-state information, which in this case seems essential.

4.5.5 Scalability

In this section, we demonstrate that the proposed algorithm is very efficient for large-scale graphs in terms of both convergence speed and execution time.

Time per update

All the algorithms are executed on a 16-core cluster with 256GB memory. We evaluate the wall-clock time cost per update. For baselines GCN and structure2vec, this corresponds to one feedforward and back propagation round with T-step embedding propagation on entire graph; for our method, this corresponds to $n_f + n_h$ mini-batch updates. Here we focus on models in GNN family. For GCN and structure2vec, we compare with $T = 1$ and $T = 5$; while in our method, $n_f = 1$ and $n_h = 5$.

The task we choose here is PageRank in Section 4.5.2. The graphs we evaluate on are generated using Barabasi-Albert model with $m = 4$ as its parameter. We vary the number of nodes in $\{100k, 1m, 10m, 100m\}$, and report the time in milliseconds in Figure 4.4a.

The results show our algorithm takes almost constant time for each update, due to its stochasticity nature. As graph size grows, the time cost for GCN and structure2vec grows linearly. For graph with $100m$ nodes, storing the intermediate updates and gradients for $T = 5$ in structure2vec is no longer feasible [4].

[4]Note that for open source implementation of GCN, the Tensorflow limits the # elements in sparse matrix. That's why it cannot work on graphs with $10m$ nodes.

Convergence

Here we compare the number of samples required for different algorithms to converge to a good solution. Figure 4.4b and 4.4c show the curves. We take the Barabasi-Albert graphs with 1,000,000 node and two different settings of $m = 1$ and $m = 4$, and fit with the PageRank scores on 50% nodes. We also visualize the test error convergence curve on the held-out 50% nodes. Both training and test curves report the RMSE (root mean square error), since we use this metric for optimization.

We compare with S2V-degree with $T = 5$, which achieves second best results in Section 4.5.2. For our algorithm, each round of updates requires $256 \times (n_f + n_h)$ samples. Here 256 is the mini-batch size we used, while S2V-degree needs the whole graph per update.

From the figures we can see our proposed algorithm converges much faster than the S2V, in terms of number of samples. The number of samples required by our algorithm is equivalent to only scanning through the entire training set for 4 or 5 passes. While for S2V-degree, it requires hundreds or thousands of passes to converge. Also note that, S2V-degree with $T = 5$ gets much worse test error in the case when $m = 1$, due to its limited ability for capturing steady-state information.

4.6 Summary

In this chapter, we presented SSE, an algorithm that can learn many steady-state algorithms over graphs. Different from graph neural network family models, SSE is trained stochastically which only requires 1-hop information, but can capture fixed point relationships efficiently and effectively. We demonstrate this in both synthetic and real-world benchmark datasets, with transductive and inductive experiments for learning various graph algorithms. The algorithm also scales well up to 100m nodes with much less training effort. Future work includes investigation in learning more complicated graph algorithms, as well as distributed training.

So far we have mostly focused on discriminative setting where the supervision is available. In the following chapters, we will present several unsupervised generative modeling methods which employs the algorithm structure as design bias.

CHAPTER 5

SEGMENTAL SEQUENCE GENERATIVE MODELING

Segmentation and labeling of high dimensional time series data has wide applications in behavior understanding and medical diagnosis. Due to the difficulty of obtaining a large amount the label information, realizing this objective in an unsupervised way is highly desirable. Hidden Semi-Markov Model (HSMM) is a classical tool for this problem. However, existing HSMM and its variants typically make strong generative assumptions on the observations within each segment, thus their ability to capture the nonlinear and complex dynamics within each segment is limited. To address this limitation, we propose to incorporate the Recurrent Neural Network (RNN) as the generative process of each segment, resulting the Recurrent HSMM (R-HSMM). To accelerate the inference while preserving accuracy, we designed a structure encoding function to mimic the exact inference. By generalizing the penalty method to distribution space, we are able to train the model and the encoding function simultaneously. We also demonstrate that the R-HSMM significantly outperforms the previous state-of-the-art on both the synthetic and real-world datasets.

5.1 Introduction

Segmentation and labeling of time series data is an important problem in machine learning and signal processing. Given a sequence of observations $\{x_1, x_2, \ldots, x_T\}$, we want to divide the T observations into several segments and label each segment simultaneously, where each segment consists of consecutive observations. The supervised sequence segmentation or labeling techniques have been well studied in recent decades [219, 134, 39]. However, for complicated signals, like human activity sensor data, accurately annotating the segmentation boundary or the activity type would be prohibitive. Therefore, it is urgent to develop unsupervised algorithms that can jointly learn segmentation and labeling

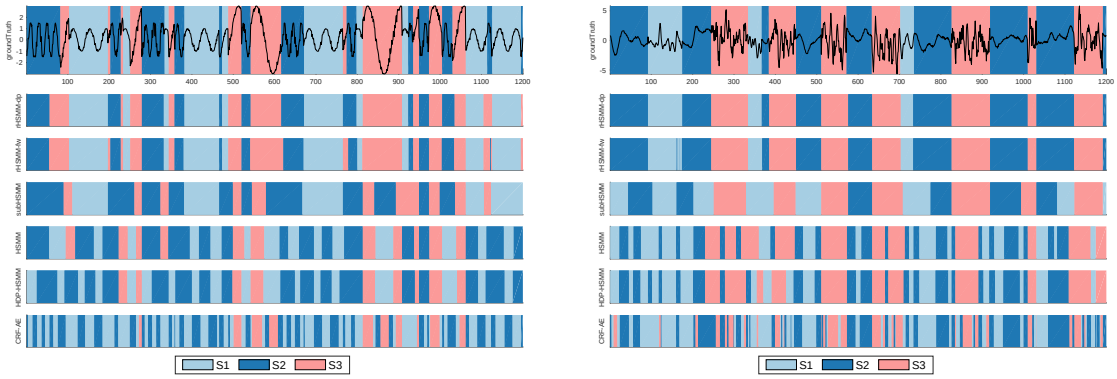
information directly from the data without supervisions. Figure 5.1 provides an illustration which we are focus on.

The Hidden Semi-Markov Model (HSMM) [164] is a powerful model for such task. It eliminates the implicit geometric duration distribution assumptions in HMM [247], thus allows the state to transit in a non-Markovian way. Most of the HSMM variants make strong parametric assumptions on the observation model [179, 117, 247]. This makes the learning and inference simple, but ignores the nonlinear and long-range dependency within a segment. Take the human activity signals as an example. The movements a person performs at a certain time step would rely heavily on the previous movements, like the interleaving actions of left hand and right hand in swimming, or more complicated dependency like shooting after jumping in playing basketball. Some models have been proposed to tackle this problem [81, 75, 149], but are limited in linear case.

Since people have justified RNN's ability in modeling nonlinear and complicated dependencies [219, 69], we introduce the recurrent neural emission model into HSMM for capturing various dependencies within each segment to address such issue. However, the flexibility of recurrent neural model comes with prices: it makes the exact Expectation-Maximization (EM) algorithm computationally too expensive.

To speed up the learning and inference, we exploit the variational encoder (VAE) framework [129]. Specifically, we propose to use bidirectional RNN (bi-RNN) encoder. Such architecture will mimic the forward-backward algorithm, and hence is expected to capture similar information as in exact posterior calculation.

It should be emphasized that due to the *discrete* nature of the latent variables in our model, the algorithm proposed in [129] and its extension on time-series models [77, 135] are not directly applicable. There are plenty of work proposed based on stochastic neuron [225, 21, 157, 182, 91, 42] to remedy such issue. However, none of these off-the-shelf methods are easy to achieve good performance according to our experiment: the hundreds or thousands layers of stochastic neuron (which is equal to the length of sequence), together



(a) Sine (b) Gaussian Process

Figure 5.1: Synthetic experiment results. Different background colors represent the segmentations with different labels. In the top row, the black curve shows the raw signal. (a) The Sine data set is generated by a HSMM with 3 hidden states, where each one has a corresponding sine function; (b) Similar to 5.1a, but the segments are generated from Gaussian processes with different kernel functions. The first two rows are our algorithms which almost exact locate every segment.

with the switching generative RNN, make the encoding function very sensitive, and thus, extremely difficult to train fully on unsupervised setting. We propose a solution, *stochastic distributional penalty method*, which introduces auxiliary distributions to separate the decoding R-HSMM and encoding bi-RNN in training procedure, and thus, reduces the learning difficulty for each component. This novel algorithm is general enough and can be applied to other VAE with discrete latent variables, which can be of independent interest. We emphasize that the proposed algorithm is maximizing *exact* the nagative Helmholtz variational free energy. It is different from [116] in which a lower bound of the variational free energy is proposed as the surrogate to be maximized for convenience.

We experimentally justified our algorithm on the synthetic datasets and three real-world datasets, namely the segmentation tasks for human activity, fruit fly behavior and heart sound records. The R-HSMM with Viterbi exact inference *significantly outperforms* basic HSMM and its variants, demonstrating the generative model is indeed flexible. Moreover, the trained bi-RNN encoder also achieve similar *state-of-the-art performances* to the exact inference, but with *400 times faster* inference speed, showing the proposed structured encoding function is able to mimic the exact inference efficiently.

(a) Classical Hidden Semi-Markov Model. (b) Recurrent Hidden Semi-Markov Model.

Figure 5.2: Graphical models of HSMM and R-HSMM. Different from classical HSMM, the R-HSMM has two-level emission structure with recurrent dependency.

5.2 Model Architecture

Given a sequence $\mathbf{x} = [x_1, x_2, \ldots, x_{|\mathbf{x}|}]$, where $x_t \in \mathbb{R}^m$ is an m dimensional observation at time t, our goal is to divide the sequence into meaningful segments. Thus, each observation x_t will have the corresponding label $z_t \in \mathbb{Z}$, where $\mathbb{Z} = \{1, 2, \ldots, K\}$ is a finite discrete label set and K is predefined. The label sequence $\mathbf{z} = [z_1, z_2, \ldots, z_{|\mathbf{x}|}]$ should have the same length of $\mathbf{x}$.

Besides labels, HSMM will associate each position t with additional variable $d_t \in \mathbb{D} = \{1, 2, \ldots, D\}$, where d_t is known as duration variable and D is the maximum possible duration. The duration variable can control the number of steps the current hidden state will remain. We use $\mathbf{d}$ to denote the duration sequence. We also use notation $\mathbf{x}_{t_1:t_2}$ to denote the substring $[x_{t_1}, x_{t_1+1}, \ldots, x_{t_2}]$ of $\mathbf{x}$. Without ambiguity, we use z as a segment label, and d as the duration.

Explicit Duration Hidden Markov Model. Similar to HMM, this model treats the pair of (z, d) as 'macro hidden state'. The probability of initial macro state is defined as $P(z, d) = P(z)P(d|z)$. We use the notation $\pi_z \triangleq P(z)$ and $P(d|z) \triangleq B_{z,d}$ to parametrize the initial probability and duration probability, respectively. $A_{i,j} \triangleq P(z_t = i|z_{t-1} = j, d_{t-1} = 1)$ is the state transition probability on the segment boundary. Here $\pi \in \mathbb{R}^K$ is in K-dimensional simplex. For each hidden state z, the corresponding rows $B_{z,:}$ and $A_{z,:}$ are also in probability simplex. Here we assume the multinomial distribution for $P(d|z)$.

In EDHMM, the transition probability of macro hidden state $P(z_t, d_t|z_{t-1}, d_{t-1})$ is de-

composed by $P(z_t|z_{t-1}, d_{t-1})P(d_t|z_t, d_{t-1})$ and thus can be defined as:

$$P(z_t|z_{t-1}, d_{t-1}) = \begin{cases} A_{z_{t-1},z_t} & \text{if } d_{t-1} = 1 \\ \mathbb{I}_{(z_t=z_{t-1})} & \text{if } d_{t-1} > 1 \end{cases}; P(d_t|z_t, d_{t-1}) = \begin{cases} B_{z_t,d_t} & \text{if } d_{t-1} = 1 \\ \mathbb{I}_{(d_t=d_{t-1}-1)} & \text{if } d_{t-1} > 1 \end{cases}. \tag{5.1}$$

The graphical model is shown in Figure 5.2a.

Recurrent Hidden Semi-Markov Model. For the simplicity of explanation, we focus our algorithm on the single sequence first. It is straightforward to apply the algorithm for dataset that has multiple sequences. Given the parameters $\{\pi, A, B\}$, the log-likelihood of a single observation sequence $\mathbf{x}$ can be written as below,

$$\mathcal{L}(\mathbf{x}) = \log \sum_{\mathbf{z},\mathbf{d}} \pi_{z_1} B_{z_1,d_1} \prod_{t=2}^{|\mathbf{x}|} P(z_t|z_{t-1}, d_{t-1}) P(d_t|z_t, d_{t-1}) P(\mathbf{x}|\mathbf{z}, \mathbf{d}), \tag{5.2}$$

where $P(\mathbf{x}|\mathbf{z}, \mathbf{d})$ is the emission probability. To define $P(\mathbf{x}|\mathbf{z}, \mathbf{d})$, we further denote the sequence variable $\boldsymbol{s} = [s_1, s_2, \ldots, s_{|\boldsymbol{s}|}]$ to be the switching position (or the beginning) of segments. Thus $s_1 = 1$, and $s_i = s_{i-1} + d_{s_{i-1}}$ and $|\boldsymbol{s}|$ is the number of segments. Traditional HSMM assumes $P(\mathbf{x}|\mathbf{z}, \mathbf{d}) = \prod_{t=1}^{|\mathbf{x}|} P(x_t|z_t)$, which ignores the dependency and some degree of dynamics exhibited in each segment. While in this chapter, we use RNN as the generative model to capture the dependent emission probability. Specifically, for the i-th segment starting from position s_i, the corresponding generative process is

$$P(\mathbf{x}_{s_i:s_i+d_{s_i}-1}|z_{s_i}, d_{s_i}) = \prod_{t=s_i}^{s_i+d_{s_i}-1} P(x_t|x_{s_i:t-1}, z_{s_i}) = \prod_{t=s_i}^{s_i+d_{s_i}-1} P(x_t|h_t, z_{s_i}) \tag{5.3}$$

where we assume that the dependency of history before time step j can be captured by a vector $h_j \in \mathbb{R}^h$. As in RNN, we use a recurrent equation to formulate the history vector,

$$h_t = \sigma(W^{(z_{s_i})} x_{t-1} + V^{(z_{s_i})} h_{t-1} + b^{(z_{s_i})}). \tag{5.4}$$

Finally, in this model, $P(\mathbf{x}|\mathbf{z}, \mathbf{d}) = \prod_{i=1}^{|\boldsymbol{s}|} P(\mathbf{x}_{s_i:s_i+d_{s_i}-1}|z_{s_i}, d_{s_i})$ is computed by the product

of generative probabilities for each segment. In Eq. 5.4, $W \in \mathbb{R}^{m \times h}$ is a weight matrix capturing the last observation x_{t-1}, and $V \in \mathbb{R}^{h \times h}$ is for the propagation of history h_{t-1}. The b is a bias term. The superscript z_{s_i} indexes the RNN we used for the corresponding segment. The segments with different labels are generated using different RNNs. So we should maintain K RNNs. $\sigma(\cdot)$ is a nonlinear activation function. We use tanh in our experiments.

At the time step t, we assume a diagonal multivariate gaussian distribution over the conditional likelihood, where the mean and covariance matrix are the output of RNN, *i.e.*,

$$P(x_t|h_t, z_{s_i}) \sim \mathcal{N}(x_t; \mu = U_\mu^{(z_{s_i})} h_t + b_\mu^{(z_{s_i})}, \Sigma = \text{Diag}(\exp(U_\Sigma^{(z_{s_i})} h_t + b_\Sigma^{(z_{s_i})}))) \quad (5.5)$$

The matrices $U_\mu, U_\Sigma \in \mathbb{R}^{m \times h}$ are used for parametrizing the mean and covariance at each time step j, given the history information. $b_\mu, b_\Sigma \in \mathbb{R}^m$ are bias terms. For simplicity, let's use $\theta_{rnn} = \{\theta_{rnn}^{(1)}, \theta_{rnn}^{(2)}, \ldots, \theta_{rnn}^{(K)}\}$ to denote the collection of parameters in each RNN. On the boundary case, *i.e.*, the starting point of each segment, we simply set $h_t = 0$, which can be viewed as the setting according to the prior knowledge (bias terms) of RNN.

The above formulation indicates that the generative model $P(x_t|h_t, z_{s_i})$ depends not only on the last step observation x_{t-1}, but also the last hidden state h_{t-1}, which is together captured in Eq. 5.4. In summary, we denote all the parameters in the proposed R-HSMM as $\theta = \{\pi, A, B, \theta_{rnn}\}$. The corresponding graphical model is shown in Figure 5.2b.

5.3 sequential variational autoencoder

To obtain the posterior or MAP in the proposed R-HSMM, the classical forward-backward algorithm or Viterbi algorithm needs to solve one dynamic programming *per sample*, which makes the inference costly, especially for the long sequence with thousands of timestamps. So instead, we treat the Bayesian inference from optimization perspective, and obtain the

posterior by maximizing the negative Helmholtz variational free energy [239, 251, 54],

$$\max_{Q(\mathbf{z},\mathbf{d}|\mathbf{x})\in\mathcal{P}} \mathcal{L}_Q^\theta(x) := \mathbb{E}_{Q(\mathbf{z},\mathbf{d}|\mathbf{x})}\left[\log P_\theta(\mathbf{x},\mathbf{z},\mathbf{d}) - \log Q(\mathbf{z},\mathbf{d}|\mathbf{x})\right], \tag{5.6}$$

over the space of all valid densities $\mathcal{P}$. To make the optimization (5.6) tractable, the variational autoencoder restricts the feasible sets to be some parametrized density Q_ψ, which can be executed efficiently comparing to the forward-backward algorithm or Viterbi algorithm. However, such restriction will introduce extra approximation error. To reduce the approximation error, we use a structured model, *i.e.*, bidirectional RNN, to mimic the dynamic programming in forward-backward algorithm. Specifically, in the forward-backward algorithm, the forward message $\alpha_t(z_t, d_t)$ and backward message $\beta_t(z_t, d_t)$ can be computed recursively, and marginal posterior at position t depends on both $\alpha_t(z_t, d_t)$ and $\beta_t(z_t, d_t)$. Similarly, in bi-RNN we embed the posterior message with RNN's latent vector, and marginal posterior is obtained from the latent vectors of two RNNs at the same time step t. Let $\psi = \{\psi_{\overrightarrow{\text{RNN}_1}}, \psi_{\overleftarrow{\text{RNN}_2}}, W_z \in \mathbb{R}^{h\times K}, W_d \in \mathbb{R}^{h\times D}\}$ be the parameters of bi-RNN encoder, the Q_ψ is decomposed as:

$$Q_\psi(\mathbf{z},\mathbf{d}|\mathbf{x}) = Q(z_1|h_1;\psi)Q(d_1|z_1,h_1;\psi)\prod_{t=2}^{|\mathbf{x}|} Q(z_t|d_{t-1},h_t;\psi)Q(d_t|z_t,d_{t-1},h_t;\psi) \tag{5.7}$$

where $h_t = [\overrightarrow{\text{RNN}_1}(\mathbf{x}_{1:t}), \overleftarrow{\text{RNN}_2}(\mathbf{x}_{t:|\mathbf{x}|})]$ is computed by bi-RNN. We use multinomial distributions $Q(z_t|h_t;\psi) = \mathcal{M}(\text{softmax}(W_z^\top h_t))$ and $Q(d_t|z_t,h_t;\psi) = \mathcal{M}(\text{softmax}(W_d^\top h_t))$. The dependency over d_{t-1} ensures that the generated segmentation $(\mathbf{z},\mathbf{d})$ is valid according to Eq. 5.1. For example, if we sampled duration $d_{t-1} > 1$ from Q_ψ at time $t-1$, then d_t and z_t should be deterministic. In our experiment, we use LSTM [103] as the recursive units in bi-RNN.

Since with any fixed θ, the negative Helmholtz variational free energy is indeed the

lower bound of the marginal likelihood, *i.e.*,

$$\log P_\theta(\mathbf{x}) \geq \mathcal{L}(\theta, \psi; \mathbf{x}) := \mathbb{E}_{Q_\psi(\mathbf{z},\mathbf{d}|\mathbf{x})}[\log P_\theta(\mathbf{x}, \mathbf{z}, \mathbf{d}) - \log Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x})], \tag{5.8}$$

therefore, we can treat it as a surrogate of the marginal log-likelihood and learn θ jointly with approximate inference, *i.e.*,

$$\max_{\theta,\psi} \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\theta, \psi; \mathbf{x}^{(n)}) \tag{5.9}$$

It should be emphasized that due to the *discrete* nature of latent variables in our model, the algorithm proposed in [129] is not directly applicable, and its extension with stochastic neuron reparametrization [21, 182, 91, 42] cannot provide satisfied results for our model according to our experiments. Therefore, we extend the penalty method to distribution space to solve optimization (5.9).

5.4 Learning via stochastic distributional penalty method

As we discussed, learning the sequential VAE with stochastic neuron reparametrization in unsupervised setting is extremely difficult, and none the off-the-shelf techniques can provide satisfied results. In this section, we introduce *auxiliary distribution* into (5.9) and generalize the penalty method [23] to distribution space.

Specifically, we first introduce an auxiliary distribution $\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x})$ for each $\mathbf{x}$ and reformulate the optimization (5.9) as

$$\begin{aligned} \max_{\theta,\psi,\{\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x}^{(n)})\}_{n=1}^N} \quad & \frac{1}{N} \sum_{n=1}^{N} \mathbb{E}_{\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x}^{(n)})} \left[\log P_\theta(\mathbf{x}^{(n)}, \mathbf{z}, \mathbf{d}) - \log \tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}^{(n)})\right], \\ \text{s.t.} \quad & KL\left(\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}^{(n)})||Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x}^{(n)})\right) = 0, \ \forall \mathbf{x}^{(n)}, n = 1, \ldots, N. \end{aligned} \tag{5.10}$$

We enforce the introduced $\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x})$ equals to $Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x})$ in term of KL-divergence, so that the optimization problems (5.9) and (5.10) are equivalent. Because of the non-

Algorithm 5 Learning sequential VAE with stochastic distributional penalty method

Input: sequences $\{\mathbf{x}^{(n)}\}_{n=1}^{N}$
Randomly initialize $\psi^{(0)}$ and $\theta^0 = \{\pi^0, A^0, B^0, \theta^0_{rnn}\}$
for $\lambda = 0, \ldots, \infty$ **do**
    for $t = 0$ **to** T **do**
        Sample $\{\mathbf{x}^{(n)}\}_{n=1}^{M}$ uniformly from dataset with mini-batch size M.
        Get $\{\mathbf{z}^{(n)}, \mathbf{d}^{(n)}\}_{n=1}^{M}$ with θ^t by dynamic programming in (5.13).
        Update π^{t+1}, A^{t+1}, B^{t+1} using rule (5.16).
        $\theta^{t+1}_{rnn} = \theta^t_{rnn}, -\gamma_t \frac{1}{M} \sum_{n=1}^{M} \nabla_{\theta^t_{rnn}} \tilde{\mathcal{L}}_\lambda(\theta, \psi | \mathbf{x}^{(n)})$
        $\psi^{t+1} = \psi^t - \eta_t \frac{1}{M} \sum_{n=1}^{M} \nabla_{\psi^t} \tilde{\mathcal{L}}_\lambda(\theta, \psi | \mathbf{x}^{(n)})$ ▷ bi-rnn sequence to sequence learning
    end for
end for

negativity of KL-divergence, itself can be viewed as the penalty function, we arrive the alternative formulation of (5.10) as

$$\max_{\theta, \psi, \{\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}^{(n)})\}_{n=1}^{N}} \frac{1}{N} \sum_{n=1}^{N} \tilde{\mathcal{L}}_\lambda(\theta, \psi | \mathbf{x}^{(n)}) \tag{5.11}$$

$\tilde{\mathcal{L}}_\lambda(\theta, \psi|\mathbf{x}) = \mathbb{E}_{\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x})} \Big[\log P_\theta(\mathbf{x}, \mathbf{z}, \mathbf{d}) - \log \tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}_i)\Big] - \lambda KL\left(\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}) || Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x})\right)$ and $\lambda \geq 0$. Obviously, as $\lambda \to \infty$, $KL(\tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}) || Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x}))$ must be 0, otherwise the $\tilde{\mathcal{L}}_\infty(\theta, \psi|\mathbf{x})$ will be $-\infty$ for arbitrary θ, ψ. Therefore, the optimization (5.11) will be equivalent to problem (5.10). Following the penalty method, we can learn the model with λ increasing from 0 to ∞ gradually. The entire algorithm is described in Algorithm 5. Practically, we can set $\lambda = \{0, \infty\}$ and do no need the gradually incremental, while still achieve satisfied performance. For each fixed λ, we optimize $\tilde{Q}$ and the parameters θ, ψ alternatively. To handle the expectation in the optimization, we will exploit the stochastic gradient descent. The update rules for θ, ψ and $\tilde{Q}$ derived below.

5.4.1 Updating $\tilde{Q}$

In fact, fix λ, Q_ψ and P_θ in optimization (5.11), the optimal solution $\tilde{Q}^*(\mathbf{z}, \mathbf{d}|\mathbf{x})$ for each $\mathbf{x}$ has closed-form.

Theorem 1 *Given fixed λ, Q_ψ and P_θ, $\tilde{Q}^*(\mathbf{z}, \mathbf{d}|\mathbf{x}) \propto Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x})^{\frac{\lambda}{1+\lambda}} P_\theta(\mathbf{x}, \mathbf{z}, \mathbf{d})^{\frac{1}{1+\lambda}}$ achieves the optimum in (5.11).*

Proof Take the functional derivative of $\tilde{\mathcal{L}}$ w.r.t. $\tilde{Q}$ and set it to zeros,

$$\nabla_{\tilde{Q}} \tilde{\mathcal{L}} = \log P_\theta(\mathbf{x}, \mathbf{z}, \mathbf{d}) + \lambda \log Q(\mathbf{z}, \mathbf{d}|\mathbf{x}) - (1+\lambda) \log \tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x}) = 0,$$

we obtain $\frac{1}{1+\lambda} \log P_\theta(\mathbf{x}, \mathbf{z}, \mathbf{d}) + \frac{\lambda}{1+\lambda} \log Q(\mathbf{z}, \mathbf{d}|\mathbf{x}) = \log \tilde{Q}(\mathbf{z}, \mathbf{d}|\mathbf{x})$. Take exponential on both sides, we achieve the conclusion. ■

In fact, because we are using the stochastic gradient for updating θ and ψ later, $\tilde{Q}^*(\mathbf{z}, \mathbf{d}|\mathbf{x})$ is never explicitly computed and only samples from it are required. Recall the fact that $Q_\psi(\mathbf{z}, \mathbf{d}|\mathbf{x})$ has a nice decomposition 5.7, we can multiply its factors into each recursion step and still get the same complexity as original Viterbi algorithm for MAP or sampling. Specifically, let's define $\alpha_t(j, r)$ to be the best joint log probability of prefix $x_{1:t}$ and its corresponding segmentation which has the last segment with label j and duration r, *i.e.*,

$$\alpha_t(j, r) \triangleq \max_{\mathbf{z}_{1:t}, \mathbf{d}_{1:t}} \log \tilde{Q}(\mathbf{z}_{1:t}, \mathbf{d}_{1:t}|\mathbf{x}_{1:t}), \text{ s.t. } z_t = j, d_t = d_{t-r} = 1, d_{t-r+1} = r \tag{5.12}$$

here $t \in \{1, 2, \ldots, |\mathbf{x}|\}$, $j \in \mathbb{Z}$, $r \in \mathbb{D}$. Then we can recursively compute the entries in α as below:

$$\alpha_t(j, r) = \begin{cases} \alpha_{t-1}(j, r-1) + \frac{1}{1+\lambda} \log(\frac{B_{j,r}}{B_{j,r-1}} P(x_t|x_{t-r+1:t-1}, z=j)) & r > 1, t > 1 \\ \qquad + \frac{\lambda}{1+\lambda} \log \frac{Q_\psi(d_{t-r+1}=r|z=j,\mathbf{x})}{Q_\psi(d_{t-r+1}=r-1|z=j,\mathbf{x})}; & \\ \max_{i \in \mathbb{Z} \setminus j} \max_{r' \in \mathbb{D}} \alpha_{t-1}(i, r') + \frac{1}{1+\lambda} \log(A_{i,j} B_{j,1} P(x_t|z=j)) & r = 1, t > 1 \\ \qquad + \frac{\lambda}{1+\lambda} \log Q_\psi(z_{t-r+1} = j, d_{t-r+1} = r|\mathbf{x}); & \\ \frac{\lambda}{1+\lambda} \log Q_\psi(z_1 = j, d_1 = r|\mathbf{x}) + \frac{1}{1+\lambda} \log(\pi_j B_{j,1} P(x_1|z=j)); & r = 1, t = 1 \\ 0. & \text{otherwise} \end{cases} \tag{5.13}$$

To construct the MAP solution, we also need to keep a back-tracing array $\beta_t(j,r)$ that records the transition path from $\alpha_{t-1}(i,r')$ to $\alpha_t(j,r)$. The sampling from $\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x})$ also can be completed with almost the same style forwards filtering backwards sampling algorithm, except replacing the max-operator by sum-operator in α propagation [165].

Without considering the complexity of computing emission probabilities, the dynamic programming needs time complexity $\mathcal{O}\left(|\mathbf{x}|K^2 + |\mathbf{x}|KD\right)$ [248] and $\mathcal{O}(|\mathbf{x}|K)$ memory. We explain the details of optimizing the time and memory requirements in Section 5.4.3.

Remark: When $\lambda = \infty$, the $\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x})$ will be exactly $Q_\psi(\mathbf{z},\mathbf{d}|\mathbf{x})$ and the algorithm will reduce to directly working on $Q_\psi(\mathbf{z},\mathbf{d}|\mathbf{x})$ without the effect from $P_\theta(\mathbf{x},\mathbf{z},\mathbf{d})$. Therefore, it is equivalent to obtaining MAP or sampling of the latent variables $\mathbf{z},\mathbf{d}$ from $Q_\psi(\mathbf{z},\mathbf{d}|\mathbf{x})$, whose cost is $\mathcal{O}(|\mathbf{x}|K)$. In practical, to further accelerate the computation, we can follow such strategy to generate samples when λ is already large enough, and thus, the effect of $P_\theta(\mathbf{x},\mathbf{z},\mathbf{d})$ is negligible.

5.4.2 Updating θ and ψ

With the fixed $\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x})$, we can update the θ and ψ by exploiting stochastic gradient descent algorithm to avoid scanning the whole training set. Sample a mini-batch of sequences $\{\mathbf{x}^n\}_{n=1}^M$ with size $M \ll N$, we proceed to update $\{\theta,\psi\}$ by optimizing the Monte Carlo approximation of (5.11),

$$\max_{\theta,\psi} \frac{1}{M}\sum_{n=1}^{M} \log P_\theta(\mathbf{x}^{(n)},\mathbf{z}^{(n)},\mathbf{d}^{(n)}) + \lambda \log Q_\psi(\mathbf{z}^{(n)},\mathbf{d}^{(n)}|\mathbf{x}^{(n)}), \tag{5.14}$$

where $\{\mathbf{z}^{(n)},\mathbf{d}^{(n)}\}$ is the MAP or a sample of $\tilde{Q}(\mathbf{z},\mathbf{d}|\mathbf{x}^{(n)})$. Note that the two parts related to θ and ψ are separated now, we can optimize them easily.

Update θ: Finding parameters to maximize the likelihood needs to solve the constrained

optimization shown below

$$\max_{\theta} \frac{1}{M} \sum_{n=1}^{M} \left(\log \pi_{z_1^{(n)}} + \sum_{i=2}^{|\boldsymbol{s}|} \log A_{z_{s_{i-1}}^{(n)}, z_{s_i}^{(n)}} + \sum_{i=1}^{|\boldsymbol{s}|} B_{z_{s_i}^{(n)}, d_{s_i}^{(n)}} + \sum_{j=s_i}^{s_i + d_{s_i}^{(n)} - 1} \log P(x_j^{(n)} | h_j^{(n)}, z_{s_i}^{(n)}; \theta_{rnn}) \right) \tag{5.15}$$

where $\{\pi, A, B\}$ are constrained to be valid probability distribution. We use stochastic gradient descent to update θ_{rnn} in totally K RNNs. For parameters π, A, B which are restricted to simplex, the stochastic gradient update will involve extra projection step. To avoid such operation which may be costly, we propose the closed-form update rule derived by Lagrangian,

$$\begin{aligned} \pi_i &= \frac{\sum_{n=1}^{M} \mathbb{I}(z_1^{(n)} = i)}{m}, \quad A_{i,j} = \frac{\sum_{n=1}^{M} \sum_{t=1}^{|\mathbf{s}^{(n)}|-1} \mathbb{I}(z_{s_t}^{(n)} = i \text{ and } z_{s_{t+1}}^{(n)} = j)}{\sum_{n=1}^{M} |\mathbf{s}^{(n)}| - M} \\ B_{j,r} &= \frac{\sum_{n=1}^{M} \sum_{t=1}^{|\mathbf{s}^{(n)}|} \mathbb{I}(d_{s_t}^{(n)} = r \text{ and } z_{s_t}^{(n)} = j)}{\sum_{n=1}^{M} |\mathbf{s}^{(n)}|} \end{aligned} \tag{5.16}$$

Since we already have the segmentation solution, the total number of training samples equals to the number of observations in dataset. Different RNNs use different parameters, and train on different parts of observations. This makes it easy for parallel training.

Update ψ: Given fixed λ, $\log Q_\psi(\mathbf{z}^{(n)}, \mathbf{d}^{(n)} | \mathbf{x}^{(n)})$ is essentially the sequence to sequence likelihood, where the input sequence is $\mathbf{x}$ and output sequence is $\{\mathbf{z}, \mathbf{d}\}$. Using the form of Q_ψ in Eq 5.7, this likelihood can be decomposed by positions. Thus we can conveniently train a bi-RNN which maximize the condition likelihood of latent variables by stochastic gradient descent.

Remark: We can get multiple samples $\{\mathbf{z}, \mathbf{d}\}$ for each $\mathbf{x}$ from $\tilde{Q}(\mathbf{z}, \mathbf{d} | \mathbf{x})$ to reduce the variance in stochastic gradient. In our algorithm, the samples of latent variable come naturally from the auxiliary distributions (which are integrated with penalty method), rather than the derivation from lower bound of objective [225, 182, 158].

5.4.3 Optimizing Dynamic Programming

Squeeze the memory requirement

In this section, we show that the Eq. 5.13 can be computed in a memory efficient way. Specifically, the dynamic programming procedure can be done with $\mathcal{O}(|\mathbf{x}|K)$ memory requirement, and caching for precomputed emission probabilities requires $\mathcal{O}(D^2K)$ memory.

Update forward variable α Note that in Eq. 5.13, when $r > 1$, we can update $\alpha_t(j, r)$ deterministically. So it is not necessary to keep the records for $r > 1$.

Specifically, let's only record $\alpha_t(j, 1)$, and do the updates in a similar way as in Eq. 5.13. The only difference is that, when constructing the answer, *i.e.*, the last segment solution, we need to do a loop over all possible z and d in order to find the best segmentation solution.

It is easy to see that the memory consumption is $\mathcal{O}(|\mathbf{x}|K)$.

Caching emission probability At each time step t, we compute $P(x_{t+r}|x_{t:t+r-1}, z = j)$ for each $j \in \mathbb{Z}$ and $r \in \mathbb{D}$. That is to say, we compute all the emission probabilities of observations starting from time t, and within max possible duration D. This can be done by performing feed-forward of K RNNs. After that, storing these results will require $\mathcal{O}(KD)$ space. For simplicity, we let $e^t_{j,r} = P(x_{t+r}|x_{t:t+r-1}, z = j)$, where $e^t \in \mathbb{R}^{K\times D}$.

Note that, at a certain time step t, we would require the emission probability of observations $P(x_t|x_{t-r+1:t-1}, z = j)$ for some $j \in \mathbb{Z}$ and $r \in \mathbb{D}$. In this case, the corresponding first observation is x_{t-r}. That is to say, we should keep $e^{t-D+1}, \ldots, e^t$ at time step t. This makes the memory consumption goes to $\mathcal{O}(KD^2)$

Squeeze the time complexity

In Eq. 5.13, the most expensive part is when $r = 1$ and $t > 1$. If we solve this in a naive way, then this step would require expensive $\mathcal{O}(|\mathbf{x}|K^2D)$ for time complexity.

Here we adopt similar technique as in [248]. Let $\gamma_t(i) = \max_{r' \in \mathbb{D}} \alpha_{t-1}(i, r')$, then:

$$\begin{aligned}\alpha_t(j,r) =& \max_{i \in \mathbb{Z}} \max_{r' \in \mathbb{D}} \alpha_{t_1}(i,r') + \frac{1}{1+\lambda} \log(A_{i,j} B_{j,1} P(x_t | z = j)) \qquad (5.17)\\ &+ \frac{\lambda}{1+\lambda} \log Q_\psi(z_{t-r+1} = j, d_{t-r+1} = r | \mathbf{x})\\ =& \max_{i \in \mathbb{Z}} \gamma_{t-1}(i) + \frac{1}{1+\lambda} \log(A_{i,j} B_{j,1} P(x_t | z = j)) \qquad (5.18)\\ &+ \frac{\lambda}{1+\lambda} \log Q_\psi(z_{t-r+1} = j, d_{t-r+1} = r | \mathbf{x})\end{aligned}$$

This reduces the complexity to be $\mathcal{O}(|\mathbf{x}|K^2)$.

5.5 Experiments

Baselines We compare with classical HSMM and two popular HSMM variants. The first one is Hierarchical Dirichlet-Process HSMM (HDP-HSMM) [117], which is the nonparametric Bayesian extension to the traditional HSMM that allows infinite number of hidden states; the second one is called subHSMM [115], which uses infinite HMM as the emission model for each segment. This model also has two-level of latent structure. It considers the dependency within each segment, which is a stronger algorithm than HDP-HSMM. We also compare with the CRF autoencoder (CRF-AE) [8], which uses markovian CRF as recognition model and conditional *i.i.d.*model for reconstruction. Comparing to HSMM, this model ignores the segmentation structures in modeling and is more similar to HMM.

Evaluation Metric We evaluate the performance of each method via the labeling accuracy. Specifically, we compare the labels of each single observations in each testing sequence. Since the labels are unknown during training, we use KM algorithm [162] to find the best mapping between predicted labels and ground-truth labels.

Settings Without explicitly mentioned, we report the mean of average accuracy for each of the sequence. We report the mean accuracy in Table 5.1. We set the truncation of max

possible duration D to be 400 for all tasks.

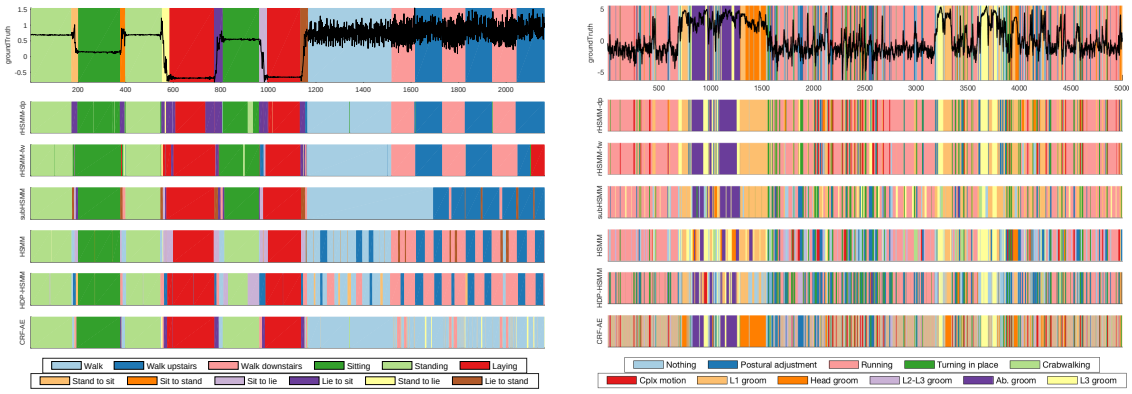
For the HDP-HSMM and subHSMM, the observation distributions are initialized as standard Multivariate Gaussian distributions. The duration is modeled by the Poisson distribution. We tune the concentration parameters $\alpha, \gamma \in \{0.1, 1, 3, 6, 10\}$. The hyperparameters are learned automatically. For subHSMM, we tune the truncation threshold of the second level infinite HMM from $\{2 \dots 15\}$.

For CRF-AE, we extend the origin model for the continuous observations, and learn all parameters similar to [151]. We use mixture of Gaussians to model the emission, where the number of mixtures is tuned in $\{1, \dots, 10\}$.

For the proposed R-HSMM, we use Adam [127] to train the K generative RNN and bi-RNN encoder. To make the learning tractable for long sequences, we use back propagation through time (BPTT) with limited budget. We also tune the dimension of hidden vector in RNN, the L_2-regularization weights and the stepsize. We implemented with CUDA that parallelized for different RNNs, and conduct experiments on K-20 enabled cluster. We include both the R-HSMM with the exact MAP via dynamic programming (rHSMM-dp) and sequential VAE with forward pass (rHSMM-fw) in experiments. In all tasks, the rHSMM-fw achieves almost the same performance to rHSMM-dp, but 400 times faster, showing the bi-RNN is able to mimic the forward-backward algorithm very well with efficient computation.

5.5.1 Segmentation Accuracy

Synthetic Experiments We first evaluate the proposed method on two 1D synthetic sequential data sets. The first data set is generated by a HSMM with 3 hidden states, where π, A, B are designed beforehand. A segment with hidden state z is a sine function $\lambda_z \sin(\omega_z x + \epsilon_1) + \epsilon_2$, where ϵ_1 and ϵ_2 are Gaussian random noises. Different hidden states use different scale parameters λ_z and frequency parameters ω_z. The second data set also has 3 hidden states, where the segment with hidden state z is sampled from a Gaussian



(a) Human activity (b) Drosophila

Figure 5.3: Segmentation results on Human activity and Drosophila datasets. Different background colors represent the segmentations with different labels. In the top row, the black cure shows the signal sequence projected to the first principle component. The following two rows are our algorithms which almost exact locate every segment. (a) The Human activity data set contains 12 hidden states, each of which corresponds to a human action; (b) The Drosophila data set contains 11 hidden states, each of which corresponds to a drosophila action.

process (GP) with kernel function $k_z(x, y)$. Different hidden states employ different kernel functions. The specific kernel functions used here are $k_1(x, y) = \exp\{-\min(|\ x-y\ |, |x+y|)^2/10\}$, $k_2(x, y) = \exp\{-(x-y)^2/10\}$ and $k_3(x, y) = (5-|x-y|)I\{(5-|x-y|) < 5\}$. For both of the Sine and GP data sets, the duration of a segment is randomly sampled from a distribution defined on $\{1, ..., 100\}$, which depends on the hidden states. Thus, the segmentation task corresponds to finding out different functions embedded in the sequences.

We visualize the segmentation results of ground truth and three competitors on Sine and GP data sets in Figure 5.1a and Figure 5.1b respectively, and report the numerical results in Table 5.1. As we can see, R-HSMM provides much better results on even small segments, dramatically outperforms HSMM variants and CRF-AE. Also note that, the sine function depicts short term dependencies, while Gaussian process has long dependency that determined by the kernel bandwidth. This demonstrates the ability of R-HSMM in capturing the long or short term dependencies.

Table 5.1: Error rate of segmentation. We report both the mean and standard deviation.

Methods	SINE	GP	HAPT	Drosophila	Heart	PN-Full
rHSMM-dp	**2.67 ± 1.13%**	**12.46 ± 2.79%**	**16.38 ± 5.03%**	**36.21 ± 1.37%**	**33.14 ± 7.87%**	**31.95 ± 4.32%**
rHSMM-fw	4.02 ± 1.37%	**13.13 ± 2.89%**	**17.74 ± 7.64%**	**35.79 ± 0.51%**	**33.36 ± 8.10%**	**32.34 ± 3.97%**
HSMM	41.85 ± 2.38%	41.15 ± 1.99%	41.59 ± 8.58%	47.37 ± 0.27%	50.62 ± 4.20 %	45.04 ± 1.87%
subHSMM	18.14 ± 2.63%	24.81 ± 4.63%	22.18 ± 4.45%	39.70 ± 2.21%	46.67 ± 4.22%	43.01 ± 2.35%
HDP-HSMM	42.74 ± 2.73%	41.90 ± 1.58%	35.46 ± 6.19%	43.59 ± 1.58%	47.56 ± 4.31%	42.58 ± 1.54%
CRF-AE	44.87 ± 1.63%	51.43 ± 2.14%	49.26 ± 10.63%	57.62 ± 0.22%	53.16 ± 4.78%	45.73 ± 0.66%

Human activity This dataset which is collected by [185] consists of signals collected from waist-mounted smartphone with accelerometers and gyroscopes. Each of the volunteers is asked to perform a protocol of activities composed of 12 activities (see Figure 5.3a for the details). Since the signals within an activity type exhibit high correlation, it is natural for RNN to model this dependency. We use these 61 sequences, where each sequence has length around 3000. Each observation is a 6 dimensional vector, consists of triaxial measures from accelerometers and gyroscopes.

Figure 5.3a shows the ground truth and the segmentation results of all methods. Both rHSMM-dp and rHSMM-fw almost perfectly recover the true segmentation. They can also capture the transition activity types, *e.g.*, stand to lie or sit to lie. The HSMM, HDP-HSMM and CRF-AE makes some fragmental but periodical segmentations for walking, caused by lacking the dependency modeling within a segment. The subHSMM also has similar problem, possibly due to the limited ability of HMM generative model.

Drosophila Here we study the behavior patterns of drosophilas. The data was collected by [118] with two dyes, two cameras and some optics to track each leg of a spontaneously behaving fruit fly. The dimension of observation in each timestamp is 45, which consists of the raw features and some higher order features.

Physionet The heart sound records, usually represented graphically by phonocardiogram (PCG), are key resources for pathology classification of patients. We collect data from PhysioNet Challenge 2016 [215], where each observation has been labeled with one of

the four states, namely Diastole, S1, Systole and S2. We experiment with both the raw signals and the signals after feature extraction. Regarding the raw signals (Heart dataset), we collect 7 1-dimensional sequences of length around 40000. The feature-rich dataset (PN-Full) contains 2750 sequences, where each of them consists of 1500 4-dimensional observations. We do 5-fold cross validation for PN-Full. As the results shown in Table 5.1, our algorithm still outperforms the baselines significantly. Also for such long raw signal sequences, the speed advantage of bi-RNN encoder over Viterbi is more significant. Viterbi takes 8min to do one inference, while bi-RNN only takes several seconds. Our framework is also flexible to incorporate prior knowledge, like the regularity of heart state transition into HSMM.

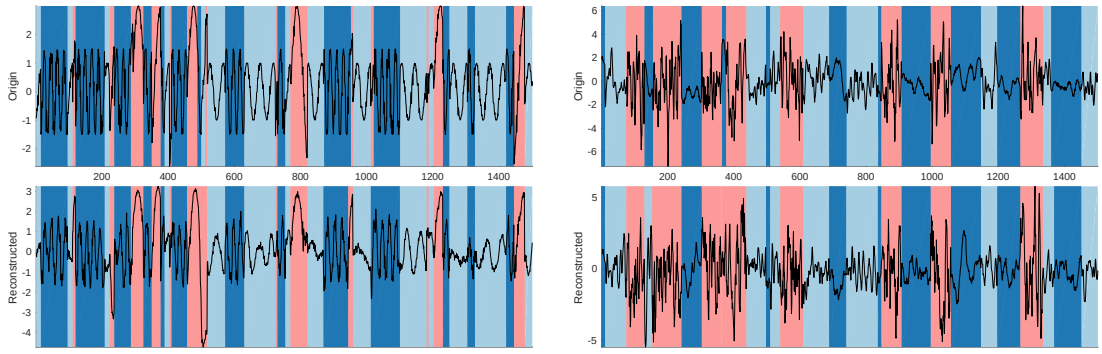
5.5.2 Reconstruction

In this section, we examine the ability of learned generative model by visualizing the reconstructed signals. Given a sequence $\mathbf{x}$, we use recognition model to get the latent variables $\mathbf{z}$ and $\mathbf{d}$, then use learned K generative RNNs to generate signals within each segment. For the ease of visualization, we show the results on 1D signal dataset in Fig. 5.4a and Fig. 5.4b.

From Fig. 5.4 we can see the generative RNN correctly captures different characteristics from signals of different segment labels, such as different frequencies and scales in Sine dataset, or the different variance patterns in GP dataset. This is essential to distinguish between different segments.

5.6 Summary

We presented the R-HSMM, a generalization of HSMM by incorporating recurrent neural generative model as the emission probability. To eliminate the difficulty caused by such flexible and powerful model in inference, we introduced the bi-RNN as the encoding distribution via the variational autoencoder framework to mimic the forward-backward al-



(a) Reconstruction illustration on Sine dataset.

(b) Reconstruction illustration on GP dataset.

Figure 5.4: Reconstruction illustration. The generative RNNs (decoders) are asked to reconstruct the signals from only the discrete labels and durations (which are generated from encoder).

gorithm. To deal with the difficulty of training VAE containing discrete latent variables, we proposed a novel stochastic distributional penalty method. We justified the modeling power of the proposed R-HSMM via segmentation accuracy and reconstruction visualization. From the comprehensive comparison, the proposed model significantly outperforms the existing models. It should be emphasized that the structured bi-RNN encoder yields similar performance as the exact MAP inference, while being 400 times faster. Future work includes further speeding up of our algorithm, as well as generalizing our learning algorithm to other discrete variational autoencoder.

So far, we have demonstrated the HSMM model structure for sequences. In the next section, we will show how to leverage compiler algorithms for graph structure generative modeling.

CHAPTER 6
GRAPH GENERATIVE MODELING WITH SYNTAX AND SEMANTICS GUIDANCE

Deep generative models have been enjoying success in modeling continuous data. However it remains challenging to capture the representations for discrete structures with formal grammars and semantics, *e.g.*, computer programs and molecular structures. How to generate both syntactically and semantically correct data still remains largely an open problem. Inspired by the theory of compiler where the syntax and semantics check is done via syntax-directed translation (SDT), we propose a novel syntax-directed variational autoencoder (SD-VAE) by introducing *stochastic lazy attributes*. This approach converts the offline SDT check into on-the-fly generated guidance for constraining the decoder. Comparing to the state-of-the-art methods, our approach enforces constraints on the output space so that the output will be not only *syntactically* valid, but also *semantically* reasonable. We evaluate the proposed model with applications in programming language and molecules, including reconstruction and program/molecule optimization. The results demonstrate the effectiveness in incorporating syntactic and semantic constraints in discrete generative models, which is significantly better than current state-of-the-art approaches.

6.1 Introduction

Recent advances in deep representation learning have resulted in powerful probabilistic generative models which have demonstrated their ability on modeling continuous data, *e.g.*, time series signals [169, 56] and images [180, 121]. Despite the success in these domains, it is still challenging to correctly generate discrete structured data, such as graphs, molecules and computer programs. Since many of the structures have syntax and semantic formalisms, the generative models without explicit constraints often produces invalid ones.

Conceptually an approach in generative model for structured data can be divided in two parts, one being the formalization of the structure generation and the other one being a (usually deep) generative model producing parameters for stochastic process in that formalization. Often the hope is that with the help of training samples and capacity of deep models, the loss function will prefer the valid patterns and encourage the mass of the distribution of the generative model towards the desired region automatically.

Arguably the simplest structured data are sequences, whose generation with deep model has been well studied under the seq2seq [219] framework that models the generation of sequence as a series of token choices parameterized by recurrent neural networks (RNNs). Its widespread success has encourage several pioneer works that consider the conversion of more complex structure data into sequences and apply sequence models to the represented sequences. [86] (CVAE) is a representative work of such paradigm for the chemical molecule generation, using the SMILES line notation [237] for representing molecules. However, because of the lack of formalization of syntax and semantics serving as the restriction of the *particular* structured data, underfitted *general-purpose* string generative models will often lead to invalid outputs. Therefore, to obtain a reasonable model via such training procedure, we need to prepare large amount of valid combinations of the structures, which is time consuming or even not practical in domains like drug discovery.

To tackle such a challenge, one approach is to incorporate the structure restrictions explicitly into the generative model. For the considerations of computational cost and model generality, context-free grammars (CFG) have been taken into account in the decoder parametrization. For instance, in molecule generation tasks, [137] proposes a grammar variational autoencoder (GVAE) in which the CFG of SMILES notation is incorporated into the decoder. The model generates the parse trees directly in a top-down direction, by repeatedly expanding any nonterminal with its production rules. Although the CFG provides a mechanism for generating *syntactic valid* objects, it is still incapable to regularize the model for generating *semantic valid* objects [137]. For example, in molecule gener-

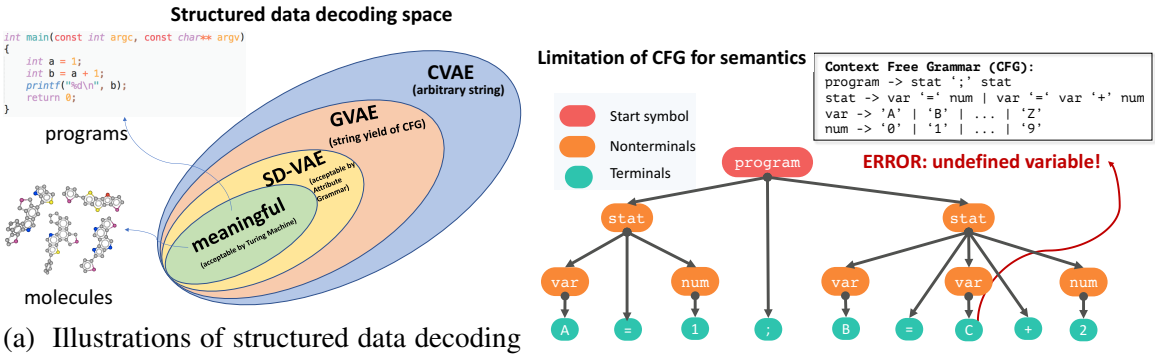
ation, the semantic of the SMILES languages requires that the rings generated must be closed; in program generation, the referenced variable should be defined in advance and each variable can only be defined exactly once in each local context (illustrated in Fig 6.1b). All the examples require cross-serial like dependencies which are not enforceable by CFG, implying that more constraints beyond CFG are needed to achieve semantic valid production in VAE.

In the theory of compiler, attribute grammars, or syntax-directed definition has been proposed for attaching semantics to a parse tree generated by context-free grammar. Thus one straightforward but not practical application of attribute grammars is, after generating a syntactic valid molecule candidate, to conduct offline semantic checking. This process needs to be repeated until a semantically valid one is discovered, which is at best computationally inefficient and at worst infeasible, due to extremely low rate of passing checking. As a remedy, we propose the *syntax-direct variational autoencoder* (SD-VAE), in which a semantic restriction component is advanced to the stage of syntax tree generator. This allows the generator with both syntactic and semantic validation. The proposed syntax-direct generative mechanism in the decoder further constraints the output space to ensure the semantic correctness in the tree generation process. The relationships between our proposed model and previous models can be characterized in Figure 6.1a.

Our method brings theory of formal language into stochastic generative model. The contribution of this chapter can be summarized as follows:

- *Syntax and semantics enforcement*: We propose a new formalization of semantics that systematically converts the offline semantic check into online guidance for stochastic generation using the proposed *stochastic lazy attribute*. This allows us effectively address both syntax and semantic constraints.

- *Efficient learning and inference*: Our approach has computational cost $O(n)$ where n is the length of structured data. This is the same as existing methods like CVAE and GVAE which do not enforce semantics in generation. During inference, the SD-VAE runs with



(a) Illustrations of structured data decoding space

(b) CFG parses program 'A=1;B=C+2'

Figure 6.1: Illustration on left shows the hierarchy of the structured data decoding space w.r.t different works and theoretical classification of corresponding strings from formal language theory. SD-VAE, our proposed model with attribute grammar reshapes the output space tighter to the meaningful target space than existing works. On the right we show a case where CFG is unable to capture the semantic constraints, since it successfully parses an invalid program.

semantic guiding on-the-fly, while the existing alternatives generate many candidates for semantic checking.

- *Strong empirical performance*: We demonstrate the effectiveness of the SD-VAE through applications in two domains, namely (1) the subset of Python programs and (2) molecules. Our approach consistently and significantly improves the results in evaluations including generation, reconstruction and optimization.

6.2 Background

Before introducing our model and the learning algorithm, we first provide some background knowledge which is important for understanding the proposed method.

6.2.1 Variational Autoencoder

The variational autoencoder [129, 186] provides a framework for learning the probabilistic generative model as well as its posterior, respectively known as decoder and encoder. We denote the observation as x, which is the structured data in our case, and the latent variable as z. The decoder is modeling the probabilistic generative processes of x given the continuous representation z through the likelihood $p_\theta(x|z)$ and the prior over the latent

variables $p(z)$, where θ denotes the parameters. The encoder approximates the posterior $p_\theta(z|x) \propto p_\theta(x|z)p(z)$ with a model $q_\psi(z|x)$ parametrized by ψ. The decoder and encoder are learned simultaneously by maximizing the evidence lower bound (ELBO) of the marginal likelihood, *i.e.*,

$$\mathcal{L}(X;\theta,\psi) := \sum_{x\in X} \mathbb{E}_{q(z|x)} \left[\log p_\theta(x|z)p(z) - \log q_\psi(z|x)\right] \leq \sum_{x\in X} \log \int p_\theta(x|z)p(z)dz, \tag{6.1}$$

where X denotes the training datasets containing the observations.

6.2.2 Context Free Grammar and Attribute Grammar

Context free grammar A context free grammar (CFG) is defined as $G = \langle \mathcal{V}, \Sigma, \mathcal{R}, s\rangle$, where symbols are divided into $\mathcal{V}$, the set of non-terminal symbols, Σ, the set of terminal symbols and $s \in \mathcal{V}$, the start symbol. Here $\mathcal{R}$ is the set of production rules. Each production rule $r \in \mathcal{R}$ is denoted as $r = \alpha \to \beta$ for $\alpha \in \mathcal{V}$ is a nonterminal symbol, and $\beta = u_1 u_2 \dots u_{|\beta|} \in (\mathcal{V} \bigcup \Sigma)^*$ is a sequence of terminal and/or nonterminal symbols.

Attribute grammar To enrich the CFG with "semantic meaning", [133] formalizes attribute grammar that introduces attributes and rules to CFG. An attribute is an attachment to the corresponding nonterminal symbol in CFG, written in the format $\langle v\rangle.a$ for $v \in \mathcal{V}$. There can be two types of attributes assigned to non-terminals in G: the *inherited* attributes and the *synthesized* attributes. An inherited attribute depends on the attributes from its parent and siblings, while a synthesized attribute is computed based on the attributes of its children. Formally, for a production $u_0 \to u_1 u_2 \dots u_{|\beta|}$, we denote $I(u_i)$ and $S(u_i)$ be the sets of *inherited* and *synthesized* attributes of u_i for $i \in \{0, \dots, |\beta|\}$, respectively.

A motivational example

We here exemplify how the above defined attribute grammar enriches CFG with non-context-free semantics. We use the following toy grammar, a subset of SMILES that gen-

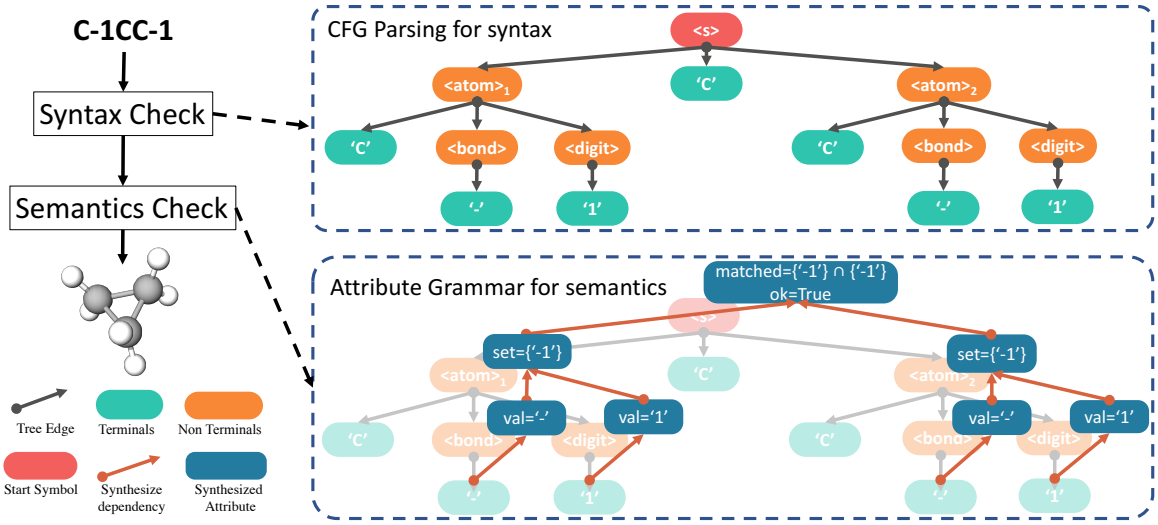


Figure 6.2: Bottom-up syntax and semantics check in compilers.

erates either a chain or a cycle with three carbons:

Production	**Semantic Rule**
$\langle s \rangle \rightarrow \langle atom \rangle_1$ 'C' $\langle atom \rangle_2$	$\langle s \rangle$.matched $\leftarrow \langle atom \rangle_1$.set $\bigcap \langle atom \rangle_2$.set,
	$\langle s \rangle$.ok $\leftarrow \langle atom \rangle_1$.set $= \langle s \rangle$.matched $= \langle atom \rangle_2$.set
$\langle atom \rangle \rightarrow$ 'C' | 'C' $\langle bond \rangle \langle digit \rangle$	$\langle atom \rangle$.set $\leftarrow \varnothing$ | concat($\langle bond \rangle$.val, $\langle digit \rangle$.val)
$\langle bond \rangle \rightarrow$ '-' | '=' | '#'	$\langle bond \rangle$.val $\leftarrow$ '-' | '=' | '#'
$\langle digit \rangle \rightarrow$ '1' | '2' | ... | '9'	$\langle digit \rangle$.val $\leftarrow$ '1' | '2' ... | '9'

where we show the production rules in CFG with $\rightarrow$ on the left, and the calculation of attributes in attribute grammar with $\leftarrow$ on the left. Here we leverage the attribute grammar to check (with attribute `matched`) whether the ringbonds come in pairs: a ringbond generated at $\langle atom \rangle_1$ should match the bond type and bond index that generated at $\langle atom \rangle_2$, also the semantic constraint expressed by $\langle s \rangle$.ok requires that there is no difference between the `set` attribute of $\langle atom \rangle_1$ and $\langle atom \rangle_2$. Such constraint in SMILES is known as *cross-serial dependencies* (CSD) [33] which is non-context-free [209]. See Appendix B.3 for more explanations. Figure 6.2 illustrates the process of performing syntax and semantics check in compilers. Here all the attributes are *synthetic*, *i.e.*, calculated in a bottom-up direction.

So generally, in the semantic correctness checking procedure, one need to perform

bottom-up procedures for calculating the attributes *after* the parse tree is generated. However, in the top-down structure generating process, the parse tree is not ready for semantic checking, since the synthesized attributes of each node require information from its children nodes, which are not generated yet. Due to such dilemma, it is nontrivial to use the attribute grammar to guide the top-down generation of the tree-structured data. One straightforward way is using acceptance-rejection sampling scheme, *i.e.*, using the decoder of CVAE or GVAE as a proposal and the semantic checking as the threshold. It is obvious that since the decoder does not include semantic guidance, the proposal distribution may raise semantically invalid candidate frequently and thus waste the computational cost.

6.3 Syntax-Directed Variational Autoencoder

As described in Section 6.2.2, directly using attribute grammar in an offline fashion (*i.e.*, after the generation process finishes) is not efficient to address both syntax and semantics constraints. In this section we describe how to bring forward the attribute grammar online and incorporate it into VAE, such that our VAE addresses both *syntactic* and *semantic* constraints. We name our proposed method Syntax-Directed VAE (SD-VAE).

6.3.1 Stochastic Syntax-Directed Decoder

By scrutinizing the tree generation, the major difficulty in incorporating the attributes grammar into the processes is the appearance of the synthesized attributes. For instance, when expanding the start symbol $\langle s \rangle$, none of its children is generated yet. Thus their attributes are also absent at this time, making the $\langle s \rangle$.`matched` unable to be computed. To enable the on-the-fly computation of the synthesized attributes for semantic validation during tree generation, besides the two types of attributes, we introduce the *stochastic lazy attributes* to enlarge the existing attribute grammar. Such *stochasticity* transforms the corresponding synthesized attribute into inherited constraints in generative procedure; and lazy linking mechanism sets the actual value of the attribute, once all the other dependent attributes are

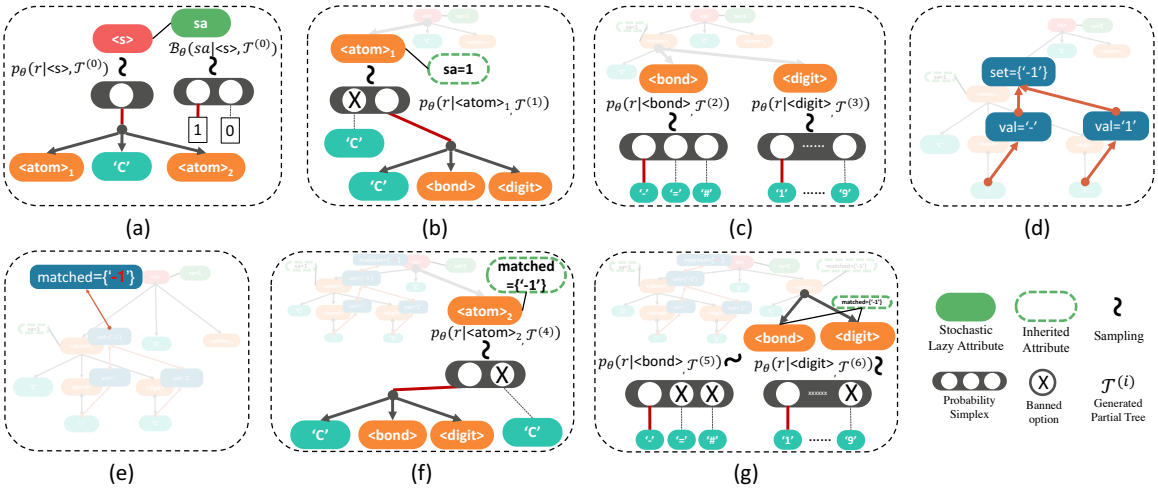


Figure 6.3: On-the-fly generative process of SD-VAE in order from (a) to (g). Steps: (a) stochastic generation of attribute; (b)(f)(g) constrained sampling with inherited attributes; (c) unconstrained sampling; (d) synthesized attribute calculation on generated subtree. (e) lazy evaluation of the attribute at root node.

ready. We demonstrate how the decoder with *stochastic lazy attributes* will generate semantic valid output through the same pedagogical example as in Section 6.2.2. Figure 6.3 visually demonstrates this process.

The tree generation procedure is indeed sampling from the decoder $p_\theta(x|z)$, which can be decomposed into several steps that elaborated below:

i) stochastic predetermination: in Figure 6.3(a), we start from the node $\langle s \rangle$ with the synthesized attributes $\langle s \rangle.\texttt{matched}$ determining the index and bond type of the ringbond that will be matched at node $\langle s \rangle$. Since we know nothing about the children nodes right now, the only thing we can do is to 'guess' a value. That is to say, we associate a stochastic attribute $\langle s \rangle.\texttt{sa} \in \{0,1\}^{C_a} \sim \prod_{i=1}^{C_a} \mathcal{B}(sa_i|z)$ as a predetermination for the sake of the absence of synthesized attribute $\langle s \rangle.\texttt{matched}$, where $\mathcal{B}(\cdot)$ is the Bernoulli distribution. Here C_a is the maximum cardinality possible [1] for the corresponding attribute a. In above example, the 0 indicates no ringbond and 1 indicates one ringbond at both $\langle atom \rangle_1$ and $\langle atom \rangle_2$, respectively.

ii) constraints as inherited attributes: we pass the $\langle s \rangle.\texttt{sa}$ as inherited constraints to

[1] Note that setting threshold for C_a assumes a *mildly context sensitive grammar* (*e.g.*, limited CSD).

Algorithm 6 Decoding with Stochastic Syntax-Directed Decoder

Global variables: CFG: $G = (\mathcal{V}, \Sigma, \mathcal{R}, s)$, decoder network parameters θ
procedure GENTREE($node$, $\mathcal{T}$)
    Sample stochastic lazy attribute $node._{sa} \sim \mathcal{B}_\theta(sa|node, \mathcal{T})$ ▷ when introduced on $node$
    Sample production rule $r = (\alpha \to \beta) \in \mathcal{R} \sim p_\theta(r|ctx, node, \mathcal{T})$. ▷ The conditioned variables encodes the semantic constraints in tree generation.
    $ctx \leftarrow \text{RNN}(ctx, r)$ ▷ update context vector
    for $i = 1, \ldots, |\beta|$ **do**
        $v_i \leftarrow \text{Node}(u_i, node, \{v_j\}_{j=1}^{i-1})$ ▷ node creation with parent and siblings' attributes
        GenTree(v_i, $\mathcal{T}$) ▷ recursive generation of children nodes
        Update synthetic and stochastic attributes of $node$ with v_i ▷ Lazy linking
    end for
end procedure

the children of node ⟨*s*⟩, *i.e.*, ⟨*atom*⟩$_1$ and ⟨*atom*⟩$_2$ to ensure the semantic validation in the tree generation. For example, Figure 6.3(b) `'sa=1'` is passed down to ⟨*atom*⟩$_1$.

iii) sampling under constraints: without loss of generality, we assume ⟨*atom*⟩$_1$ is generated before ⟨*atom*⟩$_2$. We then sample the rules from $p_\theta(r|\langle atom\rangle_1, \langle s\rangle, z)$ for expanding ⟨*atom*⟩$_1$, and so on and so forth to generate the subtree recursively. Since we carefully designed sampling distribution that is conditioning on the stochastic property, the inherited constraints will be eventually satisfied. In the example, due to the ⟨*s*⟩`.sa = '1'`, when expanding ⟨*atom*⟩$_1$, the sampling distribution $p_\theta(r|\langle atom\rangle_1, \langle s\rangle, z)$ only has positive mass on rule ⟨*atom*⟩ → `'C'` ⟨*bond*⟩ ⟨*digit*⟩.

iv) lazy linking: once we complete the generation of the subtree rooted at ⟨*atom*⟩$_1$, the synthesized attribute ⟨*atom*⟩$_1$`.set` is now available. According to the semantic rule for ⟨*s*⟩`.matched`, we can instantiate ⟨*s*⟩`.matched` = ⟨*atom*⟩$_1$`.set` = { `'-1'` }. This linking is shown in Figure 6.3(d)(e). When expanding ⟨*atom*⟩$_2$, the ⟨*s*⟩`.matched` will be passed down as inherited attribute to regulate the generation of ⟨*atom*⟩$_2$, as is demonstrated in Figure 6.3(f)(g).

In summary, the general syntax tree $\mathcal{T} \in L(G)$ can be constructed step by step, within the languages $L(G)$ covered by grammar G. In the beginning, $\mathcal{T}^{(0)} = root$, where

$root._{symbol} = s$ which contains only the start symbol s. At step t, we will choose an non-terminal node in the *frontier*[2] of partially generated tree $\mathcal{T}^{(t)}$ to expand. The generative process in each step $t = 0, 1, \ldots$ can be described as:

1. Pick node $v^{(t)} \in Fr(\mathcal{T}^{(t)})$ where its attributes needed are either satisfied, or are stochastic attributes that should be sampled first according to Bernoulli distribution $\mathcal{B}(\cdot|v^{(t)}, \mathcal{T}^{(t)})$;

2. Sample rule $r^{(t)} = \alpha^{(t)} \to \beta^{(t)} \in \mathcal{R}$ according to distribution $p_\theta(r^{(t)}|v^{(t)}, \mathcal{T}^{(t)})$, where $v^{(t)}._{symbol} = \alpha^{(t)}$, and $\beta^{(t)} = u_1^{(t)} u_2^{(t)} \ldots u_{|\beta^{(t)}|}^{(t)}$, *i.e.*, expand the nonterminal with production rules defined in CFG.

3. $\mathcal{T}^{(t+1)} = \mathcal{T}^{(t)} \bigcup \{(v^{(t)}, u_i^{(t)})\}_{i=1}^{|\beta^{(t)}|}$, *i.e.*, grow the tree by attaching $\beta^{(t)}$ to $v^{(t)}$. Now the node $v^{(t)}$ has children represented by symbols in $\beta^{(t)}$.

The above process continues until all the nodes in the frontier of $\mathcal{T}^{(T)}$ are all terminals after T steps. Then, we obtain the Algorithm 6 for sampling both syntactic and semantic valid structures.

In fact, in the model training phase, we need to compute the likelihood $p_\theta(x|z)$ given x and z. The probability computation procedure is similar to the sampling procedure in the sense that both of them requires tree generation. The only difference is that in the likelihood computation procedure, the tree structure, *i.e.*, the computing path, is fixed since x is given; While in the sampling procedure, it is sampled following the learned model. Specifically, the generative likelihood can be written as:

$$p_\theta(x|z) = \prod_{t=0}^{T} p_\theta(r_t|ctx^{(t)}, node^{(t)}, \mathcal{T}^{(t)}) \mathcal{B}_\theta(sa_t|node^{(t)}, \mathcal{T}^{(t)}) \tag{6.2}$$

where $ctx^{(0)} = z$ and $ctx^{(t)} = \text{RNN}(r_t, ctx^{(t-1)})$. Here RNN can be commonly used LSTM, *etc.*.

[2]Here frontier is the set of all nonterminal leaves in current tree.

6.3.2 Structure-Based Encoder

As we introduced in section 6.2, the encoder, $q_\psi(z|x)$ approximates the posterior of the latent variable through the model with some parametrized function with parameters ψ. Since the structure in the observation x plays an important role, the encoder parametrization should take care of such information. The recently developed deep learning models [71, 55, 140] provide powerful candidates as encoder. However, to demonstrate the benefits of the proposed syntax-directed decoder in incorporating the attribute grammar for semantic restrictions, we will exploit the same encoder in [137] for a fair comparison later.

We provide a brief introduction to the particular encoder model used in [137] for a self-contained purpose. Given a program or a SMILES sequence, we obtain the corresponding parse tree using CFG and decompose it into a sequence of productions through a pre-order traversal on the tree. Then, we convert these productions into one-hot indicator vectors, in which each dimension corresponds to one production in the grammar. We will use a deep convolutional neural networks which maps this sequence of one-hot vectors to a continuous vector as the encoder.

6.3.3 Model Learning

Our learning goal is to maximize the evidence lower bound in Eq 6.1. Given the encoder, we can then map the structure input into latent space z. The variational posterior $q(z|x)$ is parameterized with Gaussian distribution, where the mean and variance are the output of corresponding neural networks. The prior of latent variable $p(z) = \mathcal{N}(0, I)$. Since both the prior and posterior are Gaussian, we use the closed form of KL-divergence that was proposed in [129]. In the decoding stage, our goal is to maximize $p_\theta(x|z)$. Using the Equation (6.2), we can compute the corresponding conditional likelihood. During training, the syntax and semantics constraints required in Algorithm 6 can be precomputed.

In practice, we observe no significant time penalty measured in wall clock time compared to previous works.

6.4 Experiments

Code is available at https://github.com/Hanjun-Dai/sdvae.

We show the effectiveness of our proposed SD-VAE with applications in two domains, namely programs and molecules. We compare our method with CVAE [86] and GVAE [137]. CVAE only takes character sequence information, while GVAE utilizes the context-free grammar. To make a fair comparison, we closely follow the experimental protocols that were set up in [137]. The training details are included in Section 6.4.2.

Our method gets significantly better results than previous works. It yields better reconstruction accuracy and prior validity by large margins, while also having comparative diversity of generated structures. More importantly, the SD-VAE finds better solution in program and molecule regression and optimization tasks. This demonstrates that the continuous latent space obtained by SD-VAE is also smoother and more discriminative.

6.4.1 Settings

Here we first describe our datasets in detail. The programs are represented as a list of statements. Each statement is an atomic arithmetic operation on variables (labeled as `v0`, `v1`, $\cdots$, `v9`) and/or immediate numbers ($1, 2, \ldots, 9$). Some examples are listed below:

```
v3=sin(v0);v8=exp(2);v9=v3-v8;v5=v0*v9;return:v5
v2=exp(v0);v7=v2*v0;v9=cos(v7);v8=cos(v9);return:v8
```

Here `v0` is always the input, and the variable specified by `return` (respectively `v5` and `v8` in the examples) is the output, therefore it actually represent univariate functions $f : \mathbb{R} \to \mathbb{R}$. Note that a correct program should, besides the context-free grammar specified in Appendix B.1, also respect the semantic constraints. For example, a variable should be defined before being referenced. We randomly generate $130,000$ programs, where each consisting of 1 to 5 valid statements. Here the maximum number of decoding steps $T = 80$. We hold 2000 programs out for testing and the rest for training and validation.

For molecule experiments, we use the same dataset as in [137]. It contains $250,000$ SMILES strings, which are extracted from the ZINC database [86]. We use the same split as [137], where 5000 SMILES strings are held out for testing. Regarding the syntax constraints, we use the grammar specified in Appendix B.2, which is also the same as [137]. Here the maximum number of decoding steps $T = 278$.

For our SD-VAE, we address some of the most common semantics:

Program semantics We address the following: *a*) variables should be defined before use, *b*) program must return a variable, *c*) number of statements should be less than 10.

Molecule semantics The SMILES semantics we addressed includes: *a*) ringbonds should satisfy cross-serial dependencies, *b*) explicit valence of atoms should not go beyond permitted. For more details about the semantics of SMILES language, please refer to Appendix B.3. By addressing the most common semantics to harness the deep networks, it can greatly reshape the output domain of decoder [105].

6.4.2 Training Details

Since our proposed SD-VAE differentiate itself from previous works (CVAE, GVAE) on the formalization of syntax and semantics, we therefore use the same deep neural network model architecture for a fair comparison. In encoder, we use 3-layer one-dimension convolution neural networks (CNNs) followed by a full connected layer, whose output would be fed into two separate affine layers for producing μ and σ respectively as in reparameterization trick; and in decoder we use 3-layer RNNs followed by a affine layer activated by softmax that gives probability for each production rule. In detail, we use 56 dimensions the latent space and the dimension of layers as the same number as in [137]. As for implementation, we use [137]'s open sourced code for baselines, and implement our model in PyTorch framework [3]. In a 10% validation set we tune the following hyper parameters and report the test result from setting with best valid loss. For a fair comparison, all tunings are

[3] `http://pytorch.org/`

also conducted in the baselines.

We use $\texttt{ReconstructLoss} + \alpha\texttt{KLDivergence}$ as the loss function for training. A natural setting is $\alpha = 1$, but [137] suggested in their open-sourced implementation[4] that using $\alpha = 1/\texttt{LatentDimension}$ would leads to better results. We explore both settings.

6.4.3 Reconstruction Accuracy and Prior Validity

Table 6.1: Reconstructing Accuracy and Prior Validity estimated using Monte Carlo method. Our proposed method (SD-VAE) performance significantly better than existing works. * We also report the reconstruction % grouped by number of statements (3, 4, 5) in parentheses.

	Program		**Zinc SMILES**	
Methods	**Reconstruction %***	**Valid Prior %**	**Reconstruction %**	**Valid Prior %**
SD-VAE	$\mathbf{96.46}$ $(\mathbf{99.90}, \mathbf{99.12}, \mathbf{90.37})$	$\mathbf{100.00}$	$\mathbf{76.2}$	$\mathbf{43.5}$
GVAE	71.83 $(96.30, 77.28, 41.90)$	2.96	53.7	7.2
CVAE	13.79 $(40.46, 0.87, 0.02)$	0.02	44.6	0.7

We use the held-out dataset to measure the reconstruction accuracy of VAEs. For prior validity, we first sample the latent representations from prior distribution, and then evaluate how often the model can decode into a valid structure. Since both encoding and decoding are stochastic in VAEs, we follow the Monte Carlo method used in [137] to do estimation:

a) *reconstruction:* for each of the structured data in the held-out dataset, we encode it 10 times and decoded (for each encoded latent space representation) 25 times, and report the portion of decoded structures that are the same as the input ones; *b*) *validity of prior:* we sample 1000 latent representations $\mathbf{z} \sim \mathcal{N}(O, \mathbf{I})$. For each of them we decode 100 times, and calculate the portion of 100,000 decoded results that corresponds to valid Program or SMILES sequences.

Program We show in the left part of Table 6.1 that our model has near perfect reconstruction rate, and most importantly, a perfect valid decoding program from prior. This huge improvement is due to our model that utilizes the full semantics that previous

[4] `https://github.com/mkusner/grammarVAE/issues/2`

work ignores, thus in theory guarantees perfect valid prior and in practice enables high reconstruction success rate. For a fair comparison, we run and tune the baselines in 10% of training data and report the best result. In the same place we also report the reconstruction successful rate grouped by number of statements. It is shown that our model keeps high rate even with the size of program growing.

SMILES Since the settings are exactly the same, we include CVAE and GVAE results directly from [137]. We show in the right part of Table 6.1 that our model produces a much higher rate of successful reconstruction and ratio of valid prior. Figure 6.4 also demonstrates some decoded molecules from our method. Note that the results we reported have not included the semantics specific to aromaticity into account. If we use the kekulized form of SMILES to train the model, then the valid portion of prior can reach 97.3%.

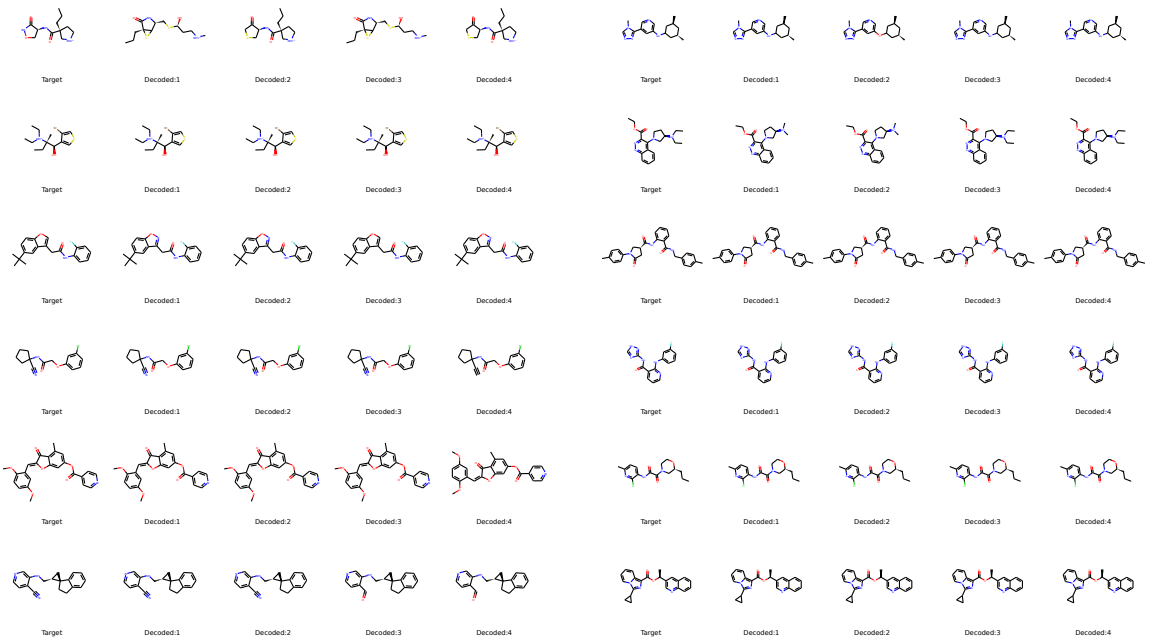


Figure 6.4: Visualization of reconstruction. The first column in each figure presents the target molecules. We first encode the target molecules, then sample the reconstructed molecules from their encoded posterior.

6.4.4 Bayesian Optimization

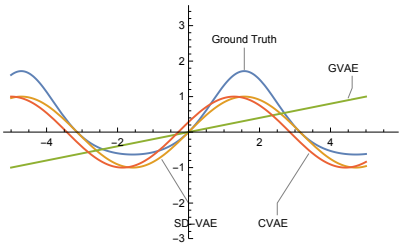
One important application of VAEs is to enable the optimization (*e.g.*, find new structures with better properties) of discrete structures in continuous latent space, and then use decoder to obtain the actual structures. Following the protocol used in [137], we use Bayesian Optimization (BO) to search the programs and molecules with desired properties in latent space. The Bayesian optimization is used for searching latent vectors with desired target property. For example, in symbolic program regression, we are interested in finding programs that can fit the given input-output pairs; in drug discovery, we are aiming at finding molecules with maximum drug likeness. To get a fair comparison with baseline algorithms, we follow the settings used in [137].

Specifically, we first train the variational autoencoder in an unsupervised way. After obtaining the generative model, we encode all the structures into latent space. Then these vectors and corresponding property values (*i.e.*, estimated errors for program, or drug likeness for molecule) are used to train a sparse Gaussian process with 500 inducing points. This is used later for predicting properties in latent space. Next, 5 iterations of batch Bayesian optimization with the expected improvement (EI) heuristic is used for proposing new latent vectors. In each iteration, 50 latent vectors are proposed. After the proposal, the newly found programs/molecules are then added to the batch for next round of iteration.

During the proposal of latent vectors in each iteration, we perform 100 rounds of decoding and pick the most frequent decoded structures. This helps regulate the decoding due to randomness, and increase the chance for baselines algorithms to propose valid ones.

Finding program In this application the models are asked to find the program which is most similar to the ground truth program. Here the distance is measured by $\log(1 + \text{MSE})$, where the MSE (Mean Square Error) calculates the discrepancy of program outputs, given the 1000 different inputs $\mathtt{v0}$ sampled evenly in $[-5, 5]$. In Figure 6.5 we show that our method finds the best program to the ground truth one compared to CVAE and GVAE.

Molecules Here we optimize the drug properties of molecules. In this problem, we



Method	Program	Score
CVAE	`v7=5+v0;v5=cos(v7);return:v5`	0.1742
	`v2=1-v0;v9=cos(v2);return:v9`	0.2889
	`v5=4+v0;v3=cos(v5);return:v3`	0.3043
GVAE	`v3=1/5;v9=-1;v1=v0*v3;return:v3`	0.5454
	`v2=1/5;v9=-1;v7=v2+v2;return:v7`	0.5497
	`v2=1/5;v5=-v2;v9=v5*v5;return:v9`	0.5749
SD-VAE	`v6=sin(v0);v5=exp(3);v4=v0*v6;return:v6`	**0.1206**
	`v5=6+v0;v6=sin(v5);return:v6`	**0.1436**
	`v6=sin(v0);v4=sin(v6);v5=cos(v4);v9=2/v4;return:v4`	**0.1456**
Ground Truth	`v1=sin(v0);v2=exp(v1);v3=v2-1;return:v3`	—

Figure 6.5: On the left are best programs found by each method using Bayesian Optimization. On the right are top 3 closest programs found by each method along with the distance to ground truth (lower distance is better). Both our SD-VAE and CVAE can find similar curves, but our method aligns better with the ground truth. In contrast the GVAE fails this task by reporting trivial programs representing linear functions.

ask the model to optimize for octanol-water partition coefficients (a.k.a *log P*), an important measurement of drug-likeness of a given molecule. As [86] suggests, for drug-likeness assessment *log P* is penalized by other properties including synthetic accessibility score [73]. In Figure 6.6 we show the the top-3 best molecules found by each method, where our method found molecules with better scores than previous works. Also one can see the molecule structures found by SD-VAE are richer than baselines, where the latter ones mostly consist of chain structure.

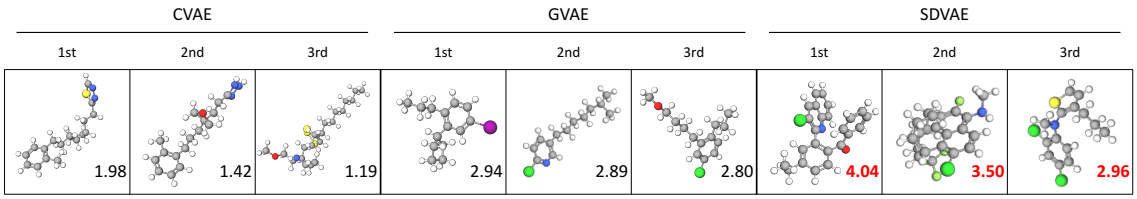


Figure 6.6: Best top-3 molecules and the corresponding scores found by each method using Bayesian Optimization.

6.4.5 Predictive performance of latent representation

The VAEs also provide a way to do unsupervised feature representation learning [86]. In this section, we seek to to know how well our latent space predicts the properties of programs and molecules. After the training of VAEs, we dump the latent vectors of each structured data, and train the sparse Gaussian Process with the target value (namely the

Table 6.2: Predictive performance using encoded mean latent vector. Test LL and RMSE are reported.

	Program		Zinc	
Method	**LL**	**RMSE**	**LL**	**RMSE**
CVAE	-4.943 $\pm$ 0.058	3.757 $\pm$ 0.026	-1.812 $\pm$ 0.004	1.504 $\pm$ 0.006
GVAE	-4.140 $\pm$ 0.038	3.378 $\pm$ 0.020	-1.739 $\pm$ 0.004	1.404 $\pm$ 0.006
SD-VAE	**-3.754 $\pm$ 0.045**	**3.185 $\pm$ 0.025**	**-1.697 $\pm$ 0.015**	**1.366 $\pm$ 0.023**

error for programs and the drug-likeness for molecules) for regression. We test the performance in the held-out test dataset. In Table 6.2, we report the result in Log Likelihood (LL) and Regression Mean Square Error (RMSE), which show that our SD-VAE always produces latent space that are more discriminative than both CVAE and GVAE baselines. This also shows that, with a properly designed decoder, the quality of encoder will also be improved via end-to-end training.

6.4.6 Diversity of generated molecules

Table 6.3: Diversity as statistics from pair-wise distances measured as $1 - s$, where s is one of the similarity metrics. So higher values indicate better diversity. We show $\mathrm{mean} \pm \mathrm{stddev}$ of $\binom{100}{2}$ pairs among 100 molecules. We report results from GVAE and our SD-VAE, because CVAE has very low valid priors and thus failed in this evaluation protocol.

Similarity Metric	MorganFp	MACCS	PairFp	TopologicalFp
GVAE	**0.92 $\pm$ 0.10**	**0.83 $\pm$ 0.15**	0.94 $\pm$ 0.10	0.71 $\pm$ 0.14
SD-VAE	**0.92 $\pm$ 0.09**	**0.83 $\pm$ 0.13**	**0.95 $\pm$ 0.08**	**0.75 $\pm$ 0.14**

Inspired by [22], here we measure the diversity of generated molecules as an assessment of the methods. The intuition is that a good generative model should be able to generate diverse data and avoid mode collapse in the learned space. We conduct this experiment in the SMILES dataset. We first sample 100 points from the prior distribution. For each point, we associate it with a molecule, which is the most frequent occurring valid SMILES decoded (we use 50 decoding attempts since the decoding is stochastic). We then, with one of the several molecular similarity metrics, compute the pairwise similarity and report

the mean and standard deviation in Table 6.3. We see both methods do not have the mode collapse problem, while producing similar diversity scores. It indicates that although our method has more restricted decoding space than baselines, the diversity is not sacrificed. This is because we never rule-out the valid molecules. And a more compact decoding space leads to much higher probability in obtaining valid molecules.

6.4.7 Visualizing the Latent Space

We seek to visualize the latent space as an assessment of how well our generative model is able to produces a coherent and smooth space of program and molecules.

Program Following [30], we visualize the latent space of program by interpolation between two programs. More specifically, given two programs which are encoded to p_a and p_b respectively in the latent space, we pick 9 evenly interpolated points between them. For each point, we pick the corresponding most decoded structure. In Table 6.4 we compare our results with previous works. Our SD-VAE can pass though points in the latent space that can be decoded into valid programs without error and with visually more smooth interpolation than previous works. Meanwhile, CVAE makes both syntactic and semantic errors, and GVAE produces only semantic errors (reference of undefined variables), but still in a considerable amount.

Table 6.4: Interpolation between two valid programs (the top and bottom ones in brown) where each program occupies a row. Programs in red are with syntax errors. Statements in blue are with semantic errors such as referring to unknown variables. Rows without coloring are correct programs. Observe that when a model passes points in its latent space, our proposed SD-VAE enforces both syntactic and semantic constraints while making visually more smooth interpolation. In contrast, CVAE makes both kinds of mistakes, GVAE avoids syntactic errors but still produces semantic errors, and both methods produce subjectively less smooth interpolations.

CVAE	GVAE	SD-VAE
v6=cos(7);v8=exp(9);v2=v8*v0;v9=v2/v6;return:v9	v6=cos(7);v8=exp(9);v2=v8*v0;v9=v2/v6;return:v9	v6=cos(7);v8=exp(9);v2=v8*v0;v9=v2/v6;return:v9
v8=cos(3);v7=exp(7);v5=v7*v0;v9=v9/v6;return:v9	v3=cos(8);v6=exp(9);v6=v8*v0;v9=v2/v6;return:v9	v6=cos(7);v8=exp(9);v2=v8*v0;v9=v2/v6;return:v9
v4=cos(3);v8=exp(3);v2=v2*v0;v9=v8/v6;return:v9	v3=cos(8);v6=2/8;v6=v5*v9;v5=v8v5;return:v5	v6=cos(7);v8=exp(9);v3=v8*v0;v9=v3/v8;return:v9
v6=cos(3);v8=sin(3);v5=v4*1;v5=v3/v4;return:v9	v3=cos(6);v6=2/9;v6=v5+v5;v5=v1+v6;return:v5	v6=cos(7);v8=v6/9;v1=7*v0;v7=v6/v1;return:v7
v9=cos(1);v7=sin(1);v3=v1*5;v9=v9+v4;return:v9	v5=cos(6);v1=2/9;v6=v3+v2;v2=v5-v6;return:v2	v6=cos(7);v8=v6/9;v1=7*v6;v7=v6+v1;return:v7
v6=cos(1);v3=sin(10;;v9=8*v8;v7=v2/v2;return:v9	v5=sin(5);v3=v1/9;v6=v3-v3;v2=v7-v6;return:v2	v6=cos(7);v8=v6/9;v1=7*v8;v7=v6+v8;return:v7
v5=exp(v0;v4=sin(v0);v3=8*v1;v7=v3/v2;return:v9	v1=sin(1);v5=v5/2;v6=v2-v5;v2=v0-v6;return:v2	v6=exp(v0);v8=v6/2;v9=6*v8;v7=v9+v9;return:v7
v5=exp(v0);v1=sin(1);v5=2*v3;v7=v3+v8;return:v7	v1=sin(1);v7=v8/2;v8=v7/v9;v4=v4-v8;return:v4	v6=exp(v0);v8=v6-4;v9=6*v8;v7=v9+v8;return:v7
v4=exp(v0);v1=v7-8;v9=8*v3;v7=v3+v8;return:v7	v8=sin(1);v2=v8/2;v8=v0/v9;v4=v4-v8;return:v4	v6=exp(v0);v8=v6-4;v9=6*v6;v7=v9+v8;return:v7
v4=exp(v0);v9=v6-8;v6=2*v5;v7=v3+v8;return:v7	v6=exp(v0);v2=v6-4;v8=v0*v1;v7=v4+v8;return:v7	v6=exp(v0);v8=v6-4;v4=4*v6;v7=v4+v8;return:v7
v6=exp(v0);v8=v6-4;v4=4*v8;v7=v4+v8;return:v7	v6=exp(v0);v8=v6-4;v4=4*v8;v7=v4+v8;return:v7	v6=exp(v0);v8=v6-4;v4=4*v8;v7=v4+v8;return:v7

SMILES For molecules, we visualize the latent space in 2 dimensions. We first embed a random molecule from the dataset into latent space. Then we randomly generate 2 orthogonal unit vectors A. To get the latent representation of neighborhood, we interpolate the 2-D grid and project back to latent space with pseudo inverse of A. Finally we show decoded molecules. In Figure 6.7, we present two of such grid visualizations. Subjectively compared with figures in [137], our visualization is characterized by having smooth differences between neighboring molecules, and more complicated decoded structures.

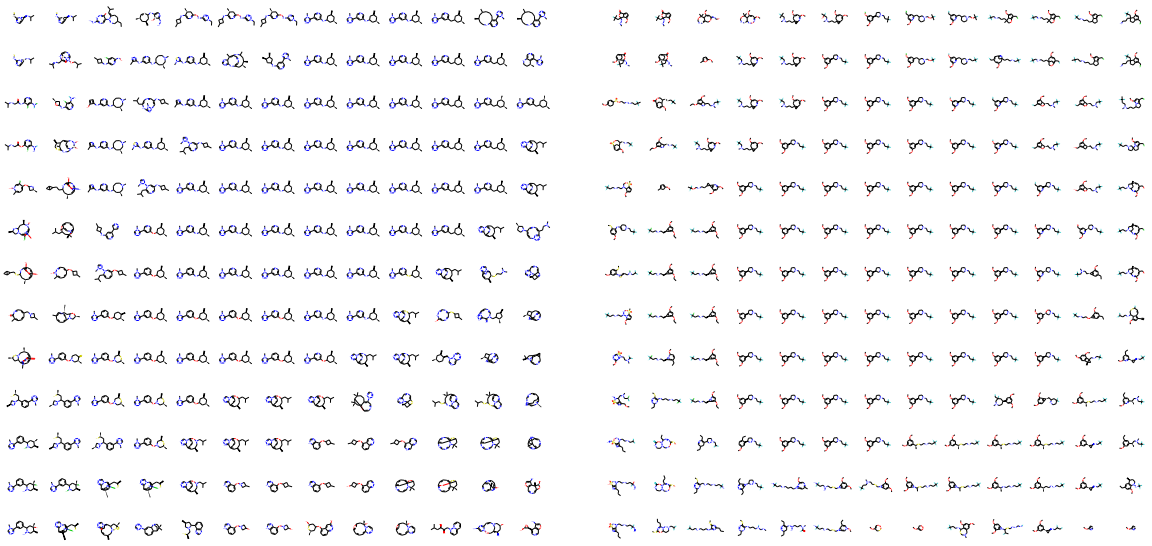


Figure 6.7: Latent Space visualization. We start from the center molecule and decode the neighborhood latent vectors (neighborhood in projected 2D space).

6.5 Summary

In this chapter we propose a new method to tackle the challenge of addressing both syntax and semantic constraints in generative model for structured data. The newly proposed *stochastic lazy attribute* presents a the systematical conversion from offline syntax and semantic check to online guidance for stochastic generation, and empirically shows consistent and significant improvement over previous models with similar computational cost.

So far we have introduced several ways to inspire deep graph learning with the classical algorithms. In next part, we will cover the deep learning enhanced graph algorithms.

PART II: Deep learning enhanced graph algorithms

Some of the human designed algorithms capture the right inductive bias of the problem, but one potential issue is their capacity. For example, in greedy algorithms the human designed heuristics are often not generic, and may fail in some special cases. Fortunately, such algorithms still provide a good framework, and the submodules like the heuristic functions inside the greedy algorithm can be potentially enhanced with deep learning.

In this part, we will focus on a set of problems over graphs that can be better tackled with deep learning enhanced algorithms.

CHAPTER 7

LEARNING HEURISTICS IN GREEDY ALGORITHMS

The design of good heuristics or approximation algorithms for NP-hard combinatorial optimization problems often requires significant specialized knowledge and trial-and-error. Can we automate this challenging, tedious process, and learn the algorithms instead? In many real-world applications, it is typically the case that the same optimization problem is solved again and again on a regular basis, maintaining the same problem structure but differing in the data. This provides an opportunity for learning heuristic algorithms that exploit the structure of such recurring problems. In this chapter, we propose a unique combination of reinforcement learning and graph embedding to address this challenge. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution, and the action is determined by the output of a graph embedding network capturing the current state of the solution. We show that our framework can be applied to a diverse range of optimization problems over graphs, and learns effective algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems.

7.1 Introduction

Combinatorial optimization problems over graphs arising from numerous application domains, such as social networks, transportation, telecommunications and scheduling, are NP-hard, and have thus attracted considerable interest from the theory and algorithm design communities over the years. In fact, of Karp's 21 problems in the seminal paper on reducibility [119], 10 are decision versions of graph optimization problems, while most of the other 11 problems, such as set covering, can be naturally formulated on graphs. Traditional approaches to tackling an NP-hard graph optimization problem have three main flavors: exact algorithms, approximation algorithms and heuristics. Exact algorithms are

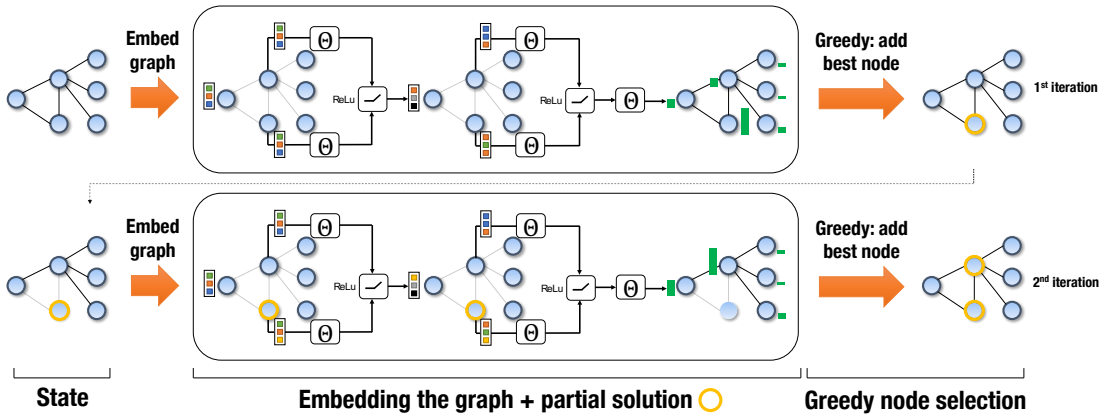


Figure 7.1: Illustration of the proposed framework as applied to an instance of Minimum Vertex Cover. The middle part illustrates two iterations of the graph embedding, which results in node scores (green bars).

based on enumeration or branch-and-bound with an integer programming formulation, but may be prohibitive for large instances. On the other hand, polynomial-time approximation algorithms are desirable, but may suffer from weak optimality guarantees or empirical performance, or may not even exist for inapproximable problems. Heuristics are often fast, effective algorithms that lack theoretical guarantees, and may also require substantial problem-specific research and trial-and-error on the part of algorithm designers.

All three paradigms seldom exploit a common trait of real-world optimization problems: instances of the same type of problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data. That is, in many applications, values of the coefficients in the objective function or constraints can be thought of as being sampled from the same underlying distribution. For instance, an advertiser on a social network targets a limited set of users with ads, in the hope that they spread them to their neighbors; such covering instances need to be solved repeatedly, since the influence pattern between neighbors may be different each time. Alternatively, a package delivery company routes trucks on a daily basis in a given city; thousands of similar optimizations need to be solved, since the underlying demand locations can differ.

Despite the inherent similarity between problem instances arising in the same domain, classical algorithms do not systematically exploit this fact. However, in industrial settings,

a company may be willing to invest in upfront, offline computation and learning if such a process can speed up its real-time decision-making and improve its quality. This motivates the main problem we address:

> **Problem Statement:** Given a graph optimization problem G and a distribution $\mathbb{D}$ of problem instances, can we learn better heuristics that generalize to unseen instances from $\mathbb{D}$?

Recently, there has been some seminal work on using deep architectures to learn heuristics for combinatorial problems, including the Traveling Salesman Problem [230, 20, 89]. However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems. As we show later, these architectures often require a huge number of instances in order to learn to generalize to new ones. Furthermore, existing works typically use the policy gradient for training [20], a method that is not particularly sample-efficient. While the methods in [230, 20] can be used on graphs with different sizes – a desirable trait – they require manual, ad-hoc input/output engineering to do so (e.g. padding with zeros).

In this chapter, we address the challenge of learning algorithms for graph problems using a unique combination of reinforcement learning and graph embedding. The learned policy behaves like a meta-algorithm that incrementally constructs a solution, with the action being determined by a graph embedding network over the current state of the solution. More specifically, our proposed solution framework is different from previous work in the following aspects:

1. Algorithm design pattern. We will adopt a *greedy* meta-algorithm design, whereby a feasible solution is constructed by successive addition of nodes based on the graph structure, and is maintained so as to satisfy the problem's graph constraints. Greedy algorithms are a popular pattern for designing approximation and heuristic algorithms for graph problems. As such, the same high-level design can be seamlessly used for different graph optimization problems.

2. Algorithm representation. We will use a *graph embedding* network, which is called `structure2vec` (S2V) [55], to represent the policy in the greedy algorithm. This novel deep learning architecture over the instance graph "featurizes" the nodes in the graph, capturing the properties of a node in the context of its graph neighborhood. This allows the policy to discriminate among nodes based on their usefulness, and generalizes to problem instances of different sizes. This contrasts with recent approaches [230, 20] that adopt a graph-agnostic sequence-to-sequence mapping that does not fully exploit graph structure.

3. Algorithm training. We will use fitted Q-learning to learn a greedy policy that is parametrized by the graph embedding network. The framework is set up in such a way that the policy will aim to optimize the objective function of the original problem instance *directly*. The main advantage of this approach is that it can deal with delayed rewards, which here represent the remaining increase in objective function value obtained by the greedy algorithm, in a data-efficient way; in each step of the greedy algorithm, the graph embeddings are updated according to the partial solution to reflect new knowledge of the benefit of *each node* to the final objective value. In contrast, the policy gradient approach of [20] updates the model parameters only once w.r.t. the whole solution (e.g. the tour in TSP).

The application of a greedy heuristic learned with our framework is illustrated in Figure 7.1. To demonstrate the effectiveness of the proposed framework, we apply it to three extensively studied graph optimization problems. Experimental results show that our framework, a single meta-learning algorithm, efficiently learns effective heuristics for each of the problems. Furthermore, we show that our learned heuristics preserve their effectiveness even when used on graphs much larger than the ones they were trained on. Since many combinatorial optimization problems, such as the set covering problem, can be explicitly or implicitly formulated on graphs, we believe that our work opens up a new avenue for graph algorithm design and discovery with deep learning.

7.2 Common Formulation for Greedy Algorithms on Graphs

We will illustrate our framework using three optimization problems over weighted graphs. Let $G(V, E, w)$ denote a weighted graph, where V is the set of nodes, E the set of edges and $w : E \to \mathbb{R}^+$ the edge weight function, i.e. $w(u, v)$ is the weight of edge $(u, v) \in E$. These problems are:

- **Minimum Vertex Cover (MVC):** Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$, and $|S|$ is minimized.
- **Maximum Cut (MAXCUT):** Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut-set $\sum_{(u,v)\in C} w(u, v)$ is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.
- **Traveling Salesman Problem (TSP):** Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph *exactly* once.

We will focus on a popular pattern for designing approximation and heuristic algorithms, namely a greedy algorithm. A greedy algorithm will construct a solution by sequentially adding nodes to a partial solution S, based on maximizing some *evaluation function* Q that measures the quality of a node in the context of the current partial solution. We will show that, despite the diversity of the combinatorial problems above, greedy algorithms for them can be expressed using a common formulation. Specifically:

1. A problem instance G of a given optimization problem is sampled from a distribution $\mathbb{D}$, i.e. the V, E and w of the instance graph G are generated according to a model or real-world data.
2. A partial solution is represented as an ordered list $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. Furthermore, we use a vector of binary decision variables x, with each dimension x_v corresponding to a

node $v \in V$, $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or extra feature on v.

3. A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.
4. The quality of a partial solution S is given by an objective function $c(h(S), G)$ based on the combinatorial structure h of S.
5. A generic greedy algorithm selects a node v to add next such that v maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as

$$S \coloneqq (S, v^*), \quad \text{where } v^* \coloneqq \operatorname{argmax}_{v \in \overline{S}} Q(h(S), v), \tag{7.1}$$

and (S, v^*) denotes appending v^* to the end of a list S. This step is repeated until a termination criterion $t(h(S))$ is satisfied.

In our formulation, we assume that the distribution $\mathbb{D}$, the helper function h, the termination criterion t and the cost function c are all given. Given the above abstract model, various optimization problems can be expressed by using different helper functions, cost functions and termination criteria:

- **MVC:** The helper function does not need to do any work, and $c(h(S), G) = -|S|$. The termination criterion checks whether all edges have been covered.
- **MAXCUT:** The helper function divides V into two sets, S and its complement $\overline{S} = V \setminus S$, and maintains a cut-set $C = \{(u, v) \,|\, (u, v) \in E, u \in S, v \in \overline{S}\}$. Then, the cost is $c(h(S), G) = \sum_{(u,v) \in C} w(u, v)$, and the termination criterion does nothing.
- **TSP:** The helper function will maintain a tour according to the order of the nodes in S. The simplest way is to append nodes to the end of partial tour in the same order as S. Then the cost $c(h(S), G) = -\sum_{i=1}^{|S|-1} w(S(i), S(i+1)) - w(S(|S|), S(1))$, and the termination criterion is activated when $S = V$. Empirically, inserting a node u in the

partial tour at the position which increases the tour length the least is a better choice. We adopt this insertion procedure as a helper function for TSP.

An estimate of the quality of the solution resulting from adding a node to partial solution S will be determined by the *evaluation function* Q, which will be learned using a collection of problem instances. This is in contrast with traditional greedy algorithm design, where the *evaluation function* Q is typically hand-crafted, and requires substantial problem-specific research and trial-and-error. In the following, we will design a powerful deep learning parameterization for the evaluation function, $\widehat{Q}(h(S), v; \Theta)$, with parameters Θ.

7.3 Representation: Graph Embedding

Since we are optimizing over a graph G, we expect that the evaluation function $\widehat{Q}$ should take into account the current partial solution S as it maps to the graph. That is, $x_v = 1$ for all nodes $v \in S$, and the nodes are connected according to the graph structure. Intuitively, $\widehat{Q}$ should summarize the state of such a "tagged" graph G, and figure out the value of a new node if it is to be added in the context of such a graph. Here, both the state of the graph and the context of a node v can be very complex, hard to describe in closed form, and may depend on complicated statistics such as global/local degree distribution, triangle counts, distance to tagged nodes, etc. In order to represent such complex phenomena over combinatorial structures, we will leverage a deep learning architecture over graphs, in particular the `structure2vec` of [55], to parameterize $\widehat{Q}(h(S), v; \Theta)$.

7.3.1 Structure2Vec

We first provide an introduction to `structure2vec`. This graph embedding network will compute a p-dimensional feature embedding μ_v for each node $v \in V$, given the current partial solution S. More specifically, `structure2vec` defines the network architecture recursively according to an input graph structure G, and the computation graph of `structure2vec` is inspired by graphical model inference algorithms, where node-

specific tags or features x_v are aggregated recursively according to G's graph topology. After a few steps of recursion, the network will produce a new embedding for each node, taking into account both graph characteristics and long-range interactions between these node features. One variant of the `structure2vec` architecture will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as

$$\mu_v^{(t+1)} \leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u\in\mathcal{N}(v)}, \{w(v,u)\}_{u\in\mathcal{N}(v)}; \Theta\right), \tag{7.2}$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph G, and F is a generic nonlinear mapping such as a neural network or kernel function.

Based on the update formula, one can see that the embedding update process is carried out based on the graph topology. A new round of embedding sweeping across the nodes will start only after the embedding update for all nodes from the previous round has finished. It is easy to see that the update also defines a process where the node features x_v are propagated to other nodes via the nonlinear propagation function F. Furthermore, the more update iterations one carries out, the farther away the node features will propagate and get aggregated nonlinearly at distant nodes. In the end, if one terminates after T iterations, each node embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood as determined by graph topology, the involved node features and the propagation function F. An illustration of two iterations of graph embedding can be found in Figure 7.1.

7.3.2 Parameterizing $\widehat{Q}(h(S), v; \Theta)$

We now discuss the parameterization of $\widehat{Q}(h(S), v; \Theta)$ using the embeddings from S2V. In particular, we design F to update a p-dimensional embedding μ_v as:

$$\mu_v^{(t+1)} \leftarrow \mathrm{relu}\big(\theta_1 x_v + \theta_2 \sum\nolimits_{u\in\mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum\nolimits_{u\in\mathcal{N}(v)} \mathrm{relu}(\theta_4\, w(v,u))\big), \tag{7.3}$$

where $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p \times p}$ and $\theta_4 \in \mathbb{R}^p$ are the model parameters, and relu is the rectified linear unit ($\mathrm{relu}(z) = \max(0, z)$) applied elementwise to its input. The summation over neighbors is one way of aggregating neighborhood information that is invariant to permutations over neighbors. For simplicity of exposition, x_v here is a binary scalar as described earlier; it is straightforward to extend x_v to a vector representation by incorporating any additional useful node information. To make the nonlinear transformations more powerful, we can add some more layers of relu before we pool over the neighboring embeddings μ_u.

Once the embedding for each node is computed after T iterations, we will use these embeddings to define the $\widehat{Q}(h(S), v; \Theta)$ function. More specifically, we will use the embedding $\mu_v^{(T)}$ for node v and the pooled embedding over the entire graph, $\sum_{u \in V} \mu_u^{(T)}$, as the surrogates for v and $h(S)$, respectively, i.e.

$$\widehat{Q}(h(S), v; \Theta) = \theta_5^\top \mathrm{relu}([\theta_6 \sum\nolimits_{u \in V} \mu_u^{(T)}, \theta_7 \, \mu_v^{(T)}]) \tag{7.4}$$

where $\theta_5 \in \mathbb{R}^{2p}$, $\theta_6, \theta_7 \in \mathbb{R}^{p \times p}$ and $[\cdot, \cdot]$ is the concatenation operator. Since the embedding $\mu_u^{(T)}$ is computed based on the parameters from the graph embedding network, $\widehat{Q}(h(S), v)$ will depend on a collection of 7 parameters $\Theta = \{\theta_i\}_{i=1}^7$. The number of iterations T for the graph embedding computation is usually small, such as $T = 4$.

The parameters Θ will be learned. Previously, [55] required a ground truth label for every input graph G in order to train the `structure2vec` architecture. There, the output of the embedding is linked with a softmax-layer, so that the parameters can by trained end-to-end by minimizing the cross-entropy loss. This approach is not applicable to our case due to the lack of training labels. Instead, we train these parameters together *end-to-end* using reinforcement learning.

7.4 Training: Q-learning

We show how reinforcement learning is a natural framework for learning the evaluation function $\widehat{Q}$. The definition of the evaluation function $\widehat{Q}$ naturally lends itself to a *reinforcement learning* (RL) formulation [220], and we will use $\widehat{Q}$ as a model for the state-value function in RL. We note that we would like to learn a function $\widehat{Q}$ *across a set of* m *graphs from distribution* $\mathbb{D}$, $\mathcal{D} = \{G_i\}_{i=1}^m$, with potentially different sizes. The advantage of the graph embedding parameterization in our previous section is that we can deal with different graph instances and sizes seamlessly.

7.4.1 Reinforcement learning formulation

We define the states, actions and rewards in the reinforcement learning framework as follows:

1. *States*: a state S is a sequence of actions (nodes) on a graph G. Since we have already represented nodes in the tagged graph with their embeddings, the state is a vector in p-dimensional space, $\sum_{v \in V} \mu_v$. It is easy to see that this embedding representation of the state can be used across different graphs. The terminal state $\widehat{S}$ will depend on the problem at hand;
2. *Transition*: transition is deterministic here, and corresponds to tagging the node $v \in G$ that was selected as the last action with feature $x_v = 1$;
3. *Actions*: an action v is a node of G that is not part of the current state S. Similarly, we will represent actions as their corresponding p-dimensional node embedding μ_v, and such a definition is applicable across graphs of various sizes;
4. *Rewards*: the reward function $r(S, v)$ at state S is defined as the change in the cost function after taking action v and transitioning to a new state $S' := (S, v)$. That is,

$$r(S, v) = c(h(S'), G) - c(h(S), G), \tag{7.5}$$

Table 7.1: Definition of reinforcement learning components for each of the three problems considered.

Problem	State	Action	Helper function	Reward	Termination
MVC	subset of nodes selected so far	add node to subset	None	-1	all edges are covered
MAXCUT	subset of nodes selected so far	add node to subset	None	change in cut weight	cut weight cannot be improved
TSP	partial tour	grow tour by one node	Insertion operation	change in tour cost	tour includes all nodes

and $c(h(\emptyset), G) = 0$. As such, the *cumulative reward* R of a terminal state $\widehat{S}$ coincides exactly with the objective function value of the $\widehat{S}$, i.e. $R(\widehat{S}) = \sum_{i=1}^{|\widehat{S}|} r(S_i, v_i)$ is equal to $c(h(\widehat{S}), G)$;

5. *Policy*: based on $\widehat{Q}$, a deterministic greedy policy $\pi(v|S) := \text{argmax}_{v' \in \overline{S}} \widehat{Q}(h(S), v')$ will be used. Selecting action v corresponds to adding a node of G to the current partial solution, which results in collecting a reward $r(S, v)$.

Table 7.1 shows the instantiations of the reinforcement learning framework for the three optimization problems considered herein. We let Q^* denote the *optimal* Q-function for each RL problem. Our graph embedding parameterization $\widehat{Q}(h(S), v; \Theta)$ from Section 7.3 will then be a function approximation model for it, which will be learned via n-step Q-learning.

7.4.2 Learning algorithm

In order to perform end-to-end learning of the parameters in $\widehat{Q}(h(S), v; \Theta)$, we use a combination of n-step Q-learning [220] and *fitted Q-iteration* [188], as illustrated in Algorithm 7. We use the term *episode* to refer to a complete sequence of node additions starting from an empty solution, and until termination; a *step* within an episode is a single action (node addition).

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared loss:

$$(y - \widehat{Q}(h(S_t), v_t; \Theta))^2, \tag{7.6}$$

where $y = \gamma \max_{v'} \widehat{Q}(h(S_{t+1}), v'; \Theta) + r(S_t, v_t)$ for a non-terminal state S_t. The n-step

Q-learning helps deal with the issue of *delayed rewards*, where the final reward of interest to the agent is only received far in the future during an episode. In our setting, the final objective value of a solution is only revealed after many node additions. As such, the 1-step update may be too myopic. A natural extension of 1-step Q-learning is to wait n steps before updating the approximator's parameters, so as to collect a more accurate estimate of the future rewards. Formally, the update is over the same squared loss (7.6), but with a different target, $y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta)$. The fitted Q-iteration approach has been shown to result in faster learning convergence when using a neural network as a function approximator [188, 159], a property that also applies in our setting, as we use the embedding defined in Section 7.3.2. Instead of updating the Q-function sample-by-sample as in Equation (7.6), the fitted Q-iteration approach uses *experience replay* to update the function approximator with a batch of samples from a dataset E, rather than the single sample being currently experienced. The dataset E is populated during previous episodes, such that at step $t+n$, the tuple $(S_t, a_t, R_{t,t+n}, S_{t+n})$ is added to E, with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, a_{t+i})$. Instead of performing a gradient step in the loss of the current sample as in (7.6), stochastic gradient descent updates are performed on a random sample of tuples drawn from E.

It is known that *off-policy* reinforcement learning algorithms such as Q-learning can be more sample efficient than their policy gradient counterparts [92]. This is largely due to the fact that policy gradient methods require *on-policy* samples for the new policy obtained after each parameter update of the function approximator.

Algorithm 7 Q-learning for the Greedy Algorithm

Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \text{argmax}_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$
    Add v_t to partial solution: $S_{t+1} \coloneqq (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over (7.6) for B
    end if
  end for
end for
return Θ

7.5 Experimental Evaluation

Instance generation. To evaluate the proposed method against other approximation/heuristic algorithms and deep learning approaches, we generate graph instances for each of the three problems. For the MVC and MAXCUT problems, we generate Erdős-Renyi (ER) [72] and Barabasi-Albert (BA) [3] graphs which have been used to model many real-world networks. For a given range on the number of nodes, e.g. 50-100, we first sample the number of nodes uniformly at random from that range, then generate a graph according to either ER or BA. For the two-dimensional TSP problem, we use an instance generator from the DIMACS TSP Challenge [114] to generate uniformly random or clustered points in the 2-D

grid. We refer the reader to the Appendix C.3.1 for complete details on instance generation. We have also tackled the Set Covering Problem, for which the description and results are deferred to Appendix C.1.

Structure2Vec Deep Q-learning. For our method, S2V-DQN, we use the graph representations and hyperparameters described in Appendix C.3.4. The hyperparameters are selected via preliminary results on small graphs, and then fixed for large ones. Note that for TSP, where the graph is fully-connected, we build the K-nearest neighbor graph ($K = 10$) to scale up to large graphs. For MVC, where we train the model on graphs with up to 500 nodes, we use the model trained on small graphs as initialization for training on larger ones. We refer to this trick as "pre-training", which is illustrated in Figure C.2.

Pointer Networks with Actor-Critic. We compare our method to a method, based on Recurrent Neural Networks (RNNs), which does not make full use of graph structure [20]. We implement and train their algorithm (PN-AC) for all three problems. The original model only works on the Euclidian TSP problem, where each node is represented by its (x, y) coordinates, and is not designed for problems with graph structure. To handle other graph problems, we describe each node by its adjacency vector instead of coordinates. To handle different graph sizes, we use a singular value decomposition (SVD) to obtain a rank-8 approximation for the adjacency matrix, and use the low-rank embeddings as inputs to the pointer network.

Baseline Algorithms. Besides the PN-AC, we also include powerful approximation or heuristic algorithms from the literature. These algorithms are specifically designed for each type of problem:

- **MVC:** *MVCApprox* iteratively selects an uncovered edge and adds both of its endpoints [172]. We designed a stronger variant, called *MVCApprox-Greedy*, that greedily picks the uncovered edge with maximum sum of degrees of its endpoints. Both algorithms are 2-approximations.
- **MAXCUT:** We include *MaxcutApprox*, which maintains the cut set $(S, V \setminus S)$ and moves

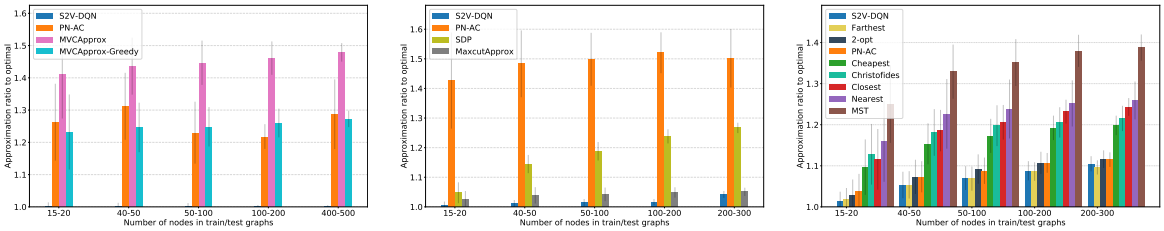
a node from one side to the other side of the cut if that operation results in cut weight improvement [132]. To make *MaxcutApprox* stronger, we greedily move the node that results in the largest improvement in cut weight. A randomized, non-greedy algorithm, referred to as SDP, is also implemented based on [85]; 100 solutions are generated for each graph, and the best one is taken.

- **TSP:** We include the following approximation algorithms: Minimum Spanning Tree (MST), Farthest insertion (Farthest), Cheapest insertion (Cheapest), Closest insertion (Closest), Christofides and 2-opt. We also add the Nearest Neighbor heuristic (Nearest); see [13] for algorithmic details.

Details on Validation and Testing. For S2V-DQN and PN-AC, we use a CUDA K80-enabled cluster for training and testing. Training convergence for S2V-DQN is discussed in Appendix C.3.6. S2V-DQN and PN-AC use 100 held-out graphs for validation, and we report the test results on another 1000 graphs. We use CPLEX[106] to get optimal solutions for MVC and MAXCUT, and Concorde [12] for TSP (details in Appendix C.3.1). All approximation ratios reported in this chapter are with respect to the best (possibly optimal) solution found by the solvers within 1 hour. For MVC, we vary the training and test graph sizes in the ranges {15–20, 40–50, 50–100, 100–200, 400–500}. For MAXCUT and TSP, which involve edge weights, we train up to 200–300 nodes due to the limited computation resource. For all problems, we test on graphs of size up to 1000–1200.

During testing, instead of using Active Search as in [20], we simply use the greedy policy. This gives us much faster inference, while still being powerful enough. We modify existing open-source code to implement both S2V-DQN [1] and PN-AC [2]. Our code is publicly available [3].

[1] https://github.com/Hanjun-Dai/graphnn
[2] https://github.com/devsisters/pointer-network-tensorflow
[3] https://github.com/Hanjun-Dai/graph_comb_opt



(a) MVC BA (b) MAXCUT BA (c) TSP random

Figure 7.2: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

7.5.1 Comparison of solution quality

To evaluate the solution quality on test instances, we use the *approximation ratio* of each method relative to the optimal solution, averaged over the set of test instances. The approximation ratio of a solution S to a problem instance G is $\mathcal{R}(S,G) = \max(\frac{OPT(G)}{c(h(S))}, \frac{c(h(S))}{OPT(G)})$, where $c(h(S))$ is the objective value of solution S, and $OPT(G)$ is the best-known solution value of instance G.

Figure 7.2 shows the average approximation ratio across the three problems; other graph types are in Figure C.1 in the appendix. In all of these figures, a lower approximation ratio is better. Overall, our proposed method, S2V-DQN, performs significantly better than other methods. In MVC, the performance of S2V-DQN is particularly good, as the approximation ratio is roughly 1 and the bar is barely visible.

The PN-AC algorithm performs well on TSP, as expected. Since the TSP graph is essentially fully-connected, graph structure is not as important. On problems such as MVC and MAXCUT, where graph information is more crucial, our algorithm performs significantly better than PN-AC. For TSP, The Farthest and 2-opt algorithm perform as well as S2V-DQN, and slightly better in some cases. However, we will show later that in real-world TSP data, our algorithm still performs better.

7.5.2 Generalization to larger instances

The graph embedding framework enables us to train and test on graphs of different sizes, since the same set of model parameters are used. How does the performance of the learned algorithm using small graphs generalize to test graphs of larger sizes? To investigate this, we train S2V-DQN on graphs with 50–100 nodes, and test its generalization ability on graphs with up to 1200 nodes. Table 7.2 summarizes the results, and full results are in Appendix C.3.3.

Table 7.2: S2V-DQN's generalization ability. Values are average approximation ratios over 1000 test instances. These test results are produced by S2V-DQN algorithms trained on graphs with 50-100 nodes.

Test Size	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
MVC (BA)	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
MAXCUT (BA)	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
TSP (clustered)	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065

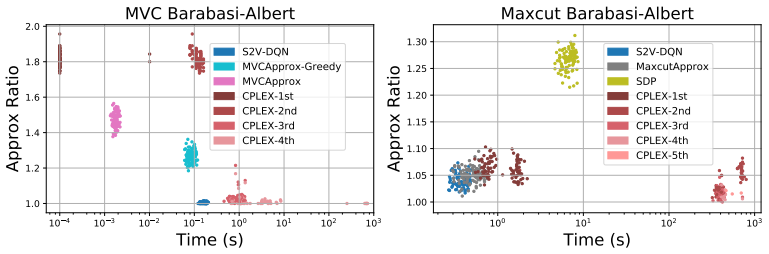
We can see that S2V-DQN achieves a very good approximation ratio. Note that the "optimal" value used in the computation of approximation ratios may not be truly optimal (due to the solver time cutoff at 1 hour), and so CPLEX's solutions do typically get worse as problem size grows. This is why sometimes we can even get better approximation ratio on larger graphs.

7.5.3 Scalability & Trade-off between running time and approximation ratio

To construct a solution on a test graph, our algorithm has polynomial complexity of $O(k|E|)$ where k is number of greedy steps (at most the number of nodes $|V|$) and $|E|$ is number of edges. For instance, on graphs with 1200 nodes, we can find the solution of MVC within 11 seconds using a single GPU, while getting an approximation ratio of 1.0062. For dense graphs, we can also sample the edges for the graph embedding computation to save time, a measure we will investigate in the future.

Figure 7.3 illustrates the approximation ratios of various approaches as a function of

running time. All algorithms report a single solution at termination, whereas CPLEX reports multiple improving solutions, for which we recorded the corresponding running time and approximation ratio. Figure C.3 (Appendix C.3.7) includes other graph sizes and types, where the results are consistent with Figure 7.3.



(a) MVC BA 200-300 (b) MAXCUT BA 200-300

Figure 7.3: Time-approximation trade-off for MVC and MAXCUT. In this figure, each dot represents a solution found for a single problem instance, for 100 instances. For CPLEX, we also record the time and quality of each solution it finds, e.g. CPLEX-1st means the first feasible solution found by CPLEX.

Figure 7.3 shows that, for MVC, we are slightly slower than the approximation algorithms but enjoy a much better approximation ratio. Also note that although CPLEX found the first feasible solution quickly, it also has much worse ratio; the second improved solution found by CPLEX takes similar or longer time than our S2V-DQN, but is still of worse quality. For MAXCUT, the observations are still consistent. One should be aware that sometimes our algorithm can obtain better results than 1-hour CPLEX, which gives ratios below 1.0. Furthermore, sometimes S2V-DQN is even faster than the *MaxcutApprox*, although this comparison is not exactly fair, since we use GPUs; however, we can still see that our algorithm is efficient.

7.5.4 Experiments on real-world datasets

In addition to the experiments for synthetic data, we identified sets of publicly available benchmark or real-world instances for each problem, and performed experiments on them. A summary of results is in Table 7.3, and details are given in Appendix C.2. S2V-DQN significantly outperforms all competing methods for MVC, MAXCUT and TSP.

Table 7.3: Realistic data experiments, results summary. Values are average approximation ratios.

Problem	Dataset	S2V-DQN	Best Competitor	2nd Best Competitor
MVC	MemeTracker	**1.0021**	1.2220 (MVCApprox-Greedy)	1.4080 (MVCApprox)
MAXCUT	Physics	**1.0223**	1.2825 (MaxcutApprox)	1.8996 (SDP)
TSP	TSPLIB	**1.0475**	1.0800 (Farthest)	1.0947 (2-opt)

7.5.5 Discovery of interesting new algorithms

We further examined the algorithms learned by S2V-DQN, and tried to interpret what greedy heuristics have been learned. We found that S2V-DQN is able to discover new and interesting algorithms which intuitively make sense but have not been analyzed before. For instance, S2V-DQN discovers an algorithm for MVC where nodes are selected to balance between their degrees and the connectivity of the remaining graph (Appendix Figures C.4 and C.7). For MAXCUT, S2V-DQN discovers an algorithm where nodes are picked to avoid cancelling out existing edges in the cut set (Appendix Figure C.5). These results suggest that S2V-DQN may also be a good assistive tool for discovering new algorithms, especially in cases when the graph optimization problems are new and less well-studied.

7.6 Summary

We presented an end-to-end machine learning framework for automatically designing greedy heuristics for hard combinatorial optimization problems on graphs. Central to our approach is the combination of a deep graph embedding with reinforcement learning. Through extensive experimental evaluation, we demonstrate the effectiveness of the proposed framework in learning greedy heuristics as compared to manually-designed greedy algorithms. The excellent performance of the learned heuristics is consistent across multiple different problems, graph types, and graph sizes, suggesting that the framework is a promising new tool for designing algorithms for graph problems.

In next chapter, we will show several extensions of this greedy algorithm learning framework to real-world application scenarios.

CHAPTER 8

EXTENSIONS OF LEARNING GREEDY ALGORITHMS OVER GRAPHS

In this chapter, we present two extensions of the deep learning enhanced greedy graph algorithm framework. In Section 8.1, we formulate the graph adversarial attack as a combinatorial optimization problem, in which a hierarchical action variant of S2V-DQN is adopted. In Section 8.2, a set of real-world applications are formulated as exploration over graph, in which a policy-gradient based method is introduced. This chapter will mainly cover the overview of problem modeling and technical contributions, and more details can be found in the original papers [57, 58].

8.1 Hierarchical action space for graph adversarial attack

Deep learning on graph structures has shown exciting results in various applications. However, few attentions have been paid to the robustness of such models, in contrast to numerous research work for image or text adversarial attack and defense. In this chapter, we focus on the adversarial attacks that fool deep learning models by modifying the combinatorial structure of data. We mainly focus on a reinforcement learning based attack method that learns the generalizable attack policy, while only requiring prediction labels from the target classifier. For the further proposed attack methods based on genetic algorithms and gradient descent in the scenario where additional prediction confidence or gradients are available, we ask readers to refer to the original paper.

8.1.1 Problem statement

Given a learned classifier f and an instance from the dataset $(G, c, y) \in \mathcal{D}$, the graph adversarial attacker $g(\cdot, \cdot) : \mathcal{G} \times \mathcal{D} \mapsto \mathcal{G}$ asks to modify the graph $G = (V, E)$ into

$\tilde{G} = (\tilde{V}, \tilde{E})$, such that

$$\begin{aligned} \max_{\tilde{G}} \quad & \mathbb{I}(f(\tilde{G}, c) \neq y) \\ s.t. \quad & \tilde{G} = g(f, (G, c, y)) \\ & \mathcal{I}(G, \tilde{G}, c) = 1. \end{aligned} \tag{8.1}$$

Here $\mathcal{I}(\cdot, \cdot, \cdot) : \mathcal{G} \times \mathcal{G} \times V \mapsto \{0, 1\}$ is an equivalency indicator that tells whether two graphs G and $\tilde{G}$ are equivalent under the classification semantics.

In this paper, we focus on the modifications to the discrete structures. The attacker g is allowed to add or delete edges from G to construct the new graph. Such type of actions are rich enough, since adding or deleting nodes can be performed by a series of modifications to the edges. Also modifying the edges is harder than modifying the nodes, since choosing a node only requires $O(|V|)$ complexity, while naively choosing an edge requires $O(|V|^2)$.

Since the attacker is aimed at fooling the classifier f, instead of actually changing the true label of the instance, the equivalency indicator should be defined first to restrict the modifications an attacker can perform. We use two ways to define the equivalency indicator:

1) Explicit semantics. In this case, a gold standard classifier f^* is assumed to be accessible. Thus the equivalency indicator $\mathcal{I}(\cdot, \cdot, \cdot)$ is defined as:

$$\mathcal{I}(G, \tilde{G}, c) = \mathbb{I}(f^*(G, c) = f^*(\tilde{G}, c)), \tag{8.2}$$

   where $\mathbb{I}(\cdot) \in \{0, 1\}$ is an indicator function.

2) Small modifications. In many cases when explicit semantics is unknown, we will ask the

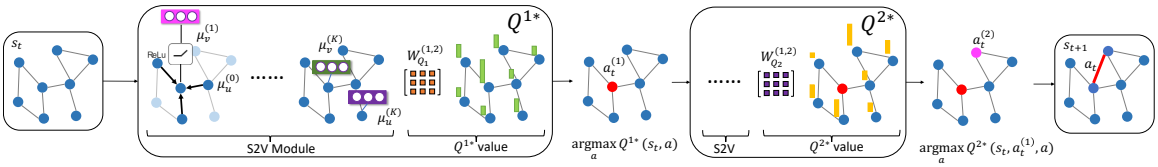


Figure 8.1: Illustration of applying hierarchical Q-function to propose adversarial attack solutions. Here adding a single edge a_t is decomposed into two decision steps $a_t^{(1)}$ and $a_t^{(2)}$, with two Q-functions Q^{1*} and Q^{2*}, respectively.

attacker to make as few modifications as possible within a neighborhood graph:

$$\mathcal{I}(G, \tilde{G}, c) = \mathbb{I}(|(E - \tilde{E}) \cup (\tilde{E} - E)| < m) \cdot \mathbb{I}(\tilde{E} \subseteq \mathcal{N}(G, b))). \tag{8.3}$$

In the above equation, m is the maximum number of edges that allowed to modify, and $\mathcal{N}(G, b) = \{(u, v) : u, v \in V, d^{(G)}(u, v) <= b\}$ defines the b-hop neighborhood graph, where $d^{(G)}(u, v) \in \{1, 2, \ldots\}$ is the distance between two nodes in graph G.

8.1.2 Main formulation

Given an instance (G, c, y) and a target classifier f, we model the attack procedure as a Finite Horizon Markov Decision Process $\mathcal{M}^{(m)}(f, G, c, y)$. The definition of such MDP is as follows:

- **Action** As we mentioned in Sec 8.1, the attacker is allowed to add or delete edges in the graph. So a single action at time step t is $a_t \in \mathcal{A} \subseteq V \times V$. However, simply performing actions in $O(|V|^2)$ space is too expensive. We will shortly show how to use hierarchical action to decompose this action space.
- **State** The state s_t at time t is represented by the tuple $(\hat{G}_t, c)$, where $\hat{G}_t$ is a partially modified graph with some of the edges added/deleted from G.
- **Reward** The purpose of the attacker is to fool the target classifier. So the non-zero

reward is only received at the end of the MDP, with reward being

$$r\big((\tilde{G}, c)\big) = \begin{cases} 1 : f(\tilde{G}, c) \neq y \\ -1 : f(\tilde{G}, c) = y \end{cases} \tag{8.4}$$

In the intermediate steps of modification, no reward will be received. That is to say, $r(s_t, a_t) = 0, \forall t = 1, 2, \ldots, m-1$. In PBA-C setting where the prediction confidence of the target classifier is accessible, we can also use $r\big((\tilde{G}, c)\big) = \mathcal{L}(f(\tilde{G}, c), y)$ as the reward.

- **Terminal** Once the agent modifies m edges, the process stops. For simplicity, we focus on the MDP with fixed length. In the case when fewer modification is enough, we can simply let the agent to modify the dummy edges.

Given the above settings, a sample trajectory from this MDP will be: $(s_1, a_1, r_1, \ldots, s_m, a_m, r_m, s_{m+1})$, where $s_1 = (G, c)$, $s_t = (\hat{G}_t, c), \forall t \in \{2, \ldots, m\}$ and $s_{m+1} = (\tilde{G}, c)$. The last step will have reward $r_m = r(s_m, a_m) = r\big((\tilde{G}, c)\big)$ and all other intermediate rewards are zero: $r_t = 0, \forall t \in \{1, 2, \ldots, m-1\}$. Since this is a discrete optimization problem with a finite horizon, we use Q-learning to learn the MDPs. In our preliminary experiments we also tried with policy optimization methods like Advantage Actor Critic, but found Q-learning works more stable. So below we focus on the modeling with Q-learning.

Q-learning is an off-policy optimization where it fits the Bellman optimality equation directly as below:

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \max_{a'} Q^*(s_{t+1}, a'). \tag{8.5}$$

This implicitly suggests a greedy policy:

$$\pi(a_t | s_t; Q^*) = \arg\max_{a_t} Q^*(s_t, a_t). \tag{8.6}$$

In our finite horizon case, γ is fixed to 1. Note that directly operating the actions in $O(|V|^2)$ space is too expensive for large graphs. Thus we propose to decompose the action $a_t \in V \times V$ into $a_t = (a_t^{(1)}, a_t^{(2)})$, where $a_t^{(1)}, a_t^{(2)} \in V$. Thus a single edge action a_t is decomposed into two ends of this edge. The hierarchical Q-function is then modeled as below:

$$\begin{aligned} Q^{1*}(s_t, a_t^{(1)}) = &\quad \max_{a_t^{(2)}} Q^{2*}(s_t, a_t^{(1)}, a_t^{(2)}) \\ Q^{2*}(s_t, a_t^{(1)}, a_t^{(2)}) = &\quad r\big(s_t, a_t = (a_t^{(1)}, a_t^{(2)})\big) + \\ &\quad \max_{a_{t+1}^{(1)}} Q^{1*}(s_t, a_{t+1}^{(1)}). \end{aligned} \tag{8.7}$$

In the above formulation, Q^{1*} and Q^{2*} are two functions that implement the original Q^*. An action is considered as completed only when a pair of $(a_t^{(1)}, a_t^{(2)})$ is chosen. Thus the reward will only be valid after $a_t^{(2)}$ is made. It is easy to see that such decomposition has the same optimality structure as in Eq (8.5), but making an action would only require $O(2 \times |V|) = O(|V|)$ complexity. Figure 8.1 illustrates this process.

Take a further look at Eq (8.7), since only the reward in last time step is non-zero, and also the budget of modification m is given, we can explicitly unroll the Bellman equations as:

$$\begin{aligned} Q^*_{1,1}(s_1, a_1^{(1)}) = &\quad \max_{a_1^{(2)}} Q^*_{1,2}(s_1, a_1^{(1)}, a_1^{(2)}) \\ Q^*_{1,2}(s_1, a_1^{(1)}, a_1^{(2)}) = &\quad \max_{a_2^{(1)}} Q^*_{2,1}(s_2, a_2^{(1)}) \\ &\cdots \\ Q^*_{m,1}(s_m, a_m^{(1)}) = &\quad \max_{a_m^{(2)}} Q^*_{m,2}(s_m, a_m^{(1)}, a_m^{(2)}) \\ Q^*_{m,2}(s_m, a_m^{(1)}, a_m^{(2)}) = &\quad r(\tilde{G}, c) \end{aligned} \tag{8.8}$$

To make notations compact, we still use $Q^* = \{Q^*_{t,1|2}\}_{t=1}^m$ to denote the Q-function. Since each sample in the dataset defines an MDP, it is possible to learn a separate Q function for each MDP $M_i^{(m)}(f, G_i, c_i, y_i), i = 1, \ldots, N$. However, we here focus on a more

practical and challenging setting, where only one Q^* is learned. The learned Q-function is thus asked to generalize or transfer over all the MDPs:

$$\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \mathbb{E}_{t,a=\arg\max_{a_t} Q^*(a_t|s_t;\boldsymbol{\theta})}[r\big((\tilde{G}_i, c_i)\big)], \tag{8.9}$$

where Q^* is parameterized by $\boldsymbol{\theta}$. From here, we can utilize the technique introduced in Chapter 7 to train the attacking model. The inference framework is depicted in Figure 8.1.

For more details on the experiments, please refer to the original paper [57].

8.2 Optimal graph touring for program and App testing

This section considers the problem of efficient exploration of unseen environments, a key challenge in AI. We propose a 'learning to explore' framework where we learn a policy from a distribution of environments. At test time, presented with an unseen environment from the same distribution, the policy aims to generalize the exploration strategy to visit the maximum number of unique states in a limited number of steps. We particularly focus on environments with graph-structured state-spaces that are encountered in many important real-world applications like software testing and map building. We formulate this task as a reinforcement learning problem where the 'exploration' agent is rewarded for transitioning to previously unseen environment states and employ a graph-structured memory to encode the agent's past trajectory.

8.2.1 Problem statement

We consider two different exploration settings. The first setting concerns *exploration in an unknown environment*, where the agent observes a graph at each step, with each node corresponding to a visited unique *environment state*, and each edge corresponding to an experienced transition. In this setting, the graph grows in size during an episode, and the agent maximizes the speed of this growth.

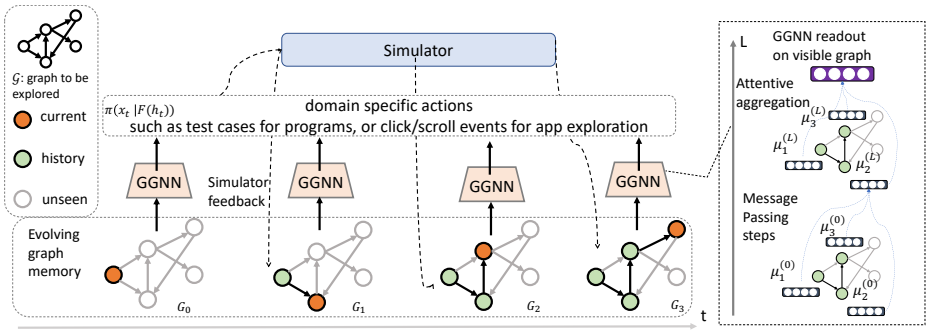


Figure 8.2: Overview of our meta exploration model for exploring a known but complicated graph structured environment. The GGNN [145] module captures the graph structures at each step, and the representations of each step are pooled together to form a representation of the exploration history.

The second setting is about *exploration in a known but complex environment*, and is motivated by program testing. In this setting, we have access to the program source code and thus also its graph structure, where the nodes in the graph correspond to the program branches and edges correspond to the syntactic and semantic relationship between branches. The challenge here is to reason about and understand the graph structure, and come up with the right actions to increase graph coverage. Each action corresponds to a test input which resides in a huge action space and has rich structures. Finding such valuable inputs is highly non-trivial in automated testing literature [84, 202, 203, 36, 141], because of challenges in modeling complex program semantics for precise logical reasoning.

8.2.2 An RL formulation for graph exploration

We formalize both settings with the same formulation. See Figure 8.2 for illustration. At each step t, the agent observes a graph $G_{t-1} = (V_{t-1}, E_{t-1})$ and a coverage mask $c_{t-1} : V_{t-1} \mapsto \{0, 1\}$, indicating which nodes have been covered in the exploration process so far. The agent generates an action x_t, the environment takes this action and returns a new graph $G_t = (V_t, E_t)$ with a new c_t. In the first setting above, the coverage mask c_t is 1 for any node $v \in V_t$ as the graph only contains visited nodes. While in the second setting, the graph G_t is constant from step to step, and the coverage mask $c_t(v) = 1$ if v is covered in the past by some actions and 0 otherwise. We set the initial observation for $t = 0$ to be

c_0 mapping any node to 0, and in the first exploration setting G_0 to be an empty graph.

The exploration process for a graph structured environment can be seen as a finite horizon Markov Decision Process (MDP), with the number of actions or steps T being the budget for exploration.

Action: The space for actions x_t is problem specific. We used the letter x instead of the more common letter a to highlight that these actions are sometimes closer to the typical inputs to a neural network, which lives in an exponentially large space with rich structures, than to the more common fixed finite action spaces in typical RL environments. In particular, for testing programs, each action is a test input to the program, which can be text (sequences of characters) or images (2D array of characters).

Our task is to provide a sequence of T actions $x_1, x_2, \ldots, x_T$ to maximize an exploration objective. An obvious choice is the number of unique nodes (environment states) covered, *i.e.* $\sum_{v \in V_T} c_T(v)$. To handle different graph sizes during training, we further normalize this objective by the maximum possible size of the graph $|\mathcal{V}|$[1], which is the number of nodes in the underlying full graph (for the second exploration setting this is the same as $|V_T|$). We therefore get the objective in (8.10).

$$\max_{\{x_1, x_2, \ldots, x_T\}} \sum_{v \in V_T} c_T(v)/|\mathcal{V}| \qquad (8.10)$$

$$r_t = \sum_{v \in V_t} c_t(v)/|\mathcal{V}| - \sum_{v \in V_{t-1}} c_{t-1}(v)/|\mathcal{V}|, \qquad (8.11)$$

Reward: Given the above objective, we can define the per-step reward r_t as in (8.11). It is easy to verify that $\sum_{t=1}^{T} r_t = \sum_{v \in V_T} c_T(v)/|\mathcal{V}|$, *i.e.*, the cumulative reward of the MDP is the same as the objective in (8.10), as $\sum_{v \in V_0} c_0(v) = 0$. In this definition, the reward at time step t is given to only the additional coverage introduced by the action x_t.

State: Instead of feeding in only the observation (G_t, c_t) at each step to the agent, we use an *agent state* representation that contains the full interaction history in the episode $h_t = \{(x_\tau, G_\tau, c_\tau)\}_{\tau=0}^{t-1}$, with $x_0 = \emptyset$. An agent policy maps each h_t to an action x_t.

[1] When it is unknown, we can simply divide the reward by T to normalize the total reward.

PART III: Towards inductive reasoning with graph structures

So far we have discussed the situations where we have an algorithm at hand, and how can we integrate the deep learning better for the specific tasks. This procedure can be viewed as *deductive reasoning*. In this section, we are taking a different perspective, where we want the neural network to create the algorithm structures. This can be seen as an *inductive reasoning* procedure.

Algorithms are specifications of computation steps with dependencies. From this aspect, the algorithm itself is with graph structure. Thus in this section, we are extending the view of structures from the domain of applications into more generic aspect, *i.e.*, the structures of computation. However, the inductive reasoning is typically hard. Like causal reasoning, there could be multiple plausible explanations of the observations. Sometime it is even hard to find such one.

In the following part of the document, we are taking an initial attempt towards this direction, where in Chapter 9 we first view the logic rules as one of the simplest form of algorithms we are going to reason about. Following this, we extend further in this direction, where we tackle the problem of retrosynthesis in Chapter 10. As will be shown in Chapter 9, the reinforcement learning is a straightforward but not sample efficient way of performing such reasoning. In the retrosynthesis work, we explore more sample efficient way of performing reasoning about the structures.

CHAPTER 9

REASONING THE LOOP INVARIANT FOR PROGRAM VERIFICATION

A fundamental problem in program verification concerns inferring loop invariants. The problem is undecidable and even practical instances are challenging. Inspired by how human experts construct loop invariants, we propose a reasoning framework CODE2INV that constructs the solution by multi-step decision making and querying an external program graph memory block. By training with reinforcement learning, CODE2INV captures rich program features and avoids the need for ground truth solutions as supervision. Compared to previous learning tasks in domains with graph-structured data, it addresses unique challenges, such as a binary objective function and an extremely sparse reward that is given by an automated theorem prover only after the complete loop invariant is proposed. We evaluate CODE2INV on a suite of 133 benchmark problems and compare it to three state-of-the-art systems. It solves 106 problems compared to 73 by a stochastic search-based system, 77 by a heuristic search-based system, and 100 by a decision tree learning-based system. Moreover, the strategy learned can be generalized to new programs: compared to solving new instances from scratch, the pre-trained agent is more sample efficient.

9.1 Introduction

The growing ubiquity and complexity of software has led to a dramatic increase in software bugs and security vulnerabilities that pose enormous costs and risks. Program verification technology enables programmers to prove the absence of such problems at compile-time before deploying their program. One of the main activities underlying this technology involves inferring a *loop invariant*—a logical formula that constitutes an abstract specification of a loop—for each loop in the program. Obtaining loop invariants enables a broad and deep range of correctness and security properties to be proven automatically by a va-

riety of program verification tools spanning type checkers, static analyzers, and theorem provers. Notable examples include Microsoft Code Contracts for .NET programs [74] and the Verified Software Toolchain spanning C source code to machine language [11].

Many different approaches have been proposed in the literature to infer loop invariants. The problem is undecidable, however, and even practical instances are challenging, which greatly limits the benefits of program verification technology. Existing approaches suffer from key drawbacks: they are purely search-based, or they use hand-crafted features, or they are based on supervised learning. The performance of search-based approaches is greatly hindered by their inability to learn from past mistakes. Hand-crafted features limit the space of possible invariants, e.g., [79] is limited to features of the form $x \pm y \leq c$ where c is a constant, and thus cannot handle invariants that involve $x + y \leq z$ for program variables x, y, z. Finally, obtaining ground truth solutions needed by supervised learning is hindered by the undecidability of the loop invariant generation problem.

In this chapter, we propose CODE2INV, an end-to-end learning-based approach to infer loop invariants. CODE2INV has the ability to automatically learn rich latent representations of desirable invariants, and can avoid repeating similar mistakes. Furthermore, it leverages reinforcement learning to discover invariants by partial feedback from trial-and-error, without needing ground truth solutions for training.

The design of CODE2INV is inspired by the reasoning exercised by human experts. Given a program, a human expert first maps the program to a well-organized structural representation, and then composes the loop invariant step by step. Based on such reasoning, different parts of the representation get highlighted at each step. To mimic this procedure, we utilize a graph neural network model (GNN) to construct the structural external memory representation of the program. The multi-step decision making is implemented by an autoregressive model, which queries the external memory using an attention mechanism. The decision at each step is a syntax- and semantics-guided decoder which generates subparts of the loop invariant.

CODE2INV employs a reinforcement learning approach since it is computationally intensive to obtain ground truth solutions. Although reinforcement learning algorithms have shown remarkable success in domains like combinatorial optimization [20, 123], our setting differs in two crucial ways: first, it has a non-continuous objective function (i.e., a proposed loop invariant is correct or not); and second, the positive reward is extremely sparse and given only after the correct loop invariant is proposed, by an automated theorem prover [161]. We therefore model the policy learning as a multi-step decision making process: it provides a fine-grained reward at each step of building the loop invariant, followed by continuous feedback in the last step based on counterexamples collected by the agent itself during trial-and-error learning.

We evaluate CODE2INV on a suite of 133 benchmark problems from recent works [65, 170, 79] and the 2017 SyGuS program synthesis competition [5]. We also compare it to three state-of-the-art systems: a stochastic search-based system C2I [204], a heuristic search-based system LOOPINVGEN [170], and and a decision tree learning-based system ICE-DT [79]. CODE2INV solves 106 problems, versus 73 by C2I, 77 by LOOPINVGEN, and 100 by ICE-DT. Moreover, CODE2INV exhibits better learning, making orders-of-magnitude fewer calls to the theorem prover than these systems.

9.2 Background

We formally define the loop invariant inference and learning problems by introducing Hoare logic [102], which comprises a set of axioms and inference rules for proving program correctness assertions. Let P and Q denote predicates over program variables and let S denote a program. We say that *Hoare triple* $\{P\}\ S\ \{Q\}$ is valid if whenever S begins executing in a state that satisfies P and finishes executing, then the resulting state satisfies Q. We call P and Q the *pre-condition* and *post-condition* respectively of S. Hoare rules allow to derive such triples inductively over the structure of S. The rule most relevant for

our purpose is that for loops:

$$\frac{P \Rightarrow I \quad (\textit{pre}) \qquad \{I \wedge B\}\ S\ \{I\} \quad (\textit{inv}) \qquad (I \wedge \neg B) \Rightarrow Q \quad (\textit{post})}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{Q\}}$$

Predicate I is called a *loop invariant*, an assertion that holds before and after each iteration, as shown in the premise of the rule. We can now formally state the problem:

Problem 1 (Loop Invariant Inference): Given a pre-condition P, a post-condition Q and a program S containing a single loop, can we find a predicate I to make $\{P\}\ S\ \{Q\}$ valid?

Given a candidate loop invariant, it is straightforward for an automated theorem prover such as Z3 [161] to check whether the three conditions denoted *pre*, *inv*, and *post* in the premise of the above rule hold, and thereby prove the property asserted in the conclusion of the rule. If any of the three conditions fails to hold, the theorem prover returns a concrete counterexample witnessing the failure.

The loop invariant inference problem is undecidable. Moreover, even seemingly simple instances are challenging, as we illustrate next using the program in Figure 9.1(a). The goal is to prove that assertion $(y > 0)$ holds at the end of the program, for every input value of integer variable y. In this case, the pre-condition P is `true` since the input value of y is unconstrained, and the post-condition Q is $(y > 0)$, the assertion to be proven. Using predicate $(x < 0 \ \vee \ y > 0)$ as the loop invariant I suffices to prove the assertion, as shown in Figure 9.1(b). Notation $\phi[e/x]$ denotes the predicate ϕ with each occurrence of variable x replaced by expression e. This loop invariant is non-trivial to infer. The reasoning is simple in the case when the input value of y is non-negative, but far more subtle in the case when it is negative: regardless of how negative it is at the beginning, the loop will iterate at least as many times as to make it positive, thereby ensuring the desired assertion upon finishing. Indeed, a state-of-the-art loop invariant generator LoopInvGen [170] crashes on this problem instance after making 1,119 calls to Z3, whereas Code2Inv successfully generates it after only 26 such calls.

$x := -50;$
while $(x < 0)$ {
  $x := x + y;$
  $y := y + 1$ }
assert$(y > 0)$

(a) An example program.

(b) A desirable loop invariant I is a predicate over x, y such that:

$$\forall x, y : \begin{cases} \texttt{true} \Rightarrow I[-50/x] & (\textit{pre}) \\ I \wedge x < 0 \Rightarrow I[(y+1)/y, (x+y)/x] & (\textit{inv}) \\ I \wedge x \geq 0 \Rightarrow y > 0 & (\textit{post}) \end{cases}$$

(c) The desired loop invariant is $(x < 0 \vee y > 0)$.

Figure 9.1: A program with a correctness assertion and a loop invariant that suffices to prove it.

The central role played by loop invariants in program verification has led to a large body of work to automatically infer them. Many previous approaches are based on exhaustive bounded search using domain-specific heuristics and are thereby limited in applicability and scalability [47, 192, 95, 206, 205, 2, 65, 78]. A different strategy is followed by data-driven approaches proposed in recent years [204, 79, 170]. These methods speculatively guess likely invariants from program executions and check their validity. In [79], decision trees are used to learn loop invariants with simple linear features, e.g. $a * x + b * y < c$, where $a, b \in \{-1, 0, 1\}, c \in Z$. In [170], these features are generalized by systematic enumeration. In [204], stochastic search is performed over a set of constraint templates. While such features or templates perform well in specific domains, however, they may fail to adapt to new domains. Moreover, even in the same domain, they do not benefit from past experiences: successfully inferring the loop invariant for one program does not speed up the process for other similar ones. We hereby formulate the second problem we aim to address:

Problem 2 (Loop Invariant Learning): Given a set of programs $\{S_i\} \sim \mathcal{P}$ that are sampled from some unknown distribution $\mathcal{P}$, can we learn from them and generalize the strategy we learned to other programs $\{\tilde{S}_i\}$ that are from the same distribution?

9.3 End-to-End Reasoning Framework

9.3.1 The reasoning process of a human expert

We start out by illustrating how a human expert might typically accomplish the task of inferring a loop invariant. Consider the example in Figure 9.2 chosen from our benchmarks.

```
1   int main() {
2     int x = 0, y = 0;
3     while (*) {
4       if (*) {
5         x++;
6         y = 100;
7       } else if (*) {
8         if (x >= 4) {
9           x++;
10          y++;
11        }
12        if (x < 0) y--;
13      }
14    }
15    assert( x < 4 || y > 2);
16  }
```

Figure 9.2: An example from our benchmarks. $*$ denotes non-deterministic choice.

An expert usually starts by reading the assertion (line 15), which contains variables `x` and `y`, then determines the locations where these two variables are initialized, and then focuses on the locations where they are updated in the loop. Instead of reasoning about the entire assertion at once, an expert is likely to focus on updates to one variable at a time. This reasoning yields the observation that `x` is initialized to zero (line 2) and may get incremented in each iteration (line 5,9). Thus, the sub goal "`x < 4`" may not always hold, given that the loop iterates non-deterministically. This in turn forces the other part "`y > 2`" to be true when "`x >= 4`". The only way `x` can equal or exceed 4 is to execute the first if branch 4 times (line 4-6), during which `y` is set to 100. Now, a natural guess for the loop invariant is "`x < 4 || y >= 100`". The reason for guessing "`y >= 100`" instead of "`y <= 100`" is because part of the proof goal is "`y > 2`". However, this guess will be rejected by the theorem prover. This is because `y` might be decreased by an arbitrary number of times in the third if-branch (line 12), which happens when `x` is less than zero; to avoid that situation, "`x >= 0`" should also be part of the loop invariant. Finally, we have the correct loop invariant:

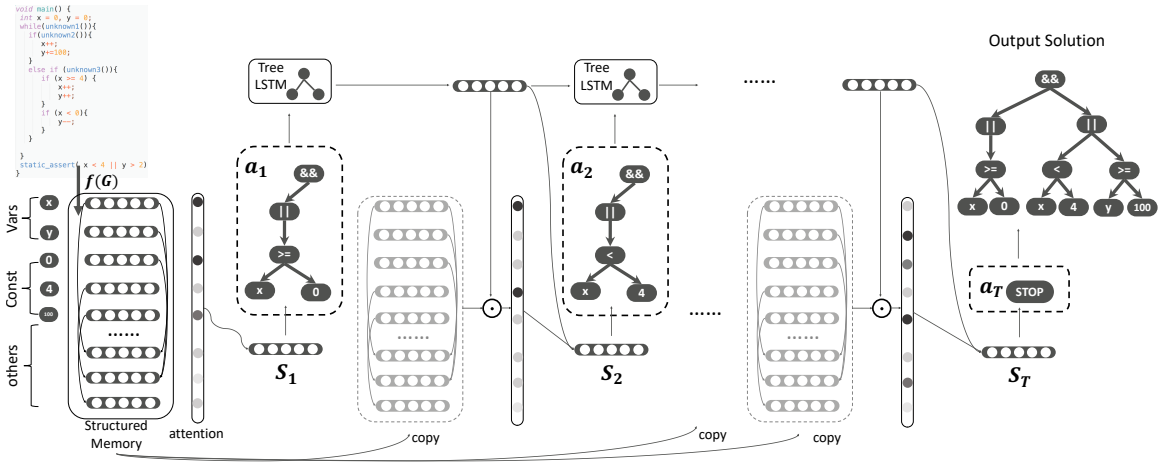


Figure 9.3: Overall framework of neuralizing loop invariant inference.

"`( x >= 0) && (x < 4 || y >= 100)`", which suffices to prove the assertion.

We observe that the entire reasoning process consists of three key components: 1) organize the program in a hierarchical-structured way rather than a sequence of tokens; 2) compose the loop invariant step by step; and 3) focus on a different part of the program at each step, depending on the inference logic, e.g., abduction and induction.

9.3.2 Programming the reasoning procedure with neural networks

We propose to use a neural network to mimic the reasoning used by human experts as described above. The key idea is to replace the above three components with corresponding differentiable modules:

- a structured external memory representation which encodes the program;
- a multi-step autoregressive model for incremental loop invariant construction; and
- an attention component that mimics the varying focus in each step.

As shown in Figure 9.3, these modules together build up the network that constructs loop invariants from programs, while being jointly trained with reinforcement learning described in Section 9.4. At each step, the neural network generates a predicate. Then, given the current generated partial tree, a TreeLSTM module summarizes what have been generated so far, and the summarization is used to read the memory using attention. Lastly,

the summarization together with the read memory is fed into next time step. We next elaborate upon each of these three components.

Structured external memory

The loop invariant is built within the given context of program. Thus it is natural to encode the program as an external memory module. However, in contrast to traditional memory networks [218, 155], where the memory slots are organized as a linear array, the information contained in a program has rich structure. A chain LSTM over program tokens can in principle capture such information but it is challenging for neural networks to understand with limited data. Inspired by [4], we instead use a graph-structured memory representation. Such a representation allows to capture rich semantic knowledge about the program such as its control-flow and data-flow.

More concretely, we first convert a given program into static single assignment (SSA) form [53], and construct a control flow graph, each of whose nodes represents a single program statement. We then transform each node into an abstract syntax tree (AST) representing the corresponding statement. Thus a program can be represented by a graph $G = (V, E)$, where V contains terminals and nonterminals of the ASTs, and the set of edges is denoted as $E = \{(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})\}_{i=1}^{|E|}$. The directed edge $(e_x^{(i)}, e_y^{(i)}, e_t^{(i)})$ starts from node $e_x^{(i)}$ to $e_y^{(i)}$, with $e_t^{(i)} \in \{1, 2, \ldots, K\}$ representing edge type. In our construction, the program graph contains 3 different edge types (and 6 after adding reversed edges).

To convert the graph into vector representation, we follow the general message passing operator introduced in graph neural network (GNN) [193] and its variants [71, 55, 4]. Specifically, the graph network will associate each node $v \in V$ with an embedding vector $\mu_v \in \mathbb{R}^d$. The embedding is updated iteratively using the general neighborhood embedding as follows:

$$\mu_v^{(l+1)} = h(\{\mu_u^{(l)}\}_{u \in \mathcal{N}^k(v), k \in \{1,2,\ldots,K\}}) \tag{9.1}$$

Here $h(\cdot)$ is a nonlinear function that aggregates the neighborhood information to update

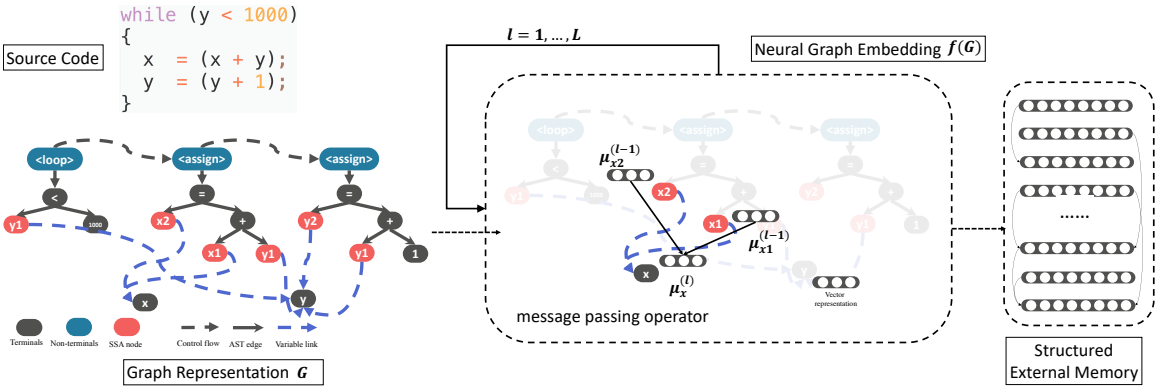


Figure 9.4: Diagram for source code graph as external structured memory. We convert a given program into a graph G, where nodes correspond to syntax elements, and edges indicate the control flow, syntax tree structure, or variable linking. We use embedding neural network to get structured memory $f(G)$.

the embedding. $\mathcal{N}^k(v)$ is the set of neighbor nodes connected to v with edge type k, *i.e.*, $\mathcal{N}^k(v) = \{u|(u, v, k) \in E\}$. Such process will be repeated for L steps, and the node embedding μ_v is set to $\mu_v^{(L)}, \forall v \in V$. Our parameterization takes the edge types into account. The specific parameterization used is shown below:

$$\mu_v^{(l+1),k} = \sigma(\textstyle\sum_{u \in \mathcal{N}^k(v)} \mathbf{W}_2 \mu_u^{(l)}), \forall k \in \{1, 2, \ldots, K\} \tag{9.2}$$

$$\mu_v^{(l+1)} = \sigma(\mathbf{W}_3[\mu_v^{(l+1),1}, \mu_v^{(l+1),2}, \ldots, \mu_v^{(l+1),K}]) \tag{9.3}$$

with the boundary case $\mu_v^{(0)} = \mathbf{W}_1 \mathbf{x}_v$. Here $\mathbf{x}_v$ represents the syntax information of node v, such as token or constant value in the program. Matrices $\mathbf{W}_{1,2,3}$ are learnable model parameters, and σ is some nonlinear activation function. Figure 9.4 shows the construction of graph structured memory using iterative message passing operator in Eq (9.1). $f(G) = \{\mu_v\}_{v \in V}$ denotes the structured memory.

Multi-step decision making process

A loop invariant itself is a mini-program that contains expressions and logical operations. Without loss of generality, we define the loop invariant to be a tree $\mathcal{T}$, in a form with

conjunctions of disjunctions:

$$\mathcal{T} = (\mathcal{T}_1 \,||\, \mathcal{T}_2 \ldots) \,\&\&\, (\mathcal{T}_{t+1} \,||\, \mathcal{T}_{t+2} \ldots) \,\&\&\, \ldots\, (\ldots \mathcal{T}_{T-1} || \,\mathcal{T}_T) \tag{9.4}$$

Each subtree $\mathcal{T}_t$ is a simple logic expression (*i.e.*, `x < y * 2 + 10 - z`). Given this representation form, it is natural to use Markov decision process (MDP) to model this problem, where the corresponding T-step finite horizon MDP is defined as $\mathcal{M}^G = (s_1, a_1, r_1, s_2, a_2, \ldots, s_T)$. Here s_t, a_t, r_t represent the state, action and reward at time step $t = 1, \ldots, T-1$, respectively. Here we describe the state and action used in the inference model, and describe the design of reward and termination in Section 9.4.

action: As defined in Eq (9.4), a loop invariant tree $\mathcal{T}$ consists of multiple subtrees $\{\mathcal{T}_t\}$. Thus we model the action at time step t as $a_t = (op_t, \mathcal{T}_t)$, where op_t can either be $||$ or $\&\&$. That is to say, at each time step, the agent first decides whether to attach the subexpression $\mathcal{T}_t$ to an existing disjunction, or create a new disjunction and add it to the list of conjunctions. We use $\mathcal{T}^{(<t)}$ to denote the partial tree generated by time t so far. So the policy $\pi(\mathcal{T}|G)$ is decomposed into:

$$\pi(\mathcal{T}|G) = \prod_{t=1}^{T} \pi(a_t|\mathcal{T}^{(<t)}, G) = \prod_{t=1}^{T} \pi(op_t, \mathcal{T}_t|\mathcal{T}^{(<t)}, G) \tag{9.5}$$

where $\mathcal{T}^{(<1)}$ is empty at the first step. The generation process of subtree $\mathcal{T}_t$ is also an autoregressive model implemented by LSTM. However, generating a valid program is non-trivial, since strong syntax and semantics constraints should be enforced. Recent advances in neural program synthesis [174, 137] utilize formal language information to help the generation process. Here we use the Syntax-Directed decoder proposed in [59] to guarantee both the syntax and semantics validity. Specifically,

- Syntax constraints: The AST generation follows the grammar of loop invariants described in Eq 9.4. Operators such as `+`, `-`, `*` are non-terminal nodes in the AST while operands such as constants or variables are leaf nodes.

- Semantic constraints: We regulate the generated loop invariant to be meaningful. For example, a valid loop invariant must contains all the variables that appear in the given assertion. Otherwise, the missing variables can take arbitrary values, causing the assertion to be violated. In contrast to offline checking which discards invalid programs after generation, such online regulation restricts the output space of the program generative model, which in turn makes learning efficient.

state: At time step $t = 1$, the state is simply the weighted average of structured memory $f(G)$. At each later time step $t > 1$, the action a_t should be conditioned on graph memory, as well as the partial tree generated so far. Thus $s_t = (G, \mathcal{T}^{(<t)})$.

Memory query with attention

At different time steps of inference, a human usually focuses on different parts of program. Thus the attention mechanism is a good choice to mimic such process. Specifically, at time step t, to summarize what we have generated so far, we use TreeLSTM [223] to embed the partial tree $\mathcal{T}^{(<t)}$. Then the embedding of partial tree $\mathbf{v}_{\mathcal{T}^{(<t)}} = \text{TreeLSTM}(\mathcal{T}^{(<t)})$ is used as the query to read the structured memory $f(G)$. Specifically, $\text{read}(f(G), \mathbf{v}_{\mathcal{T}^{(<t)}}) = \sum_{v \in \mathcal{V}} \alpha_v \mu_v$ and $\alpha_v = \frac{\exp \mu_v^\top \mathbf{v}_{\mathcal{T}^{(<t)}}}{\sum_{v \in V} \exp \mu_v^\top \mathbf{v}_{\mathcal{T}^{(<t)}}}$ are the corresponding attention weights.

9.4 Learning

The undecidability of the loop invariant generation problem hinders the ability to obtain ground truth solutions as supervisions for training. Inspired by recent advances in combinatorial optimization [20, 123], where the agent learns a good policy by trial-and-error, we employ reinforcement learning to learn to propose loop invariants. Ideally, we seek to learn a policy $\pi(\mathcal{T}|G)$ that proposes a correct loop invariant $\mathcal{T}$ for a program graph G. However, directly solving such a model is practically not feasible, since:

- In contrast to problems tackled by existing work, where the objective function is relatively continuous (*e.g.*, tour length of traveling salesman problem), the proposed loop

invariant only has binary objective (*i.e.*, correct or not). This makes the loss surface of the objective function highly non-smooth.

- Finding the loop invariant is a bandit problem where the binary reward is given only after the invariant is proposed. Also, in contrast to two player games [210] where a default policy (e.g., random rollout) can be used to estimate the reward, it is a single player game with an extremely sparse reward.

To tackle the above two challenges, the multi-step decision making model proposed in Section 9.3.2 is used, where a fine-grained reward is also designed for each step. In the last step, a continuous feedback is provided based on the counterexamples collected by the agent itself.

9.4.1 Reinforcement learning setup

Section 9.3.2 defines the state and action representation used for inference. We next describe our setup of the environment which is important to properly train a reinforcement learning agent.

Reward Design

In each intermediate step $t \in 1, \ldots, T-1$, an intermediate reward r_t is given to regulate the generation process. For example, a subexpression should be non-trivial, and it should not contradict $\mathcal{T}^{(<t)}$. In the last step, the generated loop invariant $\mathcal{T}$ is given to a theorem prover, which returns success or failure. In the latter case, the theorem prover also tells which step (*pre*, *inv*, *post*) failed, and provides a counterexample. The failure step can be viewed as a "milestone" of the verification process, providing a coarse granularity feedback. To achieve continuous (i.e. fine granularity) reward within each step, we exploit the counterexamples collected so far. For instance, the ratio of passed examples is a good indicator of the learning progress. Specifically, our reward function consists of two parts: *early reward* and *continuous reward*.

Early reward The early reward constitutes quick feedback obtained by performing light-weight structure checks during the process of loop invariant generation. The goal is to quickly remove meaningless predicates that are trivially true (e.g. "e==e") or false (e.g. "e¡e") or missing variables (e.g. "1¡2"), or simple contradictions like "e1¡e2 && e1¿e2". Early reward is computed at the end of each action; if the partially generated invariant fails to pass the above checks, the generation process terminates immediately by returning a large negative reward -4; otherwise, a positive reward 0.5 is given. Note that one promising future work could be taking advantage of UNSAT cores from counterexamples to identify contradictory parts of the candidate invariant. These contradictory parts will be "non-trivial" contradictions, compared to trivial patterns we have considered.

Continuous reward The goal of continuous reward is to reflect proof progress smoothly. It is computed after the loop invariant is generated and is based on three kinds of counterexamples. Let $ce_{pre}, ce_{inv}, ce_{post}$ denote the sets of counterexamples accumulated so far at the *pre*, *inv*, *post* step, respectively. Similarly, let $pass_{pre}, pass_{inv}, pass_{post}$ be the sets of counterexamples passed by current loop invariant candidate. The continuous reward is modeled as a function that takes these six sets of counterexamples as input and produces a scalar value. We used a simple but effective function, that is, the sum of ratios. Specifically, in the case no new counterexample is introduced, we used the sum of passed ratios of counterexamples:

$$\frac{|pass_{pre}|}{|ce_{pre}|} + \frac{|pass_{inv}|}{|ce_{inv}|} + \frac{|pass_{post}|}{|ce_{post}|}$$

When a new counterexample is returned, we used the staged sum:

$$\frac{|pass_{pre}|}{|ce_{pre}|} + [pass_{pre} = ce_{pre}]\frac{|pass_{inv}|}{|ce_{inv}|} + [pass_{pre} = ce_{pre}][pass_{inv} = ce_{inv}]\frac{|pass_{post}|}{|ce_{post}|}$$

where $[\cdot]$ is Iverson bracket. It examines counterexamples in an *ordered* way (i.e. *pre, inv, post*) so that counterexamples in the next step are considered only after all counterexamples

in the previous step get passed. When we get the highest continuous reward, which is 3, we invoke the theorem prover to verify the current loop invariant candidate; if the theorem prover accepts it, then a correct loop invariant is found; otherwise, a new counterexample is returned, and we recompute the continuous reward according to the above reward function. **termination:** There are several conditions that may trigger the termination of tree generation: (1) the agent executes the "stop" action, see in Figure 9.3; (2) the generated tree has the maximum number of branches allowed; or (3) the agent generates an invalid action.

9.4.2 Training of the learning agent

We use advantage actor critic (A2C) to train the above reinforcement learning policy. Specifically, let $\theta = \{\mathbf{W}_i\}$ be the parameters in graph memory representation $f(\cdot;\theta)$, and ϕ be the parameter used in $\pi(a_t|\mathcal{T}^{(<t)}, G; \phi)$, our objective is to maximize the expected policy reward:

$$\max_{\theta,\phi} \mathbb{E}_{\pi(op_t, \mathcal{T}_t|\mathcal{T}^{(<t)}, G;\phi)}(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(\mathcal{T}^{(<t)}, G; \psi)) \tag{9.6}$$

The baseline function $b(\mathcal{T}^{(<t)}, G; \psi)$ parameterized by ψ is used to estimate the expected return, so as to reduce the variance of policy gradient. $\mathbb{E}_{\pi,t}\| \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(\mathcal{T}^{(<t)}, G; \psi)\|$ is the objective to be minimized over. We simply apply two layer fully connected neural network to predict the expected return. γ is the discounting factor. Since the MDP is finite horizon, we use $\gamma = 1$ to address the long-term reward.

9.5 Experiments

We evaluate CODE2INV on a suite of 133 benchmark programs from recent works [65, 170, 79] and the 2017 SyGuS competition [221].[1] Each program consists of three parts: a number of assumption or assignment statements, one loop which contains nested if-else

[1] Our code and data are publicly available from https://github.com/PL-ML/code2inv

statements with arithmetic operations, and one assertion statement.

By default, the embedding size used throughout this chapter is 128. Batch size is set to 10. To compute the graph structured external memory representation, we run the message passing operator (as described in Equation (9.1)) for 20 steps. Learning rate is set to 0.001 and fixed. We maintain a circular buffer for the counterexamples. The buffer size is set to 100, i.e., we remove the old counterexamples when the buffer size is exceeded (although we seldom reached the 100 limit in our experiments). We use the counterexample to compute continuous feedback only after we collect 5 or more of them.

9.5.1 Dataset

Our dataset is collected from recent literature [65, 79] and the 2017 SyGuS competition [221]. Dillig et al. [65] create a suite of 46 C programs for evaluation of loop invariant inference, on top of which Garg et al. [79] introduce 40 more benchmarks. The 2017 SyGuS competition consists of 74 benchmarks, which is in the SMT-LIB like format [181]. We manually convert benchmarks from SyGus competition to C programs, which have some overlaps with the above two benchmark suite, so we remove the overlapped ones. Figure 9.5 shows some example programs in the SyGuS challenge dataset.

```c
int main() {
  // variable declarations
  int c;
  int n;
  // pre-conditions
  (c = 0);
  assume((n > 0));
  // loop body
  while (unknown()) {
    {
      if ( unknown() ) {
        if ( (c != n) )
        {
        (c  = (c + 1));
        }
      } else {
        if ( (c == n) )
        {
        (c  = 1);
        }
      }

    }

  }
  // post-condition
if ( (c == n) )
assert( (n > -1) );

}
```

```c
int main() {
  // variable declarations
  int x;
  int y;
  // pre-conditions
  (x = 1);
  (y = 0);
  // loop body
  while ((y < 100000)) {
    {
    (x  = (x + y));
    (y  = (y + 1));
    }

  }
  // post-condition
  assert( (x >= y) );
}
```

```c
int main() {
  // variable declarations
  int c;
  int y;
  int z;
  // pre-conditions
  (c = 0);
  assume((y >= 0));
  assume((y >= 127));
  (z = (36 * y));
  // loop body
  while (unknown()) {
    if ( (c < 36) )
    {
    (z  = (z + 1));
    (c  = (c + 1));
    }

  }
  // post-condition
if ( (c < 36) )
assert( (z < 4608) );
}
```

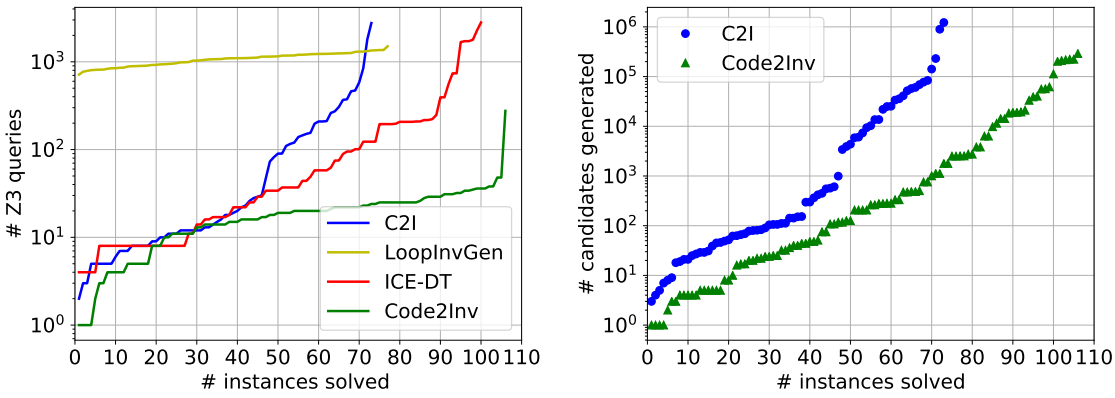
```c
int main() {
  // variable declarations
  int x;
  int y;
  int z1;
  int z2;
  int z3;
  // pre-conditions
  assume((x >= 0));
  assume((x <= 2));
  assume((y <= 2));
  assume((y >= 0));
  // loop body
  while (unknown()) {
    {
    (x  = (x + 2));
    (y  = (y + 2));
    }

  }
  // post-condition
if ( (x == 4) )
assert( (y != 0) );

}
```

Figure 9.5: Examples of programs in SyGuS challenge dataset (after converting to C).

We first evaluate CODE2INV as an out-of-the-box solver, i.e., without any training or



(a) verification cost by each solver (b) sample complexity of C2I and CODE2INV

Figure 9.6: Comparison of CODE2INV with state-of-the-art solvers on benchmark dataset.

fine-tuning with respect to the dataset. We then conduct an ablation study to justify various design choices. Finally, we evaluate the impact of training CODE2INV on a similar dataset.

9.5.2 Finding loop invariants from scratch

In this section, we study the capability of CODE2INV with no training, that is, using it as an out-of-the-box solver. We compare CODE2INV with three state-of-the-art solvers: C2I [204], which is based on stochastic search; LOOPINVGEN [170], which searches a conjunctive normal form over predicates synthesized by an underlying engine, ESCHER [2]; and ICE-DT [79], which learns a decision tree over manually designed features (e.g. predicate templates). The last two solvers are the winners of the invariant synthesis track of the SyGuS 2017 and 2016 competitions, respectively.

A uniform metric is needed to compare the different solvers since they can leverage diverse performance optimizations. For instance, CODE2INV can take advantage of GPUs and TPUs, and C2I can benefit from massive parallelization. Instead of comparing absolute running times, we observe that all four solvers are based on the Z3 theorem prover [161] and rely on the counterexamples from Z3 to adjust their search strategy. Therefore, we compare the number of queries to Z3, which is usually the performance bottleneck for verification tasks. We run all solvers on a single 2.4 GHz AMD CPU core up to 12 hours

and using up to 4 GB memory for each program.

Figure 9.6a shows the number of instances solved by each solver and the corresponding number of queries to Z3. CODE2INV solves the largest number of instances, which is **106**. In contrast, ICE-DT, LOOPINVGEN and C2I solve **100**, **77** and **74** instances, respectively. ICE-DT heavily relies on predicate templates designed by human experts, which are insufficient for 19 instances that are successfully solved by CODE2INV. Furthermore, to solve the same amount of instances, CODE2INV costs orders of magnitude fewer queries to Z3 compared to the other solvers.

We also run CODE2INV using the time limit of one hour from the 2017 SyGuS competition. CODE2INV solves **92** instances within this time limit with the same hardware configuration. While it cannot outperform existing state-of-the-art solvers based on absolute running times, however, we believe its speed can be greatly improved by (1) pre-training on similar programs, which we show in Section 9.5.4; and (2) an optimized implementation that takes advantage of GPUs or TPUs.

CODE2INV is most related to C2Isince both use accumulated counterexamples to adjust the sample distribution of loop invariants. The key difference is that C2I uses MCMC sampling whereas CODE2INV learns using RL. Figure 9.6b shows the sample complexity, i.e., number of candidates generated before successfully finding the desired loop invariant. We observe that CODE2INV needs orders of magnitude less samples which suggests that it is more efficient in learning from failures.

9.5.3 Ablation study

We next study the effectiveness of two key components in our framework via ablation experiments: counterexamples and attention mechanism. We use the same dataset as in Section 9.5.2. Table 9.1 shows our ablation study results. We see that besides providing a continuous reward, the use of counterexamples (CE) significantly reduces the verification cost, i.e., number of Z3 queries. On the other hand, the attention mechanism helps to reduce

the training cost, i.e., number of parameter updates. Also, it helps to reduce the verification cost modestly. CODE2INV achieves the best performance with both components enabled—the configuration used in other parts of our evaluation.

Additionally, to test the effectiveness of neural graph embedding, we study a simpler encoding, that is, viewing a program as a sequence of tokens and encoding the sequence using an LSTM. The performance of this setup is shown in the last row of Table 9.1. With a simple LSTM embedding, CODE2INV solves 13 fewer instances and, moreover, requires significantly more parameter updates.

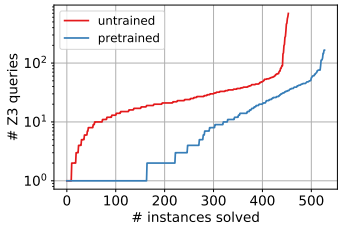
Table 9.1: Ablation study for different configurations of CODE2INV.

configuration	#solved instances	max #Z3 queries	max #parameter updates
without CE, without attention	91	415K	441K
without CE, with attention	94	147K	162K
with CE, without attention	95	392	337K
with CE, with attention	106	276	290K
LSTM embedding + CE + attention	93	32	661K

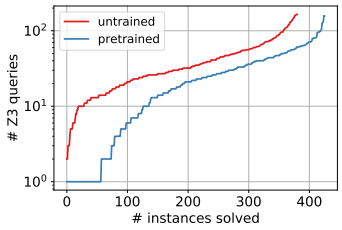
9.5.4 Boosting the search with pre-training

We next address the question: given an agent that is pre-trained on programs $P_{train} = \{p_i\} \sim \mathcal{P}$, can the agent solve new programs $P_{test} = \{\tilde{p}_i\} \sim \mathcal{P}$ faster than solving from scratch? We prepare the training and testing data as follows.

Data augmentation We augment the training dataset on top of the set of programs that CODE2INV can solve. Given a program, we first randomly select an integer K ranging from 1 to 5, which is the number of confounding variables to insert. Then, we initialize each newly created variable with a value ranging from -100 and 100. After that, we insert a statement after each statement in the loop body. The newly inserted statement only uses confounding variables and constants so that any valid loop invariant in the original program is still valid after augmentation. The *lvalue* of the statement is randomly and uniformly sampled from confounding variables, and for the *rvalue* expression of statement, we first randomly choose a depth (either 1 or 2) for the AST tree of the expression, then randomly



(a) with 1 confounding variable



(b) with 5 confounding variables

```
int main()
{
  int a = 0, b = 0, c = 0;

  while(*) {
      a += 1;
      b += 1;
      c -= 1;
  }

  if(a != b) {
    assert (b == -1);
  }
  else{
  }
}
```

(c) attention for invariant a == b

```
int main()
{
  int n, k, c = 0;
  assume (n > 0);

  while(*) {
    if(c < n) {
      c = c + 1;
      k = 2;
    }
    if(c == n) {
      c = 1;
      k = 5;
    }
  }

  if(c == n) {
    assert (n > -1);
  }
}
```

(d) attention for the first part of invariant: c ¿= -1 && n ¿= 1

Figure 9.7: (a) and (b) are verification costs of pre-trained model and untrained model; (c) and (d) are attention highlights for two example programs.

and uniformly pick an operator from $\{ +, -, * \}$, and operands from confounding variables and constants ranging between -100 and 100. For each program and each chosen parameter K, we repeat the above process a 100 times.

Finally, 90% of them serves as P_{train}, and the rest are used for P_{test}. After pre-training the agent on P_{train} for 50 epochs, we save the model and then reuse it for "fine tuning" (or active search [20]), *i.e.*, the agent continues the trial-and-error reinforcement learning, on P_{test}. Figure 9.7a and Figure 9.7b compare the verification costs between the pre-trained model and untrained model on datasets augmented with 1 and 5 confounding variables, respectively. We observe that, on one hand, the pre-trained model has a clear advantage over the untrained model on either dataset; but on the other hand, this gap reduces when more confounding variables are introduced. This result suggests an interesting future research direction: how to design a learning agent to effectively figure out loop invariant related variables from a potentially large number of confounding variables.

9.5.5 Attention visualization

Figure 9.7c and 9.7d show the attention highlights for two example programs. The original highlights are provided on the program graph representation described in Section 9.3.2. We manually converted the graphs back to source code for clarity. Figure 9.7c shows an interesting example for which CODE2INV learns a strategy of showing the assertion is actually not reachable, and thus holds trivially. Figure 9.7d shows another interesting example for which CODE2INV performs a form of abductive reasoning.

9.5.6 Discussion of limitations

We conclude our study with a discussion of limitations. For most of the instances that CODE2INV fails to solve, we observe that the loop invariant can be expressed in a compact disjunctive normal form (DNF) representation, which is more suited for the decision tree learning approach with hand-crafted features. However, CODE2INV is designed to produce loop invariants in the conjunctive normal form (CNF). The reduction of loop invariants from DNF to CNF could incur an exponential blowup in size. An interesting future research direction concerns designing a learning agent that can flexibly switch between these two.

9.6 Summary

We studied the problem of learning loop invariants for program verification. Our proposed end-to-end reasoning framework learns to compose the solution automatically without any supervision. It solves a comparable number of benchmarks as the state-of-the-art solvers while requiring much fewer queries to a theorem prover. Moreover, after being pre-trained, it can generalize the strategy to new instances much faster than starting from scratch.

In the next chapter, we will provide another probabilistic way of learning inductive rules from the data, which is supposed to be more sample efficient than reinforcement learning based methods.

CHAPTER 10
RETROSYNTHESIS PREDICTION WITH CONDITIONAL GRAPH LOGIC NETWORK

Retrosynthesis is one of the fundamental problems in organic chemistry. The task is to identify reactants that can be used to synthesize a specified product molecule. Recently, computer-aided retrosynthesis is finding renewed interest from both chemistry and computer science communities. Most existing approaches rely on template-based models that define subgraph matching rules, but whether or not a chemical reaction can proceed is not defined by hard decision rules. In this work, we propose a new approach to this task using the Conditional Graph Logic Network, a conditional graphical model built upon graph neural networks that learns when rules from reaction templates should be applied, implicitly considering whether the resulting reaction would be both chemically feasible and strategic. We also propose an efficient hierarchical sampling to alleviate the computation cost. While achieving a significant improvement of 8.1% over current state-of-the-art methods on the benchmark dataset, our model also offers interpretations for the prediction.

10.1 Introduction

Retrosynthesis planning is the procedure of identifying a series of reactions that lead to the synthesis of target product. It is first formalized by E. J. Corey [49] and now becomes one of the fundamental problems in organic chemistry. Such problem of "working backwards from the target" is challenging, due to the size of the search space–the vast numbers of theoretically-possible transformations–and thus requires the skill and creativity from experienced chemists. Recently, various computer algorithms [43] work in assistance to experienced chemists and save them tremendous time and effort.

The simplest formulation of retrosynthesis is to take the target product as input and

predict possible reactants [1]. It is essentially the "reverse problem" of reaction prediction. In reaction prediction, the reactants (sometimes reagents as well) are given as the input and the desired outputs are possible products. In this case, atoms of desired products are the subset of reactants atoms, since the side products are often ignored (see Fig 10.1). Thus models are essentially designed to identify this subset in reactant atoms and reassemble them to be the product. This can be treated as a *deductive* reasoning process. In sharp contrast, retrosynthesis is to identify the superset of atoms in target products, and thus is an *abductive* reasoning process and requires "creativity" to be solved, making it a harder problem. Although recent advances in graph neural networks have led to superior performance in reaction prediction [112, 45, 32], such advances do not transfer to retrosynthesis.

Computer-aided retrosynthesis designs have been deployed over the past years since [48]. Some of them are completely rule-based systems [222] and do not scale well due to high computation cost and incomplete coverage of the rules, especially when rules are expert-defined and not algorithmically extracted [43]. Despite these limitations, they are very useful for encoding chemical transformations and easy to interpret. Based on this, the retrosim [46] uses molecule and reaction fingerprint similarities to select the rules to apply for retrosynthesis. Other approaches have used neural classification models for this selection task [200]. On the other hand, recently there have also been attempts to use the sequence-to-sequence model to directly predict SMILES [2] representation of reactants [150, 120] (and for the forward prediction problem, products [198, 199]). Albeit simple and expressive, these approaches ignore the rich chemistry knowledge and thus require huge amount of training. Also such models lack interpretable reasoning behind their predictions.

The current landscape of computer-aided synthesis planning motivated us to pursue an algorithm that shares the interpretability of template-based methods while taking advantage of the scalability and expressiveness of neural networks to learn when such rules apply. In this chapter, we propose *Conditional Graph Logic Network* towards this direction, where

[1] We will focus on this "single step" version of retrosynthesis in this chapter.
[2] https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html.

chemistry knowledge about reaction templates are treated as logic rules and a conditional graphical model is introduced to tolerate the noise in these rules. In this model, the variables are molecules while the synthetic relationships to be inferred are defined among groups of molecules. Furthermore, to handle the potentially infinite number of possible molecule entities, we exploit the neural graph embedding in this model.

Our contribution can be summarized as follows:

1) We propose a new graphical model for the challenging retrosynthesis task. Our model brings both the benefit of the capacity from neural embeddings, and the interpretability from tight integration of probabilistic models and chemical rules.
2) We propose an efficient hierarchical sampling method for approximate learning by exploiting the structure of rules. Such algorithm not only makes the training feasible, but also provides interpretations for predictions.
3) Experiments on the benchmark datasets show a significant 8.1% improvement over existing state-of-the-art methods in top-one accuracy.

Other related work: Recently there have been works using machine learning to enhance the rule systems. Most of them treat the rule selection as multi-class classification [200] or hierarchical classification [18] where similar rules are grouped into subcategories. One potential issue is that the model size grows with the number of rules. Our work directly models the conditional joint probability of both rules and the reactants using embeddings, where the model size is invariant to the rules.

On the other hand, researchers have tried to tackle the even harder problem of multi-step retrosynthesis [201, 197] using single-step retrosynthesis as a subroutine. So our improvement in single-step retrosynthesis could directly transfer into improvement of multi-step retrosynthesis [46].

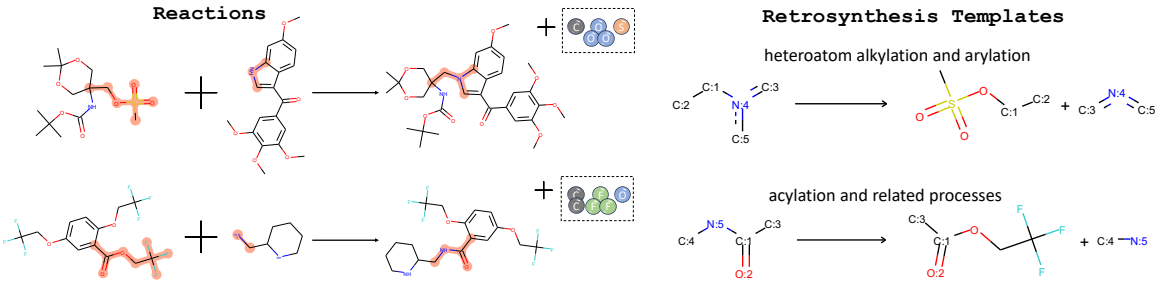


Figure 10.1: Chemical reactions and the retrosynthesis templates. The reaction centers are highlighted in each participant of the reaction. These centers are then extracted to form the corresponding template. Note that the atoms belong to the reaction side products (the dashed box in figure) are missing.

10.2 Background

A chemical reaction can be seen as a transformation from set of N reactant molecules $\{R_i\}_{i=1}^{N}$ to an outcome molecule O. Without loss of generality, we work with single-outcome reactions in this chapter, as this is a standard formulation of the retrosynthetic problem and multi-outcome reactions can be split into multiple single-outcome ones. We refer to the set of atoms changed (e.g., bond being added or deleted) during the reaction as reaction centers. Given a reaction, the corresponding retrosynthesis template T is represented by a subgraph pattern rewriting rule [3]

$$T := o^T \rightarrow r_1^T + r_2^T + \ldots + r_{N(T)}^T, \tag{10.1}$$

where $N(\cdot)$ represents the number of reactant subgraphs in the template, as illustrated in Figure. 10.1. Generally we can treat the subgraph pattern o^T as the extracted reaction center from O, and $r_i^T, i \in 1, 2, \ldots, N(T)$ as the corresponding pattern inside i-th reactant, though practically this will include neighboring structures of reaction centers as well.

We first introduce the notations to represent these chemical entities:

- **Subgraph patterns**: we use lower case letters to represent the subgraph patterns.
- **Molecule**: we use capital letters to represent the molecule graphs. By default, we use

[3] Commonly encoded using SMARTS/SMIRKS patterns

O for an outcome molecule, and R for a reactant molecule, or M for any molecule in general.

- **Set**: sets are represented by calligraphic letters. We use $\mathcal{M}$ to denote the full set of possible molecules, $\mathcal{T}$ to denote all extracted retrosynthetic templates, and $\mathcal{F}$ to denote all the subgraph patterns that are involved in the known templates. We further use $\mathcal{F}_o$ to denote the subgraphs appearing in reaction outcomes, and $\mathcal{F}_r$ to denote those appearing in reactants, with $\mathcal{F} = \mathcal{F}_o \bigcup \mathcal{F}_r$.

Task: Given a production or target molecule O, the goal of a one-step retrosynthetic analysis is to identify a set of reactant molecules $\mathcal{R} \in \mathscr{P}(\mathcal{M})$ that can be used to synthesize the target O. Here $\mathscr{P}(\mathcal{M})$ is the power set of all molecules $\mathcal{M}$.

10.3 Conditional Graph Logic Network

Let $\mathbb{I}[m \subseteq M] : \mathcal{F} \times \mathcal{M} \mapsto \{0, 1\}$ be the predicate that indicates whether subgraph pattern m is a subgraph inside molecule M. This can be checked via subgraph matching. Then the use of a retrosynthetic template $T : o^T \rightarrow r_1^T + r_2^T + \ldots + r_{N(T)}^T$ for reasoning about a reaction can be decomposed into two-step logic. First,

$$\text{I. Match template:} \quad \phi_O(T) := \mathbb{I}[o^T \subseteq O] \cdot \mathbb{I}[T \in \mathcal{T}], \tag{10.2}$$

where the subgraph pattern o^T from the reaction template T is matched against the product O, *i.e.*, o^T is a subgraph of the product O. Second,

$$\text{II. Match reactants:} \quad \phi_{O,T}(\mathcal{R}) := \phi_O(T) \cdot \mathbb{I}[|\mathcal{R}| = N(T)] \cdot \prod_{i=1}^{N(T)} \mathbb{I}[r_i^T \subseteq R_{\pi(i)}], \tag{10.3}$$

where the set of subgraph patterns $\{r_1, \ldots, r_{N(T)}\}$ from the reaction template are matched against the set of reactants $\mathcal{R}$. The logic is that the size of the set of reactant $\mathcal{R}$ has to match the number of patterns in the reaction template T, and there exists a permutation $\pi(\cdot)$ of the elements in the reactant set $\mathcal{R}$ such that each reactant matches a corresponding subgraph

pattern in the template.

Since there will still be uncertainty in whether the reaction is possible from a chemical perspective even when the template matches, we want to capture such uncertainty by allowing each template/or logic reasoning rule to have a different confidence score. More specifically, we will use a template score function $w_1(T, O)$ given the product O, and the reactant score function $w_2(\mathcal{R}, T, O)$ given the template T and the product O. Thus the overall probabilistic models for the reaction template T and the set of molecules $\mathcal{R}$ are designed as

$$\text{I. Match template:} \qquad p(T|O) \propto \exp\left(w_1(T, O)\right) \cdot \phi_O(T), \tag{10.4}$$

$$\text{II. Match reactants:} \qquad p(\mathcal{R}|T, O) \propto \exp\left(w_2(\mathcal{R}, T, O)\right) \cdot \phi_{O,T}(\mathcal{R}). \tag{10.5}$$

Given the above two step probabilistic reasoning models, the joint probability of a single-step retrosythetic proposal using reaction template T and reactant set $\mathcal{R}$ can be written as

$$p\left(\mathcal{R}, T|O\right) \propto \exp\left(w_1\left(T, O\right) + w_2\left(\mathcal{R}, T, O\right)\right) \cdot \phi_O\left(T\right) \phi_{O,T}\left(\mathcal{R}\right), \tag{10.6}$$

In this energy-based model, whether the graphical model (GM) is directed or undirected is a design choice. We will present our directed GM design and the corresponding partition function in Sec 10.4 shortly. We name our model as *Conditional Graph Logic Network (GLN)* (Fig. 10.2), as it is a conditional graphical model defined with logic rules, where the logic variables are graph structures (*i.e.*, molecules, subgraph patterns, *etc.*). In this model, we assume that satisfying the templates is a necessary condition for the retrosynthesis, *i.e.*, $p\left(\mathcal{R}, T|O\right) \neq 0$ only if $\phi_O\left(T\right)$ and $\phi_{O,T}\left(\mathcal{R}\right)$ are nonzero. Such restriction provides sparse structures into the model, and makes this abductive reasoning feasible.

Reaction type conditional model: In some situations when performing the retrosynthetic analysis, the human expert may already have a certain type c of reaction in mind. In

this case, our model can be easily adapted to incorporate this as well:

$$p(\mathcal{R}, T|O, c) \quad \propto \quad \exp\left(w_1\left(T, O\right) + w_2\left(\mathcal{R}, T, O\right)\right) \cdot \phi_O\left(T\right) \phi_{O,T}\left(\mathcal{R}\right) \mathbb{I}[T \in \mathcal{T}_c] \quad (10.7)$$

where $\mathcal{T}_c$ is the set of retrosynthesis templates that belong to reaction type c.

Remark: Similar to the proposed model, the Markov logic network (MLN) [187] is an alternative to introduce uncertainty into logic rules. However, there is significant difference in the way the retrosynthetic templates are treated. The proposed model considers the templates as separate variables that will be inferred for the target molecules together with the reactions. The explicit probabilistic modeling of templates makes it more straightforward to interpret the prediction. The MLN instead sets the logic rules (the templates) as features in the energy-based model, *i.e.*, $p\left(\mathcal{R}|O\right) \propto \exp\left(\sum_{T\in\mathcal{T}} w_{T,O}\phi_O\left(T\right) + w_{T,\mathcal{R},O}\phi_{O,T}\left(\mathcal{R}\right)\right)$, upon which the template inference is not well-defined. Moreover, our model will also lead to efficient sampling and inference, avoiding the MCMC on combinatorial space $\mathscr{P}\left(\mathcal{M}\right)$ in the MLN, which accelerates the model learning.

So in summary:

- GLN is a directed graphical model while MLN is undirected.
- MLN treats the predicates of logic rules as latent variables, and the inference task is to get the posterior of them. While in GLN, the task is the structured prediction, and the predicates are implemented with subgraph matching.
- Due to the above two, GLN can be implemented with efficient hierarchical sampling. However for MLN, generally the expensive MCMC in combinatorial space is needed for both training and inference.

10.4 Model Design

Although the model we defined so far has some nice properties, the design of the components plays a critical role in capturing the uncertainty in the retrosynthesis. We first

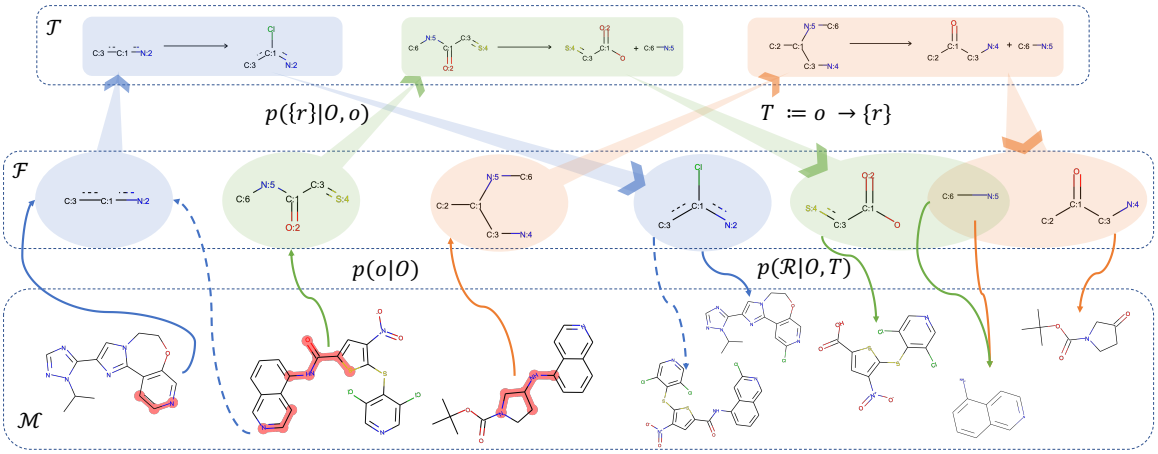


Figure 10.2: Retrosynthesis pipeline with GLN. The three dashed boxes from top to bottom represent set of templates $\mathcal{T}$, subgraphs $\mathcal{F}$ and molecules $\mathcal{M}$. Different colors represent retrosynthesis routes with different templates. The dashed lines represent potentially possible routes that are not observed. Reaction centers in products O are highlighted.

describe a decomposable design of $p(T|O)$ in Sec. 10.4.1, for learning and sampling efficiency consideration; then in Sec. 10.4.2 we describe the parameterization of the scoring functions w_1, w_2 in detail.

10.4.1 Decomposable design of $p(T|O)$

Depending on how specific the reaction rules are, the template set $\mathcal{T}$ could be as large as the total number of reactions in extreme case. Thus directly model $p(T|O)$ can lead to difficulties in learning and inference. By revisiting the logic rule defined in Eq. (10.2), we can see the subgraph pattern o^T plays a critical role in choosing the template. Since we represent the templates as $T = (o^T \rightarrow \{r_i^T\}_{i=1}^{N(T)})$, it is natural to decompose the energy function $w_1(T, O)$ in Eq. (10.4) as $w_1(T, O) = v_1\left(o^T, O\right) + v_2\left(\{r_i^T\}_{i=1}^{N(T)}, O\right)$. Meanwhile, recall the template matching rule is also decomposable, so we obtain the resulting template probability model as:

$$p(T|O) = p(o^T, \{r_i^T\}_{i=1}^{N(T)}|O) \tag{10.8}$$
$$= \frac{1}{Z(O)}\left(\exp\left(v_1(o^T, O)\right) \cdot \mathbb{I}\left[o^T \in O\right]\right)\left(\exp\left(v_2\left(\{r_i^T\}_{i=1}^{N(T)}, O\right)\right) \cdot \mathbb{I}[(o^T \rightarrow \{r_i^T\}_{i=1}^{N(T))\in\mathcal{T}}\right),$$

where the partition function $Z(O)$ is defined as:

$$Z(O) = \sum_{o \in \mathcal{F}} \exp\left(v_1(o, O)\right) \cdot \mathbb{I}\left[o \in O\right] \cdot \left(\sum_{\{r\} \in \mathscr{P}(\mathcal{F})} \exp\left(v_2\left(\{r\}, O\right)\right) \cdot \mathbb{I}[(o \to \{r\}) \in \mathcal{T}]\right) \quad (10.9)$$

Here we abuse the notation a bit to denote the set of subgraph patterns as $\{r\}$.

With such decomposition, we can further speed up both the training and inference for $p(T|O)$, since the number of valid reaction centers per molecule and number of templates per reaction center are much smaller than total number of templates. Specifically, we can sample $T \sim p(T|O)$ by first sampling reaction center $p(o|O) \propto \exp\left(v_1(o, O)\right) \cdot \mathbb{I}\left[o \in O\right]$ and then using $p(\{r\}|O, o) \propto \exp\left(v_2\left(\{r\}, O\right)\right) \cdot \mathbb{I}[(o \to \{r\}) \in \mathcal{T}]$ to choose the subgraph patterns for reactants. In the end we obtain the templated represented as $(o \to \{r\})$.

In the literature there have been several attempts for modeling and learning $p(T|O)$, *e.g.*, multi-class classification [200] or multiscale model with human defined template hierarchy [18]. The proposed decomposable design follows the template specification naturally, and thus has nice graph structure parameterization and interpretation as will be covered in the next subsection.

Finally the directed graphical model design of Eq. (10.6) is written as

$$p(\mathcal{R}, T|O) = \frac{1}{Z(O)Z(T,O)} \exp\left(\left(v_1\left(o^T, O\right) + v_2\left(\left\{r_i^T\right\}_{i=1}^{N(T)}\right) + w_2\left(\mathcal{R}, T, O\right)\right)\right) \cdot \phi_O\left(T\right) \phi_{O,T}\left(\mathcal{R}\right) \quad (10.10)$$

where $Z(T, O) = \sum_{\mathcal{R} \in \mathscr{P}(\mathcal{M})} \exp\left(w_2(\mathcal{R}, T, O)\right) \cdot \phi_{O,T}(\mathcal{R})$ sums over all subsets of molecules.

10.4.2 Graph Neuralization for v_1, v_2 and w_2

Since the arguments of the energy functions w_1, w_2 are molecules, which can be represented by graphs, one natural choice is to design the parameterization based on the recent advances in graph neural networks (GNN). Here we first present a brief review of the general form of GNNs, and then explain how we can utilize them to design the energy functions.

The graph embedding is a function $g : \mathcal{M} \bigcup \mathcal{F} \mapsto \mathbb{R}^d$ that maps a graph into d-

dimensional vector. We denote $G = (\mathcal{V}^G, \mathcal{E}^G)$ as the graph representation of some molecule or subgraph pattern, where $\mathcal{V}^G = \{v_i\}_{i=1}^{|\mathcal{V}^G|}$ is the set of atoms (nodes) and the set of bonds (edges) is $\mathcal{E}^G = \{e_i = (e_i^1, e_i^2)\}_{i=1}^{|\mathcal{E}^G|}$. We represent each undirected bond as two directional edges. Generally, the embedding of the graph is computed through the node embeddings h_{v_i} that are computed in an iterative fashion. Specifically, let $h_{v_i}^0 = x_{v_i}$ initially, where x_{v_i} is a vector of node features, like the atomic number, aromaticity, *etc.* of the corresponding atom. Then the following update operator is applied recursively:

$$h_v^{l+1} = F(x_v, \left\{(h_u^l, x_{u\to v}\right\}_{u\in\mathcal{N}(v)}) \quad \text{where} \quad x_{u\to v} \text{ is the feature of edge } u \to v. \tag{10.11}$$

This procedure repeats for L steps. While there are many design choices for the so-called message passing operator F, we use the `structure2vec` [55] due to its simplicity and efficient `c++` binding with RDKit. Finally we have the parameterization

$$h_v^{l+1} = \sigma(\theta_1 x_v + \theta_2 \sum_{u\in\mathcal{N}(v)} h_u^l + \theta_3 \sum_{u\in\mathcal{N}(v)} \sigma(\theta_4 x_{u\to v})) \tag{10.12}$$

where $\sigma(\cdot)$ is some nonlinear activation function, *e.g.*, `relu` or `tanh`, and $\theta = \{\theta_1, \ldots, \theta_4\}$ are the learnable parameters. Let the node embedding $h_v = h_v^L$ be the last output of F, then the final graph embedding is obtained via averaging over node embeddings: $g(G) = \frac{1}{|\mathcal{V}^G|}\sum_{v\in\mathcal{V}^G} h_v$. Note that attention [229] or other order invariant aggregation can also be used for such aggregation.

With the knowledge of GNN, we introduce the concrete parametrization for each component:

- **Parameterizing v_1:** Given a molecule O, v_1 can be viewed as a scoring function of possible reaction centers inside O. Since the subgraph pattern o is also a graph, we parameterize it with inner product, *i.e.*, $v_1(o, O) = g_1(o)^\top g_2(O)$. Such form can be treated as computing the compatibility between o and O. Note that due to our design choice, $v_1(o, O)$ can be written as $v_1(o, O) = \sum_{v\in\mathcal{V}^O} h_v^\top g_1(o)$. Such form allows us to see the

contribution of compatibility from each atom in O.

- **Parameterizing v_2:** The size of set of subgraph patterns $\left\{r_i^T\right\}_{i=1}^{N(T)}$ varies for different template T. Inspired by the DeepSet [250], we use average pooling over the embeddings of each subgraph pattern to represent this set. Specifically,

$$v_2(\left\{r_i^T\right\}_{i=1}^{N(T)}, O) = g_3(O)^\top \left(\frac{1}{N(T)} \sum_{i=1}^{N(T)} g_4(r_i^T)) \right) \tag{10.13}$$

- **Parameterizing w_2:** This energy function also needs to take the set as input. Following the same design as v_2, we have

$$w_2(\mathcal{R}, T, O) = g_5(O)^\top \left(\frac{1}{|\mathcal{R}|} \sum_{R \in \mathcal{R}} g_6(R) \right). \tag{10.14}$$

Note that our GLN framework isn't limited to the specific parameterization above and is compatible with other parametrizations. For example, one can use condensed graph of reaction [104] to represent $\mathcal{R}$ as a single graph. Other chemistry specialized GNNs [112, 83] can also be easily applied here. For the ablation study on these design choices, please refer to Section 10.6.4.

10.5 MLE with Efficient Inference

Given dataset $\mathcal{D} = \{(O_i, T_i, \mathcal{R}_i)\}_{i=1}^{|\mathcal{D}|}$ with $|\mathcal{D}|$ reactions, we denote the parameters in $w_1(T, O), w_2(T, \mathcal{R}, O)$ as $\Theta = (\theta_1, \theta_2)$, respectively. The maximum log-likelihood estimation (MLE) is a natural choice for parameter estimation. Since $\forall (O, T, \mathcal{R}) \sim \mathcal{D}$, $\phi_O(T) = 1$ and $\phi_{O,T}(\mathcal{R}) = 1$, we have the MLE optimization as

$$\begin{aligned} \max_{\Theta} \ \ell(\Theta) &:= \widehat{\mathbb{E}}_{\mathcal{D}} \left[\log p(\mathcal{R}|T, O) p(T|O) \right] \\ &= \widehat{\mathbb{E}}_{\mathcal{D}} \left[w_1(T, O) + w_2(\mathcal{R}, T, O) - \log Z(O) - \log Z(O, T) \right], \end{aligned} \tag{10.15}$$

The gradient of $\ell(\Theta)$ w.r.t. Θ can be derived[4] as

$$\begin{aligned}\nabla_\Theta \ell(\Theta) &= \widehat{\mathbb{E}}_{\mathcal{D}}\left[\nabla_\Theta w_1(T,O)\right] - \widehat{\mathbb{E}}_O \mathbb{E}_{T|O}\left[\nabla_\Theta w_1(T,O)\right] \\ &\quad + \widehat{\mathbb{E}}_{\mathcal{D}}\left[\nabla_\Theta w_2(\mathcal{R},T,O)\right] - \widehat{\mathbb{E}}_{O,T}\mathbb{E}_{\mathcal{R}|T,O}\left[\nabla_\Theta w_2(\mathcal{R},T,O)\right],\end{aligned} \tag{10.16}$$

where $\mathbb{E}_{T|O}[\cdot]$ and $\mathbb{E}_{\mathcal{R}|O,T}[\cdot]$ stand for the expectation w.r.t. current model $p(T|O)$ and $p(\mathcal{R},T|O)$, respectively. With the gradient estimator (10.16), we can apply the stochastic gradient descent (SGD) algorithm for optimizing (10.15).

Efficient inference for gradient approximation: Since $\mathcal{R} \in \mathscr{P}(\mathcal{M})$ is a combinatorial space, generally the expensive MCMC algorithm is required for sampling from $p(\mathcal{R}|T,O)$ to approximate (10.16). However, this can be largely accelerated by scrutinizing the logic property in the proposed model. Recall that the matching between template and reactants is the necessary condition for $p(\mathcal{R},T|O) \geq 0$ by design.

On the other hand, given O, only a few templates T with reactants $\mathcal{R}$ have nonzero $\phi_O(T)$ and $\phi_{O,T}(\mathcal{R})$. Then, we can sample T and $\mathcal{R}$ by importance sampling on *restricted supported templates* instead of MCMC over $\mathscr{P}(\mathcal{M})$. Rigorously, given O, we denote the matched templates as $\mathcal{T}_O$ and the matched reactants based on T as $\mathcal{R}_{T,O}$, where

$$\mathcal{T}_O = \{T : \phi_O(T) \neq 0, \forall T \in \mathcal{T}\} \text{ and } \mathcal{R}_{T,O} = \{\mathcal{R} : \phi_{O,T}(\mathcal{R}) \neq 0, \forall \mathcal{R} \in \mathscr{P}(\mathcal{M})\} \tag{10.17}$$

Then, the importance sampling leads to an *unbiased* gradient approximation $\widehat{\nabla}_\Theta \ell(\Theta)$ as illustrated in Algorithm 8. To make the algorithm more efficient in practice, we have adopted the following accelerations:

- **1)** Decomposable modeling of $p(T|O)$ as described in Sec. 10.4.1;
- **2)** Cache the computed $\mathcal{T}_O$ and $\mathcal{R}(T,O)$ in advance.

In a dataset with 5×10^4 reactions, $|\mathcal{T}_O|$ is about 80 and $|\mathcal{R}_{T,O}|$ is roughly 10 on average. Therefore, we reduce the actual computational cost to a manageable constant. We further

[4]We adopt the conventions $0 \log 0 = 0$ [50], which is justified by continuity since $x \log x \to 0$ as $x \to 0$.

Algorithm 8 Importance Sampling for $\widehat{\nabla}_{\Theta}\ell\left(\Theta\right)$

1: Input $(\mathcal{R}, T, O) \sim \mathcal{D}$, $p\left(\mathcal{R}|T, O\right)$ and $p\left(T|O\right)$.
2: Construct $\mathcal{T}_O$ according to $\phi_O\left(T\right)$.
3: Sample $\tilde{T} \propto \exp\left(w_1\left(T, O\right)\right),\ \forall T \in \mathcal{T}_O$ in hierarchical way, as in Sec. 10.4.1.
4: Construct $\mathcal{R}_{T,O}$ according to $\phi_{O,T}\left(\mathcal{R}\right)$.
5: Sample $\tilde{\mathcal{R}} \propto \exp\left(w_2\left(\mathcal{R}, T, O\right)\right)$.
6: Compute stochastic approximation $\widehat{\nabla}_{\Theta}\ell\left(\Theta\right)$ with sample $\left(\mathcal{R}, T, \tilde{\mathcal{R}}, \tilde{T}, O\right)$ by (10.16).

reduce the computation cost of sampling by generating the T and $\mathcal{R}$ uniformly from the support. Although these samples only cover the support of the model, we avoid the calculation of the forward pass of neural networks, achieving better computational complexity. In our experiment, such an approximation already achieves state-of-the-art results. We would expect recent advances in energy based models would further boost the performance, which we leave as future work to investigate.

Remark on $\mathcal{R}_{T,O}$: Note that to get all possible sets of reactants that match the reaction template T and product O, we can efficiently use graph edit tools without limiting the reactants to be known in the dataset. This procedure works as follows: given a template $T = o^T \rightarrow r_1^T + \ldots + r_N^T$,

1) Enumerate all matches between subgraph pattern o^T and target product O.

2) Instantiate a copy of the reactant atoms according to $r_1^T, \ldots, r_N^T$ for each match.

3) Copy over all of the connected atoms and atom properties from O.

This process is a routine in most Cheminformatics packages. In this chapter `runReactants` from RDKit with the improvement of stereochemistry handling [5] is used to realize this.

Further acceleration via beam search: Given a product O, the prediction involves finding the pair $(\mathcal{R}, T)$ that maximizes $p(\mathcal{R}, T|O)$. One possibility is to first enumerate $T \in T(O)$ and then $\mathcal{R} \in \mathcal{R}_{T,O}$. This is acceptable by exploiting the sparse support property induced by logic rules.

A more efficient way is to use beam search with size k. Firstly we find k reaction

[5] https://github.com/connorcoley/rdchiral.

centers $\{o_i\}_{i=1}^k$ with top $v_1(o, O)$. Next for each $o \in \{o_i\}_{i=1}^k$ we score the corresponding $v_2(\{r\}, O) \cdot \mathbb{I}[(o \to \{r\}) \in \mathcal{T}]$. In this stage the top k pairs $\{(o_{T_j}, \{r_i^{T_j}\})\}_{j=1}^k$ (*i.e.*, the templates) that maximize $v_1(o|O) + v_2(\{r\}, O)$ are kept. Finally using these templates, we choose the best $\mathcal{R} \in \bigcup_{j=1}^k \mathcal{R}_{T_j,O}$ that maximizes total score $w_1(T, O) + w_2(\mathcal{R}, T, O)$. Fig. 10.2 provides a visual explanation.

10.6 Experiment

Dataset: We mainly evaluate our method on a benchmark dataset named USPTO-50k, which contains 50k reactions of 10 different types in the US patent literature. We use exactly the same training/validation/test splits as [46], which contain 80%/10%/10% of the total 50k reactions. Table 10.1 contains the detailed information about the benchmark. Additionally, we also build a dataset from the entire USPTO 1976-2016 to verify the scalability of our method.

Baselines: Baseline algorithms consist of rule-based ones and neural network-based ones, or both. The expertSys is an expert system based on retrosynthetic reaction rules, where the rule is selected according to the popularity of the corresponding reaction type. The seq2seq [150] and transformer [120] are neural seq2seq learning model [219] implemented with LSTM [103] or Transformer [228]. These models encode the canonicalized SMILES representation of the target compound as input, and directly output canonical SMILES of reactants. We also include some data-driven template-based models. The retrosim [46] uses direct calculation of molecular similarities to rank the rules and resulting reactants. The neuralsym [200] models $p(T|O)$ as multi-class classification using MLP. All the results except neuralsym are obtained from their original reports, since we have the same experiment setting. Since neuralsym is not open-source, we reimplemented it using their best reported ELU512 model with the same method for parameter tuning.

Evaluation metric: The evaluation metric we used is the top-k exact match accuracy, which is commonly used in the literature. This metric compares whether the predicted set

of reactants are *exactly* the same as ground truth reactants. The comparison is performed between canonical SMILES strings generated by RDKit.

Setup of GLN: We use rdchiral [44] to extract the retrosynthesis templates from the training set. After removing duplicates, we obtained 11,647 unique template rules in total for USPTO-50k. These rules represent 93.3% coverage of the test set. That is to say, for each test instance we try to apply these rules and see if any of the rules gives exact match. Thus this is the theoretical upper bound of the rule-based approach using this particular degree of specificity, which is high enough for now. For more information about the statistics of these rules, please refer to Table 10.2.

We train our model for up to 150k updates with batch size of 64. It takes about 12 hours to train with a single GTX 1080Ti GPU. We tune embedding sizes in $\{128, 256\}$, GNN layers $\{3, 4, 5\}$ and GNN aggregation in $\{\texttt{max}, \texttt{mean}, \texttt{sum}\}$ using validation set. The preprocessing of $\mathcal{T}_O$ and $\mathcal{R}_{T,O}$ is relatively expensive, since theoretically the subgraph isomorphism check is NP-hard. However, since the processing is embarrassingly parallelizable, it took about 1 hour on a cluster with 48 CPU cores for 50k reactions. We implement the entire model using pytorch. The optimizer we used is Adam [128] with a fixed learning rate of $1e-3$ and a gradient clip of 5.0. In all the experiments, the graph embedding module is implemented using s2v [55]. The best embedding size we used has size of 256 for representing each molecule or subgraph structure, and `relu` is used as nonlinear activation function. For the aggregation used in $g(\cdot)$, in DeepSet module used for representation of $r_i^{T\ N(T)}_{i=1}$ for a specific T, or in DeepSet module for molecule set $\mathcal{R}$, we tried $\{\texttt{max}, \texttt{sum}, \texttt{average}\}$-pooling, and found the performance is about the same. We use $average$-pooling since it offers the scoring of each node embedding within the graphs. The visualization in Fig 10.5 relies on this trick.

Our code is released at https://github.com/Hanjun-Dai/GLN.

Table 10.1: Dataset information.

USPTO 50k	
# train	40,008
# val	5,001
# test	5,007
# rules	11,647
# reaction types	10

Table 10.2: Reaction and template set information.

Rule coverage	93.3%
# unique centers	9,078
Avg. # centers per mol	29.31
Avg. # rules per mol	83.85
Avg. # reactants	1.71

Table 10.3: Top-k exact match accuracy.

methods	Top-k accuracy %					
	1	3	5	10	20	50
Reaction class unknown						
transformer[120]	37.9	57.3	62.7	/	/	/
retrosim[46]	37.3	54.7	63.3	74.1	82.0	85.3
neuralsym[200]	44.4	65.3	72.4	78.9	82.2	83.1
GLN	**52.5**	**69.0**	**75.6**	**83.7**	**89.0**	**92.4**
Reaction class given as prior						
expertSys[150]	35.4	52.3	59.1	65.1	68.6	69.5
seq2seq[150]	37.4	52.4	57.0	61.7	65.9	70.7
retrosim[46]	52.9	73.8	81.2	88.1	91.8	92.9
neuralsym[200]	55.3	76.0	81.4	85.1	86.5	86.9
GLN	**64.2**	**79.1**	**85.2**	**90.0**	**92.3**	**93.2**

10.6.1 Main results

We present the top-k exact match accuracy in Table 10.3, where $k \in \{1, 3, 5, 10, 20, 50\}$. We evaluate both the reaction class unknown and class conditional settings. Using the reaction class as prior knowledge represents some situations where the chemists already have an idea of how they would like to synthesize the product.

In all settings, our proposed GLN outperforms the baseline algorithms. And particularly for top-1 accuracy, our model performs significantly better than the second best method, with 8.1% higher accuracy with unknown reaction class, and 8.9% higher with reaction class given. This demonstrates the advantage of our method in this difficult setting and potential applicability in reality.

Moreover, our performance in the reaction class unknown setting even outperforms expertSys and seq2seq in the reaction conditional setting. Since the transformer paper didn't report top-k performance for $k > 10$, we leave it as blank. Meanwhile, [120] also reports the result when training using training+validation set and tuning on the test set. With this extra priviledge, the top-1 accuracy of transformer is 42.7% which is still worse

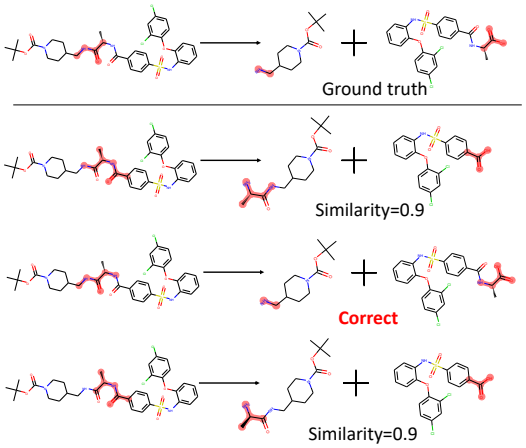


Figure 10.3: Example successful predictions.

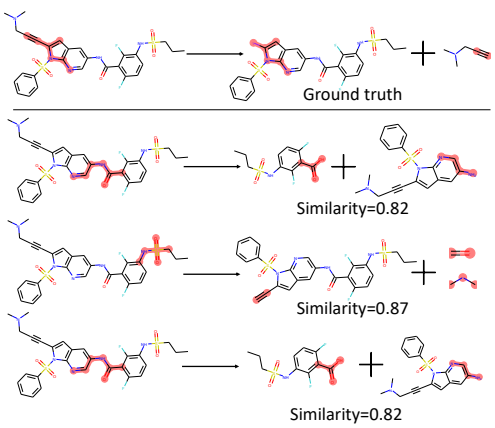


Figure 10.4: Example failed predictions.

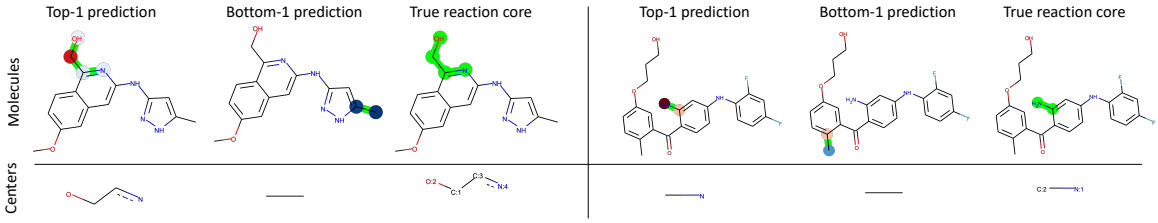


Figure 10.5: Reaction center prediction visualization. Red atoms indicate positive match scores, while blue ones having negative scores. The darkness of the color shows the magnitude of the score. Green parts highlight the substructure match between molecules and center structures.

than our performance. This shows the benefit of our logic powered deep neural network model comparing to purely neural models, especially when the amount of data is limited.

Since the theoretical upper bound of this rule-based implementation is 93.3%, the top-50 accuracy for our method in each setting is quite close to this limit. This shows the probabilistic model we built matches the actual retrosynthesis target well.

10.6.2 Interpret the predictions

Visualizing the predicted synthesis: In Fig 10.3 and 10.4, we visualize the ground truth reaction and the top 3 predicted reactions. For each reaction, we also highlight the corresponding reaction cores (*i.e.*, the set of atoms get changed). This is done by matching the subgraphs from predicted retrosynthesis template with the target compound and generated

reactants, respectively. Fig 10.3 shows that our correct prediction also gets almost the same reaction cores predicted as the ground truth. In this particular case, the explanation of our prediction aligns with the existing reaction knowledge.

Fig 10.4 shows a failure mode where none of the top-3 prediction matches. In this case we calculated the similarity between predicted reactants and ground truth ones using Dice similarity from RDKit. We find these are still similar in the molecule fingerprint level, which suggests that these predictions could be the potentially valid but unknown ones in the literature.

Visualizing the reaction center prediction: Here we visualize the prediction of probabilistic modeling of reaction center. This is done by calculating the inner product of each atom embedding in target molecule with the subgraph pattern embedding. Fig 10.5 shows the visualization of scores on the atoms that are part of the reaction center. The top-1 prediction assigns positive scores to these atoms (red ones), while the bottom-1 prediction (*i.e.*, prediction with least probability) assigns large negative scores (blue ones). Note that although the reaction center in molecule and the corresponding subgraph pattern have the same structure, the matching scores differ a lot. This suggests that the model has learned to predict the activity of substructures inside molecule graphs.

10.6.3 Large scale experiments on USPTO-full

To see how this method scales up with the dataset size, we create a large dataset from the entire set of reactions from USPTO 1976-2016. There are 1,808,937 raw reactions in total. For the reactions with multiple products, we duplicate them into multiple ones with one product each. After removing the duplications and reactions with wrong atom mappings, we obtain roughly 1M unique reactions, which are further divided into train/valid/test sets with size 800k/100k/100k.

Table 10.4: Top-k accuracy on USPTO-full.

	`retrosim`	`neuralsym`	GLN
top-1	32.8	35.8	**39.3**
top-10	56.1	60.8	**63.7**

Table 10.5: Ablation study on USPTO-50k with different representations.

	s2v-3	GGNN	MPNN	GIN	ECFP	s2v-0	s2v-1	s2v-2
top-1	52.6	51.6	50.4	51.8	51.9	40.7	47.0	51.3
top-10	83.1	81.8	83.2	83.3	81.5	78.1	80.4	82.2

We train on single GPU for 3 days and report with the model having best validation accuracy. The results are presented in Table 10.4. We compare with the best two baselines from previous sections. Despite the noisiness of the full USPTO set relative to the clean USPTO-50k, our method still outperforms the two best baselines in top-k accuracies.

10.6.4 Ablation study of design choices

Our GLN provides a general graphical model to retrosynthesis problem, which is compatible with many reasonable choices of the representation of graphs. In addition to S2V with 3 layers (s2v-3) we used in this chapter, we provide more ablation studies using different widely used GNNs and different number of "message-passing" layers.

The rationale behind the choices are: 1) the GNNs should be able to take both atom and bond features into consideration; 2) according to [240], the family of message-passing GNNs should have similar representation power as WL graph isomorphism check at best. We adopt the s2v in this chapter since it satisfies these requirements. Meanwhile, it comes with efficient `c++` binding of `RDKit`.

We use 2 layers of GNN by default, or use -k after the name in Table 10.5 to denote k-layer design. We can see that most variations of GNNs can achieve similar performances with enough number of message-passing like propagations. Based on this, for the experiment on the full USPTO dataset we simply use ECFP-2 provided by RDKit, as it is WL-isomorphism check based method with enough expressiveness [240] but faster to run.

Besides the choice of GNN, we also compare the choices of v_1, v_2 and w_2 mentioned in Section 10.4.2. Basically all these functions are comparing the compatibility of two vectors $\vec{x}, \vec{y}$. In this chapter, we simply used inner-product $\vec{x}^\top \vec{y}$. Here we also studied $MLP([\vec{x}, \vec{y}])$

Class	Fraction %
1	30.3
2	23.8
3	11.3
4	1.8
5	1.3
6	16.5
7	9.2
8	1.6
9	3.7
10	0.5

Figure 10.6: Reaction distribution over 10 types.

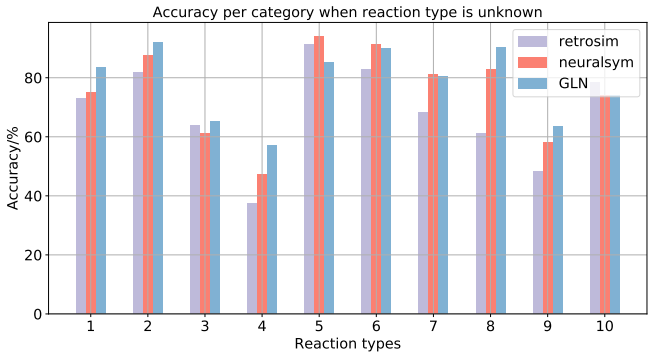


Figure 10.7: Top-10 accuracy per each reaction type.

and bilinear $\vec{x}^\top A\vec{y}$. For top-1, the inner-prod, MLP and bilinear gets 52.6, 52.7 and 53.5, respectively. So our GLN could be further improved with better design choices.

10.6.5 Per-category performance

We study the performance per each reaction category. Following the setting of baseline methods, we report the top-10 accuracy. As is shown in Table 10.6, the distribution of reaction types is highly unbalanced. From Fig 10.7 we can see our performances are better than retrosim in most classes, including the most common cases like class 1 and 2, or rare cases like class 4 or 8. This shows that our performance is not obtained by overfitting to one particular category of reactions. Such property is also important, as the retrosynthesis could involve rare reactions that haven't been well studied in the literature.

10.6.6 Reaction conditional performance

In Figure 10.8 we show the per-class performance when the reaction type is given as prior. As is shown Table 10.6, the distribution of reaction types is not uniform, where some reactions only get less than 5% of the total data. In this case, it is important to have a flexible model that can take the reaction type into account. Training one model per each reaction class is not a good idea in this case due to the imbalance of distribution.

From Figure 10.8 we can see our performances are comparable to retrosim in all classes,

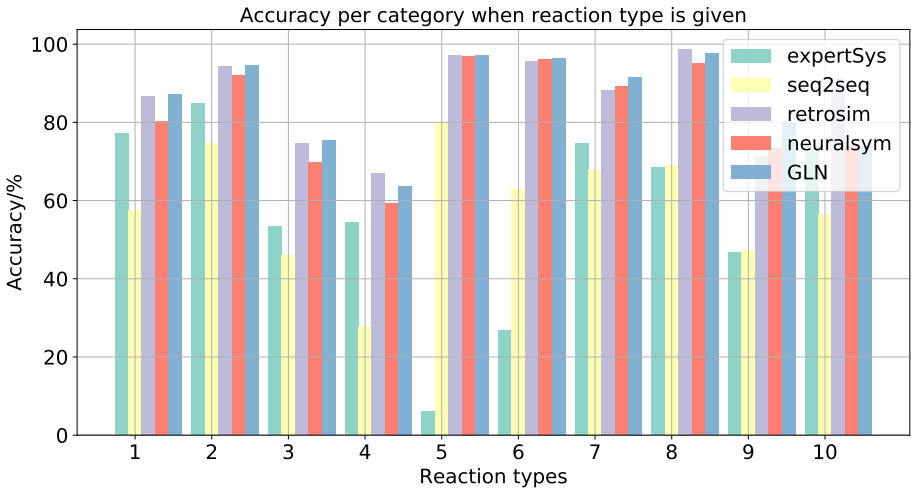


Figure 10.8: Top-10 accuracy per reaction class, when the reaction class is given during training.

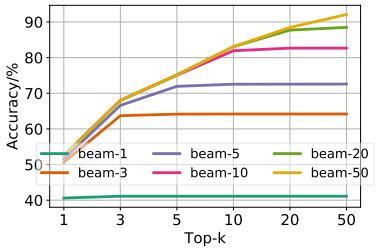


Figure 10.9: Top-k accuracy with different beam sizes.

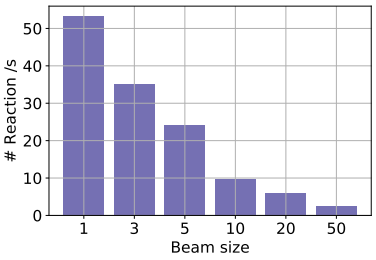


Figure 10.10: Inference speed with different beam sizes.

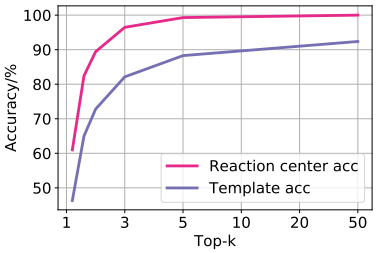


Figure 10.11: Top-k accuracy of reaction center and template.

while being much better than expertSys and seq2seq. Even in rare classes like class 9 or 10, we can still get best or second best performance. This shows the effectiveness and the flexibility of our GLN.

10.6.7 Effect of beam size

Beam size In Section. 10.6.1 we reported the top-k accuracy with beam size of 50, since k is at most 50. Here we study the performance of GLN using different beam sizes. Figure 10.9 shows the top-k accuracy for different k and different beam sizes. Overall the performance gets consistently better with larger beam sizes, for all top-k predictions. We can also see that the top-1 accuracy improved about 10% from beam size 1 (*i.e.*, greedy inference) to

beam size 3. Note that the curve of beam size s flattened after top-s predictions, since generally it didn't produce more predictions than s.

We also report the speed for inference in Figure 10.10. Such information during inference is averaged over 5,007 test predictions. The majority of the time is spent during applying the template via the call to RDKit, thus the time required grows up linearly with the beam size, as the number of RDKit calls grows linearly with the beam size.

Accuracy of $p(T|O)$ In Figure 10.11 we show the accuracy of $p(T|O)$, which decomposes into the reaction center identification accuracy and the template selection accuracy related to that reaction center. Here the beam size is fixed to 50. Predicting the reaction center is relatively easy and GLN achieves 99% top-20 accuracy. These results indicate that the current bottleneck in performance is in the template selection part, which is reasonably good now but can definitely be further improved by capturing more reaction features.

10.7 Summary

Evaluation: Retrosynthesis usually does not have a single right answer. Evaluation in this work is to reproduce what is reported for single-step retrosynthesis. This is a good, but imperfect benchmark, since there are potentially many reasonable ways to synthesize a single product.

Limitations: We share the limitations of all template-based methods. In our method, the template designs, more specifically, their specificities, remain as a design art and are hard to decide beforehand. Also, the scalability is still an issue since we rely on subgraph isomorphism during preprocessing.

Future work: The subgraph isomorphism part can potentially be replaced with predictive model, while during inference the fast inner product search [96] can be used to reduce computation cost. Also actively building templates or even inducing new ones could enhance the capacity and robustness.

CHAPTER 11
CONCLUSION

The ubiquitous graph structures have been used in many real-world applications including Chemistry, Bioinformatics, Social Networks, *etc.*. With the emerging of big data in these domains, efficient and expressive methods are needed for representation learning, generative modeling, as well as the combinatorial optimization.

Our thesis work has tightly connected the neural networks with the classical algorithms for graphs in the following aspects:

- using the existing algorithms to provide the inspiration of deep architecture design;
- enhancing the procedures of existing algorithm frameworks with deep learning;
- performing inductive reasoning over structured data.

11.1 Contribution and impact of the thesis work

Contribution The projects have been done in this thesis have made contributions in both academic and industrial domains. Our works have contributed to the modeling, learning and also new angles of several applications in graph learning domain, respectively.

- **Modeling:** Our work has created new models for multiple problems. These include a new form of graph neural network for supervised learning, a new generative model with autoregressive dependency structure for sequence modeling and a new graphical model with logic rules for structured prediction.
- **Learning:** We proposed a new stochastic learning method for large scale graph embedding based on fixed point iteration and a Q-learning based framework for a family of graph combinatorial optimization.

- **New angles with state-of-the-art results:** Our work has achieved state-of-the-art results that provides new angles for several problem domains. For loop invariant prediction, we build a general purpose tool with RL that requires little human effort. For retrosynthesis, we made a novel combination of deep graph learning with reaction template logics that outperforms seq2seq based methods.

Impact The thesis work has also received recognitions from the academia and industry. Specifically, the academic awards include best paper award in NIPS workshop in molecules and materials, and spotlight paper in NIPS 2017 and NeurIPS 2018. Our work has been covered in medias including Gatech CSE News, Cybersecurity Lectures, National Science Review, NVIDIA, etc. Also, some of our work has been officially integrated into deep learning package for graphs like Deep Graph Library (DGL [236]), which enables training on large graphs with millions of nodes.

11.2 Limitation and future work

Despite the progress we have made, there are still many limitations of the current thesis work, in which we will address in our future work.

Probabilistic modeling of graph The tractable probabilistic model of structured data often has limited form, *e.g.*, autoregressive. Such modeling will also limit the optimization and inference algorithms associated. A more general model would be the energy based models that can specify a broad family of distributions. However, such flexibility doesn't come for free. The corresponding inference and learning will also be more challenging.

Optimization in discrete space In general, the combinatorial optimization is NP-hard. Though empirically methods like reinforcement learning, imitation learning, *etc.*, has achieved improvements in some aspects, a smarter search algorithm will be needed in the end.

Understanding symbolic structures With the representation learning over graphs, people can handle symbolic forms like programs, logics, *etc.*. However, there is still gap between semantic meaning and syntax form of the representation. For example, a recursion in program will be subtle in its syntactic representation, but should be quite different in semantic space. Bridging the gap between these two will help both the representation learning, as well as symbolic reasoning.

Inductive/causal inference Having structural prior knowledge will greatly help with the generalization or even extrapolation. However, manually specifying such priors would be prohibitive in general. Though we have made several attempts in this direction in part III, the problem is yet resolved, in terms of sample efficiency and scalability.

Appendices

APPENDIX A
DERIVATION OF EMBEDDING FOR GRAPHICAL MODEL INFERENCE ALGORITHMS

By recognizing inference as computational expressions, inference machines [189] incorporate learning into the messages passing inference for CRFs. More recently, [101, 254, 148] designed specific recurrent neural networks and convolutional neural networks for imitating the messages in CRFs. Although these methods share the similarity, *i.e.*, bypassing learning potential function, to the proposed framework, there are significant differences comparing to the proposed framework.

The most important difference lies in the learning setting. In these existing messages learning work [101, 254, 148, 38], the learning task is still estimating the messages represented graphical models with designed function forms, *e.g.*, RNN or CNN, by maximizing loglikelihood. While in our work, we represented each structured data as a distribution, and the learning task is regression or classification over these distributions. Therefore, we treat the embedded models as samples, and learn the nonlinear mapping for embedding, and regressor or classifier, $f : \mathcal{P} \to \mathcal{Y}$, over these distributions jointly, with task-dependent user-specified loss functions.

Another difference is the way in constructing the messages forms, and thus, the neural networks architecture. In the existing work, the neural networks forms are constructed *strictly* follows the message updates forms (3.8) or (3.11). Due to such restriction, these works only focus on discrete variables with finite values, and is difficult to extend to continuous variables because of the integration. However, by exploiting the embedding point of view, we are able to build the messages with more *flexible* forms without losing the dependencies. Meanwhile, the difficulty in calculating integration for continuous variables is no longer a problem with the reasoning (3.3) and (3.4).

A.1 Derivation of the Fixed-Point Condition for Mean-Field Inference

In this section, we derive the fixed-point equation for mean-field inference in Section 3.4. As we introduced, the mean-field inference is indeed minimizing the variational free energy,

$$\min_{q_1,\dots,q_d} L(\{q_i\}_{i=1}^d) := \int_{\mathcal{H}^d} \prod_{i\in\mathcal{V}} q_i(h_i) \log \frac{\prod_{i\in\mathcal{V}} q_i(h_i)}{p(\{h_i\} \mid \{x_i\})} \prod_{i\in\mathcal{V}} dh_i.$$

Plug the MRF (3.5) into objective, we have

$$\begin{aligned} L(\{q_i\}_{i=1}^d) =& -\int_{\mathcal{H}^d} \prod_{i\in\mathcal{V}} q_i(h_i) \Bigg(\log \Phi(h_i, x_i) + \sum_{j\in\mathcal{N}(i)} \log \left(\Psi(h_i, h_j)\Phi(h_j, x_j)\right) \\ &+ \sum_{k\notin\mathcal{N}(i)} \log \Bigg(\prod_{(k,l)\in\mathcal{E}} \Psi(h_k, h_l)\Phi(h_k, x_k) \Bigg) \Bigg) \prod_{i\in\mathcal{V}} dh_i + \sum_{i\in\mathcal{V}} \int_{\mathcal{H}} q_i(h_i) \log q_i(h_i) dh_i \\ =& \quad -\int q_i(h_i) \log \Phi(h_i, x_i) dh_i + \sum_{i\in\mathcal{V}} \int_{\mathcal{H}} q_i(h_i) \log q_i(h_i) dh_i \\ &- \sum_{j\in\mathcal{N}(i)} \int q_i(h_i) \left(\int q_j(h_j) \log \left(\Psi(h_i, h_j)\Phi(h_j, x_j)\right) dh_j \right) dh_i + c_i \end{aligned} \tag{A.1}$$

where $c_i = -\int \prod_{k\notin\mathcal{N}(i)} q_k(h_k) \left(\sum_{k\notin\mathcal{N}(i)} \log \left(\prod_{(k,l)\in\mathcal{E}} \Psi(h_k, h_l)\Phi(h_k, x_k) \right) \right) \prod_{k\notin\mathcal{N}(i)} dh_k$. Take functional derivatives of $L(\{q_i\}_{i=1}^d)$ w.r.t. $q_i(h_i)$, and set them to zeros, we obtain the fixed-point condition in Section 3.4,

$$\log q_i(h_i) = c_i + \log \Phi(h_i, x_i) + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \left(\Psi(h_i, h_j)\Phi(h_j, x_j)\right) dh_j. \tag{A.2}$$

This fixed-point condition could be further reduced due to the independence between

h_i and x_j given h_j, *i.e.*,

$$\begin{aligned}\log q_i(h_i) &= c_i + \log \Phi(h_i, x_i) + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \Psi(h_i, h_j) dh_j \\ &\quad + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \Phi(h_j, x_j) dh_j \\ &= c_i' + \log \Phi(h_i, x_i) + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \Psi(h_i, h_j) dh_j, \qquad \text{(A.3)}\end{aligned}$$

where $c_i' = c_i + \sum_{j\in\mathcal{N}(i)} \int q_j(h_j) \log \Phi(h_j, x_j) dh_j$.

A.2 Derivation of the Fixed-Point Condition for Loopy BP

The derivation of the fixed-point condition for loopy BP can be found in [243]. However, to keep the paper self-contained, we provide the details here. The objective of loopy BP is

$$\begin{aligned}\min_{\{q_{ij}\}_{(i,j)\in\mathcal{E}}} \quad & -\sum_i (|\mathcal{N}(i)| - 1) \int_{\mathcal{H}} q_i(h_i) \log \frac{q_i(h_i)}{\Phi(h_i, x_i)} dh_i + \\ & \sum_{i,j} \int_{\mathcal{H}^2} q_{ij}(h_i, h_j) \log \frac{q_{ij}(h_i, h_j)}{\Psi(h_i, h_j)\Phi(h_i, x_i)\Phi(h_j, x_j)} dh_i dh_j \\ \text{s.t.} \quad & \int_{\mathcal{H}} q_{ij}(h_i, h_j) dh_j = q_i(h_i), \quad \int_{\mathcal{H}} q_{ij}(h_i, h_j) dh_j = q_i(h_i), \quad \int_{\mathcal{H}} q_i(h_i) dh_i = 1.\end{aligned}$$

Denote $\lambda_{ij}(h_j)$ is the multiplier to marginalization constraints $\int_{\mathcal{H}} q_{ij}(h_i, h_j) dh_i - q_j(h_j) = 0$, the Lagrangian is formed as

$$\begin{aligned}L(\{q_{ij}\}, \{q_i\}, \{\lambda_{ij}\}, \{\lambda_{ji}\}) \quad &= -\sum_i (|\mathcal{N}(i)| - 1) \int_{\mathcal{H}} q_i(h_i) \log \frac{q_i(h_i)}{\Phi(h_i, x_i)} dh_i \\ &\quad + \sum_{i,j} \int_{\mathcal{H}^2} q_{ij}(h_i, h_j) \log \frac{q_{ij}(h_i, h_j)}{\Psi(h_i, h_j)\Phi(h_i, x_i)\Phi(h_j, x_j)} dh_i dh_j \\ &\quad - \sum_{i,j} \int_{\mathcal{H}} \lambda_{ij}(h_j) \Big(\int_{\mathcal{H}} q_{ij}(h_i, h_j) dh_i - q_j(h_j) \Big) dh_j \\ &\quad - \sum_{i,j} \int_{\mathcal{H}} \lambda_{ji}(h_i) \Big(\int_{\mathcal{H}} q_{ij}(h_i, h_j) dh_j - q_i(h_i) \Big) dh_i\end{aligned}$$

with normalization constraints $\int_{\mathcal{H}} q_i(h_i)dh_i = 1$. Take functional derivatives of $L(\{q_{ij}\}, \{q_i\}, \{\lambda_{ij}\}, \{\lambda_{ji}\})$ with respect to $q_{ij}(h_i, h_j)$ and $q_i(h_i)$, and set them to zero, we have

$$\begin{aligned} q_{ij}(h_i, h_j) &\propto \Psi(h_i, h_j)\Phi(h_i, x_i)\Phi(h_j, x_j)\exp(\lambda_{ij}(h_j) + \lambda_{ji}(h_i)), \\ q_i(h_i) &\propto \Phi(h_i, x_i)\exp\left(\frac{\sum_{k\in\mathcal{N}(i)}\lambda_{ki}(h_i)}{|\mathcal{N}(i)| - 1}\right). \end{aligned}$$

We set $m_{ij}(h_j) = \frac{q_j(h_j)}{\Phi(h_i, x_i)\exp(\lambda_{ij}(h_j))}$, therefore,

$$\prod_{k\in\mathcal{N}(i)} m_{ki}(h_i) \propto \exp\left(\frac{\sum_{k\in\mathcal{N}(i)}\lambda_{ki}(h_i)}{|\mathcal{N}(i)| - 1}\right).$$

Plug it into $q_{ij}(h_i, h_j)$ and $q_i(h_i)$, we recover the loopy BP update for marginal belief and

$$\exp(\lambda_{ji}(h_i)) = \frac{q_i(h_i)}{\Phi(h_i, x_i)m_{ji}(h_i)} \propto \prod_{k\in N(i)\setminus j} m_{ki}(h_i).$$

The update rule for message $m_{ij}(h_j)$ can be recovered using the marginal consistency constraints,

$$\begin{aligned} m_{ij}(h_j) &= \frac{q_j(h_j)}{\Phi(h_i, x_i)\exp(\lambda_{ij}(h_j))} = \frac{\int_{\mathcal{H}} q_{ij}(h_i, h_j)dh_i}{\Phi(h_i, x_i)\exp(\lambda_{ij}(h_j))} \\ &= \Phi(h_j, x_j)\exp(\lambda_{ij}(h_j))\frac{\int_{\mathcal{H}}\Psi(h_i, h_j)\Phi(h_i, x_i)\exp(\lambda_{ji}(h_i))dh_i}{\Phi(h_i, x_i)\exp(\lambda_{ij}(h_j))} \\ &\propto \int_{\mathcal{H}}\Psi(h_i, h_j)\Phi(h_i, x_i)\prod_{k\in N(i)\setminus j} m_{ki}(h_i)dh_i. \end{aligned}$$

Moreover, we also obtain the other important relationship between $m_{ij}(h_j)$ and $\lambda_{ji}(h_i)$ by marginal consistency constraint and the definition of $m_{ij}(h_j)$,

$$m_{ij}(h_j) \propto \int \Psi(h_i, h_j)\Phi(h_j, x_j)\exp(\lambda_{ji}(h_i))dh_i.$$

APPENDIX B

SYNTAX, SEMANTICS AND ATTRIBUTE GRAMMAR IN SD-VAE

B.1 Grammar for Program Syntax

The syntax grammar for program is a generative contest free grammar starting with $\langle program \rangle$.

⟨*program*⟩ → ⟨*stat list*⟩

⟨*stat list*⟩ → ⟨*stat*⟩ '`;`' ⟨*stat list*⟩ | ⟨*stat*⟩

⟨*stat*⟩ → ⟨*assign*⟩ | ⟨*return*⟩

⟨*assign*⟩ → ⟨*lhs*⟩ '=' ⟨*rhs*⟩

⟨*return*⟩ → '`return:`' ⟨*lhs*⟩

⟨*lhs*⟩ → ⟨*var*⟩

⟨*var*⟩ → '`v`' ⟨*var id*⟩

⟨*digit*⟩ → '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

⟨*rhs*⟩ → ⟨*expr*⟩

⟨*expr*⟩ → ⟨*unary expr*⟩ | ⟨*binary expr*⟩

⟨*unary expr*⟩ → ⟨*unary op*⟩ ⟨*operand*⟩ | ⟨*unary func*⟩ '`(`' ⟨*operand*⟩ '`)`'

⟨*binary expr*⟩ → ⟨*operand*⟩ ⟨*binary op*⟩ ⟨*operand*⟩

⟨*unary op*⟩ → '`+`' | '`-`'

⟨*unary func*⟩ → '`sin`' | '`cos`' | '`exp`'

⟨*binary op*⟩ → '`+`' | '`-`' | '`*`' | '`/`'

⟨*operand*⟩ → ⟨*var*⟩ | ⟨*immediate number*⟩

⟨*immediate number*⟩ → ⟨*digit*⟩ '`.`' ⟨*digit*⟩

⟨*digit*⟩ → '`0`' | '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

B.2 Grammar for Molecule Syntax

Our syntax grammar for molecule is based on OpenSMILES standard, a context free grammar starting with $\langle s \rangle$.

⟨*s*⟩ → ⟨*atom*⟩

⟨*smiles*⟩ → ⟨*chain*⟩

⟨*atom*⟩ → ⟨*bracket atom*⟩ | ⟨*aliphatic organic*⟩ | ⟨*aromatic organic*⟩

⟨*aliphatic organic*⟩ → ‘B’ | ‘C’ | ‘N’ | ‘O’ | ‘S’ | ‘P’ | ‘F’ | ‘I’ | ‘Cl’ | ‘Br’

⟨*aromatic organic*⟩ → ’c’ | ’n’ | ’o’ | ’s’

⟨*bracket atom*⟩ → ‘[’ ⟨*bracket atom (isotope)*⟩ ‘]’

⟨*bracket atom (isotope)*⟩ → ⟨*isotope*⟩ ⟨*symbol*⟩ ⟨*bracket atom (chiral)*⟩

| ⟨*symbol*⟩ ⟨*bracket atom (chiral)*⟩

| ⟨*isotope*⟩ ⟨*symbol*⟩ | ⟨*symbol*⟩

⟨*bracket atom (chiral)*⟩ → ⟨*chiral*⟩ ⟨*bracket atom (h count)*⟩

| ⟨*bracket atom (h count)*⟩

| ⟨*chiral*⟩

⟨*bracket atom (h count)*⟩ → ⟨*h count*⟩ ⟨*bracket atom (charge)*⟩

| ⟨*bracket atom (charge)*⟩

| ⟨*h count*⟩

⟨*bracket atom (charge)*⟩ → ⟨*charge*⟩

⟨*symbol*⟩ → ⟨*aliphatic organic*⟩ | ⟨*aromatic organic*⟩

⟨*isotope*⟩ → ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digit*⟩ ⟨*digit*⟩

⟨*digit*⟩ → ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’

⟨*chiral*⟩ → ‘@’ | ‘@@’

⟨*h count*⟩ → ‘H’ | ‘H’ ⟨*digit*⟩

⟨*charge*⟩ → ’-’ | ’-’ ⟨*digit*⟩ | ’+’ | ’+’ ⟨*digit*⟩

⟨*bond*⟩ → ‘–’ | ‘=’ | ‘#’ | ‘/’ | ‘\’

⟨*ringbond*⟩ → ⟨*digit*⟩

⟨*branched atom*⟩ → ⟨*atom*⟩ | ⟨*atom*⟩ ⟨*branches*⟩ | ⟨*atom*⟩ ⟨*ringbonds*⟩

| ⟨*atom*⟩ ⟨*ringbonds*⟩ ⟨*branches*⟩

⟨*ringbonds*⟩ → ⟨*ringbonds*⟩ ⟨*ringbond*⟩ | ⟨*ringbond*⟩

⟨*branches*⟩ → ⟨*branches*⟩ ⟨*branch*⟩ | ⟨*branch*⟩

⟨*branch*⟩ → ’(’ ⟨*chain*⟩ ’)’ | ’(’ ⟨*bond*⟩ ⟨*chain*⟩ ’)’

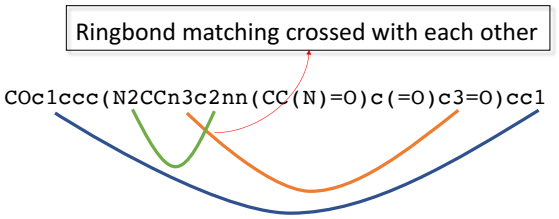


Figure B.1: Example of cross-serial dependencies (CSD) that exhibits in SMILES language.

$$\begin{aligned} \langle chain \rangle \quad &\rightarrow \langle branched\ atom \rangle \mid \langle chain \rangle\ \langle branched\ atom \rangle \\ &\mid \langle chain \rangle\ \langle bond \rangle\ \langle branched\ atom \rangle \end{aligned}$$

B.3 Examples of SMILES semantics

Here we provide more explanations of the semantics constraints that contained in SMILES language for molecules.

Specifically, the semantics we addressed here are:

1. **Ringbond matching:** The ringbonds should come in pairs. Each pair of ringbonds has an index and a bond-type associated. What the SMILES semantics requires is exactly the same as the well-known cross-serial dependencies (CSD) in formal language. CSD also appears in some natural languages, such as Dutch and Swiss-German. Another example of CSD is a sequence of multiple different types of parentheses where each separately balanced disregarding the others. See Figure B.1 for an illustration.

2. **Explicit valence control:** Intuitively, the semantics requires that each atom cannot have too many bonds associated with it. For example, a normal carbon atom has maximum valence of 4, which means associating a Carbon atom with two triple-bonds will violate the semantics.

B.4 Dependency graph introduced by attribute grammar

Suppose there is a production $r = u_0 \rightarrow u_1 u_2 \dots u_{|\beta|} \in \mathcal{R}$ and an attribute $u_i.a$ we denote the dependency set $D^r(u_i.a) = \{u_j.b | u_j.b \text{ is required for calculating } u_i.a\}$. The union of all dependency sets $\mathcal{D}_{\mathcal{T}}^{(att)} = \bigcup_{r \in \mathcal{T}, u_i \in r} D^r(u_i.a)$ induces a dependency graph, where nodes are the attributes and directed edges represents the dependency relationships between those attributes computation. Here $\mathcal{T}$ is an (partial or full) instantiation of the generated syntax tree of grammar G. Let $D^r(u_i) = \{u_j | \exists a, b : u_j.b \in D^r(u_i.a)\}$ and $\mathcal{D}_{\mathcal{T}} = \bigcup_{r \in \mathcal{T}, u_i \in r} D^r(u_i)$, that is, $\mathcal{D}_{\mathcal{T}}$ is constructed from $\mathcal{D}_{\mathcal{T}}^{(att)}$ by merging nodes with the same symbol but different attributes, we call $\mathcal{D}_{\mathcal{T}}^{(att)}$ is noncircular if the corresponding $\mathcal{D}_{\mathcal{T}}$ is noncircular.

In our paper, we assume the noncircular property of the dependency graph. Such property will be exploited for top-down generation in our decoder.

APPENDIX C
EXPERIMENTAL DETAILS OF S2V-DQN

C.1 Set Covering Problem

We also applied our framework to the classical Set Covering Problem (SCP). SCP is interesting because it is not a graph problem, but can be formulated as one. Our framework is capable of addressing such problems seamlessly, as we will show in the coming sections of the appendix which detail the performance of S2V-DQN as compared to other methods.

Set Covering Problem (SCP): Given a bipartite graph G with node set $V \coloneqq \mathcal{U} \cup \mathcal{C}$, find a subset of nodes $S \subseteq \mathcal{C}$ such that every node in $\mathcal{U}$ is covered, i.e. $u \in \mathcal{U} \Leftrightarrow \exists s \in S$ s.t. $(u, s) \in E$, and $|S|$ is minimized. Note that an edge $(u, s), u \in \mathcal{U}, s \in \mathcal{C}$, exists whenever subset s includes element u.

Meta-algorithm: Same as MVC; the termination criterion checks whether all nodes in $\mathcal{U}$ have been covered.

RL formulation: In SCP, the state is a function of the subset of nodes of $\mathcal{C}$ selected so far; an action is to add node of $\mathcal{C}$ to the partial solution; the reward is -1; the termination criterion is met when all nodes of $\mathcal{U}$ are covered; no helper function is needed.

Baselines for SCP: We include *Greedy*, which iteratively selects the node of $\mathcal{C}$ that is not in the current partial solution and that has the most uncovered neighbors in $\mathcal{U}$ [132]. We also used *LP*, another $O(\log |\mathcal{U}|)$-approximation that solves a linear programming relaxation of SCP, and rounds the resulting fractional solution in decreasing order of variable values (SortLP-1 in [176]).

C.2 Experimental Results on Realistic Data

In this section, we show results on realistic instances for all four problems. In particular, for MVC and SCP, we used the MemeTracker graph to formulate network diffusion optimization problems. For MAXCUT and TSP, we used benchmark instances that arise in physics and transportation, respectively.

C.2.1 Minimum Vertex Cover

As mentioned in the introduction, the MVC problem is related to the efficient spreading of information in networks, where one wants to cover as few nodes as possible such that all nodes have at least one neighbor in the cover. The MemeTracker graph [1] is a network of who-copies-whom, where nodes represent news sites or blogs, and a (directed) edge from u to v means that v frequently copies phrases (or memes) from u. The network is learned from real traces in [87], having 960 nodes and 5000 edges. The dataset also provides the average transmission time $\Delta_{u,v}$ between a pair of nodes, i.e. how much later v copies u's phrases after their publication online, on average. As done in [124], we use these average transmission times to compute a diffusion probability $P(u, v)$ on the edge, such that $P(u, v) = \alpha \cdot \frac{1}{\Delta_{u,v}}$, where α is a parameter of the diffusion model. In both MVC and SCP, we use $\alpha = 0.1$, but results are consistent for other values we have considered. For pairs of nodes that have edges in both directions, i.e. (u, v) and (v, u), we take the average probability to obtain an undirected version of the graph, for which MVC is defined.

Following the widely-adopted Independent Cascade model (see [70] for example), we sample a diffusion cascade from the full graph by independently keeping an edge with probability $P(u, v)$. We then consider the largest connected component in the graph as a single training instance, and train S2V-DQN on a set of such sampled diffusion graphs. The aim is to test the learned model on the (undirected version of the) *full* MemeTracker graph.

Experimentally, an optimal cover has 473 nodes, whereas S2V-DQN finds a cover with

[1] http://snap.stanford.edu/netinf/#data

474 nodes, only one more than the optimum, at an approximation ratio of 1.002. In comparison, MVCApprox and MVCApprox-Greedy find much larger covers with 666 and 578 nodes, at approximation ratios of 1.408 and 1.222, respectively.

C.2.2 Maximum Cut

A library of Maximum Cut instances is publicly available [2], and includes synthetic and realistic instances that are widely used in the optimization community (see references at library website). We perform experiments on a subset of the instances available, namely ten problems from Ising spin glass models in physics, given that they are realistic and manageable in size (the first 10 instances in Set2 of the library). All ten instances have 125 nodes and 375 edges, with edge weights in $\{-1, 0, 1\}$.

To train our S2V-DQN model, we constructed a training dataset by perturbing the instances, adding random Gaussian noise with mean 0 and standard deviation 0.01 to the edge weights. After training, the learned model is used to construct a cut-set greedily on each of the ten instances, as before.

Table C.1 shows that S2V-DQN finds near-optimal solutions (optimal in 3/10 instances) that are much better than those found by competing methods.

C.2.3 Traveling Salesman Problem

We use the standard TSPLIB library [184] which is publicly available [3]. We target 38 TSPLIB instances with sizes ranging from 51 to 318 cities (or nodes). We do not tackle larger instances as we are limited by the memory of a single graphics card. Nevertheless, most of the instances addressed here are larger than the largest instance used in [20].

We apply S2V-DQN in "Active Search" mode, similarly to [20]: no upfront training phase is required, and the reinforcement learning algorithm 7 is applied on-the-fly on each instance. The best tour encountered over the episodes of the RL algorithm is stored.

[2] http://www.optsicom.es/maxcut/#instances
[3] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

Table C.1: MAXCUT results on the ten instances described in C.2.2; values reported are cut weights of the solution returned by each method, where larger values are better (best in bold). Bottom row is the average approximation ratio (lower is better).

Instance	OPT	S2V-DQN	MaxcutApprox	SDP
G54100	110	**108**	80	54
G54200	112	**108**	90	58
G54300	106	**104**	86	60
G54400	114	**108**	96	56
G54500	112	**112**	94	56
G54600	110	**110**	88	66
G54700	112	**108**	88	60
G54800	108	**108**	76	54
G54900	110	**108**	88	68
G5410000	112	**108**	80	54
Approx. ratio	1	**1.02**	1.28	1.90

Table C.2 shows the results of our method and six other TSP algorithms. On all but 6 instances, S2V-DQN finds the best tour among all methods. The average approximation ratio of S2V-DQN is also the smallest at 1.05.

C.2.4 Set Covering Problem

The SCP is also related to the diffusion optimization problem on graphs; for instance, the proof of hardness in the classical [122] paper uses SCP for the reduction. As in MVC, we leverage the MemeTracker graph, albeit differently.

We use the same cascade model as in MVC to assign the edge probabilities, and sample graphs from it in the same way. Let $\mathcal{R}^G(u)$ be the set of nodes reachable from u in a sampled graph G. For every node u in G, there are two corresponding nodes in the SCP instance, $u_{\mathcal{C}} \in \mathcal{C}$ and $u_{\mathcal{U}} \in \mathcal{U}$. An edge exists between $u_{\mathcal{C}} \in \mathcal{C}$ and $v_{\mathcal{U}} \in \mathcal{U}$ if and only if $v \in \mathcal{R}^G(u)$. In other words, each node in the sampled graph G has a set consisting of the other nodes that it can reach in G. As such, the SCP reduces to finding the smallest set of nodes whose union can reach all other nodes. We generate training and testing graphs according to this same process, with $\alpha = 0.1$.

Table C.2: TSPLIB results: Instances are sorted by increasing size, with the number at the end of an instance's name indicating its size. Values reported are the cost of the tour found by each method (lower is better, best in bold). Bottom row is the average approximation ratio (lower is better).

Instance	OPT	S2V-DQN	Farthest	2-opt	Cheapest	Christofides	Closest	Nearest	MST
eil51	426	**439**	467	446	494	527	488	511	614
berlin52	7,542	**7,542**	8,307	7,788	9,013	8,822	9,004	8,980	10,402
st70	675	**696**	712	753	776	836	814	801	858
eil76	538	**564**	583	591	607	646	615	705	743
pr76	108,159	**108,446**	119,692	115,460	125,935	137,258	128,381	153,462	133,471
rat99	1,211	**1,280**	1,314	1,390	1,473	1,399	1,465	1,558	1,665
kroA100	21,282	**21,897**	23,356	22,876	24,309	26,578	25,787	26,854	30,516
kroB100	22,141	**22,692**	23,222	23,496	25,582	25,714	26,875	29,158	28,807
kroC100	20,749	**21,074**	21,699	23,445	25,264	24,582	25,640	26,327	27,636
kroD100	21,294	22,102	**22,034**	23,967	25,204	27,863	25,213	26,947	28,599
kroE100	22,068	22,913	23,516	**22,800**	25,900	27,452	27,313	27,585	30,979
rd100	7,910	**8,159**	8,944	8,757	8,980	10,002	9,485	9,938	10,467
eil101	629	**659**	673	702	693	728	720	817	847
lin105	14,379	**15,023**	15,193	15,536	16,930	16,568	18,592	20,356	21,167
pr107	44,303	**45,113**	45,905	47,058	52,816	49,192	52,765	48,521	55,956
pr124	59,030	**61,623**	65,945	64,765	65,316	64,591	68,178	69,297	82,761
bier127	118,282	**121,576**	129,495	128,103	141,354	135,134	145,516	129,333	153,658
ch130	6,110	**6,270**	6,498	6,470	7,279	7,367	7,434	7,578	8,280
pr136	96,772	**99,474**	105,361	110,531	109,586	116,069	105,778	120,769	142,438
pr144	58,537	**59,436**	61,974	60,321	73,032	74,684	73,613	61,652	77,704
ch150	6,528	**6,985**	7,210	7,232	7,995	7,641	7,914	8,191	9,203
kroA150	26,524	**27,888**	28,658	29,666	29,963	32,631	31,341	33,612	38,763
kroB150	26,130	**27,209**	27,404	29,517	31,589	33,260	31,616	32,825	35,289
pr152	73,682	**75,283**	75,396	77,206	88,531	82,118	86,915	85,699	90,292
u159	42,080	**45,433**	46,789	47,664	49,986	48,908	52,009	53,641	54,399
rat195	2,323	**2,581**	2,609	2,605	2,806	2,906	2,935	2,753	3,163
d198	15,780	16,453	**16,138**	16,596	17,632	19,002	17,975	18,805	19,339
kroA200	29,368	**30,965**	31,949	32,760	35,340	37,487	36,025	35,794	40,234
kroB200	29,437	31,692	**31,522**	33,107	35,412	34,490	36,532	36,976	40,615
ts225	126,643	**136,302**	140,626	138,101	160,014	145,283	151,887	152,493	188,008
tsp225	3,916	**4,154**	4,280	4,278	4,470	4,733	4,780	4,749	5,344
pr226	80,369	**81,873**	84,130	89,262	91,023	98,101	100,118	94,389	114,373
gil262	2,378	**2,537**	2,623	2,597	2,800	2,963	2,908	3,211	3,336
pr264	49,135	**52,364**	54,462	54,547	57,602	55,955	65,819	58,635	66,400
a280	2,579	**2,867**	3,001	2,914	3,128	3,125	2,953	3,302	3,492
pr299	48,191	**51,895**	51,903	54,914	58,127	58,660	59,740	61,243	65,617
lin318	42,029	45,375	45,918	**45,263**	49,440	51,484	52,353	54,019	60,939
linhp318	41,345	45,444	45,918	**45,263**	49,440	51,484	52,353	54,019	60,939
Approx. ratio	1	**1.05**	1.08	1.09	1.18	1.2	1.21	1.24	1.37

Experimentally, we test S2V-DQN and the other baseline algorithms on a set of 1000 test graphs. S2V-DQN achieves an average approximation ratio of 1.001, only slightly behind LP, which achieves 1.0009, and well ahead of Greedy at 1.03.

C.3 Experiment Details

C.3.1 Problem instance generation

Minimum Vertex Cover

For the Minimum Vertex Cover (MVC) problem, we generate random Erdős-Renyi (edge probability 0.15) and Barabasi-Albert (average degree 4) graphs of various sizes, and use the integer programming solver CPLEX 12.6.1 with a time cutoff of 1 hour to compute optimal solutions for the generated instances. When CPLEX fails to find an optimal solution, we report the best one found within the time cutoff as "optimal". All graphs were generated using the NetworkX [4] package in Python.

Maximum Cut

For the Maximum Cut (MAXCUT) problem, we use the same graph generation process as in MVC, and augment each edge with a weight drawn uniformly at random from $[0, 1]$. We use a quadratic formulation of MAXCUT with CPLEX 12.6.1. and a time cutoff of 1 hour to compute optimal solutions, and report the best solution found as "optimal".

Traveling Salesman Problem

For the (symmetric) 2-dimensional TSP, we use the instance generator of the 8th DIMACS Implementation Challenge [5] [114] to generate two types of Euclidean instances: "random" instances consist of n points scattered uniformly at random in the $[10^6, 10^6]$ square, while

[4] https://networkx.github.io/
[5] http://dimacs.rutgers.edu/Challenges/TSP/

"clustered" instances consist of n points that are clustered into $n/100$ clusters; generator details are described in page 373 of [114].

To compute optimal TSP solutions for both TSP, we use the state-of-the-art solver, Concorde [6] [12], with a time cutoff of 1 hour.

Set Covering Problem

For the SCP, given a number of node n, roughly $0.2n$ nodes are in node-set $\mathcal{C}$, and the rest in node-set $\mathcal{U}$. An edge between nodes in $\mathcal{C}$ and $\mathcal{U}$ exists with probability either 0.05 or 0.1, which can be seen as "density" values, and commonly appear for instances used in optimization papers on SCP [15]. We guarantee that each node in $\mathcal{U}$ has at least 2 edges, and each node in $\mathcal{C}$ has at least one edge, a standard measure for SCP instances [15]. We also use CPLEX 12.6.1. with a time cutoff of 1 hour to compute a near-optimal or optimal solution to a SCP instance.

C.3.2 Full results on solution quality

Table C.1 is a complete version of Table 7.2 that appears in the main text.

C.3.3 Full results on generalization

The full generalization results can be found in Table C.3, C.4, C.5, C.6, C.7, C.8 , C.9 and C.10.

Table C.3: S2V-DQN's generalization on MVC problem in ER graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0032	1.0883	1.0941	1.0710	1.0484	1.0365	1.0276	1.0246	1.0111
40-50	\	1.0037	1.0076	1.1013	1.0991	1.0800	1.0651	1.0573	1.0299
50-100	\	\	1.0079	1.0304	1.0570	1.0532	1.0463	1.0427	1.0238
100-200	\	\	\	1.0102	1.0095	1.0136	1.0142	1.0125	1.0103
400-500	\	\	\	\	\	\	1.0021	1.0027	1.0057

[6] http://www.math.uwaterloo.ca/tsp/concorde/

Table C.4: S2V-DQN's generalization on MVC problem in BA graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0016	1.0027	1.0039	1.0066	1.0093	1.0106	1.0125	1.0150	1.0491
40-50	\	1.0027	1.0051	1.0092	1.0130	1.0144	1.0161	1.0170	1.0228
50-100	\	\	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
100-200	\	\	\	1.0016	1.0020	1.0019	1.0021	1.0026	1.0060
400-500	\	\	\	\	\	\	1.0025	1.0026	1.0030

Table C.5: S2V-DQN's generalization on MAXCUT problem in ER graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0034	1.0167	1.0407	1.0667	1.1067	1.1489	1.1885	1.2150	1.1488
40-50	\	1.0127	1.0154	1.0089	1.0198	1.0383	1.0388	1.0384	1.0534
50-100	\	\	1.0112	1.0024	1.0109	1.0467	1.0926	1.1426	1.1297
100-200	\	\	\	1.0005	1.0021	1.0211	1.0373	1.0612	1.2021
200-300	\	\	\	\	1.0106	1.0272	1.0487	1.0700	1.1759

Table C.6: S2V-DQN's generalization on MAXCUT problem in BA graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0055	1.0119	1.0176	1.0276	1.0357	1.0386	1.0335	1.0411	1.0331
40-50	\	1.0107	1.0119	1.0139	1.0144	1.0119	1.0039	1.0085	0.9905
50-100	\	\	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
100-200	\	\	\	1.0166	1.0183	1.0166	1.0104	1.0166	1.0156
200-300	\	\	\	\	1.0420	1.0394	1.0290	1.0319	1.0244

Table C.7: S2V-DQN's generalization on TSP in random graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0147	1.0511	1.0702	1.0913	1.1022	1.1102	1.1124	1.1156	1.1212
40-50	\	1.0533	1.0701	1.0890	1.0978	1.1051	1.1583	1.1587	1.1609
50-100	\	\	1.0701	1.0871	1.0983	1.1034	1.1071	1.1101	1.1171
100-200	\	\	\	1.0879	1.0980	1.1024	1.1056	1.1080	1.1142
200-300	\	\	\	\	1.1049	1.1090	1.1084	1.1114	1.1179

Table C.8: S2V-DQN's generalization on TSP in clustered graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0214	1.0591	1.0761	1.0958	1.0938	1.0966	1.1009	1.1012	1.1085
40-50	\	1.0564	1.0740	1.0939	1.0904	1.0951	1.0974	1.1014	1.1091
50-100	\	\	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065
100-200	\	\	\	1.1009	1.0979	1.1013	1.1059	1.1048	1.1091
200-300	\	\	\	\	1.1012	1.1049	1.1080	1.1067	1.1112

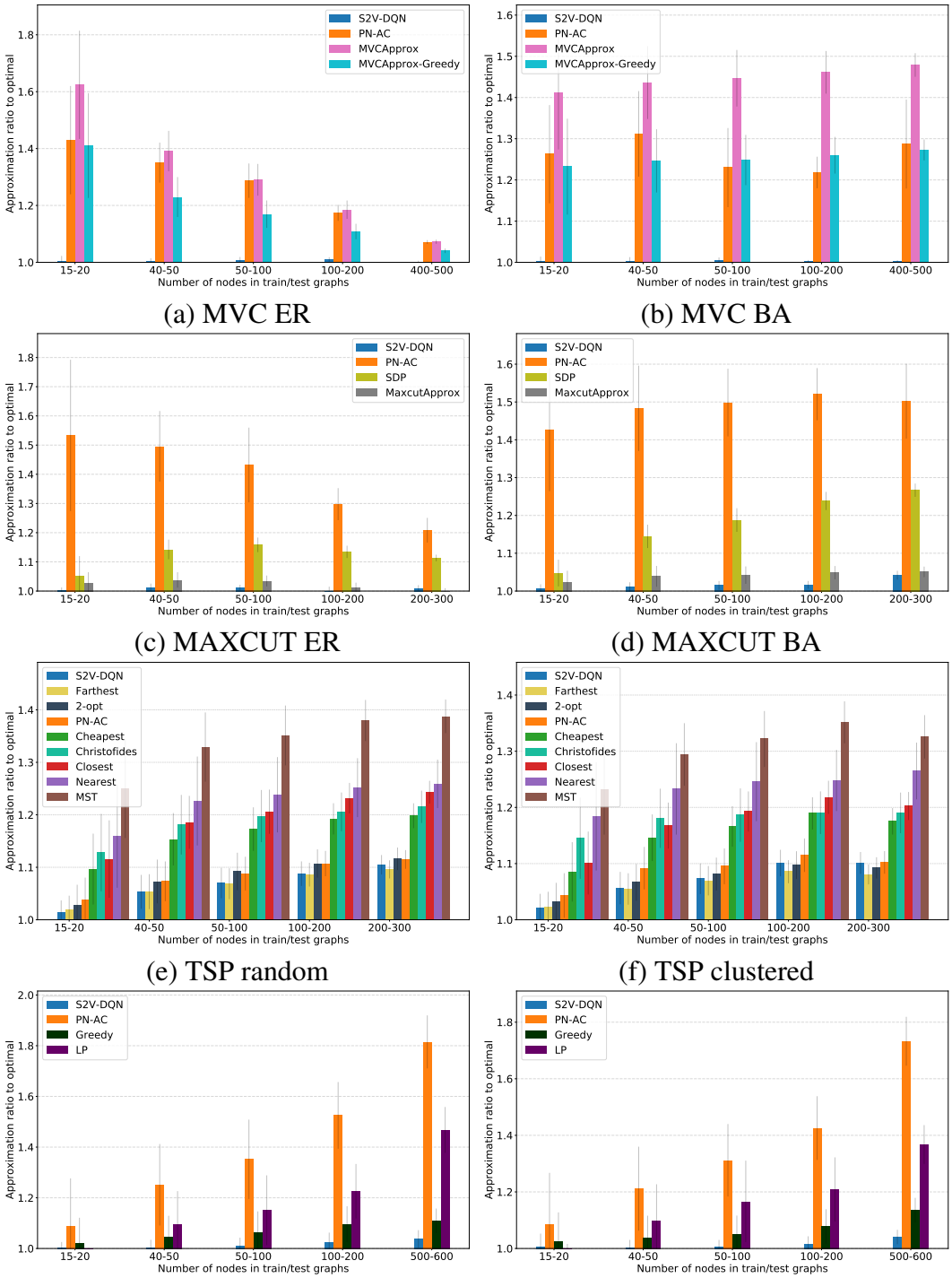


Figure C.1: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

Table C.9: S2V-DQN's generalization on SCP with edge probability 0.05.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0055	1.0170	1.0436	1.1757	1.3910	1.6255	1.8768	2.1339	3.0574
40-50	\	1.0039	1.0083	1.0241	1.0452	1.0647	1.0792	1.0858	1.0775
50-100	\	\	1.0056	1.0199	1.0382	1.0614	1.0845	1.0821	1.0620
100-200	\	\	\	1.0147	1.0270	1.0417	1.0588	1.0774	1.0509
200-300	\	\	\	\	1.0273	1.0415	1.0828	1.1357	1.2349

Table C.10: S2V-DQN's generalization on SCP with edge probability 0.1.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0015	1.0200	1.0369	1.0795	1.1147	1.1290	1.1325	1.1255	1.0805
40-50	\	1.0048	1.0137	1.0453	1.0849	1.1055	1.1052	1.0958	1.0618
50-100	\	\	1.0090	1.0294	1.0771	1.1180	1.1456	1.2161	1.0946
100-200	\	\	\	1.0231	1.0394	1.0564	1.0702	1.0747	2.5055
200-300	\	\	\	\	1.0378	1.0517	1.0592	1.0556	1.3192

C.3.4 Experiment Configuration of S2V-DQN

The node/edge representations and hyperparameters used in our experiments is shown in Table C.11. For our method, we simply tune the hyperparameters on small graphs (i.e., the graphs with less than 50 nodes), and fix them for larger graphs.

Table C.11: S2V-DQN's configuration used in Experiment.

Problem	**Node tag**	**Edge feature**	**Embedding size** p	T	**Batch size**	**n-step**
Minimum Vertex Cover	0/1 tag	N/A	64	5	128	5
Maximum Cut	0/1 tag	edge length; end node tag	64	3	64	1
Traveling Salesman Problem	coordinates; 0/1 tag; start/end node	edge length; end node tag	64	4	64	1
Set Covering Problem	0/1 tag	N/A	64	5	64	2

C.3.5 Stabilizing the training of S2V-DQN

For the learning rate, we use exponential decay after a certain number of steps, where the decay factor is fixed to 0.95. We also anneal the exploration probability ϵ from 1.0 to 0.05 in a linear way. For the discounting factor used in MDP, we use 1.0 for MVC, MAXCUT and SCP. For TSP, we use 0.1.

We also normalize the intermediate reward by the maximum number of nodes. For Q-learning, it is also important to disentangle the actual Q with obsolete $\tilde{Q}$, as mentioned in [160].

Also for TSP with insertion helper function, we find it works better with *negative* version of designed reward function. This sounds counter intuitive at the beginning. However, since typically the RL agent will bias towards most recent rewards, flipping the sign of reward function suggests a focus over future rewards. This is especially useful with the insertion construction. But it shows that designing a good reward function is still challenging for learning combinatorial algorithm, which we will investigate in our future work.

C.3.6 Convergence of S2V-DQN

In Figure C.2, we plot our algorithm's convergence with respect to the held-out validation performance. We first obtain the convergence curve for each type of problem under every graph distribution. To visualize the convergence at the same scale, we plot the approximate ratio.

Figure C.2 shows that our algorithm converges nicely on the MVC, MAXCUT and SCP problems. For the MVC, we use the model trained on small graphs to initialize the model for training on larger ones. Since our model also generalizes well to problems with different sizes, the curve looks almost flat. For TSP, where the graph is essentially fully connected, it is harder to learn a good model based on graph structure. Nevertheless, as shown in previous section, the graph embedding can still learn good feature representations with multiple embedding iterations.

C.3.7 Complete time v/s approximation ratio plots

Figure C.3 is a superset of Figure 7.3, including both graph types and three graph size ranges for MVC, MAXCUT and SCP.

C.3.8 Additional analysis of the trade-off between time and approx. ratio

Tables C.12 and C.13 offer another perspective on the trade-off between the running time of a heuristic and the quality of the solution it finds. We ran CPLEX for MVC and MAXCUT for 10 minutes on the 200-300 node graphs, and recorded the time and value of all the solutions found by CPLEX within the limit; results shown next carry over to smaller graphs. Then, for a given method M that terminates in T seconds on a graph G and returns a solution with approximation ratio R, we asked the following 2 questions:

1. If CPLEX is given the same amount of time T for G, how well can CPLEX do?
2. How long does CPLEX need to find a solution of same or better quality than the one the heuristic has found?

For the first question, the column "Approx. Ratio of Best Solution" in Tables C.12 and C.13 shows the following:

– MVC (Table C.12): The larger values for S2V-DQN imply that solutions we find quickly are of higher quality, as compared to the MVCApprox/Greedy baselines.

– MAXCUT (Table C.13): On most of the graphs, CPLEX cannot find any solution at all if given the same time as S2V-DQN or MaxcutApprox. SDP (solved with state-of-the-art CVX solver) is so slow that CPLEX finds solutions that are 10% better than those of SDP if given the same time as SDP (on ER graphs), which confirms that SDP is not time-efficient. One possible interpretation of the poor performance of SDP is that its theoretical guaranteed of 0.87 is *in expectation* over the solutions it can generate, and so the variance in the approximation ratios of these solutions may be very large.

For the second question, the column "Additional Time Needed" in Tables C.12 and C.13 shows the following:

– MVC (Table C.12): The larger values for S2V-DQN imply that solutions we find are harder to improve upon, as compared to the MVCApprox/Greedy baselines.

– MAXCUT (Table C.13): On ER (BA) graphs, CPLEX (10 minute-cutoff) cannot find a solution that is better than those of S2V-DQN or MaxcutApprox on many instances (e.g. the value (59) for S2V-DQN on ER graphs means that on $41 = 100 - 59$ graphs, CPLEX could not find a solution that is as good as S2V-DQN's). When we consider only those graphs for which CPLEX could find a better solution, S2V-DQN's solutions take significantly more time for CPLEX to beat, as compared to MaxcutApprox and SDP. The negative values for SDP indicate that CPLEX finds a solution better than SDP's in a shorter time.

Table C.12: Minimum Vertex Cover (100 graphs with 200-300 nodes): Trade-off between running time and approximation ratio. An "Approx. Ratio of Best Solution" value of 1.x% means that the solution found by CPLEX if given the same time as a certain heuristic (in the corresponding row) is x% worse, on average. "Additional Time Needed" in seconds is the additional amount of time needed by CPLEX to find a solution of value at least as good as the one found by a given heuristic; negative values imply that CPLEX finds such solutions faster than the heuristic does. Larger values are better for both metrics. The values in parantheses are the number of instances (out of 100) for which CPLEX finds some solution in the given time (for "Approx. Ratio of Best Solution"), or finds some solution that is at least as good as the heuristic's (for "Additional Time Needed").

	Approx. Ratio of Best Solution		Additional Time Needed	
	ER	BA	ER	BA
S2V-DQN	1.09 (100)	1.81 (100)	2.14 (100)	137.42 (100)
MVCApprox-Greedy	1.07 (100)	1.44 (100)	1.92 (100)	0.83 (100)
MVCApprox	1.03 (100)	1.24 (98)	2.49 (100)	0.92 (100)

Table C.13: Maximum Cut (100 graphs with 200-300 nodes): please refer to the caption of Table C.12.

	Approx. Ratio of Best Solution		Additional Time Needed	
	ER	BA	ER	BA
S2V-DQN	N/A (0)	1081.45 (1)	8.99 (59)	402.05 (34)
MaxcutApprox	1.00 (48)	340.11 (3)	-0.23 (50)	218.19 (57)
SDP	0.90 (100)	0.84 (100)	-6.06 (100)	-5.54 (100)

C.3.9 Visualization of solutions

In Figure C.4, C.5 and C.6, we visualize solutions found by our algorithm for MVC, MAXCUT and TSP problems, respectively. For the ease of presentation, we only visualize small-size graphs. For MVC and MAXCUT, the graph is of the ER type and has 18 nodes. For TSP, we show solutions for a "random" instance (18 points) and a "clustered" one (15 points).

For MVC and MAXCUT, we show two step by step examples where S2V-DQN finds the optimal solution. For MVC, it seems we are picking the node which covers the most edges in the current state. However, in a more detailed visualization in Appendix C.3.10, we show that our algorithm learns a smarter greedy or dynamic programming like strategy. While picking the nodes, it also learns how to keep the connectivity of graph by scarifying the intermediate edge coverage a little bit.

In the example of MAXCUT, it is even more interesting to see that the algorithm did not pick the node which gives the largest intermediate reward at the beginning. Also in the intermediate steps, the agent seldom chooses a node which would cancel out the edges that are already in the cut set. This also shows the effectiveness of graph state representation, which provides useful information to support the agent's node selection decisions. For TSP, we visualize an optimal tour and one found by S2V-DQN for two instances. While the tours found by S2V-DQN differ slightly from the optimal solutions visualized, they are of comparable cost and look qualitatively acceptable. The cost of the tours found by S2V-DQN is within 0.07% and 0.5% of optimum, respectively.

C.3.10 Detailed visualization of learned MVC strategy

In Figure C.7, we show a detailed comparison with our learned strategy and two other simple heuristics. We find that the S2V-DQN can learn a much smarter strategy, where the agent is trying to maintain the connectivity of graph during node picking and edge removal.

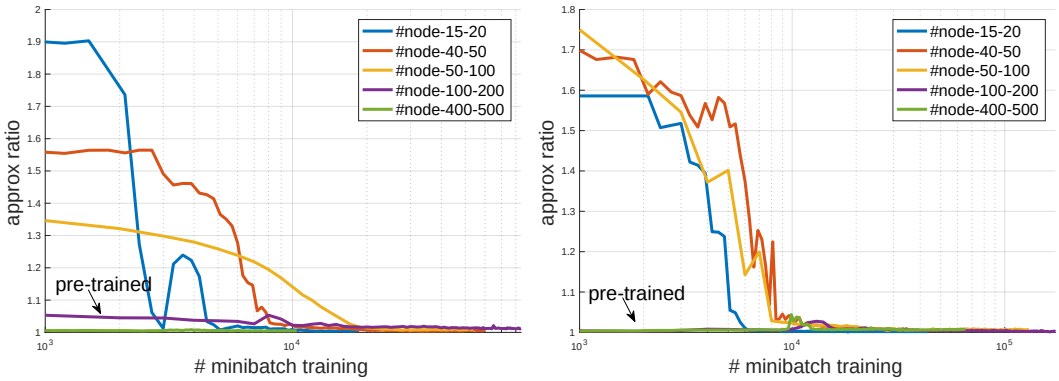
C.3.11 Experiment Configuration of PN-AC

We implemented PN-AC to the best of our capabilities. Note that it is quite possible that there are minor differences between our implementation and [20] that might have resulted in performance not as good as reported in that paper.

For experiments of PN-AC across all tasks, we follow the configurations provided in [20]: *a*) For the input data, we use mini-batches of 128 sequences with 0-paddings to the maximal input length (which is the maximal number of nodes) in the training data. *b*) For node representation, we use coordinates for TSP, so the input dimension is 2. For MVC, MAXCUT and SCP, we represent nodes based on the adjacency matrix of the graph. To get a fixed dimension representation for each node, we use SVD to get a low-rank approximation of the adjacency matrix. We set the rank as 8, so that each node in the input sequence is represented by a 8-dimensional vector. *c*) For the network structure, we use standard single-layer LSTM cells with 128 hidden units for both encoder and decoder parts of the pointer networks. *d*) For the optimization method, we train the PN-AC model with the Adam optimizer [127] and use an initial learning rate of 10^{-3} that decay every 5000 steps by a factor of 0.96. *e*) For the glimpse trick, we exactly use one-time glimpse in our implementation, as described in the original PN-AC paper. *f*) We initialize all the model parameters uniformly random within $[-0.08, 0.08]$ and clip the $L2$ norm of the gradients to 1.0. *g*) For the baseline function in the actor-critic algorithm, we tried the critic network in our implementation, but it hurts the performance according to our experiments. So we use the exponential moving average performance of the sampled solution from the pointer network as the baseline.

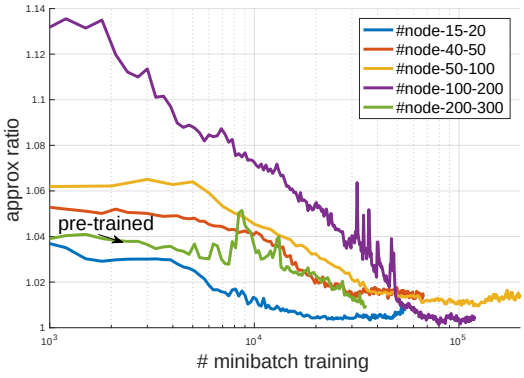
Consistency with the results from [20] Though our TSP experiment setting is not exactly the same as [20], we still include some of the results directly here, for the sake of completeness. We applied the insertion heuristic to PN-AC as well, and all the results reported in this chapter are with the insertion heuristic. We compare the approximation ratio reported by [20] verses which reported by our implementation. For TSP20: 1.02 vs

1.03 (reported in this chapter); TSP50: 1.05 vs 1.07 (reported in this chapter); TSP100: 1.07 vs 1.09 (reported in this chapter). Note that we have variable graph size in each setting (where the original PN-AC is only reported on fixed graph size), which makes the task more difficult. Therefore, we think the performance gap here is pretty reasonable.

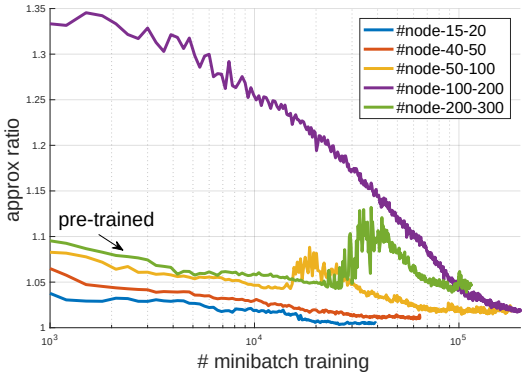


(a) MVC ER

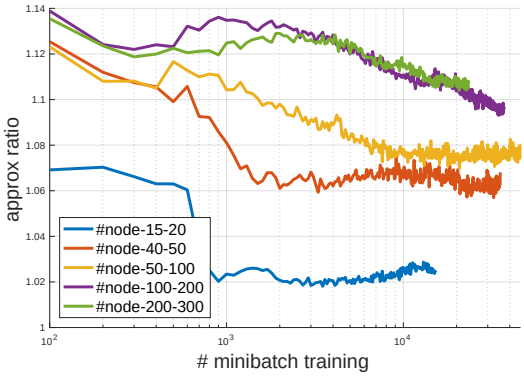
(b) MVC BA



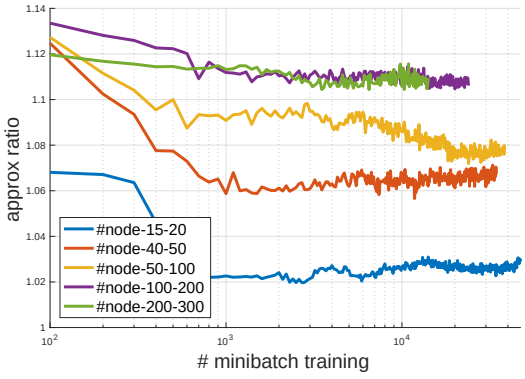
(c) MAXCUT ER



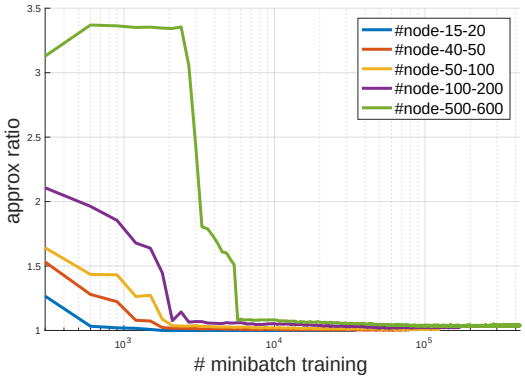
(d) MAXCUT BA



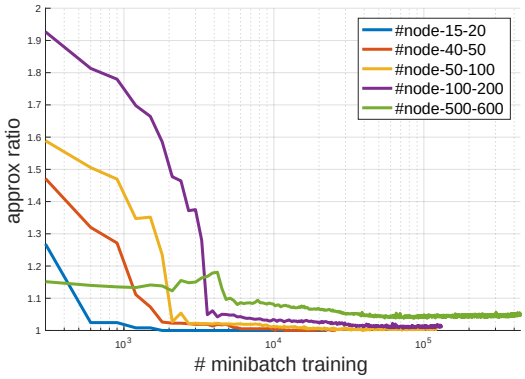
(e) TSP random



(f) TSP clustered

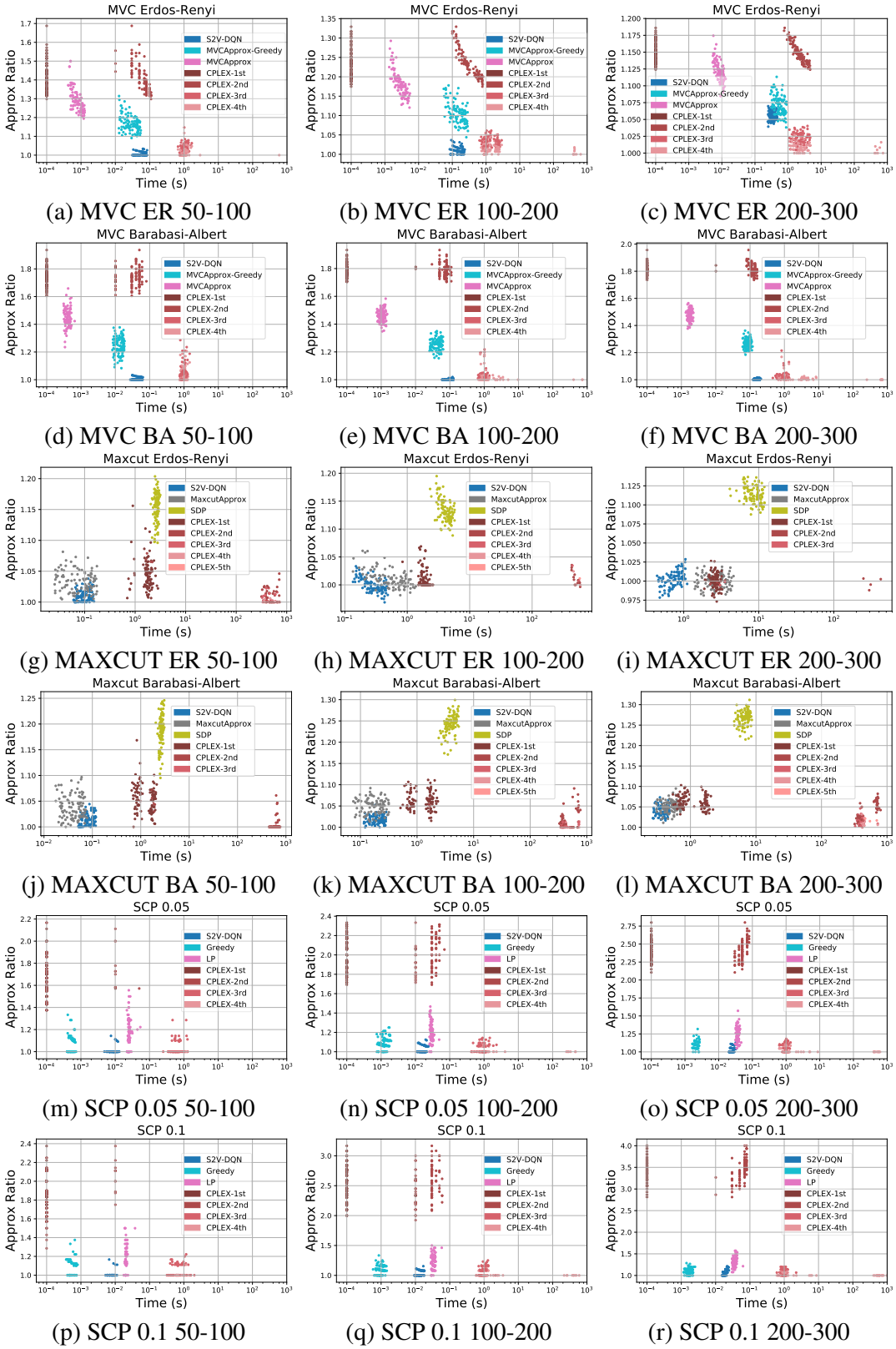


(g) SCP 0.1



(h) SCP 0.05

Figure C.2: S2V-DQN convergence measured by the held-out validation performance.



(a) MVC ER 50-100 (b) MVC ER 100-200 (c) MVC ER 200-300

(d) MVC BA 50-100 (e) MVC BA 100-200 (f) MVC BA 200-300

(g) MAXCUT ER 50-100 (h) MAXCUT ER 100-200 (i) MAXCUT ER 200-300

(j) MAXCUT BA 50-100 (k) MAXCUT BA 100-200 (l) MAXCUT BA 200-300

(m) SCP 0.05 50-100 (n) SCP 0.05 100-200 (o) SCP 0.05 200-300

(p) SCP 0.1 50-100 (q) SCP 0.1 100-200 (r) SCP 0.1 200-300

Figure C.3: Time-approximation trade-off for MVC, MAXCUT and SCP. In this figure, each dot represents a solution found for a single problem instance. For CPLEX, we also record the time and quality of each solution it finds. For example, CPLEX-1st means the first feasible solution found by CPLEX.

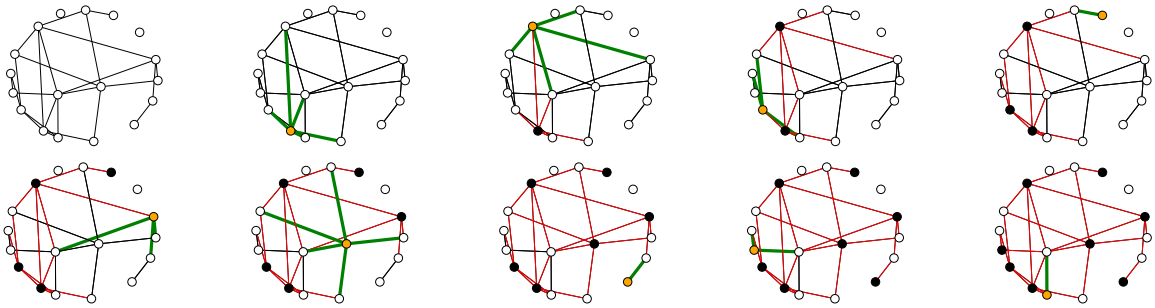


Figure C.4: Minimum Vertex Cover: an optimal solution to an ER graph instance found by S2V-DQN. Selected node in each step is colored in orange, and nodes in the partial solution up to that iteration are colored in black. Newly covered edges are in thick green, previously covered edges are in red, and uncovered edges in black. We show that the agent is not only picking the node with large degree, but also trying to maintain the connectivity after removal of the covered edges. For more detailed analysis, please see Appendix C.3.10.

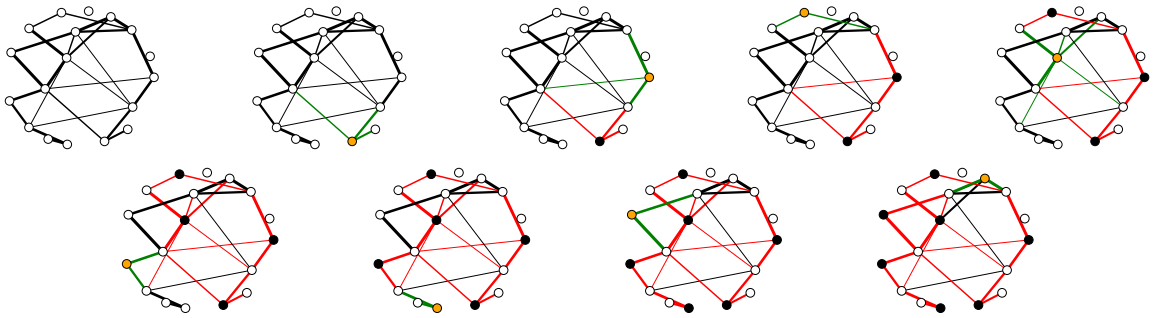


Figure C.5: Maximum Cut: an optimal solution to ER graph instance found by S2V-DQN. Nodes are partitioned into two sets: white or black nodes. At each iteration, the node selected to join the set of black nodes is highlighted in orange, and the new cut edges it produces are in green. Cut edges from previous iteration are in red (Best viewed in color). It seems the agent will try to involve the nodes that won't cancel out the edges in current cut set.

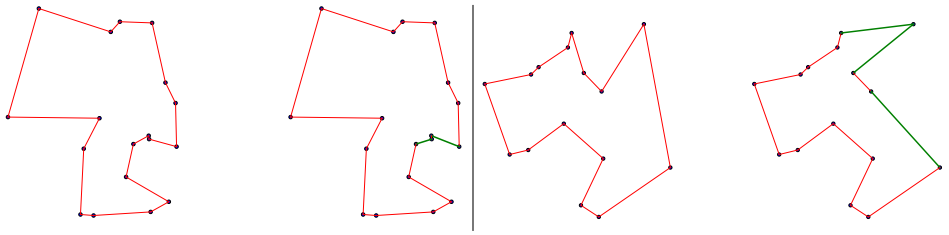


Figure C.6: Traveling Salesman Problem. Left: optimal tour to a "random" instance with 18 points (all edges are red), compared to a tour found by our method next to it. For our tour, edges that are not in the optimal tour are shown in green. Our tour is 0.07% longer than an optimal tour. Right: a "clustered" instance with 15 points; same color coding as left figure. Our tour is 0.5% longer than an optimal tour. (Best viewed in color).

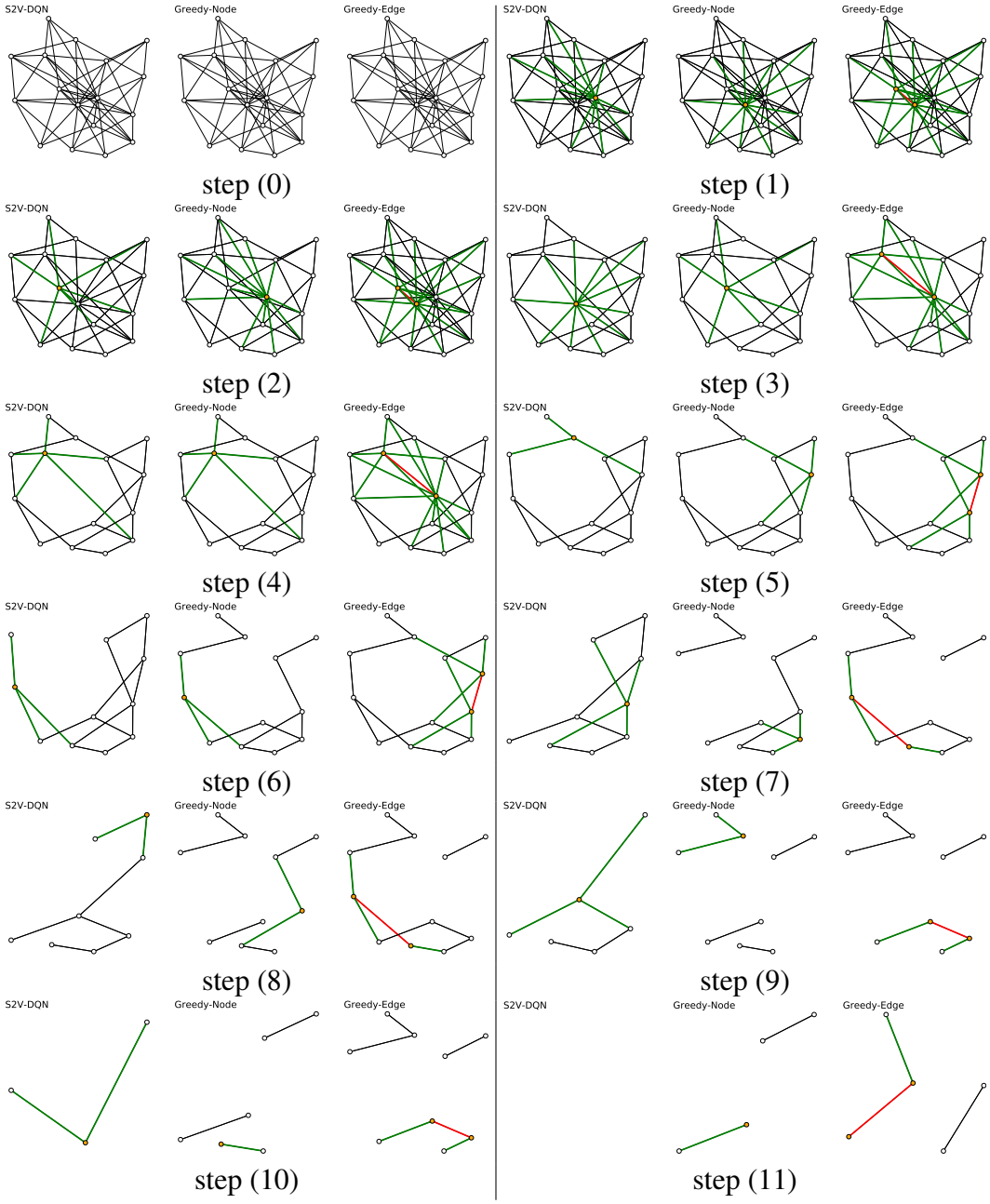


Figure C.7: Step-by-step comparison between our S2V-DQN and two greedy heuristics. We can see our algorithm will also favor the large degree nodes, but it will also do something smartly: instead of breaking the graph into several disjoint components, our algorithm will try the best to keep the graph connected.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2013.

[3] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[5] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2013.

[6] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[7] D. Alvarez-Melis and T. S. Jaakkola, "Tree-structured decoding with doubly-recurrent neural networks," 2016.

[8] W. Ammar, C. Dyer, and N. A. Smith, "Conditional random field autoencoders for unsupervised structured prediction," in *Advances in Neural Information Processing Systems*, 2014, pp. 3311–3319.

[9] A. Andreeva, D. Howorth, S. E. Brenner, T. J. Hubbard, C. Chothia, and A. G. Murzin, "Scop database in 2004: Refinements integrate structure and sequence family data," *Nucleic acids research*, vol. 32, no. suppl 1, pp. D226–D229, 2004.

[10] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *Advances in Neural Information Processing Systems*, 2016, pp. 3981–3989.

[11] A. W. Appel, "Verified Software Toolchain," in *Proceedings of the 20th European Symposium on Programming (ESOP)*, 2011.

[12] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, *Concorde TSP solver*, 2006.

[13] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[14] J. Atwood and D. Towsley, "Diffusion-convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 1993–2001.

[15] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study," *Combinatorial Optimization*, pp. 37–60, 1980.

[16] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[17] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[18] J. L. Baylon, N. A. Cilfone, J. R. Gulcher, and T. W. Chittenden, "Enhancing retrosynthetic reaction prediction with deep learning using multiscale reaction classification," *Journal of chemical information and modeling*, vol. 59, no. 2, pp. 673–688, 2019.

[19] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," The University of Chichago, Tech. Rep. TR-2002-01, 2002, http://www.cs.uchicago.edu/research/publications/techreports/TR-2002-01.

[20] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *CoRR*, vol. abs/1611.09940, 2016.

[21] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[22] M. Benhenda, "Chemgan challenge for drug discovery: Can ai reproduce natural chemical diversity?" *arXiv preprint arXiv:1708.08227*, 2017.

[23] D. P. Bertsekas, *Nonlinear Programming*, Second. Belmont, MA: Athena Scientific, 1999.

[24] P. Bielik, V. Raychev, and M. Vechev, "Phog: Probabilistic model for code," in *Proceedings of the International Conference on Machine Learning*, 2016.

[25] E. J. Bjerrum, "Smiles enumeration as data augmentation for neural network modeling of molecules," *arXiv preprint arXiv:1703.07076*, 2017.

[26] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann, "Netgan: Generating graphs via random walks," *arXiv preprint arXiv:1803.00816*, 2018.

[27] B. Bollobás and O. Riordan, "The diameter of a scale-free random graph," *Combinatorica*, vol. 24, no. 1, pp. 5–34, 2004.

[28] K. M. Borgwardt, "Graph kernels," PhD thesis, Ludwig-Maximilians-University, Munich, Germany, 2007.

[29] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs.," in *Proc. Intl. Conf. Data Mining*, 2005, pp. 74–81.

[30] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Jozefowicz, and S. Bengio, "Generating sentences from a continuous space," *arXiv preprint arXiv:1511.06349*, 2015.

[31] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, vol. 1, no. Nov, pp. 77–112, 2000.

[32] J. Bradshaw, M. J. Kusner, B. Paige, M. H. Segler, and J. M. Hernández-Lobato, "A generative model for electron paths," 2018.

[33] J. Bresnan, R. M. Kaplan, S. Peters, and A. Zaenen, "Cross-serial dependencies in dutch," 1982.

[34] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[35] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli, "Leveraging grammar and reinforcement learning for neural program synthesis," in *International Conference on Learning Representations*, 2018.

[36] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[37] C. Chang and C. Lin, *LIBSVM: A library for support vector machines*, Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm, 2001.

[38] L.-C. Chen, A. G. Schwing, A. L. Yuille, and R. Urtasun, "Learning deep structured models," *arXiv preprint arXiv:1407.2538*, 2014.

[39] X. Chen, X. Qiu, C. Zhu, and X. Huang, "Gated recursive neural network for chinese word segmentation," in *Proceedings of Annual Meeting of the Association for Computational Linguistics*, 2015.

[40] X. Chen, C. Liu, and D. Song, "Towards synthesizing complex programs from input-output examples," in *International Conference on Learning Representations*, 2018.

[41] Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, and N. de Freitas, "Learning to learn for global optimization of black box functions," *arXiv preprint arXiv:1611.03824*, 2016.

[42] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," *arXiv preprint arXiv:1609.01704*, 2016.

[43] C. W. Coley, W. H. Green, and K. F. Jensen, "Machine learning in computer-aided synthesis planning," *Accounts of Chemical Research*, vol. 51, no. 5, pp. 1281–1289, May 15, 2018.

[44] C. W. Coley, W. H. Green, and K. F. Jensen, "Rdchiral: An rdkit wrapper for handling stereochemistry in retrosynthetic template extraction and application," 2019.

[45] C. W. Coley, W. Jin, L. Rogers, T. F. Jamison, T. S. Jaakkola, W. H. Green, R. Barzilay, and K. F. Jensen, "A graph-convolutional neural network model for the prediction of chemical reactivity," *Chemical Science*, vol. 10, no. 2, pp. 370–377, Jan. 2, 2019.

[46] C. W. Coley, L. Rogers, W. H. Green, and K. F. Jensen, "Computer-assisted retrosynthesis based on molecular similarity," *ACS Central Science*, vol. 3, no. 12, pp. 1237–1245, 2017.

[47] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2003.

[48] E. Corey and W. T. Wipke, "Computer-assisted design of complex organic syntheses," *Science*, vol. 166, no. 3902, pp. 178–192, 1969.

[49] E. J. Corey, "The logic of chemical synthesis: Multistep synthesis of complex carbogenic molecules (nobel lecture)," *Angewandte Chemie International Edition in English*, vol. 30, no. 5, pp. 455–465, 1991.

[50] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[51] T. F. Cox and M. A. A. Cox, *Multidimensional Scaling*. London: Chapman and Hall, 1994.

[52] B. C. Csáji, "Approximation with artificial neural networks,"

[53] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, 1991.

[54] B. Dai, N. He, H. Dai, and L. Song, "Provable bayesian inference via particle mirror descent," in *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, 2016, pp. 985–994.

[55] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, 2016, pp. 2702–2711.

[56] H. Dai, B. Dai, Y.-M. Zhang, S. Li, and L. Song, "Recurrent hidden semi-markov model," 2017.

[57] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song, "Adversarial attack on graph structured data," *arXiv preprint arXiv:1806.02371*, 2018.

[58] H. Dai, Y. Li, C. Wang, R. Singh, P.-S. Huang, and P. Kohli, "Learning transferable graph exploration," in *Advances in Neural Information Processing Systems*, 2019, pp. 2514–2525.

[59] H. Dai, Y. Tian, B. Dai, S. Skiena, and L. Song, "Syntax-directed variational autoencoder for structured data," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[60] D. P. De Farias and B. Van Roy, "The linear programming approach to approximate dynamic programming," *Operations research*, vol. 51, no. 6, pp. 850–865, 2003.

[61] ——, "On constraint sampling in the linear programming approach to approximate dynamic programming," *Mathematics of operations research*, vol. 29, no. 3, pp. 462–478, 2004.

[62] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *J Med Chem*, vol. 34, pp. 786–797, 1991.

[63] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in Neural Information Processing Systems*, 2016, pp. 3837–3845.

[64] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. rahman Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy i/o," in *Proceedings of the International Conference on Machine Learning*, 2017.

[65] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2013.

[66] P. D. Dobson and A. J. Doig, "Distinguishing enzyme structures from non-enzymes without alignments," *J Mol Biol*, vol. 330, no. 4, pp. 771–783, 2003.

[67] J. G. Doench, E. Hartenian, D. B. Graham, Z. Tothova, M. Hegde, I. Smith, M. Sullender, B. L. Ebert, R. J. Xavier, and D. E. Root, "Rational design of highly active sgrnas for crispr-cas9-mediated gene inactivation," *Nature biotechnology*, vol. 32, no. 12, pp. 1262–1267, 2014.

[68] L. Dong and M. Lapata, "Language to logical form with neural attention," *arXiv preprint arXiv:1601.01280*, 2016.

[69] N. Du, H. Dai, R. Trivedi, U. Upadhyay, M. Gomez-Rodriguez, and L. Song, "Recurrent marked temporal point processes: Embedding event history to vector," in *KDD*, 2016.

[70] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha, "Scalable influence estimation in continuous-time diffusion networks," in *NIPS*, 2013.

[71] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in Neural Information Processing Systems*, 2015, pp. 2215–2223.

[72] P. Erdos and A Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungar. Acad. Sci*, vol. 5, pp. 17–61, 1960.

[73] P. Ertl and A. Schuffenhauer, "Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions," *Journal of cheminformatics*, vol. 1, no. 1, p. 8, 2009.

[74] M. Fahndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software*, 2010.

[75] E. Fox, E. B. Sudderth, M. I. Jordan, and A. S. Willsky, "Nonparametric bayesian learning of switching linear dynamical systems," in *Advances in Neural Information Processing Systems*, 2009, pp. 457–464.

[76] N. Fusi, I. Smith, J. Doench, and J. Listgarten, "In silico predictive modeling of crispr/cas9 guide efficiency," *bioRxiv*, 2015.

[77] Y. Gao, E. W. Archer, L. Paninski, and J. P. Cunningham, "Linear dynamical neural population models through nonlinear embeddings," in *Advances in Neural Information Processing Systems*, 2016, pp. 163–171.

[78] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Ice: A robust framework for learning invariants," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2014.

[79] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2016.

[80] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Proc. Annual Conf. Computational Learning Theory*, B. Schölkopf and M. K. Warmuth, Eds., Springer, 2003, pp. 129–143.

[81] Z. Ghahramani and G. E. Hinton, "Variational learning for switching state-space models," *Neural computation*, vol. 12, no. 4, pp. 831–864, 2000.

[82] R. C. S. L. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, MIT Press, vol. 13, 2001, p. 402.

[83] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *arXiv preprint arXiv:1704.01212*, 2017.

[84] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[85] M. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *Journal of the ACM*, vol. 42, no. 6, pp. 1115–1145, 1995.

[86] R. Gómez-Bombarelli, D. Duvenaud, J. M. Hernández-Lobato, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik, "Automatic chemical design using a data-driven continuous representation of molecules," *arXiv preprint arXiv:1610.02415*, 2016.

[87] M. Gomez-Rodriguez, J. Leskovec, and A. Krause, "Inferring networks of diffusion and influence," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2010, pp. 1019–1028.

[88] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014, pp. 2672–2680.

[89] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, pp. 471–476, 2016.

[90] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *KDD*, 2016.

[91] S. Gu, S. Levine, I. Sutskever, and A. Mnih, "Muprop: Unbiased backpropagation for stochastic neural networks," *arXiv preprint arXiv:1511.05176*, 2015.

[92] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," *arXiv preprint arXiv:1611.02247*, 2016.

[93] G. L. Guimaraes, B. Sanchez-Lengeling, P. L. C. Farias, and A. Aspuru-Guzik, "Objective-reinforced generative adversarial networks (organ) for sequence generation models," *arXiv preprint arXiv:1705.10843*, 2017.

[94] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[95] S. Gulwani and N. Jojic, "Program verification as probabilistic inference," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2007.

[96] R. Guo, S. Kumar, K. Choromanski, and D. Simcha, "Quantization based fast inner product search," in *Artificial Intelligence and Statistics*, 2016, pp. 482–490.

[97] J. Hachmann, R. Olivares-Amaya, S. Atahan-Evrenk, C. Amador-Bedolla, R. S. Sánchez-Carrera, A. Gold-Parker, L. Vogt, A. M. Brockway, and A. Aspuru-Guzik, "The harvard clean energy project: Large-scale computational screening and design of organic photovoltaics on the world community grid," *The Journal of Physical Chemistry Letters*, vol. 2, no. 17, pp. 2241–2251, 2011.

[98] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Laboratory (LANL), Tech. Rep., 2008.

[99] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 1024–1034.

[100] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," in *Advances in Neural Information Processing Systems*, 2014, pp. 3293–3301.

[101] J. R. Hershey, J. L. Roux, and F. Weninger, "Deep unfolding: Model-based inspiration of novel deep architectures," *arXiv preprint arXiv:1409.2574*, 2014.

[102] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, Oct. 1969.

[103] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[104] F. Hoonakker, N. Lachiche, A. Varnek, and A. Wagner, "Condensed graph of reaction: Considering a chemical reaction as one single pseudo molecule,"

[105] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing, "Harnessing deep neural networks with logic rules," *arXiv preprint arXiv:1603.06318*, 2016.

[106] IBM, *CPLEX User's Manual, Version 12.6.1*, version Version 12.6.1, 2014.

[107] T. S. Jaakkola and D. Haussler, "Exploiting generative models in discriminative classifiers," in *Advances in Neural Information Processing Systems 11*, M. S. Kearns, S. A. Solla, and D. A. Cohn, Eds., MIT Press, 1999, pp. 487–493.

[108] D. Janz, J. van der Westhuizen, and J. M. Hernández-Lobato, "Actively learning what makes a discrete sequence valid," *arXiv preprint arXiv:1708.04465*, 2017.

[109] D. Janz, J. van der Westhuizen, B. Paige, M. J. Kusner, and J. M. Hernández-Lobato, "Learning a generative model for validity in complex discrete structures," *arXiv preprint arXiv:1712.01664*, 2017.

[110] T. Jebara, R. Kondor, and A. Howard, "Probability product kernels," *J. Mach. Learn. Res.*, vol. 5, pp. 819–844, 2004.

[111] W. Jin, R. Barzilay, and T. Jaakkola, "Junction tree variational autoencoder for molecular graph generation," *arXiv preprint arXiv:1802.04364*, 2018.

[112] W. Jin, C. Coley, R. Barzilay, and T. Jaakkola, "Predicting organic reaction outcomes with weisfeiler-lehman network," in *Advances in Neural Information Processing Systems*, 2017, pp. 2607–2616.

[113] W. Jitkrittum, A. Gretton, N. Heess, S. M. A. Eslami, B. Lakshminarayanan, D. Sejdinovic, and Z. Szabó, "Kernel-based just-in-time learning for passing expectation propagation messages," in *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI 2015, July 12-16, 2015, Amsterdam, The Netherlands*, 2015, pp. 405–414.

[114] D. S. Johnson and L. A. McGeoch, "Experimental analysis of heuristics for the stsp," in *The traveling salesman problem and its variations*, Springer, 2007, pp. 369–443.

[115] M. Johnson and A. Willsky, "Stochastic variational inference for bayesian time series models," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 1854–1862.

[116] M. J. Johnson, D. Duvenaud, A. B. Wiltschko, S. R. Datta, and R. P. Adams, "Structured vaes: Composing probabilistic graphical models and variational autoencoders," *arXiv preprint arXiv:1603.06277*, 2016.

[117] M. J. Johnson and A. S. Willsky, "Bayesian nonparametric hidden semi-markov models," *The Journal of Machine Learning Research*, vol. 14, no. 1, pp. 673–701, 2013.

[118] J. Kain, C. Stokes, Q. Gaudry, X. Song, J. Foley, R. Wilson, and B. de Bivort, "Leg-tracking and automated behavioural classification in drosophila," *Nature communications*, vol. 4, p. 1910, 2013.

[119] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[120] P. Karpov, G. Godin, and I Tetko, "A transformer model for retrosynthesis," 2019.

[121] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196*, 2017.

[122] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *KDD*, ACM, 2003, pp. 137–146.

[123] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348–6358.

[124] E. B. Khalil, B. Dilkina, and L. Song, "Scalable diffusion-aware optimization of network topology," in *Knowledge Discovery and Data Mining (KDD)*, 2014.

[125] E. B. Khalil, B. Dilkina, G. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[126] E. B. Khalil, P. Le Bodic, L. Song, G. L. Nemhauser, and B. N. Dilkina, "Learning to branch in mixed integer programming.," in *AAAI*, 2016, pp. 724–731.

[127] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[128] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[129] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[130] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[131] ——, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[132] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[133] D. E. Knuth, "Semantics of context-free languages," *Theory of Computing Systems*, vol. 2, no. 2, pp. 127–145, 1968.

[134] L. Kong, C. Dyer, and N. A. Smith, "Segmental recurrent neural networks," *arXiv preprint arXiv:1511.06018*, 2015.

[135] R. G. Krishnan, U. Shalit, and D. Sontag, "Deep kalman filters," *arXiv preprint arXiv:1511.05121*, 2015.

[136] R. Kuang, E. Ie, K. Wang, K. Wang, M. Siddiqi, Y. Freund, and C. Leslie, "Profile-based string kernels for remote homology detection and motif extraction," *Journal of bioinformatics and computational biology*, vol. 3, no. 03, pp. 527–550, 2005.

[137] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato, "Grammar variational autoencoder," *arXiv preprint arXiv:1703.01925*, 2017.

[138] M. G. Lagoudakis and M. L. Littman, "Learning to select branching rules in the dpll procedure for satisfiability," *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 344–359, 2001.

[139] G Landrum, *Rdkit: Open-source cheminformatics (2013)*, 2012.

[140] T. Lei, W. Jin, R. Barzilay, and T. Jaakkola, "Deriving neural architectures from sequence and graph kernels," *arXiv preprint arXiv:1705.09037*, 2017.

[141] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 475–485.

[142] C. Leslie, E. Eskin, and W. S. Noble, "The spectrum kernel: A string kernel for SVM protein classification," in *Proceedings of the Pacific Symposium on Biocomputing*, Singapore: World Scientific Publishing, 2002, pp. 564–575.

[143] C. Leslie, E. Eskin, J. Weston, and W. S. Noble, "Mismatch string kernels for SVM protein classification," in *Advances in Neural Information Processing Systems 15*, S. Becker, S. Thrun, and K. Obermayer, Eds., vol. 15, Cambridge, MA: MIT Press, 2002.

[144] K. Li and J. Malik, "Learning to optimize," *arXiv preprint arXiv:1606.01885*, 2016.

[145] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[146] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.

[147] C. Liang, J. Berant, Q. Le, K. D. Forbus, and N. Lao, "Neural symbolic machines: Learning semantic parsers on freebase with weak supervision," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 23–33.

[148] G. Lin, C. Shen, I. Reid, and A. van den Hengel, "Deeply learning the messages in message passing inference," in *Advances in Neural Information Processing Systems (NIPS'15)*, 2015.

[149] S. W. Linderman, A. C. Miller, R. P. Adam, D. M. Blei, L. Paninski, and M. J. Johnson, "Recurrent switching linear dynamical systems," *arXiv preprint arXiv:1610.08466*, 2016.

[150] B. Liu, B. Ramsundar, P. Kawthekar, J. Shi, J. Gomes, Q. Luu Nguyen, S. Ho, J. Sloane, P. Wender, and V. Pande, "Retrosynthetic reaction prediction using neural sequence-to-sequence models," *ACS Central Science*, vol. 3, no. 10, pp. 1103–1113, 2017.

[151] K. S. M. Schmidt, *Crfchain: Matlab code for chain-structured conditional random fields with categorical features.* 2008.

[152] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proceedings of the International Conference on Machine Learning*, 2014.

[153] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," in *Communications of the ACM*, 1971.

[154] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[155] A. Miller, A. Fisch, J. Dodge, A.-H. Karimi, A. Bordes, and J. Weston, "Key-value memory networks for directly reading documents," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[156] T. Minka, "The EP energy function and minimization schemes," *See www. stat. cmu. edu/minka/papers/learning. html, August*, 2001.

[157] A. Mnih and K. Gregor, "Neural variational inference and learning in belief networks," *arXiv preprint arXiv:1402.0030*, 2014.

[158] A. Mnih and D. J. Rezende, "Variational inference for monte carlo objectives," *arXiv preprint arXiv:1602.06725*, 2016.

[159] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[160] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[161] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[162] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of SIAM*, vol. 5, no. 1, pp. 32–38, 1957.

[163] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[164] K. P. Murphy, "Hidden semi-markov models (hsmms)," 2002.

[165] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT Press, 2012.

[166] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *UAI*, 1999, pp. 467–475.

[167] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2015.

[168] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *Proceedings of The 33rd International Conference on Machine Learning*, 2016, pp. 2014–2023.

[169] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[170] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[171] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Stanford InfoLab, Tech. Rep., 1999.

[172] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. New Jersey: Prentice-Hall, 1982.

[173] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, ACM, 2017, pp. 506–519.

[174] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[175] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman, 1988.

[176] D. Peleg, G. Schechtman, and A. Wool, "Approximating bounded 0-1 integer linear programs," in *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, IEEE, 1993, pp. 69–77.

[177] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," *arXiv preprint arXiv:1403.6652*, 2014.

[178] E. O. Pyzer-Knapp, K. Li, and A. Aspuru-Guzik, "Learning from the harvard clean energy project: The use of neural networks to accelerate materials discovery," *Advanced Functional Materials*, vol. 25, no. 41, pp. 6495–6502, 2015.

[179] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[180] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[181] M. Raghothaman and A. Udupa, "Language to specify syntax-guided synthesis problems," 2014.

[182] T. Raiko, M. Berglund, G. Alain, and L. Dinh, "Techniques for learning binary stochastic feedforward neural networks," *arXiv preprint arXiv:1406.2989*, 2014.

[183] J. Ramon and T. Gärtner, "Expressivity versus efficiency of graph kernels," in *Proceedings of the first international workshop on mining graphs, trees and sequences*, 2003, pp. 65–74.

[184] G. Reinelt, "Tsplib—a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[185] J.-L. Reyes-Ortiz, L. Oneto, A. Samà, X. Parra, and D. Anguita, "Transition-aware human activity recognition using smartphones," *Neurocomputing*, vol. 171, pp. 754–767, 2016.

[186] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014, pp. 1278–1286.

[187] M. Richardson and P. Domingos, "Markov logic networks," *Machine learning*, vol. 62, no. 1-2, pp. 107–136, 2006.

[188] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, Springer, 2005, pp. 317–328.

[189] S. Ross, D. Munoz, M. Hebert, and J. A. Bagnell, "Learning message-passing inference machines for structured prediction," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, IEEE, 2011, pp. 2737–2744.

[190] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with uct," in *CPAIOR*, Springer, 2012, pp. 356–361.

[191] H. Samulowitz and R. Memisevic, "Learning to solve QBF," in *AAAI*, 2007.

[192] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using Gröbner bases," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2004.

[193] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *Neural Networks, IEEE Transactions on*, vol. 20, no. 1, pp. 61–80, 2009.

[194] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Houston, Texas, USA, 2013, ISBN: 978-1-4503-1870-9.

[195] B. Schölkopf, K. Tsuda, and J.-P. Vert, *Kernel Methods in Computational Biology*. Cambridge, MA: MIT Press, 2004.

[196] B. Schölkopf and A. J. Smola, *Learning with Kernels*. Cambridge, MA: MIT Press, 2002.

[197] J. S. Schreck, C. W. Coley, and K. J. Bishop, "Learning retrosynthetic planning through self-play," *arXiv preprint arXiv:1901.06569*, 2019.

[198] P. Schwaller, T. Gaudin, D. Lanyi, C. Bekas, and T. Laino, ""found in translation": Predicting outcomes of complex organic chemistry reactions using neural sequence-to-sequence models," *Chemical science*, vol. 9, no. 28, pp. 6091–6098, 2018.

[199] P. Schwaller, T. Laino, T. Gaudin, P. Bolgar, C. Bekas, and A. A. Lee, "Molecular transformer for chemical reaction prediction and uncertainty estimation," Nov. 6, 2018.

[200] M. H. S. Segler and M. P. Waller, "Neural-symbolic machine learning for retrosynthesis and reaction prediction," *Chemistry – A European Journal*, vol. 23, no. 25, pp. 5966–5971, May 2, 2017.

[201] M. H. Segler, M. Preuss, and M. P. Waller, "Planning chemical syntheses with deep neural networks and symbolic ai," *Nature*, vol. 555, no. 7698, p. 604, 2018.

[202] K. Sen, "DART: directed automated random testing," in *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers*, 2009, p. 4.

[203] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 263–272.

[204] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2014.

[205] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2011.

[206] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *Proceedings of the European Symposium on Programming (ESOP)*, 2013.

[207] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, no. Sep, pp. 2539–2561, 2011.

[208] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Proc. Intl. Conference on Artificial Intelligence and Statistics*, M. Welling and D. van Dyk, Eds., Society for Artificial Intelligence and Statistics, 2009.

[209] S. M. Shieber, "Evidence against the context-freeness of natural language," 1985.

[210] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[211] A. Smola, A. Gretton, L. Song, and B. Schölkopf, "A hilbert space embedding for distributions," in *Algorithmic learning theory*, Springer, 2007, pp. 13–31.

[212] L. Song, A. Gretton, D. Bickson, Y. Low, and C. Guestrin, "Kernel belief propagation," in *Proc. Intl. Conference on Artificial Intelligence and Statistics*, ser. JMLR workshop and conference proceedings, vol. 10, 2011.

[213] L. Song, A. Gretton, and C. Guestrin, "Nonparametric tree graphical models," in *13th Workshop on Artificial Intelligence and Statistics*, ser. JMLR workshop and conference proceedings, vol. 9, 2010, pp. 765–772.

[214] L. Song, J. Huang, A. J. Smola, and K. Fukumizu, "Hilbert space embeddings of conditional distributions," in *Proceedings of the International Conference on Machine Learning*, 2009.

[215] D. Springer, L. Tarassenko, and G. Clifford, "Logistic regression-hsmm-based heart sound segmentation," 2015.

[216] B. Sriperumbudur, A. Gretton, K. Fukumizu, G. Lanckriet, and B. Schölkopf, "Injective Hilbert space embeddings of probability measures," in *Proc. Annual Conf. Computational Learning Theory*, 2008, pp. 111–122.

[217] M. Sugiyama and K. Borgwardt, "Halting in random walk kernels," in *Advances in Neural Information Processing Systems*, 2015, pp. 1630–1638.

[218] S. Sukhbaatar, J. Weston, R. Fergus, *et al.*, "End-to-end memory networks," in *Neural Information Processing Systems*, 2015.

[219] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[220] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[221] SyGuS Competition, http://sygus.seas.upenn.edu/SyGuS-COMP2017.html, 2017.

[222] S. Szymkuc, E. P. Gajewska, T. Klucznik, K. Molga, P. Dittwald, M. Startek, M. Bajczyk, and B. A. Grzybowski, "Computer-assisted synthetic planning: The end of the beginning," *Angew. Chem., Int. Ed.*, vol. 55, no. 20, pp. 5904–5937, 2016.

[223] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the Association for Computational Linguistics (ACL)*, 2015.

[224] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th International Conference on World Wide Web*, International World Wide Web Conferences Steering Committee, 2015, pp. 1067–1077.

[225] Y. Tang and R. R. Salakhutdinov, "Learning stochastic feedforward neural networks," in *Advances in Neural Information Processing Systems*, 2013, pp. 530–538.

[226] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, pp. 2319–2322, 2000.

[227] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun, "Large margin methods for structured and interdependent output variables," *Journal of machine learning research*, vol. 6, no. Sep, pp. 1453–1484, 2005.

[228] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[229] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[230] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[231] S. V. N. Vishwanathan, N. N. Schraudolph, I. R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, 2010, In press.

[232] S. V. N. Vishwanathan and A. J. Smola, "Fast kernels for string and tree matching," in *Advances in Neural Information Processing Systems 15*, S. Becker, S. Thrun, and K. Obermayer, Eds., Cambridge, MA: MIT Press, 2003, pp. 569–576.

[233] M. Wainwright, T. Jaakkola, and A. Willsky, "Tree-reweighted belief propagation and approximate ML estimation by pseudo-moment matching," in *9th Workshop on Artificial Intelligence and Statistics*, 2003.

[234] M. J. Wainwright and M. I. Jordan, "Graphical models, exponential families, and variational inference," *Foundations and Trends in Machine Learning*, vol. 1, no. 1 – 2, pp. 1–305, 2008.

[235] N. Wale, I. A. Watson, and G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," *Knowledge and Information Systems*, vol. 14, no. 3, pp. 347–375, 2008.

[236] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[237] D. Weininger, "Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules," *Journal of chemical information and computer sciences*, vol. 28, no. 1, pp. 31–36, 1988.

[238] B. Weisfeiler and A. A. Lehman, "A reduction of a graph to a canonical form and an algebra arising during this reduction," *Nauchno-Technicheskaya Informatsia*, vol. 2, no. 9, pp. 12–16, 1968.

[239] P. M. Williams, "Bayesian conditionalisation and the principle of minimum information," *British Journal for the Philosophy of Science*, vol. 31, no. 2, pp. 131–144, 1980.

[240] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[241] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[242] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Generalized belief propagation," in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds., MIT Press, 2001, pp. 689–695.

[243] J. Yedidia, W. Freeman, and Y. Weiss, "Bethe free energy, kikuchi approximations and belief propagation algorithms," Mitsubishi Electric Research Laboratories, Tech. Rep., 2001.

[244] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," in *Advances in Neural Information Processing Systems*, 2018, pp. 6410–6421.

[245] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: Generating realistic graphs with deep auto-regressive models," *arXiv preprint arXiv:1802.08773*, 2018.

[246] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[247] S.-Z. Yu, "Hidden semi-markov models," *Artificial Intelligence*, vol. 174, no. 2, pp. 215–243, 2010.

[248] S.-Z. Yu and H. Kobayashi, "An efficient forward-backward algorithm for an explicit-duration hidden markov model," *Signal Processing Letters, IEEE*, vol. 10, no. 1, pp. 11–14, 2003.

[249] A. L. Yuille, "Cccp algorithms to minimize the bethe and kikuchi free energies: Convergent alternatives to belief propagation," *Neural Computation*, vol. 14, no. 7, pp. 1691–1722, Jul. 2002.

[250] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, "Deep sets," in *Advances in neural information processing systems*, 2017, pp. 3391–3401.

[251] A. Zellner, "Optimal Information Processing and Bayes's Theorem," *The American Statistician*, vol. 42, no. 4, Nov. 1988.

[252] W. Zhang and T. G. Dietterich, "Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling," *Journal of Artificial Intelligence Reseach*, vol. 1, pp. 1–38, 2000.

[253] X. Zhang, L. Lu, and M. Lapata, "Top-down tree long short-term memory networks," *arXiv preprint arXiv:1511.00060*, 2015.

[254] S. Zheng, S. Jayasumana, B. Romera-Paredes, V. Vineet, Z. Su, D. Du, C. Huang, and P. Torr, "Conditional random fields as recurrent neural networks," *arXiv preprint arXiv:1502.03240*, 2015.