UNIVERSITY OF CALIFORNIA

Los Angeles

Neuro-Symbolic AI: A Probabilistic Perspective

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Kareem Ahmed Yousry Abdelraof Ahmed

2024

ABSTRACT OF THE DISSERTATION

Neuro-Symbolic AI: A Probabilistic Perspective

by

Kareem Ahmed Yousry Abdelraof Ahmed
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2024
Professor Guy Van den Broeck, Co-Chair
Professor Kai-Wei Chang, Co-Chair

The last decade has witnessed an explosion of interest in Artificial Intelligence, not only among researchers, but also in the public eye. This has led to machine learning (ML) systems being increasingly deployed in domains with high personal and societal impact. Consequently, there's a growing need to reason about the behavior of these machine learning systems, with the ultimate goal of mitigating unwanted behaviors, ensuring reliable outputs, and fostering trustworthy interactions with the end user. This dissertation lays the foundations for reasoning about the behavior of such systems, especially when we have access to *domain knowledge*—sets of rules, or constraints, that characterize the set of valid predictions given the problem at hand. In particular, a core contribution of this dissertation is *viewing ML systems as inducing probability distributions over output spaces*. To reason about the behavior of these systems, one must then reason about the behavior of the underlying probability distributions. Building upon that perspective, the dissertation begins by developing methods that *minimize the probability* of ML systems producing invalid outputs, even when the constraints and distributions are theoretically intractable. It then goes further, developing methods that *provide guarantees* on the outputs of these ML systems, developing probabilistically-

sound approaches to gradient estimation when the constraints are embedded within the architecture. The developed methods make use of *tractable circuits* whose structure differs from that of typical ML architectures. This dissertation, therefore, develops frameworks for *expressing constraints as Python functions* that can then be *efficiently computed on* GPUs. Lastly, this dissertation showcases the scalability and efficacy of the developed methods on real-word applications.

The dissertation of Kareem Ahmed Yousry Abdelraof Ahmed is approved.

Sameer Singh

Yizhou Sun

Kai-Wei Chang, Committee Co-Chair

Guy Van den Broeck, Committee Co-Chair

University of California, Los Angeles

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My PhD has been one of the most challenging experiences of my life. And definitely the most rewarding. Growing up, I was always exposed to the Hollywood version of scientist: a brilliant lone wolf with eureka moments. What I've learned during my PhD is that you have to work hard, often very hard, for these moments of eureka. Which do eventually show up, late as they may be. I've also learnt that science is a team sport, and that any one person who endeavors to do science must stand on the shoulders of giants, past and present.

First and foremost, I am extremely grateful to my advisors Guy Van den Broeck and Kai-Wei Chang. They both took a chance on an idealistic PhD student who had lost his way. They mentored me, offered me guidance, as well as what felt like unlimited freedom to explore my own ideas.

I remain indebted to Arthur Choi for all his help, support—technical and emotional—and friendship throughout my PhD. I can not recount an instance when I reached out, and he was not available and willing to help or listen. I am very grateful for Antonio Vergari, who has always made the time and space to hear me out during my hardship, and for our long walks during COVID.

I am honored to have Professors Sameer Singh and Yizhou Sun on my doctoral committee. I am grateful for their valuable feedback on this thesis as well as insightful questions and discussions.

I consider myself very lucky to have shared this journey with my academic siblings as well as UCLA friends: Aishwarya Sivaraman, Zeina Migeed, Tal Friedman, Yitao Liang, Steven Holtzen, Pasha Khosravi, Zhe Zeng, Honghua Zhang, Anji Liu, Meihua Dang, Poorva Garg and Tao Meng. All of you made the hard times easier, and the easy times fun.

To my beautiful Madeline: My blessing, my biggest blessing. We've shared a tough year on the job market. Your love and support keep me going. Thank you for putting up with my crankiness and anxiety. My days are infinitely sweeter, and my life infinitely better because I have you in it.

Last but by no means least, I am grateful for my family, day and night. Mom, Dad, Omar and Zeina, your love and support propels me forward. I truly could not have made it through this journey without your unconditional love and support.

VITA

2009–2014 B.Sc. in Computer Science. The German University in Cairo.

2014–2017 M.Sc. in Computer Science. The Technical University of Munich.

PUBLICATIONS

Anji Liu, **Kareem Ahmed**, and Guy Van den Broeck. Scaling Tractable Probabilistic Circuits: A Systems Perspective. *In Proceedings of the 41st International Conference on Machine Learning (ICML), 2024.*

Chendi Qian*, Andrei Manolache*, **Kareem Ahmed**, Zhe Zeng, Guy Van den Broeck, Mathias Niepert, and Christopher Morris. Probabilistic Task-Adaptive Graph Rewiring. *In Proceedings of the 12th International Conference on Learning Representations (ICLR), 2024.*

Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. A Pseudo-Semantic Loss for Deep Autoregressive Models with Logical Constraints. *In Advances in Neural Information Processing Systems 36 (NeurIPS), 2023.*

Vinay Shukla, Zhe Zeng*, **Kareem Ahmed***, and Guy Van den Broeck. A Unified Approach to Count-Based Weakly-Supervised Learning. *In Advances in Neural Information Processing Systems 36 (NeurIPS), 2023.*

Kareem Ahmed*, Zhe Zeng*, Mathias Niepert, and Guy Van den Broeck. SIMPLE: A gradient estimator for k-subset sampling. *In Proceedings of the 11th International Conference on Learning Representations (ICLR), 2023.*

Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. Semantic Strengthening of Neuro-Symbolic Learning. *In Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS), 2023.*

Kareem Ahmed, Stefano Teso, Kai-Wei Chang, Guy Van den Broeck, and Antonio Vergari. Semantic Probabilistic Layers for Neuro-Symbolic Learning. *In Advances in Neural Information Processing Systems 35 (NeurIPS), 2022.*

Kareem Ahmed, Eric Wang, Kai-Wei Chang, and Guy Van den Broeck. Neuro- Symbolic Entropy Regularization. *In Proceedings of the 38th Conference on Uncertainty in Artificial Intelligence (UAI), 2022.* ***Selected for oral Presentation.***

Kareem Ahmed, Tao Li, Thy Ton, Quan Guo, Kai-Wei Chang, Parisa Kordjamshidi, Vivek Srikumar, Guy Van den Broeck, and Sameer Singh. PYLON: A Pytorch Framework for Learning with Constraints. *In Proceedings of the 36th AAAI Conference on Artificial Intelligence (Demo Track), 2022.*

Kareem Ahmed, Tao Li, Thy Ton, Quan Guo, Kai-Wei Chang, Parisa Kordjamshidi, Vivek Srikumar, Guy Van den Broeck, and Sameer Singh. PYLON: A Pytorch Framework for Learning with Constraints. *In Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track, 2022.*

CHAPTER 1

Introduction

Neural networks have achieved resounding success across many domains, leading to their widespread adoption across many walks of life, from drug design to self-driving cars. In such settings, we do not want to predict a single output, e.g., whether an image contains a cat, but many outputs that *constitute a semantically meaningful object*, e.g., a protein structure. This leads to output spaces that scale exponentially in the number of variables, and it is not possible to see enough data to *faithfully* recover the true underlying function. How can we then hope to develop *trustworthy* systems?

Knowledge is everywhere! Sets of rules, or constraints, specified in formal language, characterizing the set of predictions admissible in a given problem domain. The presence of such constraints is, in principle, very beneficial. During training time, they greatly restrict our hypothesis space by precluding models inconsistent with the constraints, reducing the amount of data required to recover the target function. At inference time, they restrict the set of possible predictions to those admissible in the current domain. The challenge, however, then becomes: how do we consolidate the *continuous nature of gradient-based learning* with the *discrete nature of constraints*?

By taking a probabilistic approach to this problem, viewing neural networks as inducing probability distributions over output spaces reasoning about such distributions, this dissertation aims to develop, from first principles, *tractable* approaches that leverage probabilistic semantics to consolidate purely *statistical approaches to learning*, chiefly using neural networks, with purely *symbolic approaches to reasoning*. The Holy Grail of this research is to conceive of methods that address shortcomings of both paradigms: developing scalable approaches that learn from unstructured data while leveraging constraints to ensure the *explainable* and *trustworthy* behavior of neural networks.

1.1 Structure of the Dissertation

Chapter 2, based on Ahmed et al. [2022c], Ahmed et al. [2023a], and Ahmed et al. [2023b] starts with a starry-eyed perspective to integrating logical reasoning with statistical learning, where a simple means to integrating the constraint into learning is by optimizing for the set of parameters that maximize the probability of not only the data but also the constraint. It then proceeds to exploit a common assumption in machine learning, namely that data belonging to the same class tend to form discrete clusters to propose a new neuro-symbolic loss termed *neuro-symbolic entropy*. It then moves on to relax the assumption at the core of many exact neuro-symbolic approaches, the ability to compile a given constraint into a compact circuit, and derives a probabilistic approach to scaling probabilistic inference for neuro-symbolic learning while retaining the sound semantics of the logic. Lastly, it moves beyond minimizing the probability of the constraint under fully-factorized output distributions and towards autoregressive distributions, approximating the probability of the constraint w.r.t. the autoregressive distribution by its probability in a local pseudolikelihood distribution centered around a model sample, resulting in an efficient and high-fidelity approximation.

Chapter 3, based on Ahmed et al. [2022b] and Ahmed et al. [2023c], moves towards developing methods that provide *guarantees* on a system's behavior. To that end, it proposes *semantic probabilistic layers (SPLs)*, drop-in replacements for the traditional Softmax layer that guarantee the neural network's predictions are consistent with a set of constraints, while being amenable to end-to-end learning. Very often the utility of constraints can extend beyond just the output layer of a neural network to being part of the neural network architecture. This necessitates taking discrete samples and taking gradients w.r.t. these samples, which are inherently differentiable. Reparameterizing the samples in terms of the marginals, this dissertation proceeds to show that the gradient of the loss w.r.t. the samples can be estimated as the gradient w.r.t. the marginals of the distribution over all subsets of size k. This constitutes a new, general gradient estimator, SIMPLE, that exhibits lower bias and lower variance compared to state-of-the-art gradient estimators, and which can be applied in the context of any constraint that can be compiled into a tractable circuit.

Chapter 4, based on Ahmed et al. [2022a] and Liu et al. [2024a], develops a unifying framework that seamlessly integrates with existing deep learning code and allows users to easily specify and utilize constraints, augmenting procedurally trained neural networks with declaratively specified constraints. When the PyTorch function can be compiled into a tractable circuit Pylon makes use of PyJuice, another framework developed by this dissertation, to efficiently compute the loss. PyJuice interprets the circuit as a layerwise computational graph, leading to maximal GPU utilization and running times orders of magnitude faster than older implementations.

Chapter 5, based on Shukla et al. [2023] and Qian et al. [2023] showcases the scalability and efficacy of the approaches developed throughout the dissertation. One such application is weakly-supervised learning, where high-quality labels are often very scarce, whereas data with partial labels is more readily available due to privacy or budget constraints. The dissertation derives a count loss penalizing the model for deviations in its distribution from an arithmetic constraint defined over label counts. Another application is learning the structure of graph neural networks, where this dissertation proposed probabilistically rewired message-passing graph neural networks (PR-MPNNs). Building upon SIMPLE, the approach learns to add relevant edges while omitting less beneficial ones, overcoming many of the pitfalls of state-of-the-art algorithms.

Finally, Chapter 6 concludes by summarizing the thesis and discussing future directions.

CHAPTER 2

Learning with Constraints

A simple means to integrating constraints into learning is by optimizing for the set of parameters that maximize the probability of not only the data but also the constraint. We can compile the constraint into a *tractable circuit*, a compact computational representation which we can parameterize to induce a distribution over the possible worlds of the constraint. This yields the probability we seek to maximize. An assumption core to learning is that data belonging to the same class tend to form discrete clusters. Minimizing the entropy [Grandvalet and Bengio, 2005] of the distribution can thus be regarded as minimizing a measure of class overlap under the learned representation. That is, we should aim to learn minimum entropy distributions satisfying the constraint. Naively we might consider maximizing the probability of the constraints while minimizing the entropy. However, the entropy is agnostic to the constraint and, therefore, likely to steer the network towards predictions that violate it. We propose a new loss, *neuro-symbolic entropy regularization*, minimizing the distribution's entropy restricted to the possible worlds of the constraint and derived an efficient algorithm for its computation using tractable circuits.

An assumption at the heart of many exact neuro-symbolic approaches is the ability to compile a given constraint into a compact circuit by exploiting the structure inherent in the problem. Unfortunately, many constraints do not exhibit sufficient structure to yield a compact circuit, which can very often grow exponentially. We propose a new approach, *semantic strengthening*, a probabilistic approach to scaling probabilistic inference for neuro-symbolic learning while retaining the sound semantics of the logic. We start by assuming that the probability of the constraint decomposes, conditioned on the networks learned features. We thereby reduce the (often intractable)

problem of probabilistically satisfying the global constraint, e.g., that a Sudoku puzzle is valid, to the (tractable) problem of probabilistically satisfying the local constraints, e.g. the uniqueness of the elements of a row, column, or square. This, however, introduces inconsistencies: an assignment that satisfies one constraint might violate another, leading to misaligned gradients. We give an algorithm for tractably computing the conditional mutual information, a measure of modeling error incurred by assuming the constraints to be independent when they are in fact dependent. We interleave learning with semantic strengthening, iteratively tightening our approximation, using the network to guide constraint joining.

Previous approaches to neuro-symbolic AI assume the outputs of a neural network to be conditionally independent given the learned features and therefore the distribution over the output space to be fully-factorized, disregarding any potential correlations among the individual variables. In my work, *pseudo-semantic loss*, we move beyond fully-factorized output distributions and towards *autoregressive distributions*, including those *induced by LLMs* such as GPT, where the output at any given time step depends on the outputs at all previous time steps. Computing the probability of an arbitrary constraint under a fully-factorized distribution is already computationally hard, owing to the exponentially-many possible worlds of the constraint and the lack of structure to the solution space. Under an autoregressive distribution, however, computing the probability of even a single literal as a constraint is hard. To that end, we approximate the probability of the constraint w.r.t. the autoregressive distribution by its probability in a local pseudolikelihood distribution centered around a model sample resulting in a factorizable, efficient and high-fidelity approximation.

2.1 Neuro-Symbolic Entropy Regularization

Neural networks have achieved breakthroughs across a wide range of domains. Such breakthroughs are often only possible in the presence of large labeled datasets, which can be hard to obtain. Increasing efforts are therefore being devoted to approaches that utilize alternate sources of supervision in lieu of *more* labeled data. Entropy regularization constitutes one such approach [Grandvalet

and Bengio, 2005; Chapelle et al., 2010]. It posits that data belonging to the same class tend to form discrete clusters. Minimizing the entropy of the predictive distribution can thus be regarded as minimizing a measure of class overlap under the learned representation. Intuitively, a classifier guessing uniformly at random has *maximum entropy* and has not learned features that are informative of the underlying class. Consequently, we prefer a *minimum entropy* classifier that learns features *maximally informative* of the underlying class, even on unlabeled data.

The need for labeled data is only exacerbated in structured prediction, where the objective is to predict multiple interdependent output variables representing a discrete object. Viewed as traditional classification, the number of classes in structured prediction is exponential in the number of output variables—all possible output configurations. Neuro-symbolic methods can provide additional supervision, leveraging symbolic knowledge regarding the structure of the output space [De Raedt et al., 2020]. This knowledge characterizes the set of valid structures; for instance, a path in a graph is a series of *connected* edges connecting the source and destination vertices.

Here, we take a principled approach to unifying the aforementioned forms of supervision. Naively, we might consider simply optimizing both losses simultaneously. However, computed in that manner, entropy regularization does not account for the structure of the output space and is therefore likely to push the network towards invalid structures. Instead, we restrict the entropy loss to the network's distribution over the valid structures, as characterized by the constraint, as opposed to the entire predictive distribution, proposing *neuro-symbolic entropy regularization*. That is, we require that the network's output distribution be maximally informative of the target *subject to the constraint*. Intuitively, the network should "know" the right structure among the valid structures. Computing the entropy of a distribution subject to a constraint is, in general, computationally hard. We provide an algorithm leveraging structural properties of tractable logical circuits to efficiently compute this quantity. Our framework integrates seamlessly with other neuro-symbolic approaches that maximize the constraint probability, in effect "eliminating" invalid structures.

Empirically, we evaluate our loss on four structured prediction tasks, where we observe it leads to models whose predictions are more accurate, and more likely to satisfy the constraint.

2.1.1 Neuro-Symbolic Entropy Loss

We first introduce background on logical constraints and probability distributions over output structures. Afterwards, we motivate and define our neuro-symbolic entropy loss.

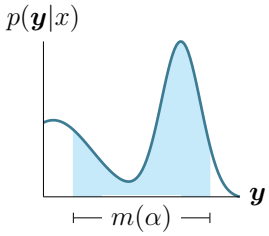
2.1.1.1 Background

We write uppercase letters (X, Y) for Boolean variables and lowercase letters (x, y) for their instantiation ($Y = 0$ or $Y = 1$). Sets of variables are written in bold uppercase ($\mathbf{X}$, $\mathbf{Y}$), and their joint instantiation in bold lowercase ($\mathbf{x}$, $\mathbf{y}$). A literal is a variable (Y) or its negation ($\neg Y$). A logical sentence (α or β) is constructed from variables and logical connectives ($\wedge$, $\vee$, etc.), and is also called a (logical) formula or constraint. A state or world $\mathbf{y}$ is an instantiation to all variables $\mathbf{Y}$. A state $\mathbf{y}$ satisfies a sentence α, denoted $\mathbf{y} \models \alpha$, if the sentence evaluates to true in that world. A state $\mathbf{y}$ that satisfies a sentence α is also said to be a model of α. We denote by $m(\alpha)$ the set of all models of α. The notation for states $\mathbf{y}$ is used to refer to an assignment, the logical sentence enforcing the assignment, or the binary output vector capturing the assignment. A sentence α entails another sentence β, denoted $\alpha \models \beta$, if all worlds that satisfy α also satisfy β.

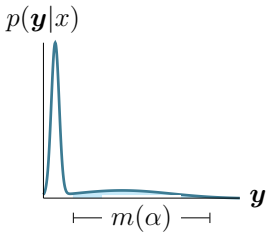
A Probability Distribution over Possible Structures Let α be a logical sentence defined over Boolean variables $\mathbf{Y} = \{Y_1, \ldots, Y_n\}$. Let p be a vector of probabilities for the same variables $\mathbf{Y}$, where p_i denotes the predicted probability of variable Y_i and corresponds to a single output of the neural network. The neural network's outputs induce a probability distribution $\mathrm{P}(\cdot)$ over all possible states $\mathbf{y}$ of $\mathbf{Y}$:

$$\mathrm{P}(\boldsymbol{y}) = \prod_{i:\boldsymbol{y}\models\mathbf{Y}_i} \mathsf{p}_i \prod_{i:\boldsymbol{y}\models\neg\mathbf{Y}_i} (1 - \mathsf{p}_i). \tag{2.1}$$

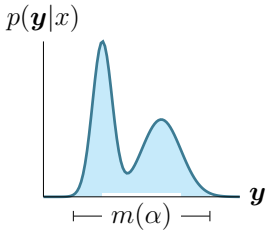
Semantic Loss The semantic loss [Xu et al., 2018a] is a function of the logical constraint α and a probability vector p. It quantifies how close the neural network comes to satisfying the constraint by computing the probability of the constraint under the distribution $\mathrm{P}(\cdot)$ induced by



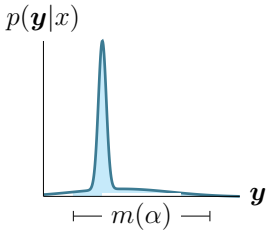
(a) Network uncertain over valid and invalid predictions



(b) Network allocating most mass to one invalid prediction



(c) Network allocating most mass to valid predictions



(d) Network allocating most mass to one valid prediction

Figure 2.1: A network's predictive distribution can be uncertain or certain ($\leftrightarrow$), and it can allow or disallow invalid predictions under the constraint α ($\updownarrow$). Entropy regularization steers the network towards confident, possibly invalid predictions (b). Neuro-symbolic learning steers the network towards valid predictions without necessarily being confident (c). Neuro-symbolic entropy-regularization guides the network to valid and confident predictions (d).

p. It does so by reducing the problem of probability computation to weighted model counting (WMC): summing up the models of α, each weighted by its likelihood under $\mathrm{P}(\cdot)$. It, therefore, maximizes the probability mass allocated by the network to the models of α

$$\mathbb{E}_{\boldsymbol{y}\sim\mathrm{P}}\left[\mathbb{1}\{\boldsymbol{y} \models \alpha\}\right] = \sum_{\boldsymbol{y}\models\alpha} \mathrm{P}(\boldsymbol{y}). \tag{2.2}$$

Taking the negative logarithm recovers semantic loss. We make use of semantic loss in our experiments to "eliminate" invalid structures under the neural network's distribution.

2.1.1.2 Motivation and Definition

Consider the plots in Figure 2.1. For any given data point x, the neural network can be fairly uncertain regarding the target class, accommodating for both valid and invalid structured predictions under its predicted distribution.

A common underlying assumption in many machine learning methods is that data belonging to the same class tend to form discrete clusters [Chapelle et al., 2010]—an assumption deemed justified on the sheer basis of the existence of classes. Consequently, a classifier is expected to favor decision boundaries lying in regions of low data density, separating the clusters. Entropy-regularization [Grandvalet and Bengio, 2005] directly implements the above assumption, requiring that the classifier output confident—low-entropy—predictive distributions, pushing the decision boundary away from unlabeled points, thereby supplementing scarce labeled data with abundant unlabeled data. Seen through that lens, minimizing the entropy of the predictive distribution can be regarded as minimizing a measure of class overlap as a function of the learned features.

Entropy regularization, however, remains agnostic to the underlying domain, failing to exploit situations where we have knowledge characterizing valid predictions in the domain. Therefore, it can often be harmful to a model's performance, causing it to grow confident in invalid predictions.

Conversely, neuro-symbolic approaches steer the network towards distributions disallowing invalid predictions, by maximizing the constraint probability, but do little to ensure the network learn features conducive to classification.

Clearly then, there is a benefit to combining the merits of both approaches. We restrict the entropy computation to the distribution over models of the logical formula, ensuring the network only grow confident in valid predictions. Complemented with maximizing the constraint probability, the network learns to allocate all of its mass to models of the constraint, while being maximally informative of the target.

Defining the Loss More precisely, let $\mathbf{Y}$ be a random variable distributed according to Equation 2.1: $\mathbf{Y} \sim \mathrm{P}$. We are interested in minimizing the entropy of $\mathbf{Y}$ conditioned on the constraint α

$$\begin{aligned} H(\mathbf{Y}|\alpha) &= -\sum_{\boldsymbol{y} \models \alpha} \mathrm{P}(\boldsymbol{y}|\alpha) \log \mathrm{P}(\boldsymbol{y}|\alpha) \\ &= -\mathbb{E}_{\mathbf{Y}|\alpha} \left[\log \mathrm{P}(\mathbf{Y}|\alpha)\right]. \end{aligned} \tag{2.3}$$

Algorithm 1 $\textsc{Ent}(\alpha, \mathrm{P}, \mathsf{c})$

Input: a smooth, deterministic and decomposable logical circuit α, a fully-factorized probability distribution $\mathrm{P}(\cdot)$ over states of α, and a cache c for memoization
Output: $H(\mathbf{Y}|\alpha)$, where $\mathbf{Y} \sim \mathrm{P}(\cdot)$

if $\alpha \in \mathsf{c}$ **then return** $\mathsf{c}(\alpha)$
if α is a literal **then**
    $e \leftarrow 0$
else if α is an AND gate **then**
    $e \leftarrow \textsc{Ent}(\beta, \mathrm{P}, \mathsf{c}) + \textsc{Ent}(\gamma, \mathrm{P}, \mathsf{c})$
else if α is an OR gate **then**
    $e \leftarrow \sum_{i=1}^{|\mathsf{ch}(\alpha)|} \mathrm{P}(\beta_i) \log \mathrm{P}(\beta_i) + \mathrm{P}(\beta_i)\, \textsc{Ent}(\beta_i, \mathrm{P}, \mathsf{c})$
$\mathsf{c}(\alpha) \leftarrow e$
return e

2.1.2 Computing the Loss

The above loss is, in general, hard to compute. To see this, consider the uniform distribution over models of a constraint α. That is, let $\mathrm{P}(\boldsymbol{y}|\alpha) = \frac{1}{|m(\alpha)|}$ for all $\boldsymbol{y} \models \alpha$. Then, $H(\mathbf{Y}|\alpha) = -\sum_{\boldsymbol{y} \models \alpha} \frac{1}{|m(\alpha)|} \log \frac{1}{|m(\alpha)|} = \log |m(\alpha)|$. This tells us how many models of α there are, which is a well-known #P-hard problem [Valiant, 1979a,b]. We will show that, through compilation into tractable circuits, we can compute Equation 2.3 in time linear in the size of the circuit.

2.1.2.1 Computation through Compilation

Tractable Circuit Compilation We resort to knowledge compilation techniques—a class of methods that transform, or *compile*, a logical theory into a target form with certain properties that allow certain probabilistic queries to be answered efficiently. More precisely, we know of circuit languages that compute the probability of constraints [Darwiche, 2000], and that are amenable to backpropagation. We use the circuit compilation techniques in Darwiche [2011a] to build a logical circuit representing our constraint. Due to the structural properties of this circuit form, we can use it to compute both the probability of the constraint as well as its gradients with respect to the network's weights, in time linear in the size of the circuit [Darwiche and Marquis, 2002]. This does not, in general, escape the complexity of the computation: worst case, the compiled circuit can

be exponential in the size of the constraint. In practice, however, constraints often exhibit enough structure (repeated sub-problems) to make compilation feasible. We refer to Section A.1.1 for an illustrative example of such a compilation.

Logical Circuits More formally, a *logical circuit* is a directed, acyclic computational graph representing a logical formula. Each node n in the DAG encodes a logical sub-formula, denoted $[n]$. Each inner node in the graph is either an AND or an OR gate, and each leaf node encodes a Boolean literal (Y or $\neg Y$). We denote by $\mathsf{ch}(n)$ the set of n's children, i.e., the operands of its logical gate.

Structural Properties As already alluded to, circuits enable the tractable computation of certain classes of queries over encoded functions granted that a set of structural properties are enforced.

A circuit is *decomposable* if the inputs of every AND gate depend on disjoint sets of variables i.e. for $\alpha = \beta \wedge \gamma$, $vars(\beta) \cap vars(\gamma) = \varnothing$. Intuitively, decomposable AND nodes encode local factorizations of the function. For the sake of simplicity, we assume that decomposable AND gates always have two inputs, a condition that can be enforced on any circuit in exchange for a polynomial increase in its size [Vergari et al., 2015; Peharz et al., 2020a].

A second useful property is *smoothness*. A circuit is *smooth* if the children of every OR gate depend on the same set of variables i.e. for $\alpha = \bigvee_i \beta_i$, we have that $vars(\beta_i) = vars(\beta_j)\ \forall i, j$. Decomposability and smoothness are a sufficient and necessary condition for tractable integration over arbitrary sets of variables in a single pass, as they allow larger integrals to decompose into smaller ones [Choi et al., 2020a].

Lastly, a circuit is said to be *deterministic* if, for any input, at most one child of every OR node has a non-zero output i.e. for $\alpha = \bigvee_i \beta_i$, we have that $\beta_i \wedge \beta_j = \bot$ for all $i \neq j$. Figure 2.2 shows an example of smooth, decomposable and deterministic circuit.

2.1.2.2 Algorithm

Let α be a *smooth*, *deterministic* and *decomposable* logical circuit encoding our constraint, defined over Boolean variables $\mathbf{Y} = \{Y_1, \ldots, Y_n\}$. We now show that we can compute the constrained entropy in Equation 2.3 in time linear in the size of α. The key insight is that we are able to efficiently decompose an expectation with respect to a fully-factorized distribution by alternately splitting the query variables and the support of the distribution until we reach the leaves of the circuit, which are simple literals. In what follows, in a slight abuse of notation for brevity, all unconditional probabilities are implicitly conditioned on constraint α; i.e., we redefine $\mathrm{P}(\cdot)$ as $\mathrm{P}(\cdot|\alpha)$.

Base Case: α is a literal

When α is a literal, $\alpha = Y_i$ or $\alpha = \neg Y_i$, we have that

$$\mathrm{P}(y_i|\alpha) = \mathbb{1}\{y_i \models [\alpha]\}, \text{ and}$$
$$H(y_i|\alpha) = -\mathrm{P}(y_i|\alpha)\log \mathrm{P}(y_i|\alpha) = 0.$$

Intuitively, a literal has no uncertainty associated with it.

Recursive Case: α is a conjunction

When α is a conjunction, decomposability enables us to write

$$\mathrm{P}(\boldsymbol{y}|\alpha) = \mathrm{P}(\boldsymbol{y}_1|\beta)\,\mathrm{P}(\boldsymbol{y}_2|\gamma), \text{ where } vars(\beta) \cap vars(\gamma) = \varnothing$$

as it decomposes α into two independent constraints β and γ, and $\boldsymbol{y}$ into two independent assignments $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$. The neuro-symbolic entropy $-\mathbb{E}_{\mathbf{Y}|\alpha}\left[\log \mathrm{P}(\mathbf{Y}|\alpha)\right]$ is then

$$-\mathbb{E}_{\{\mathbf{Y}_1,\mathbf{Y}_2\}|\alpha}\Big[\log \mathrm{P}(\mathbf{Y}_1|\beta) + \log \mathrm{P}(\mathbf{Y}_2|\gamma)\Big]$$

$$= -\Big[\mathbb{E}_{\mathbf{Y}_1|\beta}\big[\log \mathrm{P}(\mathbf{Y}_1|\beta)\big] + \mathbb{E}_{\mathbf{Y}_2|\gamma}\big[\log \mathrm{P}(\mathbf{Y}_2|\gamma)\big]\Big].$$

That is, the entropy if a decomposable conjunction α is the sum of entropies of the conjuncts.

Recursive Case: α is a disjunction

When α is a smooth and deterministic disjunction, we have that $\alpha = \bigvee_i \beta_i$, where the β_is are mutually exclusive, and therefore partition α. Consequently, we have that

$$\mathrm{P}(\boldsymbol{y}|\alpha) = \sum_i \mathrm{P}(\beta_i) \cdot \mathrm{P}(\boldsymbol{y}|\beta_i).$$

The neuro-symbolic entropy decomposes as well:

$$\begin{aligned}
&- \mathbb{E}_{\mathbf{Y}|\alpha}\left[\log \mathrm{P}(\mathbf{Y}|\alpha)\right] = -\sum_{\boldsymbol{y}\models\alpha} \mathrm{P}(\boldsymbol{y}|\alpha) \log \mathrm{P}(\boldsymbol{y}|\alpha) \\
&= -\sum_{\boldsymbol{y}\models\alpha} \sum_i \mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i) \log \Big[\sum_j \mathrm{P}(\beta_j)\, \mathrm{P}(\boldsymbol{y}|\beta_j)\Big] \\
&= -\sum_{\boldsymbol{y}\models\alpha} \sum_i \mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i) [\![\boldsymbol{y} \models \beta_i]\!] \\
&\qquad\qquad \log \Big[\sum_j \mathrm{P}(\beta_j)\, \mathrm{P}(\boldsymbol{y}|\beta_j) [\![\boldsymbol{y} \models \beta_j]\!]\Big],
\end{aligned}$$

where by determinism, we have that, for any $\boldsymbol{y}$ such that $\boldsymbol{y} \models \alpha$, $\boldsymbol{y} \models \beta_i \implies \boldsymbol{y} \not\models \beta_j$ for all $i \neq j$. In other words, any state that satisfies the constraint α satisfies one and only one of its terms, and therefore, the above expression equals

$$\begin{aligned}
&-\sum_{\boldsymbol{y}\models\alpha} \sum_i \mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i) \log \Big[\mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i)\Big] [\![\boldsymbol{y} \models \beta_i]\!] \\
&= -\sum_i \sum_{\boldsymbol{y}\models\beta_i} \mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i) \log \Big[\mathrm{P}(\beta_i)\, \mathrm{P}(\boldsymbol{y}|\beta_i)\Big].
\end{aligned}$$

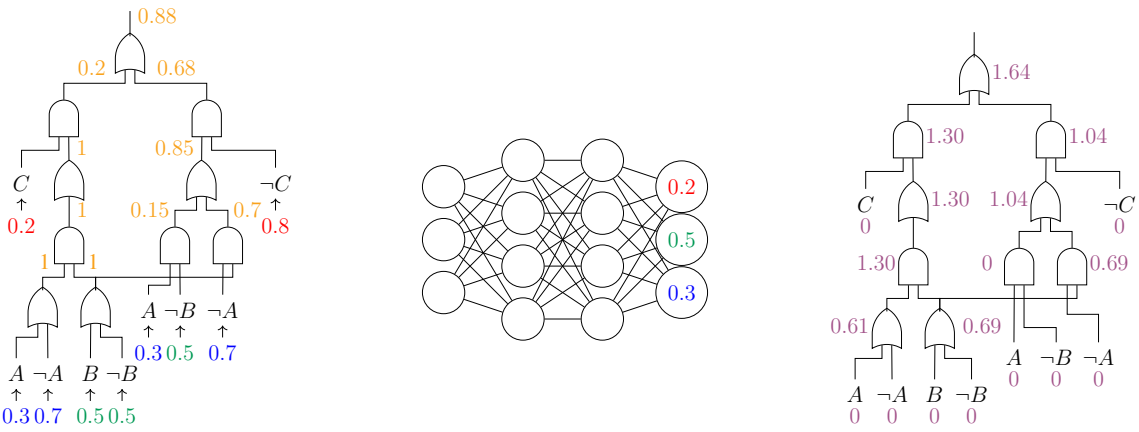


Figure 2.2: For a given data point, the network (middle) outputs a distribution over classes A, B and C, highlighted in blue, green and red, respectively. The circuit encodes the constraint $(A \wedge B) \implies C$. For each leaf node l, we plug in $\mathrm{P}(l)$ and $1 - \mathrm{P}(l)$ for positive and negative literals, respectively. The computation proceeds bottom-up, taking products at AND gates and summations at OR gates. The value accumulated at the root of the circuit (left) is the probability allocated by the network to the constraint. The weights accumulated on edges from OR gates to their children are of special significance: OR nodes induce a partitioning of the distribution's support, and the weights correspond to the mass allocated by the network to each mutually-exclusive event. Complemented with a second upward pass, where the entropy of an OR node is the entropy of the distribution over its children plus the expected entropy of its children, and the entropy of an AND node is the product of its children's entropies, we get the entropy of the distribution over the constraint's models—the neuro-symbolic entropy regularization loss (right).

Further simplifying the expression, expanding the logarithm, and using the fact that probability sums to 1 yields

$$
\begin{aligned}
&= -\sum_i \mathrm{P}(\beta_i) \log \mathrm{P}(\beta_i) \sum_{\boldsymbol{y} \models \beta_i} \mathrm{P}(\boldsymbol{y} | \beta_i) \\
&\qquad + \mathrm{P}(\beta_i) \sum_{\boldsymbol{y} \models \beta_i} \mathrm{P}(\boldsymbol{y} | \beta_i) \log \mathrm{P}(\boldsymbol{y} | \beta_i) \\
&= -\sum_i \mathrm{P}(\beta_i) \log \mathrm{P}(\beta_i) + \mathrm{P}(\beta_i) \mathbb{E}_{\mathbf{Y} | \beta_i} \Big[\log \mathrm{P}(\mathbf{Y} | \beta_i) \Big].
\end{aligned}
$$

That is, the entropy of the random variable $\mathbf{Y}$ conditioned on a disjunction α is the sum of the entropy of the distribution induced on the children of α, and the average entropy of its children. The full algorithm is illustrated in Algorithm 1.

2.1.3 An Illustrative example

Consider Figure 2.2. Given a data point, the neural network defines a distribution over Boolean random variables A, B, and C, where $\mathrm{P}(A) = \mathsf{p}_0$ and $\mathrm{P}(\neg A) = 1 - \mathsf{p}_0$, $\mathrm{P}(B) = \mathsf{p}_1$ and $\mathrm{P}(\neg B) = 1 - \mathsf{p}_1$, etc. The circuit encodes the constraint $(A \wedge B) \implies C$. To compute the the probability of the constraint under the network's distribution, we feed the probabilities into the circuit, proceeding in a bottom-up fashion, taking products at AND gates and summations at OR gates, accumulating intermediate computations on the edges of the circuit. The value accumulated at the root of the circuit is the probability mass allocated by the network to models of the formula, and corresponds to the probability of the constraint under the network's distribution – this is exactly the semantic loss, up to a negative logarithm. The weights accumulated on edges from OR gates to their children are of special significance: OR nodes induce a partitioning of the distribution's support, and the weights correspond to the mass allocated by the network to each mutually-exclusive event. Complemented with another upward pass, where the entropy of every OR node is the entropy of the distribution over it's children plus the expected entropy of its children, and the entropy of every AND node is the product of its children's entropies, we calculate the entropy of the distribution over models of the constraint – this is exactly the neuro-symbolic entropy regularization. Therefore, performing two upward sweeps of the circuit, we are able to compute the neuro-symbolic entropy regularization and the semantic loss

2.1.4 Experimental Evaluation

In this section we set out to empirically test our neuro-symbolic entropy loss. To that end, we devise a series of semi-supervised and fully-supervised structured prediction experiments. Such are settings where, contrary to the their dominant use, classifiers are expected to predict structured objects rather than scalar, discrete or real values. Such objects are defined in terms of constraints: a set of rules characterizing the set of solutions. We aim to answer the following:

1. Does entropy regularization lead to predictive models with improved generalization?

Table 2.1: Experimental results for entity-relation extraction on ACE05 and SciERC. #Labels indicates the number of labeled data points available to the network per relation. The remaining training set is stripped of labels and utilized in an unsupervised manner. We report the F1-score where a prediction is correct if the relation and its entities are correct.

# Labels		3	5	10	15	25	50	75
ACE05	Baseline	4.92 ± 1.12	7.24 ± 1.75	13.66 ± 0.18	15.07 ± 1.79	21.65 ± 3.41	28.96 ± 0.98	33.02 ± 1.17
	Self-training	7.72 ± 1.21	12.83 ± 2.97	16.22 ± 3.08	17.55 ± 1.41	27.00 ± 3.66	32.90 ± 1.71	37.15 ± 1.42
	Product t-norm	8.89 ± 5.09	14.52 ± 2.13	19.22 ± 5.81	21.80 ± 7.67	30.15 ± 1.01	34.12 ± 2.75	37.35 ± 2.53
	Semantic Loss	12.00 ± 3.81	14.92 ± 3.14	22.23 ± 3.64	27.35 ± 3.10	30.78 ± 0.68	36.76 ± 1.40	38.49 ± 1.74
	+ Full Entropy	**14.80** ± 3.70	15.78 ± 1.90	23.34 ± 4.07	28.09 ± 1.46	31.13 ± 2.26	36.05 ± 1.00	39.39 ± 1.21
	+ NeSy Entropy	14.72 ± 1.57	**18.38** ± 2.50	**26.41** ± 0.49	**31.17** ± 1.68	**35.85** ± 0.75	**37.62** ± 2.17	**41.28** ± 0.46
SciERC	Baseline	2.71 ± 1.10	2.94 ± 1.00	3.49 ± 1.80	3.56 ± 1.10	8.83 ± 1.00	12.32 ± 3.00	12.49 ± 2.60
	Self-training	3.56 ± 1.40	3.04 ± 0.90	4.14 ± 2.60	3.73 ± 1.10	9.44 ± 3.80	14.82 ± 1.20	13.79 ± 3.90
	Product t-norm	**6.50** ± 2.00	8.86 ± 1.20	10.92 ± 1.60	13.38 ± 0.70	13.83 ± 2.90	19.20 ± 1.70	19.54 ± 1.70
	Semantic Loss	6.47 ± 1.02	**9.31** ± 0.76	11.50 ± 1.53	12.97 ± 2.86	14.07 ± 2.33	20.47 ± 2.50	23.72 ± 0.38
	+ Full Entropy	6.26 ± 1.21	8.49 ± 0.85	11.12 ± 1.22	14.10 ± 2.79	17.25 ± 2.75	**22.42** ± 0.43	24.37 ± 1.62
	+ NeSy Entropy	6.19 ± 2.40	8.11 ± 3.66	**13.17** ± 1.08	**15.47** ± 2.19	**17.45** ± 1.52	22.14 ± 1.46	**25.11** ± 1.03

2. If the answer to the above question is positive, it is our expectation that restricting the distribution acted upon by entropy regularization to that over just the models of the constraint might seem more sensible as compared to entropy-regularizing the entire predictive distribution–including non-models of the constraint. Do the results support this?

3. Finally, entropy regularization can be interpreted as clustering the different classes, and has intimate connections to transductive Support Vector Machines [Chapelle et al., 2010]. Does such an interpretation carry over to models and non-models of the constraint? Put differently, can we expect entropy-regularized predictive models to better conform to our constraints, measured by the percentage of predictions satisfying the constraint?.

2.1.4.1 Semi-Supervised: Entity-Relation Extraction

We begin by testing our research questions in the semi-supervised setting. Here the model is presented with only a portion of the labeled training set, with the rest used exclusively in an unsupervised manner by the respective approaches.

We make use of the natural ontology of entity types and their relations present when dealing with relational data. This defines a set of relations and their permissible argument types. As is with all of our constraints, we express the aforementioned ontology in the language of Boolean logic.

Our approach to recognizing the named entities and their pairwise relations is most similar to Zhong and Chen [2020]. Contextual embeddings are first procured for every token in the sentence. These are then fed into a named entity recognition module that outputs a vector of per-class probability for every entity. A classifier then classifies the concatenated contextual embeddings and entity predictions into a relation.

We employ two entity-relation extraction datasets, the Automatic Content Extraction (ACE) 2005 [Walker et al., 2006] and SciERC datasets [Luan et al., 2018]. ACE05 defines an ontology over 7 entities and 18 relations from mixed-genre text, whereas SciERC defines 6 entity types with 7 possible relation between them and includes annotations for scientific entities and there relations, assimilated from 12 AI conference/workshop proceedings. We report the percentage of coherent predictions: data points for which the predicted entity types, as well as the relations are correct.

We compare against five baselines. The first baseline is a purely supervised model which makes no use of unlabeled data. The second is a classical self-training approach based off of Chang et al. [2007], and uses integer linear programming to impute the unlabeled data's most likely labels subject to the constraint, and consequently augment the (small) labeled set. The third baseline is a popular instantiation of a broad class of methods, fuzzy logics, which replace logical operators with their fuzzy t-norms and logical implications with simple inequalities. Lastly, we compare our proposed method, dubbed "NeSy Entropy", to vanilla semantic loss as proposed in Xu et al. [2018a] as well as another entropy-regularized baseline, dubbed "Full Entropy", which minimizes the entropy of the entire predictive distribution, as opposed to just the distribution over the constraint's models.

Our results are shown in Table 2.1. We observe that semantic loss outperforms the baseline, self-training, and product t-norm across the board. We attribute such a performance to the exactness of semantic loss, and its faithfulness to the underlying constraint. We also observe that

Table 2.2: Grid shortest path test results

Test accuracy %	Coherent	Incoherent	Constraint
5-layer MLP	5.62	**85.91**	6.99
Semantic loss	28.51	83.14	69.89
+ Full Entropy	29.02	83.76	75.23
+ NeSy Entropy	**30.12**	83.01	**91.61**

Table 2.3: Preference prediction test results

Test accuracy %	Coherent	Incoherent	Constraint
3-layer MLP	1.01	**75.78**	2.72
Semantic loss	15.03	72.43	69.83
+ Full Entropy	17.52	71.80	80.21
+ NeSy Entropy	**18.17**	71.51	**96.04**

entropy-regularizing the predictive model, in conjunction with training using semantic loss leads to better predictive models, as compared with models trained solely using semantic loss. Furthermore, it turns out that restricting entropy to the distribution over the constraint's models, models that we know constitute the set of valid predictions, compared to the model's entire predictive distribution leads to a significant increase in the accuracy of predictions.

2.1.4.2 Fully-Supervised Learning

We now turn our attention to testing our hypotheses in a fully supervised setting, where our aim is to examine the effect of constraints enforced on the training set. We note that this is a seemingly harder setting in the following sense: In a semi- supervised setting we might make the argument that, despite its abundance, imposing an auxiliary loss on unlabeled data provides the predictive model with an unfair advantage as compared to the baseline. We concern ourselves with two tasks: predicting paths in a grid and preference learning.

Predicting Simple Paths For this task, our aim is to find the shortest path in a graph, or more specifically a 4-by-4 grid, $G = (V, E)$ with uniform edge weights. Our input is a binary vector of length $|V| + |E|$, with the first $|V|$ variables indicating the source and destination, and the next $|E|$ variables encoding a subgraph $G' \subseteq G$. Each label is a binary vector of length $|E|$ encoding the shortest *simple* path in G', a requirement that we enforce through our constraint. We follow the algorithm proposed by Nishino et al. [2017] to generate a constraint for each simple path in the grid, conjoined with indicators specifying the corresponding source-destination pair. Our constraint is

Table 2.4: Warcraft shortest path prediction results

Test accuracy %	Coherent	Incoherent	Constraint
ResNet-18	44.8	97.7	56.9
Semantic loss	50.9	97.7	67.4
+ Full Entropy	51.5	97.6	67.7
+ NeSy Entropy	**55.0**	**97.9**	**69.8**

then the disjunction of all such conjunctions.

To generate the data, we begin by randomly removing one third of the edges in the graph G, resulting in a subgraph, G'. Subsequently, we filter out connected components in G' with fewer than 5 nodes to reduce degenerate cases. We then sample a source and destination node uniformly at random. The latter constitutes a single data point. We generate a dataset of 1600 examples, with a $60/20/20$ train/validation/test split.

Preference Learning We also consider the task of preference learning. Given the user's ranking of a subset of elements, we wish to predict the user's preferences over the remaining elements of the set. We encode an ordering over n items as a binary matrix X_{ij}, where for each $i, j \in 1, \ldots, n$, X_{ij} denotes that item i is at position j. Our constraint α requires that the network's output be a valid total ordering. We use preference ranking data over 10 types of sushi for $5,000$ individuals, taken from PREFLIB [Mattei and Walsh, 2013a], split 60/20/20. Our inputs consist of the user's preference over 6 sushi types, with the model tasked to predict the user's preference, a *strict* total order, over the remaining 4.

Tables 2.2 and 2.3 compare the baseline to the same MLP augmented with semantic loss, semantic loss with entropy regularization over the entire predictive distribution, "Full Entropy", and entropy regularization over the distribution over the constraint's models, "NeSy Entropy".

Similar to Xu et al. [2018a], we observe that the semantic loss has a marginal effect on incoherent accuracy, but significantly improves the networks ability to output coherent predictions. We also observe that, similar to semi-supervised settings, entropy-regularization leads to more coher-

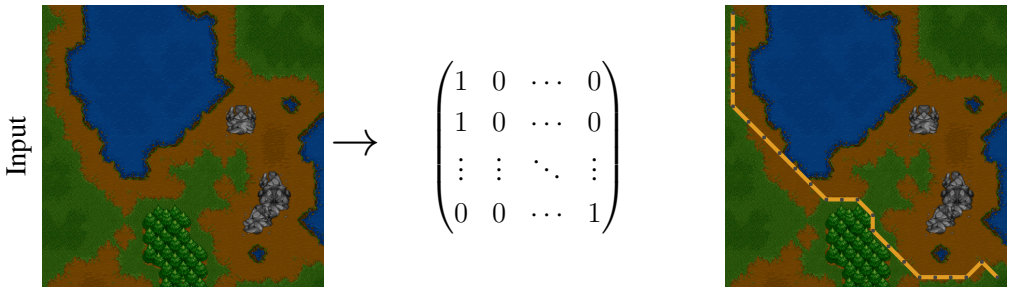


Figure 2.3: Warcraft dataset. Each input (left) is a 12×12 grid corresponding to a Warcraft II terrain map, the output is a matrix (middle) indicating the shortest path from top left to bottom right (right).

ent predictions using both "Full Entropy" and "NeSy Entropy", with "NeSy Entropy" leading to the best performing predictive models. Remarkably, we also observe that "NeSy Entropy" leads to predictive models whose predictions almost always satisfy the constraint, denoted "Constraint".

Warcraft Shortest Path Lastly, we consider a more real-world variant of the task of predicting simple paths. Following [Pogančić et al., 2019], our training set consists of $10,000$ terrain maps curated using Warcraft II tileset. Each map encodes an underlying grid of dimension 12×12, where each vertex is assigned a cost depending on the type of terrain it represents (e.g. earth has lower cost than water). The shortest (minimum cost) path between the top left and bottom right vertices is encoded as an indicator matrix, and serves as label. Figure 2.3 shows an example input presented to the network, the groundtruth, and the input with the annotated shortest path. Figure 2.4 shows examples of baseline predictions and those obtained by training with constraints.

Presented with an image of a terrain map, a convolutional neural network – following [Pogančić et al., 2019], we use ResNet18 [He et al., 2016a] – outputs a 12×12 binary matrix indicating the vertices that constitute the minimum cost path. We report three metrics: "Coherent" denotes the percentage of optimal-cost predictions, "Incoherent" denotes the percentage of individual vertices matching the groundtruth, and "Constraint" indicates the percentage of predictions that constitute valid paths. Our results are shown in Table 2.4.

In line with our previous experiments, we observe that incorporating constraints into learning

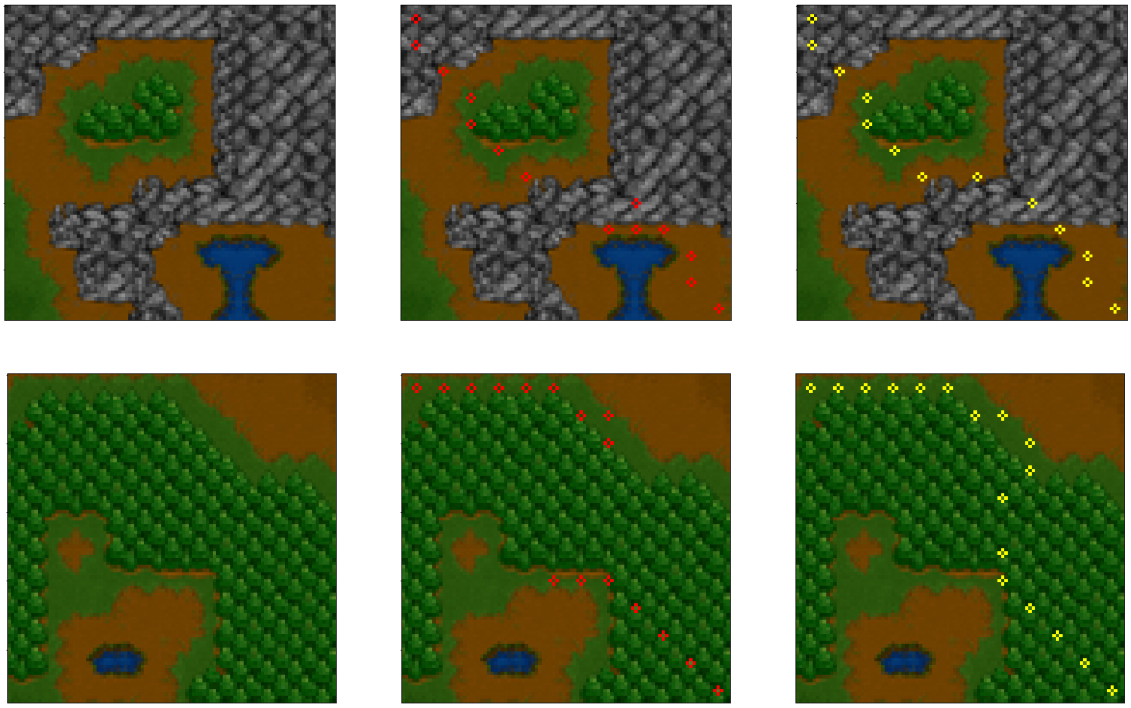


Figure 2.4: Example maps from the Warcraft dataset (left) annotated with the baseline predictions in red (center), and the predictions obtained using constraints in yellow (right)

improves the "Coherent" metric from 44.8% to 50.9%, and of the "Coherent" metric from 56.9% to 67.4%. Augmenting semantic loss with the entropy over the network's predictive distribution, "Full Entropy", we attain a modest improvement from 50.9% to 51.5% and 67.4% to 67.7% for the "Coherent" and "Constraint" metrics respectively. Restricting the entropy minimization to models of the constraint, "NeSy Entropy", we observe that we attain a large improvement to 55.0% and 69.8% for the "Coherent" and "Constraint" metrics, respectively.

2.1.5 Related Work

The idea of using a model's predictions to obtain artificial labels for unlabeled data is as old as time [Scudder, 1965; McLachlan, 1975], and has often known throughout the literature as pseudo-labeling or self-training. Self-training is an iterative process by which a learner imputes the labels of examples which have been confidently classified in the previous step, and can therefore be

viewed as implicitly minimizing the model's entropy. This is done explicitly in Grandvalet and Bengio [2005] with a loss term which minimizes the entropy of the model's predicted distribution for any given unlabeled data point, thereby rendering the entropy computation amenable to differentiation, and allowing finer control on the influence of the unlabeled data. It has been applied successfully across many domains, including NLP [McClosky et al., 2006], object detection [Rosenberg et al., 2005], image classification [Lee, 2013; Xie et al., 2019], domain adaptation [Zou et al., 2018]. It has also been used recently by a plethora of semi-supervised learning algorithms as a constituent of their training pipelines [Arazo et al., 2019; Pham and Le, 2019; Miyato et al., 2018; Berthelot et al., 2019]. This is in contrast to entropy maximization in reinforcement learning where the aim is to capture the range of low-cost behaviors [Toussaint, 2009].

In an acknowledgment to the need for both symbolic as well as sub-symbolic reasoning, there has been a plethora of recent works studying how to best combine neural networks and logical reasoning, dubbed *neuro-symbolic reasoning*. The focus of such approaches is making probabilistic reasoning tractable through first-order approximations, and differentiable, reducing logical formulas into arithmetic objectives, replacing logical operators with their fuzzy t-norms, and implications with inequalities [Kimmig et al., 2012; Rocktäschel et al., 2015; Fischer et al., 2019].

Diligenti et al. [2017a] and Donadello et al. [2017] use first-order logic to specify constraints on outputs of a neural network. They employ fuzzy logic to reduce logical formulas into differential, arithmetic objectives denoting the extent to which neural network outputs violate the constraints, thereby supporting end-to-end learning under constraints. More recently, Xu et al. [2018a] introduced semantic loss, which circumvents the shortcomings of fuzzy approaches, while still supporting end-to-end learning under constraints. More precisely, *fuzzy reasoning* is replaced with *exact probabilistic reasoning*, made possible by compiling logical formulae into structures supporting efficient probabilistic queries.

Another class of neuro-symbolic approaches have their roots in logic programming. DeepProbLog [Manhaeve et al., 2018] extends ProbLog, a probabilistic logic programming language, with the capacity to process neural predicates, whereby the network's outputs are construed as the

probabilities of the corresponding predicates. This simple idea retains all essential components of ProbLog: the semantics, inference mechanism, and the implementation. In a similar vein, Dai et al. [2018] combine domain knowledge specified as purely logical Prolog rules with the output of neural networks, dealing with the network's uncertainty through revising the hypothesis by iteratively replacing the output of the neural network with anonymous variables until a consistent hypothesis can be formed. Bošnjak et al. [2017] present a framework combining prior procedural knowledge, as a Forth program, with neural functions learned through data. The resulting neural programs are consistent with specified prior knowledge and optimized with respect to data.

2.2 Semantic Strengthening of Neuro-Symbolic Learning

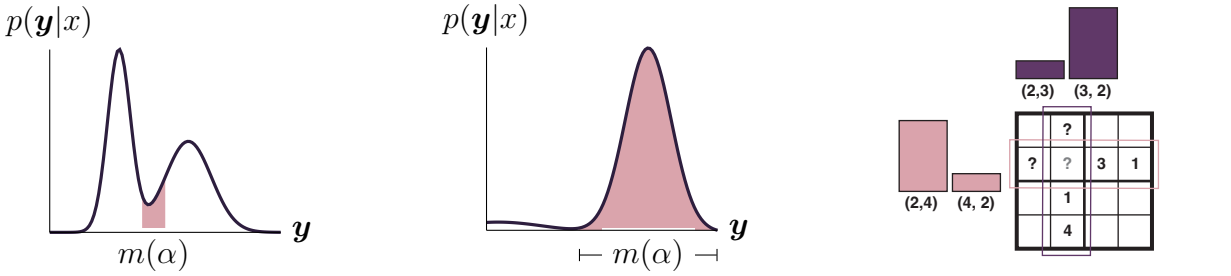
Neural networks are often only able to achieve decent *label-level* accuracy, with a complete disregard to the structure *jointly* encoded by the individual labels. Neuro-symbolic approaches [De Raedt et al., 2020] hope to remedy the problem by injecting into the training process knowledge regarding the underlying problem domain. We have seen that this can be achieved by maximizing the probability allocated by the neural network to outputs satisfying the rules of the underlying domain. Computing this quantity is, a #P-hard problem [Valiant, 1979b], which while tractable for a range of practical problems [Xu et al., 2018a; Ahmed et al., 2022c], precludes many problems of interest.

A common approach is to side step the hardness of computing the probability *exactly* by replacing logical operators with their fuzzy t-norms, and logical implications with simple inequalities [Medina Grespan et al., 2021; van Krieken et al., 2020]. This, however, does not preserve the sound probabilistic semantics of the underlying logical statement: equivalent logic statements no longer correspond to the same set of satisfying assignments, to different probability distributions, and consequently, vastly different constraint probabilities. On the other hand, obtaining a Monte Carlo estimate of the probability [Ahmed et al., 2022a] is infeasible in exponentially-sized output spaces where the valid outputs represent only a sliver of the distribution's support.

In this work, starting from first principles, we derive a probabilistic approach to scaling prob-

abilistic inference for neuro-symbolic learning while retaining the sound semantics of the underlying logic. Namely, we start by assuming that the probability of the constraint decomposes, conditioned on the network's learned features. That is, we assume the events encoded by the logical formula to be *mutually independent* given the learned features, and therefore, joint probability factorizes as a product of probabilities. This generalizes the prolific assumption that the probabilities of the variables are *mutually-independent* conditioned on the network's learned features [Mullenbach et al., 2018; Xu et al., 2018a; Giunchiglia and Lukasiewicz, 2020] to events over arbitrary number of atoms. This reduces the (often intractable) problem of probabilistically satisfying the constraint, the validity of a Sudoku puzzle, to the (tractable) problem of probabilistically satisfying the individual local constraints, e.g. the uniqueness of the elements of a row, column, or square. This, however, introduces inconsistencies: an assignment that satisfies one constraint might violate another, leading to misaligned gradients. More precisely, for each pair of constraints, we are interested in the penalty incurred, in terms of modeling error, by assuming the constraints to be independent when they are in fact dependent, conditioned on the features learned by the neural network. This corresponds exactly to the conditional mutual information, a quantity notoriously hard to calculate. We give an algorithm for tractably computing the conditional mutual information, given that our constraints are represented as circuits satisfying certain structural properties. Training then proceeds, where we interleave the process of learning the neural network, with the process of *semantic strengthening*, where we iteratively tightening our approximation, using the neural network to guide us to which constraints need to be made dependent.

We test our approach on three different tasks: predicting a minimum-cost path in a Warcraft terrain, predicting a minimum-cost perfect matching, as well as solving Sudoku puzzles, where we observe that our approach greatly improves upon the baselines all for a minuscule increase in computation time, thereby sidestepping the intractability of the problem.



(a) Setting where satisfying assignments are only fraction of distribution support.

(b) A network allocating *most* of probability mass to satisfying assignments.

(c) Distributions over empty entries of Sudoku row and col modeled separately.

Figure 2.5: Estimating the probability of a constraint using sampling can fail when, (a) the set of satisfying assignments represents only a minuscule subset of the distribution's support, or, (b) when the network already largely satisfies the constraints, and consequently, we are very unlikely to sample very low-probability assignments violating the constraint. Using product t-norm, (c), to model the probability of satisfying constraints reduces the problem to satisfying the constraints locally, which can often lead to conflicting probabilities, and therefore, conflicting gradients. Here, e.g., according to the distribution over the Sudoku row, 3 is the likely value of the cell in grey, where as, according to the distribution over the Sudoku column, 4 is the likely value.

2.2.1 Problem Statement and Motivation

Recall from section Sec. 2.1.1.1 that a neural network induces a distribution over the output space, and that we can define a semantic loss [Xu et al., 2018a], as in Equation 2.2 to minimize the probability mass allocated to invalid outputs under the neural network's distribution.

Computing the above expectation is generally #P-hard [Valiant, 1979b]: there are potentially exponentially many models of α. For instance, there are 6.67×10^{21} valid 9×9 Sudokus [Felgenhauer and Jarvis, 2005], where as the number of valid matchings or paths in a $n \times n$ grid grows doubly-exponentially in the grid size [Strehl, 2001].

A common approach resorts to *relaxing* the logical statements, replacing logical operators with their fuzzy t-norms, and implications with simple inequalities, and come in different flavors: Product [Rocktäschel et al., 2015; Li and Srikumar, 2019; Asai and Hajishirzi, 2020], Gödel [Minervini et al., 2017], and Łukasiewicz [Bach et al., 2017], which differ only in their interpretation of the logical operators. Medina Grespan et al. [2021] offer a comprehensive theoretical, and empirical,

treatment of the subject matter.

While attractive due to their tractability, t-norms suffer from a few major drawbacks. First, they *lose the precise meaning of the logical statement*, i.e. the satisfying and unsatisfying assignments of the relaxed logical formula differ from those of the original logical formula. Second, the logic is no longer consistent, i.e. logical statements that are otherwise equivalent correspond to different truth values, as the relaxations are a function of their syntax rather than their semantics. Lastly, the relaxation sacrifices sound probabilistic semantics, unlike other approaches [Xu et al., 2018a; Manhaeve et al., 2018] where the output probability corresponds to the probability mass allocated to truth assignments of the logical statement, the output probability has no sound probabilistic interpretation [Medina Grespan et al., 2021].

A slightly more benign relaxation [Rocktäschel et al., 2015] only assumes that, for a constraint $\alpha = \beta_1 \wedge \ldots \wedge \beta_n$, a neural network $f(\cdot)$, and an input $\mathbf{x}$, the events β_i are mutually independent conditioned on the features learned by the neural network. That is, the probability of the constraint factorizes as $\mathrm{P}(\alpha|f(\mathbf{x})) = \mathrm{P}(\beta_1|f(\mathbf{x})) \times \ldots \times \mathrm{P}(\beta_n|f(\mathbf{x}))$. This recovers the true probabilistic semantics of the logical statement when $\beta_1, \ldots, \beta_n$ are over disjoint sets of variables, i.e. $\forall_{i,j}$ $vars(\beta_i) \cap vars(\beta_j) = \emptyset$ for $i \neq j$ and can otherwise be thought of as a *tractable* approximation, the basis of which is the neural network's ability to sufficiently encode the dependencies shared between the constraints, rendering them conditionally independent given the learned features. That is assuming the neural network makes almost-deterministic predictions of the output variables given the embeddings. Even assuming the true function is deterministic, there is still the problem of an imperfect embedding giving probabilistic predictions whereby clauses are dependent.

The above relaxation reduces the *intractable* problem of satisfying the global constraint to the *tractable* problem of satisfying the local constraints, and can therefore often lead to *misaligned gradients*. Consider cell $(1, 1)$ of the Sudoku in Figure 2.5. Consider the two constraints asserting that the elements of row 2 and that the elements of column 2 are unique, and assume the probability distribution induced by the network over row and column assignments are as shown in Figure 2.5, right. This leads to opposing gradients for cell $(1, 1)$: On the one hand, the gradient from maximiz-

ing the probability of the column constraint pushes it to 2, whereas the gradient from maximizing the probability of the row constraint pushes it to 4. The problem here is modeling as independent two constraints that are strongly coupled so much that one determines the value of the other.

Recently, Ahmed et al. [2022a] proposed using sampling to obtain a Monte Carlo estimate of the probability of the constraint being satisfied. This offers the convenience of specifying constraints as PyTorch functions, as well as accommodating non-differentiable elements in the training pipeline of the constraint, especially in cases where the training pipeline includes non-differentiable elements. However, when problems are intractable, this is often accompanied by a state space that is combinatorial in size, meaning that the probability of sampling a valid structure drops precipitously as a function of the size of the state space, making it near impossible to obtain any learning signal, as almost all the sampled states will necessarily violate our constraint. The same applies when the constraint is almost satisfied, meaning we never sample low-probability assignment that violate the constraint.

That is not to mention the pitfalls of sampling: Ahmed et al. [2022a] employ the REINFORCE gradient estimator, which while unbiased in the limit of many samples, exhibits variances that makes it very hard to learn. Even gradient estimators that do not exhibit this problem of variance, trade off variance for bias, making it unlikely to obtain the true gradient.

2.2.2 Semantic Strengthening

We are interested in an approach that, much like the approaches discussed in Sec. 2.2.1 is tractable, but retains sound probabilistic semantics, and yields a non-zero gradient when the constraint is locally, or globally, violated.

Let our constraint α be given by a conjunctive normal form (CNF), $\alpha = \beta_1 \wedge \ldots \wedge \beta_n$. We start by assuming that, for a neural network $f(\cdot)$, and an input $\mathbf{x}$, the clauses β_i are mutually independent conditioned on the features learned by the neural network i.e. the probability of the constraint factorizes as $\mathrm{P}(\alpha|f(\mathbf{x})) = \mathrm{P}(\beta_1|f(\mathbf{x})) \times \ldots \times \mathrm{P}(\beta_n|f(\mathbf{x}))$, where the probability of each of

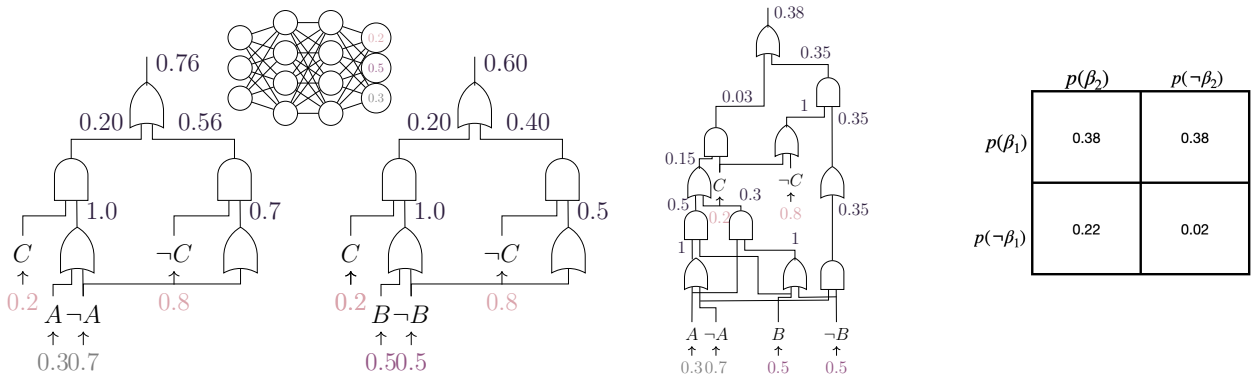


Figure 2.6: (Left) Example of two compatible constraint circuits parameterized by the outputs of a neural network. To compute the probability of a circuit, we plug in the output of the neural network $\boldsymbol{p}_i$ and $1-\boldsymbol{p}_i$ for positive and negative literal i, respectively. The computation proceeds bottom-up, taking products at AND gates and summations at OR gates, and the probability is accumulated at the root of the circuit. (Right) the conjunction of the two constraint circuits, its probability, computing the probabilities required for the mutual information using the law of total probability.

the clauses, $\mathrm{P}(\beta_i)$, can be computed tractably. This recovers the true probabilistic semantics of the logical statement when $\beta_1, \ldots, \beta_n$ are over disjoint sets of variables, i.e. $\forall_{i,j}\ vars(\beta_i) \cap vars(\beta_j) = \emptyset$ for $i \neq j$, and can otherwise be thought of as a *tractable* approximation, the basis of which is the neural network's ability to sufficiently encode the dependencies shared between the constraints, rendering them conditionally independent given the learned features, again, assuming the true function is deterministic, with no inherent uncertainty.

The above approximation is semantically sound in the sense that, the probability of each term $\mathrm{P}(\beta_i)$ accounts for all the truth assignment of the clause β_i. It is also guaranteed to yield a semantic loss value of 0, and therefore a zero gradient if and only if all the clauses, β_i, are satisfied.

However, as discussed in Sec. 2.2.1, training the neural network to satisfy the local constraints can often be problematic: two *dependent* constraints assumed independent can often disagree on the value of their shared variables leading to opposing gradients. If we are afforded more computational resources, we can start strengthening our approximation by relaxing some of the independence assumptions made in our model.

2.2.2.1 Deriving the Criterion

The question then becomes, *which independence assumptions to relax*. We are, of course, interested in relaxing the independence assumptions that have the most positive impact on the quality of the approximation. Or, put differently, we are interested in relaxing the independence assumptions for which we incur the most penalty for assuming, otherwise dependent constraints, to be independent. For each pair of constraints β_i and β_j, for all $i \neq j$, this corresponds to the Kullback-Leibler divergence of the product of their marginals from their joint distribution, and is a measure of the modeling error we incur, in bits, by assuming the independence of the two constraints

$$\mathrm{KL}\big(\mathrm{P}_{(X,Y)} \,\|\, \mathrm{P}_X \cdot \mathrm{P}_Y\big) \tag{2.4}$$

where X and Y are Bernoulli random variables, $X \sim \mathrm{P}(\beta_i)$, $Y \sim \mathrm{P}(\beta_j)$, and $(X, Y) \sim \mathrm{P}(\beta_i, \beta_j)$, for all i, j such that $i \neq j$. Equation 2.4 equivalently corresponds to the *mutual information* $I(X;Y)$ given by

$$I(X;Y) = \mathbf{E}_{(X,Y)}\left[\log \frac{\mathrm{P}_{(X,Y)}(X,Y)}{\mathrm{P}_X(X) \cdot \mathrm{P}_Y(Y)}\right], \tag{2.5}$$

between the random variables X and Y, or the measure of *dependence* between them. Intuitively, mutual information captures the information shared between X and Y: it measures how much knowing one reduces about the uncertainty of the other. When they are independent, then knowing one does not give any information about the other, and therefore the mutual information is 0. At the other extreme, one is a deterministic function of the other, and therefore, the mutual information is maximized and equals to their entropy. Note that the expectations in both Equation 2.4 and Equation 2.5 are over the joint distribution $P_{(X,Y)}$.

We would be remiss, however, to dismiss the features learned by the network, as they already encode some of the dependencies between the constraints, affording us the ability to make stronger approximations. That is, we are interested in the mutual information between all pairs of constraints β_i, β_j *conditioned* on the neural network's features. Let $\mathcal{D}$ be our data distribution, and Z

Algorithm 2 $\texttt{MI}(\beta_1; \beta_2 \mid f(\mathbf{x}))$

Input: Two compatible constraint circuits β_1 and β_2
Output: Mutual Information of β_1 and β_2 given features

// Conjoin β_1 and β_2
$\alpha = \beta_1 \wedge \beta_2$
// Compute the probability of α, β_1 and β_2 c.f. Figure 2.6
$p_\alpha, p_{\beta_1}, p_{\beta_2}$ = prob(α), prob(β_1), prob(β_2)
// Calculate marginals and joint using total probability
$p_{\mathbf{X}} = [1 - p_{\beta_1}, p_{\beta_1}]$, $p_{\mathbf{Y}} = [1 - p_{\beta_2}, p_{\beta_2}]$
$p_{(\mathbf{X},\mathbf{Y})} = [[1 - p_{\beta_1} - p_{\beta_2} - p_\alpha, p_{\beta_2} - p_\alpha], [p_{\beta_1} - p_\alpha, p_\alpha]]$
mi $= 0$
for x, y **in** product($[0, 1]$) **do**
    mi $+= p_{(\mathbf{X},\mathbf{Y})}[x][y] \times \log(\frac{p_{(\mathbf{X},\mathbf{Y})}[x][y]}{p_{\mathbf{X}}[x] \times p_{\mathbf{Y}}[y]})$
return mi

Algorithm 3 $\texttt{SemanticStrengthening}(\text{constraints}, \kappa)$

Input: Current set of constraint circuits
Output: *Strengthened* set of constraints

pwmi $= [\,]$
for β_1, β_2 **in** product(constraints) **do**
    if disjoint($vars(\beta_i), vars(\beta_j)$) **then continue**
// Keep track of constraints with mutual information
    pwmi.append(($\texttt{MI}(\beta_i, \beta_j), \beta_i, \beta_j$))
// Consider only the top κ pairs of constraints
to_merge $=$ sorted(pwmi, reverse=True)[: κ]
for mi, β_1, β_2 **in** (to_merge) **do**
    constraints.remove(β_i, β_j)
    constraints.append($\beta_i \wedge \beta_j$)
return constraints

be a random variable distributed according to D, we are interested in computing

$$I(X; Y \mid Z) = \mathbf{E}_Z \left[\mathbf{E}_{(X,Y)|Z} \left[\log \frac{\mathrm{P}(x, y \mid \mathbf{z})}{\mathrm{P}(x \mid \boldsymbol{z}) \cdot \mathrm{P}(y \mid \mathbf{z})} \right] \right] \tag{2.6}$$

$$= \mathbf{E}_Z \left[\sum_{x=0}^{1} \sum_{y=0}^{1} \mathrm{P}(x, y \mid \mathbf{z}) \left[\log \frac{\mathrm{P}(x, y \mid \mathbf{z})}{\mathrm{P}(x \mid \boldsymbol{z}) \cdot \mathrm{P}(y \mid \mathbf{z})} \right] \right], \tag{2.7}$$

where, as is common place, we estimate the outer expectation using Monte Carlo sampling.

Perhaps rather surprisingly, not withstanding the expectation w.r.t the data distribution, the quantity in Equation 2.6 is hard to compute. This is not only due to the intractability of the probability, which as we have already stated is #P-hard in general, but also due to the hardness of conjunction, in general. Loosely speaking, one could have constraints β_i and β_j for which the probability computation, $\mathrm{P}(\beta_i)$ and $\mathrm{P}(\beta_j)$ is tractable, yet computing $\mathrm{P}(\alpha)$, where once again $\alpha = \beta_i \wedge \beta_j$, is hard [Shen et al., 2016; Khosravi et al., 2019a]. Intuitively, the hardness of conjunction comes from finding the intersection of the satisfying assignments without enumeration. We formalize this in Sec. 2.2.2.3.

2.2.2.2 The Semantic Strengthening Algorithm

For the purposes of this section, we will assume we can tractably compute the conditional mutual information in Equation 2.6, and proceed with giving our Semantic Strengthening algorithm. The idea is, simply put, to use the neural network to guide the process of relaxing the independence assumptions introduced between the constraints. Specifically, we are given an interval, η, a constraint budget, κ, and a computational budget τ. We initiate the process of training the neural network, interrupting training every η epochs, computing the conditional mutual information between pairs of constraints, considering only those pairs sharing at least one variable (e.g. the two constraints asserting the uniqueness of the first and last row, respectively, do not share variables, are therefore independent, and by definition have a mutual information of 0, so we need not consider joining them, *yet*). Subsequently, we identify the κ pairs of constraints with the highest pairwise conditional mutual information, and that therefore, have the most detrimental effect on the quality of our approximation. We detect the strongly connected components of constraints, and conjoin them: if β_1 and β_2 should be made dependent, and β_2 and β_3 should be made dependent, then β_1 , β_2 and β_3 are made dependent. We delete the old constraints from, and add the new constraints, to our set of constraints, and resume training. This process is repeated every η epochs until we have exhausted our computational budget τ. Our full algorithm is shown in Algorithm 3.

2.2.2.3 Tractably Computing the Criterion

Recall the definition of tractable circuits and their structural properties given in Sec. 2.1.2.1. Determinism, taken together with smoothness and decomposability, allows us to tractably compute the probability of a constraint [Darwiche and Marquis, 2002].

What remains, is to show that we can tractably conjoin two constraints. Conjoining two decomposable and deterministic circuits is NP-hard if we wish the result to also be decomposable and deterministic, which as we mentioned is a requirement for tractable probability computation [Darwiche and Marquis, 2002; Shen et al., 2016; Khosravi et al., 2019a]. To guarantee the tractabil-

Figure 2.7: An example of a Warcraft terrain map (left) and an MNIST grid, and the corresponding groundtruth labels.

ity of the probability computation of the conjoined constraint, we will, therefore, need to introduce one last structural property, namely the notion of *compatibility* between two circuits [Vergari et al., 2021]. Two circuits, c_1 and c_2 over variables $\mathbf{Y}$ are said to be compatible if (1) they are smooth and decomposable, and (2) any pair of AND nodes, $n \in c_1$ and $m \in c_2$ with the same scope over $\mathbf{Y}$ can be rearranged to be mutually compatible and decompose in the same way i.e. $vars(n) = vars(m) \implies vars(n_i) = vars(m_i)$, and n_i and m_i are compatible, for some arrangement of the inputs n_i and m_i of n and m. A sufficient condition for compatibility is that both c_1 and c_2 share the exact same hierarchical scope partitioning [Vergari et al., 2021], sometimes called a vtree or variable ordering [Choi et al., 2020a; Pipatsrisawat and Darwiche, 2008]. Intuitively, the two circuits should share the order in which they factorize the function over its variables. Figure 2.6 shows an example of smooth, decomposable, deterministic and compatible circuits.

At a high level, there exist off-the-shelf compilers utilizing SAT solvers, essentially through case analysis, to compile a logical formula into a tractable logical circuit. We are agnostic to the exact flavor of circuit so long as the properties outlined herein are respected. In our experiments, we use PySDD[1] – a Python SDD compiler [Darwiche, 2011a; Choi and Darwiche, 2013].

Now that we have shown that we can tractably compute the probabilities $\mathrm{P}(\beta_1), \mathrm{P}(\beta_2)$ and $\mathrm{P}(\alpha)$, we can utilize the law of total probability (c.f. Figure 2.6) to compute the remaining probabilities, and therefore, the mutual information. Our algorithm is shown in Algorithm 2.

[1] https://github.com/wannesm/PySDD

2.2.3 Experimental Evaluation

We evaluated our approach, semantic strengthening, on several neuro-symbolic tasks, namely Warcraft minimum-cost path finding, minimum-cost perfect matching of MNIST digits, as well as the task of training neural networks to solve Sudoku puzzles. The challenge with all of the above tasks, when looked at through a neuro-symbolic lens, is the vastness of the state space: as previously mentioned, there are 6.6×10^{21} valid 9×9 Sudokus, and the number of valid matchings, or paths in a grid grows doubly-exponentially in the grid size—simply too much to enumerate. Even approaches like semantic loss which rely on circuit approaches to exploit the local structure in the problem, essentially through caching solutions to repeated subproblems, do not scale to large instances of these tasks.

As has been established in previous work [Xu et al., 2018a; Ahmed et al., 2022c,b], label-level accuracy, or the accuracy of predicting individual labels is very often a poor indication of the performance of the neural network, and is often uninteresting in neuro-symbolic settings, where we are rather more interested in the accuracy of our predicted structure object *exactly* matching the ground truth, e.g., *is the prediction a shortest path?*, a metric which we denote "Exact" in our experiments, as well as the accuracy of predicting objects that are *consistent* with the constraint, e.g., *is the prediction a valid path?*, a metric which we denote "Consistent" in our experiments. Note that, unlike the other two tasks, for the case of Sudoku, these measures are one and the same: a valid Sudoku has a single *unique* solution.

In all of our experiments, we compare against two baselines: a neural network, whose architecture we specify in the corresponding experimental section, and the same neural network augmented with product t-norm, where we assume the independence of constraints throughout training.

Warcraft Shortest Path We evaluate our approach, semantic strengthening, on the challenging task of predicting the minimum-cost path in a weighted grid imposed over Warcraft terrain maps. Following Pogančić et al. [2019], our training set consists of $10,000$ terrain maps curated using the Warcraft II tileset. Each map encodes an underlying grid of dimension 12×12, where each

vertex is assigned a cost depending on the type of terrain it represents (e.g. earth has lower cost than water). The shortest (minimum cost) path between the top left and bottom right vertices is encoded as an indicator matrix, and serves as label. Figure 2.7 shows an example input presented to the network and the input with an annotated shortest path as a groundtruth. Presented with an image of a terrain map, a convolutional neural network—similar to Pogančić et al. [2019], we use ResNet18 [He et al., 2016a]—outputs a 12×12 binary matrix indicating a set of vertices. Note that the minimum-cost path is not unique: there may exist several paths sharing the same minimum cost, all of which are considered to be correct by our metrics. Table 2.5 shows our results.

Table 2.5: Warcraft shortest path prediction results

Test accuracy %	Exact	Consistent
ResNet-18	44.80	56.90
+ Product t-norm	50.40	63.20
+ Semantic Strengthening	$\mathbf{61.20}$	$\mathbf{72.70}$

We observe that incorporating constraints into learning improves the accuracy of predicting the optimal path from 44.80% to 50.40%, and the accuracy of predicting a *valid* path from 56.90% to 63.20%, as denoted by the "Exact" and "Consistent" metrics, respectively. Furthermore, and perhaps more interestingly, we see that our approach, *semantic strengthening*, greatly improves upon the baseline, as well as product t-norm improving the accuracy of predicting the optimal path from 44.80% and 50.40% to 61.20%, while greatly improving the accuracy of predicting a valid path from 56.90% and 63.20% to 72.70%.

MNIST Perfect Matching Our next task consists in predicting a minimum-cost perfect-matching of a set of k^2 MNIST digits arranged in a $k \times k$ grid, where diagonal matchings are not permitted. We consider the problem for the instance when $k = 10$. Similar to Pogančić et al. [2019], we generate the ground truth by considering the underlying $k \times k$ grid graph, and solving a min-cost perfect-matching problem using Blossom V [Kolmogorov, 2009], where the edge weights are given simply by reading the two vertex digits as a two-digit number, reading downwards for verti-

cal edges, and left to right for horizontal edges. The minimum-cost perfect matching label is then encoded as an indicator vector for the subset of the selected edges. Similar to the Warcraft experiment, the grid image is input to a (pretrained) ResNet-18, which simply outputs a set of predicted edges. Table 2.6 shows our results.

Table 2.6: Perfect Matching prediction test results

Test accuracy %	Exact	Consistent
ResNet-18	9.30	10.00
+ Product t-norm	12.70	12.90
+ Semantic Strengthening	**15.50**	**18.40**

Similar to the Warcraft experiment, we observe that incorporating constraints into learning improves the accuracy of predicting the optimal perfect matching from 9.30% to 12.70%, and the accuracy of predicting a *valid* perfect matching from 10.00% to 12.90%, as denoted by the "Exact" and "Consistent" metrics, respectively. Furthermore, we see that our approach, *semantic strengthening*, greatly improves upon the baseline, as well as product t-norm improving the accuracy of predicting the optimal perfect matching from 09.30% and 12.70% to 15.50%, while greatly improving the accuracy of predicting a valid perfect matching from 10.00% and 12.90% to 18.40%.

Sudoku Lastly, we consider the task of predicting a solution to a given Sudoku puzzle. Here the task is, given a 9×9 partially-filled grid of numbers to fill in the remaining cells in the grid such that the entries each row, column, and 3×3 square are unique i.e. each of the numbers from 1 through 9 appears exactly once.

We use the dataset provided by Wang et al. [2019], consisting of 10K Sudoku puzzles, split into 9K training examples, and 1K test samples, all puzzles having 10 missing entries.

As our baseline, we follow Wang et al. [2019] in using a convolutional neural network modeled on that of Park [2018]. The input to the neural network is given as a bit representation of the initial Sudoku board, along with a mask representing the bits to be learned, i.e. the bits in the empty

Sudoku cells. The network interprets the bit inputs as 9 input image channels (one for each square in the board) and uses a sequence of 10 convolutional layers (each with 512 3×3 filters) to output the solution, with the mask input as a set of additional image channels in the same format as the board. Table 2.7 shows our results.

Table 2.7: Sudoku test results

Test accuracy %	Exact	Consistent
10-Layer ConvNet	16.80	16.80
+ Product t-norm	22.10	22.10
+ Semantic Strengthening	**28.00**	**28.00**

In line with our previous experiments, we observe that incorporating constraints into learning improves the accuracy of predicting correct Sudoku solutions, the "Exact" metric from 16.80% to 22.10%. Furthermore, we see that our approach, *semantic strengthening*, greatly improves upon the baseline, as well as product t-norm, improving the accuracy from 16.80% and 22.10% to 28.00%.

2.2.4 Related Work

There has been increasing interest in combining neural learning with symbolic reasoning, a class of methods that has been termed *neuro-symbolic* methods, studying how to best combine both paradigms in a bid to accentuate their positives and mitigate their negatives. The focus of many such approaches has therefore been on making probabilistic reasoning tractable through first-order approximations, and differentiable, through reducing logical formulas into arithmetic objectives, replacing logical operators with their fuzzy t-norms, and implications with inequalities [Kimmig et al., 2012; Rocktäschel et al., 2015; Fischer et al., 2019; Pryor et al., 2022].

Diligenti et al. [2017a] and Donadello et al. [2017] use first-order logic to specify constraints on outputs of a neural network. They employ fuzzy logic to reduce logical formulas into differential, arithmetic objectives denoting the extent to which neural network outputs violate the constraints, thereby supporting end-to-end learning under constraints. More recently, Xu et al. [2018a] intro-

duced semantic loss, which circumvents the shortcomings of fuzzy approaches, while supporting end-to-end learning under constraints. More precisely, *fuzzy reasoning* is replaced with *exact probabilistic reasoning*, by compiling logical formulae into structures supporting efficient probabilistic queries. Liu et al. [2023a] use semantic loss to simultaneously learn a neural network and extract generalized logic rules. Different from other neural-symbolic methods that require background knowledge and candidate logical rules, they aim to induce task semantics with minimal priors.

Another class of neuro-symbolic approaches have their roots in logic programming. DeepProbLog [Manhaeve et al., 2018] extends ProbLog, a probabilistic logic programming language, with the capacity to process neural predicates, whereby the network's outputs are construed as the probabilities of the corresponding predicates. This simple idea retains all essential components of ProbLog: the semantics, inference mechanism, and the implementation. Manhaeve et al. [2021] attempts to scale DeepProbLog by considering only the top-k proof paths. In a similar vein, Dai et al. [2018] combine domain knowledge specified as purely logical Prolog rules with the output of neural networks, dealing with the network's uncertainty through revising the hypothesis by iteratively replacing the output of the neural network with anonymous variables until a consistent hypothesis can be formed. Bošnjak et al. [2017] present a framework combining prior procedural knowledge, as a Forth program, with neural functions learned through data. The resulting neural programs are consistent with specified prior knowledge and optimized with respect to data.

2.3 A Pseudo-Semantic Loss for Autoregressive Models with Logical Constraints

All of the methods developed so far, as well as those existing in the literature assume the outputs of the neural network to be conditionally independent given the learned features, and therefore the distribution over the solutions of the constraint is assumed to be fully-factorized.

In this work we move beyond fully-factorized output distributions and towards autoregressive ones, including those induced by large language models such as GPT [Radford et al., 2019], where the output at any given time step depends on the outputs at all previous time steps. Computing the probability of an arbitrary constraint under fully-factorized output distributions is #P-hard. Intuitively, the hardness of the problem can be attributed to the possibly exponentially-many solutions of the constraint. Under an autoregressive distribution, however, computing the probability of even a single literal as a constraint is #P-hard Roth [1996]. That is, under autoregressive distributions, the hardness of computing the probability of an arbitrary constraint is now due to two distinct factors: the hardness of the logical constraint as well as the hardness of the distribution. Throughout this paper, we will assume the inherent hardness of the constraint can be sidestepped: for many applications, we can come up with compact representations of the constraint's solutions that are amenable to computing its probability under the fully-factorized distribution efficiently. When such compact representations are unavailable, we can fall back to approximate representations of the constraint [Ahmed et al., 2023a].

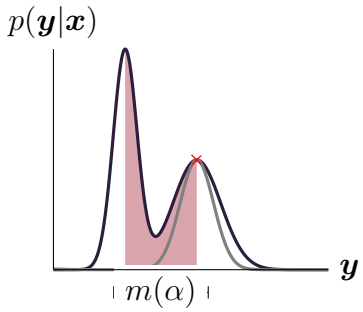


Figure 2.8: **Our approach in a nutshell**. Given a data point x, we approximate the likelihood of the constraint α (area shaded in pink) with the pseudolikelihood (shown in gray) of the constraint in the neighborhood of a sample (denoted $\times$), where $m(\alpha)$ denotes the region of the constraint support.

Unlike previous works that are only able to approximately handle simple constraints under relaxations of autoregressive distributions [Ganchev et al., 2010; Zhang et al., 2017; Hu et al., 2018; Yu et al., 2022], our approach injects non-trivial constraints, that don't easily factorize, as part of the training process, computing the probability of the constraint exactly w.r.t. an approximate distribution. Concretely, we approximate the likelihood of the constraint w.r.t. the autoregressive distribution with its probability in a local pseudolikelihood distribution—a product of conditionals—centered around a model sample. This leads to a factorizable objective which allows us to efficiently compute the probability of constraints by reusing solutions to common sub-problems.

Experiments show our approximation is low-entropy, allocating most of its mass around the sample, and has low KL- divergence from the true distribution. Intuitively, we want to stay close to the sample to ensure high fidelity, while retaining a distribution to ensure differentiability and maximum generality within tractability bounds. An overview of our approach is depicted in Figure 2.8.

Empirically, we start by evaluating our approach on the tasks of solving a Sudoku puzzle and generating a shortest path in a given Warcraft map where, conditioned on the input puzzle (map, resp.), the neural network autoregressively generates a Sudoku solution (shortest path, resp.), taking into account generations at previous time steps. We observe that our autoregressive models improve upon the non-autoregressive baselines, and that our approach leads to models whose predictions are even more accurate, and even more likely to satisfy the constraint. Lastly, we evaluated our approach on the challenging task of detoxifying pretrained large language models where the aim is to move the model's distribution away from toxic generations and towards nontoxic ones without sacrificing the model's overall language modeling abilities. We show that, perhaps surprisingly, using only a simple constraint disallowing a list of toxic words, the model exhibits a great reduction in the toxicity of the generated sentences, as measured using the perspective API[2], at almost no cost in terms of the model's language modeling capabilities, measured in perplexity.of logical constraints in such task.

2.3.1 An autoregressive Probability Distribution over Possible Structures

Let α be a logical sentence defined over Boolean variables $\mathbf{Y} = \{Y_{11}, \ldots, Y_{nk}\}$, where n denotes the number of time steps in the sequence, and k denotes the number of possible classes.

The neural network's outputs induce a probability distribution $p(\cdot)$ over possible states $\boldsymbol{y}$. However, the neural network will ensure that, for each time step i, there is exactly one class being predicted in each possible state. That is, exactly one Boolean variable $\{Y_{i1}, \ldots, Y_{ik}\}$ can be set to true for each time step i. We will use $\boldsymbol{y}_i$ to denote that variable Y_{ij} is set to true in state $\boldsymbol{y}$. More

[2] `https://www.perspectiveapi.com/`

precisely, we let $\boldsymbol{y}_i \in \{0,1\}^k$ be the one-hot encoding of Y_{ij} being set to 1 among $\{Y_{i1}, \ldots, Y_{ik}\}$. By the chain rule, the probability assigned by the autoregressive neural network to a state $\boldsymbol{y}$ is then

$$p(\boldsymbol{y}) = \prod_{i=1}^{n} p(\boldsymbol{y}_i \mid \boldsymbol{y}_{<i}), \tag{2.8}$$

where $\boldsymbol{y}_{<i}$ denotes the prefix $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{i-1}$. The most common approaches [Mullenbach et al., 2018; Xu et al., 2018a; Giunchiglia and Lukasiewicz, 2020] to neuro-symbolic learning assume the conditional independence of the network outputs given the learned embeddings. More precisely, let f be a neural network that maps inputs $\boldsymbol{x}$ to M-dimensional embeddings $\boldsymbol{z} = f(\boldsymbol{x})$. Under such assumption, we obtain the *fully-factorized* distribution

$$p(\boldsymbol{y} \mid \boldsymbol{z}) = \prod_{i=1}^{n} p(\boldsymbol{y}_i \mid \boldsymbol{z}). \tag{2.9}$$

We no longer have a notion of ordering under the fully-factorized distribution—and each possible $p(\boldsymbol{y}_i \mid \boldsymbol{z})$ is computed as $\sigma(\mathbf{w}_i^\top \boldsymbol{z})$ where $\mathbf{w}_i \in \mathbb{R}^M$ is a vector of parameters and $\sigma(x)$ is the softmax function. The appeal of such distribution is that it enables the tractability of many reasoning tasks, but the downside is that it dismisses any correlation between the output labels. As we will show in our experimental section (cf. Sec. 2.3.4), using autoregressive distributions, even simple ones such as LSTMs, already outperforms a neural network where the labels are assumed to be independent.

2.3.2 The Pseudo-Semantic loss

Recall that in neuro-symbolic learning, we often assume access to symbolic knowledge connecting the different outputs of a neural network and are concerned with maximizing the likelihood of the constraint w.r.t. the network's parameters $\boldsymbol{\theta}$:

$$\arg\max_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\alpha) = \arg\max_{\boldsymbol{\theta}} \mathbf{E}_{\boldsymbol{y} \sim p_{\boldsymbol{\theta}}} \left[\mathbb{1}\{\boldsymbol{y} \models \alpha\}\right] = \arg\max_{\boldsymbol{\theta}} \sum_{\boldsymbol{y} \models \alpha} p_{\boldsymbol{\theta}}(\boldsymbol{y}), \tag{2.10}$$

where we can use tractable circuit to compute the above expectation exactly when the output of the neural networks is fully-factorized. Unfortunately, as previously mentioned, moving beyond the fully-factorized distribution, we are faced with another source of intractability: the hardness of the distribution w.r.t. which the expectation in Equation 2.10 is being computed. Assuming a deep generative model whose distribution p can capture a Bayesian network distribution, the problem of computing even a single marginal—i.e., the marginal probability of a single variable—is known to be #P-hard [Roth, 1996]. This class of models includes the autoregressive distribution. Intuitively, a constraint might have exponentially-many solutions, yet lend itself nicely to reusing of solutions to sub-problems, and therefore a tractable calculation of the expectation in Equation 2.10. An example being the n choose k constraint Ahmed et al. [2023c], where the expectation in Equation 2.10 can be computed in quadratic time under the fully-factorized distribution, despite having a normally-prohibitive number of solutions. Moving away from the fully-factorized distribution, however, entails that in the worst case, we would need to compute a sub-problem combinatorial number of times—for all possible sequences—for exponentially many solutions of the constraint.

To sidestep the intractability of the expectation in Equation 2.10, as a first step, we consider the *pseudolikelihood* $\tilde{p}(\cdot)$ of a set of parameters given an assignment [Besag, 1975], as a surrogate for its likelihood i.e.,

$$p(\boldsymbol{y}) \approx \tilde{p}(\boldsymbol{y}) \coloneqq \prod_i p(\boldsymbol{y}_i \mid \boldsymbol{y}_{-i}), \tag{2.11}$$

where $\boldsymbol{y}_{-i}$ denotes $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{i-1}, \boldsymbol{y}_{i+1}, \ldots, \boldsymbol{y}_n$. Consequently, we can consider *the pseudolikelihood* of a set of parameters given a logical constraint α as a surrogate for its true likelihood i.e.,

$$p(\alpha) \approx \tilde{p}(\alpha) = \mathbf{E}_{\boldsymbol{y} \sim \tilde{p}} \left[\mathbb{1}\{\boldsymbol{y} \models \alpha\} \right] = \sum_{\boldsymbol{y} \models \alpha} \tilde{p}(\boldsymbol{y}). \tag{2.12}$$

Intuitively, the pseudolikelihood objective aims to measure our ability to predict the value of each variable given a full observation of all other variables. The pseudolikelihood objective attempts to match the model's conditional distributions to the conditional distributions computed from thedata. If it succeeds in matching them exactly, then a Gibbs sampler run on the model's conditional

distributions attains the same invariant distribution as a Gibbs sampler run on the true distribution.

On its own, the above would still not be sufficient to ensure the tractability of the expectation in Equation 2.10. Intuitively, different solutions depend on different sets of conditionals, meaning we would have to compute the probabilities of many of the solutions of the constraint from scratch. Instead, *we compute the pseudolikelihood of the constraint in the neighborhood of a model sample*[3]

$$\tilde{p}(\alpha) = \mathbf{E}_{\boldsymbol{y}\sim\tilde{p}}\left[\mathbb{1}\{\boldsymbol{y} \models \alpha\}\right] \approx \mathbf{E}_{\boldsymbol{y}\sim p}\mathbf{E}_{\tilde{\boldsymbol{y}}\sim\tilde{p}_{\boldsymbol{y}}}\left[\mathbb{1}\{\tilde{\boldsymbol{y}} \models \alpha\}\right] = \mathbf{E}_{\boldsymbol{y}\sim p}\tilde{p}_{\boldsymbol{y}}(\alpha) = \mathbf{E}_{\boldsymbol{y}\sim p}\sum_{\tilde{\boldsymbol{y}}\models\alpha}\tilde{p}_{\boldsymbol{y}}(\tilde{\boldsymbol{y}}), \quad (2.13)$$

$$\text{where } \tilde{p}_{\boldsymbol{y}}(\tilde{\boldsymbol{y}}) \coloneqq \prod_i p(\tilde{\boldsymbol{y}}_i \mid \boldsymbol{y}_{-i}) \quad (2.14)$$

which is the pseudolikelihood $\tilde{p}(\cdot)$ of an assignment in the neighborhood of a sample $\boldsymbol{y}$. Crucially this distribution is fully-factorized, making it amenable to neuro-symbolic loss functions.

Definition 1 (Pseudo-Semantic Loss)**.** *Let α be a sentence in Boolean logic, and let $\tilde{p}_{\boldsymbol{y}}(\cdot)$ be the pseudolikelihood function parameterized by $\boldsymbol{\theta}$ and centered around state $\boldsymbol{y}$, as defined in Equation* 2.14*. Then, we define the pseudo-semantic loss between α and $\boldsymbol{\theta}$ to be*

$$\mathcal{L}^{\mathsf{SL}}_{\mathsf{pseudo}}(\alpha, p_{\boldsymbol{\theta}}) \coloneqq -\log \mathbf{E}_{\boldsymbol{y}\sim p}\tilde{p}_{\boldsymbol{y}}(\alpha) = -\log \mathbf{E}_{\boldsymbol{y}\sim p}\sum_{\tilde{\boldsymbol{y}}\models\alpha}\tilde{p}_{\boldsymbol{y}}(\tilde{\boldsymbol{y}}). \quad (2.15)$$

Intuitively, our pseudo-semantic loss between α and $p_{\boldsymbol{\theta}}$ can be thought of as penalizing the neural network for all the local perturbations $\tilde{\boldsymbol{y}}$ of the model sample $\boldsymbol{y}$ that violate the constraint.

2.3.3 The Algorithm

We will now give a walk through of computing our pseudo-semantic loss. We note that our algorithm is implemented in log-space to preserve numerical stability and uses PyTorch Paszke et al. [2019]. Our full algorithm is shown in Algorithm 4. We sample an assignment $\boldsymbol{y} \sim p_{\boldsymbol{\theta}}$ from the

[3]We sample y_1 conditioned on the beginning-of-sentence token, then y_2 conditioned on the sampled y_1, followed by y_3 conditioned on both y_1 and y_2 and so on until we the end-of-sentence token is sampled.

model (line 4). We now need to compute the pseudolikelihood of the sample

$$\log \tilde{p}_\theta(\boldsymbol{y}) = \sum_i \log p(\boldsymbol{y}_i \mid \boldsymbol{y}_{-i})$$
$$= \sum_i \log p(\boldsymbol{y}_i, \boldsymbol{y}_{-i}) - \underset{\boldsymbol{y}'_i}{\text{LSE}} \log p(\boldsymbol{y}'_i, \boldsymbol{y}_{-i}),$$

where LSE is the logsumexp function. That is, for every element in the sequence, we need to marginalize over all categories $\boldsymbol{y}'_i$. This entails, for every element in the sampled sequence, we need to substitute each of the categories (lines 9-10) and compute the probability of the sample under the model (line 12), obtaining sequence length $\times$ number of categories sequences. Now we can compute the log-conditional probabilities $\log p(\boldsymbol{y}_i \mid \boldsymbol{y}_{-i})$. We marginalize over the categories $\boldsymbol{y}'_i$ to obtain the log-marginal $\log p(\boldsymbol{y}_{-i})$ = LSE $\boldsymbol{y}'_i(\log p(\boldsymbol{y}'_i, \boldsymbol{y}_{-i}))$. We then condition the probability of every sequence by subtracting the log-marginals i.e., $\log p(\boldsymbol{y}_i, \boldsymbol{y}_{-i}) - \log p(\boldsymbol{y}_{-i})$ (line 15). We use these conditionals to compute the pseudolikelihood assigned by the neural network to local perturbations of the model sample y that satisfy the constraint (line 18). As per Sec. 2.2.2.3, we can compute the pseudolikelihood of a constraint α locally around the sample y by pushing the computed conditionals at the respective input nodes of c_α, propagating them through the circuit, taking sums and products. Figure 2.9 shows a toy example run of our algorithm in non-log space.

Algorithm 4 $\mathcal{L}^{\mathsf{SL}}_{\mathsf{pseudo}}(\alpha; p_{\boldsymbol{\theta}})$

1: **Input**: Logical constraint α and model $p_{\boldsymbol{\theta}}$.
2: **Output**: Pseudo-semantic loss of α w.r.t. $\boldsymbol{\theta}$
3: // Obtain sample y from $p_{\boldsymbol{\theta}}$
4: $\boldsymbol{y} \sim p_{\boldsymbol{\theta}}$
5: // Get sequence length and num. of categories
6: seq, cats = $\boldsymbol{y}$.shape()
7: // Expand the batch to contain all perturbations
8: // of $\boldsymbol{y}$ that are a Hamming distance of 1 away
9: $\boldsymbol{y} = \boldsymbol{y}$.expand(seq, cats)
10: $\boldsymbol{y}$[:, range(seq), :, range(seq)] = range(cats)
11: // Evaluate expanded samples through model
12: $\log p_{\boldsymbol{\theta}} = p_{\boldsymbol{\theta}}(\boldsymbol{y})$.log_softmax(dim= -1)
13: // Compute the conditional probabilities:
14: // $\log \tilde{p}_{\boldsymbol{\theta}}[i][j] = \log p_{\boldsymbol{\theta}}(\boldsymbol{y}_j | \boldsymbol{y}_{-j})$
15: $\log \tilde{p}_{\boldsymbol{\theta}} = \log p_{\boldsymbol{\theta}} - \log p_{\boldsymbol{\theta}}$.logsumexp(dim=$-1$)
16: // Compute the probability of α under $\tilde{p}_{\boldsymbol{y}}$
17: // by propagating the conditionals through c_α
18: **return** $-\log \tilde{p}_{\boldsymbol{y}}(\alpha)$

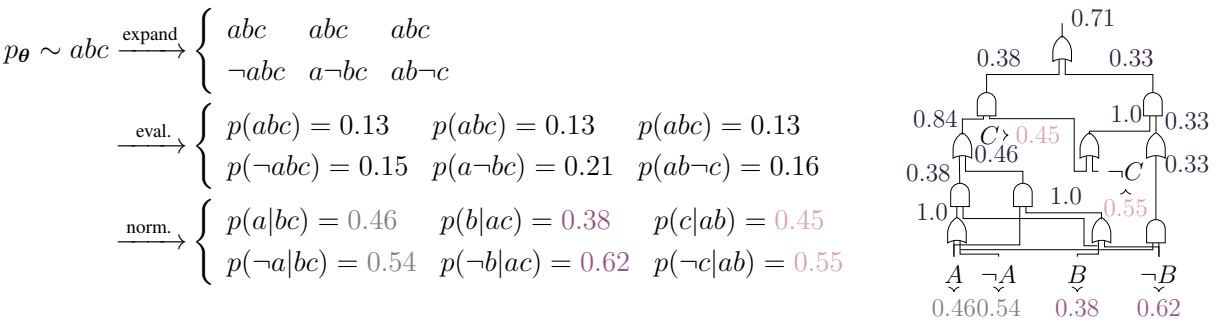


Figure 2.9: **An example of our pipeline.** (Left) We start by sampling an assignment from the model $p_{\boldsymbol{\theta}}$. Our goal is to compute the pseudolikelihood of the model sample—the product of the sample's conditionals. We start by expanding the model sample to include all samples that are a Hamming distance of 1 away from the sample. We proceed by (batch) evaluating the samples through the model, obtaining the joint probability of each sample. We then normalize along each column, obtaining the conditionals. (Right) A logical circuit encoding constraint $(\text{Cat} \implies \text{Animal}) \wedge (\text{Dog} \implies \text{Animal})$, with variable A mapping to Cat, variable B mapping to dog and variable C mapping to Animal. To compute the pseudolikelihood of the constraint in the neighborhood of the sample abc, we feed the computed conditional at the corresponding literals. We push the probabilities upwards, taking products at AND nodes and sums at OR nodes. The number accumulated at the root of the circuit is the pseudolikelihood of the constraint in the neighborhood of the sample abc.

2.3.4 Experimental Evaluation

We evaluate our pseudo-semantic loss on several tasks, spanning a number of domains. We start by evaluating on Warcraft shortest-path finding, where we are given an image of a Warcraft tilemap, and are tasked with *autoregressively* generating one of the potentially many minimum-cost paths between two end points conditioned on the map, where the cost is determined by the *underlying* cost of the tiles spanned by the path. We move on to evaluating on the classic, yet challenging, task of solving a 9×9 Sudoku puzzle where, once again, the generation proceeds autoregressively, conditioned on the input Sudoku puzzle. It is worth noting that such tasks have been considered as a test bed for other neuro-symbolic approaches before, but never for autoregressive generation.

We also evaluate on the task of large language models (LLMs) detoxification. In this task, we are interested in the generations produced by an LLM when presented by a prompt input by the user. More specifically, we are interested not only in how good these models are at the modeling aspect, but also how *toxic* their outputs might be, a measure which includes sexual explicitness, identity attacks, and profanity, among others. Our goal in this task is then to shift the model's distribution

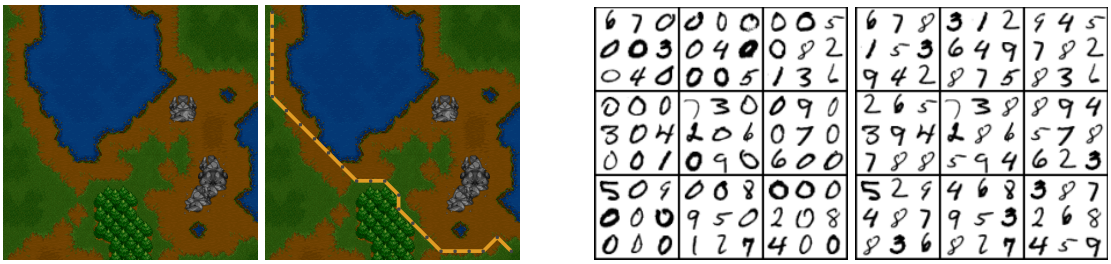


Figure 2.10: **Example inputs and groundtruth labels for two of the three tasks considered in our experimental evaluation.** (Left) Example Warcraft terrain map and a possible (non-unique) minimum-cost shortest path. (Right) Example Sudoku puzzle and its corresponding solution.

away from toxic generations, and toward nontoxic ones, all while maintaining its original ability to model text. We believe this to be a timely and important problem due to their recent prevalence and widespread usage coupled with the fact that previous work [Gehman et al., 2020] has found non-negligible amounts of toxic, harmful, and abusive text in the corpora used to train LLMs.

Lastly, we evaluated our approximation's fidelity by comparing the entropy of our local approximation against that of the GPT-2 distribution, as well as how close our approximation is to the true likelihood in the proximity of the sampled data point as measured by the KL-divergence between the two. All experimental details, hardware specifications, as well as training details are provided in the appendix.

Warcraft Shortest Path For this task, we follow the experimental setting set forth by Pogančić et al. [2019], where our training set consists of $10,000$ terrain maps curated using Warcraft II tileset. Each map encodes a 12×12 grid superimposed on a Warcraft terrain map, where each vertex is weighted according to the cost of the tile, which in turn depends on type of terrain it represents e.g., earth has lower cost than water. These costs are *not* presented to the network. The task is then to generate a minimum-cost path from the upper left to the lower right vertices, where the cost of a path is defined as the sum of costs of the vertices visted by the edges along the path, and the minimum-cost path is not unique, i.e., there exists many paths with the minimum cost, and are all considered correct. The minimum cost path between the top left and bottom right vertices is encoded as an indicator matrix, and serves as a label. Figure 2.10 shows an example input to the

Table 2.8: Results on Sudoku.

Test accuracy %	Exact	Consistent
ConvNet	16.80	16.80
ConvNet + SL	22.10	22.10
RNN	22.40	22.40
RNN + PSEUDOSL	**28.20**	**28.20**

Table 2.9: Results on Warcraft.

Test accuracy %	Exact	Consistent
ResNet-18	55.00	56.90
ResNet-18 + SL	59.40	61.20
CNN-LSTM	62.00	76.60
CNN-LSTM + PSEUDOSL	**66.00**	**79.00**

network, and the input annotated with a possible path.

We use a CNN-LSTM model, where, presented with an image of a terrain map, we use a ResNet18 [He et al., 2016a] to obtain a 128 image embedding, which is then passed on to an LSTM with a single layer, a hidden dim of size 512, and at every time step predicts the next edge in the path conditioned on the image embedding and previous edges. The constraint being maximized by pseudo-semantic loss in this task is that the predicted edges form a valid path.

As has been established in previous work [Xu et al., 2018a; Ahmed et al., 2022c,b], the accuracy of predicting individual labels is often a poor indicator of the performance of the neural network in neuro-symbolic settings, where we are rather more interested in the accuracy of our predicted structure object *exactly* matching the groundtruth label , e.g., *is the prediction a shortest path?*, a metric which we denote "Exact" in our experiments, as well as the accuracy of predicting objects that are *consistent* with the constraint, e.g., *is the prediction a valid path?*, a metric denoted "Consistent". Our results are shown in Table 2.9.

As alluded to repeatedly throughout the course of the paper, the first observation is that using an autoregressive model to predict the shortest path in the grid, even a simple single layer LSTM outperforms both a ResNet-18, as well as a ResNet-18 trained with semantic loss, improving the exact match from 55.00% and 59.40% to 62.00%, and greatly improving the consistency of the predicted paths to 76.00%, an improvement by almost 15%. We also see that using our pseudo-semantic loss, denoted PSEUDOSL, we improve the exact and consistent accuracies to 66.00% and 79.00%, resp.

Sudoku Next, we consider the task of predicting a solution to a given Sudoku puzzle. Here the task is, given a 9×9 partially-filled grid of numbers to fill in the remaining cells such that the entries each row, column, and 3×3 square are unique i.e., each number from 1 to 9 appears exactly once. We use the dataset provided by Wang et al. [2019], consisting of 10K Sudoku puzzles, split into 9K training examples, and 1K test samples, all puzzles having 10 missing entries. As our baseline, we use a 5-layer RNN with a hidden dimension of 128, $\tanh$ non-linearity and a dropout of 0.2. At each time step, the RNN predicts the next cell given as input a one-hot encoding of the previous cell, and conditioned on the partially filled Sudoku. The constraint being maximized by pseudo-semantic loss is that entries in each row, column, and 3×3 squares are unique. Our results are shown in Table 2.8.

In line with our previous experiment, we observe that, once again, a simple RNN outperforms the non-autoregressive model, as well as the same model augmented with semantic loss, although the difference is not that big with regards to semantic loss. Augmenting that same autoregressive model with pseudo-semantic loss, however, increases the gap to a convolutional network, and the same convolutional network augmented with semantic loss to 11.40 and 7.10, respectively.

LLM detoxification Lastly, we consider the task of LLM detoxification. That is, we investigate the effectiveness of logical constraints, enforced using pseudo-semantic loss, at steering the model away from toxic prompted-generations. We choose a *very* simple constraint to be *minimized* by pseudo-semantic loss throughout this task, namely we minimize the probability that any of a list of profanity, slurs, and swear words[4] appear as part of the model generations. Following previous work [Gehman et al., 2020; Wang et al., 2022], we evaluate on the REALTOXICITYPROMPTS, a dataset of almost 100k prompts ranging from nontoxic, assigned a toxicity score of 0, to very toxic, assigned a toxicity score of 1. We focus on GPT-2 [Radford et al., 2019] as a base model for detoxification. As is customary, [Gehman et al., 2020; Wang et al., 2022], we use Perspective API, an online automated model for toxic language and hate speech detection, to score the toxicity of

[4]List downloaded from here.

Table 2.10: Evaluation of LLM toxicity and quality across different detoxification methods on GPT-2 with 124 million parameters. Model toxicity is evaluated on the REALTOXICITYPROMPTS benchmark through Perspective API. **Full**, **Toxic** and **Nontoxic** refer to the full, toxic and nontoxic subsets of the prompts, respectively. **PPL** refers to the model perplexity on the WebText validation set. **PPL** of word banning is evaluated on the 50% nontoxic portion of the WebText validation set. In line with previous work Gehman et al. [2020]; Wang et al. [2022], we characterize toxicity using two metrics: the **Expected Maximum Toxicity** over 25 generations, and the **Toxicity Probability** of a completion at least once over 25 generations. Setting the probabilities of toxic words to zero sending the perplexity of to infinity. We, therefore, report the perplexity on the 50% least toxic prompts dataset for **Word Banning** variants.

Models		Exp. Max. Toxicity (↓)			Toxicity Prob. (↓)			PPL (↓)
		Full	Toxic	Nontoxic	Full	Toxic	Nontoxic	
	GPT-2	0.44	0.62	0.39	34.11%	67.27%	24.85%	25.85
Domain-Adaptive	SGEAT Wang et al. [2022]	0.32	0.46	0.28	14.05%	35.72%	7.99%	28.72
	PseudoSL *(ours)*	**0.29**	0.38	**0.27**	9.80%	20.07%	6.93%	28.14
Word Banning	GPT-2	0.40	0.55	0.36	27.92%	57.86%	19.56%	22.24
	SGEAT Wang et al. [2022]	0.30	0.41	**0.27**	10.73%	27.05%	**6.17%**	24.91
	PseudoSL *(ours)*	**0.29**	**0.37**	**0.27**	**9.20%**	**18.71%**	6.55%	24.19

our predictions. It returns scores in the range 0 to 1.0, corresponding to nontoxic on the one end, and extremely toxic on the other. Though not without limitations, studies [Wang et al., 2022; Welbl et al., 2021] have shown that the toxicity scores from Perspective API are strongly correlated with human evaluations.

We compare GPT-2 against SGEAT [Wang et al., 2022]—which finetunes GPT-2 on the nontoxic portion of its self generation, performing unconditional text generation and retaining only generations with toxicity < 0.5—and against SGEAT augmented with pseudo-semantic loss. We report the *Expected Maximum Toxicity* and the *Toxicity Probability*. The *Expected Maximum Toxicity* measures the worst-case toxicity by calculating the maximum toxicity over 25 generations under the same prompt with different random seeds, and averaging the maximum toxicity over all prompts. *Toxicity Probability* estimates the empirical probability of generating toxic language by evaluating the fraction of times a toxic continuation is generated at least once over 25 generations with different random seeds for all prompts. To understand the impact of detoxification, we evalu-

ate the quality of the LLM using perplexity on the validation split of WebText, used to train GPT-2. Our results are shown in Table 2.10.

Domain-Adaptive Training It was previously shown that SGEAT lowers the toxicity of the generations produced by GPT-2, albeit at a slight cost in terms of perplexity. This is confirmed by our numbers, where we see that SGEAT reduces the average worst-case toxicity as well as the probability of producing a toxic generation when prompted with either toxic or nontoxic prompts. We also observe that using PseudoSL loss alongside SGEAT *further* reduces the overall average worst-case toxicity as well as the probability of producing a toxic generation, while producing a better language model compared to SGEAT. Much of this reduction in toxicity appears to stem primarily from a reduction in the average worst-case toxicity as well as toxicity probability given *toxic prompts*.

Decoding-Time Methods We also compared GPT-2, SGEAT and PseudoSL with variants thereof obtained through augmentation with a decoding-time algorithm, *Word Banning* [Gehman et al., 2020]. *Word Banning* sets the probability of generating any of the words from the aforementioned list of profanity, slurs and swearwords to zero during decoding. We also attempted to compare against NeuroLogic decoding, a search-based decoding algorithm utilizing look-ahead heuristics to optimize for not only the probability of the generated sentence but also the lexical constraints being satisfied. However, attempting to run NeuroLogic decoding on the entire dataset of prompts (100k) using the maximum batch size we could fit on a 48GB GPU yielded an estimated time of 165 hours. Considering a randomly-sampled subset of the prompts, we obtained empty generations for at least 30% of the prompts. The *PPL* of word banning goes to infinity as the probabilities of some banned words are set to zero. We, therefore, report the perplexity on the 50% least toxic portion of the prompts dataset. We observe that augmenting all of the domain-adaptive training baselines with Word Banning reduces their average worst-case toxicity as well as the their toxicity probability. SGEAT augmented with Word Banning exhibits lower toxicity, both average worst-case toxicity and toxicity probability, than all other non-PseudoSL variants. Interestingly, even

augmented with Word Banning, SGEAT exhibits higher toxicity than the base PseudoSL model. PSeudoSL augmented with Word Banning exhibits the lowest overall toxicity, both average worst-case toxicity and toxicity probability, with the gap to the second best in terms of toxicity probability being particularly stark on the toxic prompts. We also note that, in our evaluation of PPL for Word Banning, having discarded the toxic sentences, assigned a lower probability under our model, the perplexity of our model is now closer to that of GPT-2.

Fidelity evaluation Lastly, we evaluated the fidelity of our approximation. We start comparing the entropy of our approximate distribution to the true distribution. We want this quantity to be low, as it would mean our approximation only considers assignments centered around the model sample. We also evaluate the KL-divergence of our approximate distribution from the true distribution in the neighborhood of a model sample. We want this quantity to be low as well, as it corresponds to how faithful our approximation is to the true distribution in the neighborhood of the model sample. Intuitively, the KL-divergence measures how many extra bits are needed to encode samples from our approximation using a code optimized for GPT-2, and is zero when the two distributions coincide. We find the entropy of GPT-2 is 80.89 bits while the entropy of our approximation is, on average, 35.08 bits. We also find the KL-divergence $D_{\mathrm{KL}}(\tilde{p}_{\boldsymbol{y}} \mid\mid p_{\boldsymbol{\theta}})$ is on average 4.8 bits. That is we only need 4 extra bits, on average, to encode the true distribution w.r.t. our approximation distribution. Intuitively, we want to stay close to the sample to ensure high fidelity, while retaining a distribution to ensure differentiability and maximum generality within tractability bounds.

2.3.5 Related Work

In an acknowledgment to the need for both symbolic as well as sub-symbolic reasoning, there has been a plethora of recent works studying how to best combine neural networks and logical reasoning, dubbed *neuro-symbolic AI*. The focus of such approaches is typically making probabilistic reasoning tractable through first-order approximations, and differentiable, through reducing logical formulas into arithmetic objectives, replacing logical operators with their fuzzy t-norms, and

implications with inequalities [Kimmig et al., 2012; Rocktäschel et al., 2015; Fischer et al., 2019].

Another class of neuro-symbolic approaches have their roots in logic programming. DeepProbLog [Manhaeve et al., 2018] extends ProbLog, a probabilistic logic programming language, with the capacity to process neural predicates, whereby the network's outputs are construed as the probabilities of the corresponding predicates. This simple idea retains all essential components of ProbLog: the semantics, inference mechanism, and the implementation. In a similar vein, Dai et al. [2018] combine domain knowledge specified as purely logical Prolog rules with the output of neural networks, dealing with the network's uncertainty through revising the hypothesis by iteratively replacing the output of the neural network with anonymous variables until a consistent hypothesis can be formed. Bošnjak et al. [2017] present a framework combining prior procedural knowledge, as a Forth program, with neural functions learned through data. The resulting neural programs are consistent with specified prior knowledge and optimized with respect to data.

Diligenti et al. [2017a] and Donadello et al. [2017] use first-order logic to specify constraints on outputs of a neural network. They employ fuzzy logic to reduce logical formulas into differential, arithmetic objectives denoting the extent to which neural network outputs violate the constraints, thereby supporting end-to-end learning under constraints. Xu et al. [2018a] introduced semantic loss, which circumvents the shortcomings of fuzzy approaches, while still supporting end-to-end learning under constraints. More precisely, *fuzzy reasoning* is replaced with *exact probabilistic reasoning*, made possible by compiling logical formulae into structures supporting efficient probabilistic queries.

There has recently been a plethora of approaches ensuring consistency by embedding the constraints as predictive layers, including semantic probabilistic layers (SPLs) [Ahmed et al., 2022b], MultiplexNet [Hoernle et al., 2022] and C-HMCNN [Giunchiglia and Lukasiewicz, 2020]. Much like semantic loss [Xu et al., 2018a], SPLs maintain sound probabilistic semantics, and while displaying impressive scalability to real world problems, but might struggle with encoding harder constraints. MultiplexNet is able to encode only constraints in disjunctive normal form, which is problematic for generality and efficiency as neuro-symbolic tasks often involve an intractably

large number of clauses. HMCCN encodes label dependencies as fuzzy relaxation and is the current state-of-the-art model for hierarchical multi-label classification [Giunchiglia and Lukasiewicz, 2020], but, similar to its recent extension [Giunchiglia and Lukasiewicz, 2021], is restricted to a certain family of constraints.

A related line of research focuses on constrained text generation, modifying the decoding algorithm to inject *lexical* constraints into the beam search process. Such methods include constrained beam search [Post and Vilar, 2018], NeuroLogic Decoding [Lu et al., 2021] and A*esque NeuroLogic Decoding [Lu et al., 2022b]. And although they can be easily applied to various language models without training, these search-based methods can be inefficient as they suffer from large search spaces. Recent works like NADO [Meng et al., 2022] and FUDGE [Yang and Klein, 2021] train auxiliary neural models to provide token-level guidance for autoregressive generation. In a similar vein, GeLaTo [Zhang et al., 2023a] augments a large language model with guidance from a tractable probabilistic model to guarantee the keyword tokens are part of the generated sentence while retaining its fluency. Another family of approaches that enforce keyword-type constraints are insertion-based language models [Lu et al., 2022a; Susanto et al., 2020], where the initial sequences only consist of the desired keywords and the transition phrases are repeatedly inserted to complete the sentences.

Throughout this work, we assumed that the constructing a logical circuit from a logical formula was easy. This is, in general, not the case. Ahmed et al. [2023a] offer an approach, by assuming the sub-problems are independent, and iteratively relaxing the independence assumption according to the sub-problems that most violate that assumption as measured using mutual information.

CHAPTER 3

Guarantees Within and Without Neural Networks

It is very often desirable, if not crucial, to provide *guarantees* on a system's behavior. Much of the neuro-symbolic AI literature has been focused on biasing neural networks towards predictions that satisfy the constraint, but fall short of providing any such guarantees. To that end, we propose *semantic probabilistic layers (SPLs)*: drop-in replacements for the traditional softmax layer that guarantee the neural network's predictions are consistent with a set of constraints, while being amenable to end-to-end learning. SPLs combine exact probabilistic inference with logical reasoning in a modular way, learning arbitrarily-complex distributions over the variables and restricting their support to the possible worlds of the constraint.

Very often the utility of constraints can extend beyond just the output layer of a neural network to being part of the neural network architecture. Take for instance the task of *learning to explain (L2X)*, where we are interested in learning the k-subset of n words that best explain a classifier's predicted sentiment given a user review. This necessitates sampling from the distribution over all subsets of size k, a task we show to be tractable for any n and k. Having evaluated the loss on the sampled model, a more substantial challenge presents itself: how do we propagate the error through discrete sampling, an inherently non-differentiable operation? Reparameterizing the samples in terms of the marginals, we show that the gradient of the loss w.r.t. the samples can be estimated as the gradient w.r.t. the marginals of the distribution over all subsets of size k. We show that the distribution's marginals can be computed tractably, and easily, using auto-differentiation. This constitutes a new, general purpose gradient estimator, which we termed *SIMPLE*, that exhibits lower bias and lower variance compared to state-of-the-art gradient estimators.

3.1 Semantic Probabilistic Layers for Neuro-Symbolic Learning

Modularity is among the major factors that propelled the Cambrian explosion of deep learning [Goodfellow et al., 2016]. By stacking multiple *differentiable* layers together, practitioners are able to train deep classifiers in an end-to-end fashion with little effort. However, despite its flexibility, *this modular approach to learning does not guarantee that the predictions of these models conform to our expectations of what makes sense*. On the contrary, unconstrained deep classifiers are notorious for leading to predictions that are inconsistent with the logical constraints governing an underlying domain.

This is even more evident in, and crucial for, structured output prediction (SOP) tasks, where classifiers have to predict hundreds of mutually constrained labels [Tsochantaridis et al., 2004; Borchani et al., 2015]. Consider for example a classical SOP task such as multi-label classification (MLC) [Tsoumakas and Katakis, 2007]. Learning a multi-label classifier that disregards the correlations among labels, e.g., by considering them *fully independent* given the inputs, yields sub-optimal results [Bielza et al., 2011]. In more challenging tasks such as hierarchical MLC (HMLC) [Sorower, 2010] or pathfinding [Pogančić et al., 2019], leveraging the domain's logical constraints (encoding, e.g., the label hierarchy or acyclicity and connectedness of a path) at training time can improve prediction accuracy [Levatić et al., 2015], but it cannot guarantee that the predictions are always *consistent* with the constraints at inference time [Giunchiglia and Lukasiewicz, 2020]. Figure 3.1 illustrates this problem in the context of pathfinding: constraint-unaware neural networks systematically fail to predict label configurations that form a valid path. In many safety-critical scenarios such as protein function [Radivojac et al., 2013] and interaction prediction [Sacca et al., 2014], and drug discovery [De Cao and Kipf, 2018; Di Liello et al., 2020], predicting inconsistent solutions can not only be harmful but also highly expensive [Amodei et al., 2016; Giunchiglia et al., 2022].

Unsurprisingly, due to their discrete nature, injecting logical constraints into deep neural networks while retaining modularity and differentiability is extremely challenging, as demonstrated

GROUND TRUTH RESNET-18 SEMANTIC LOSS SPL (ours)

Figure 3.1: **Neural nets struggle with satisfying validity constraints in complex semantic SOP tasks** such as predicting the lowest-cost path from the top-left to the bottom-right corners of a Warcraft map. Even state-of-the-art neuro-symbolic approaches like the Semantic Loss [Xu et al., 2018a] fail to ensure consistency with hard rules (c). SPLs in contrast guarantees validity while retaining modularity, expressiveness and efficiency. See Sec. 3.1.4 for complete experimental details and additional results.

in the *neuro-symbolic learning* literature [Sarker et al., 2021]. One such attempt has been to learn neural networks that satisfy the logical constraints by explicitly minimizing a differentiable loss term encoding the probability that the networks violates the constraint for a given prediction. And while successful, such approaches do not guarantee consistency of the predictions at test time. More recently, researchers have proposed predictive layers that do guarantee consistency, but these are restricted to specific kinds of symbolic knowledge [Giunchiglia and Lukasiewicz, 2020; Sivaraman et al., 2020] or become intractable for even moderately complex logical constraints [Hoernle et al., 2022].

Motivated by these observations, we introduce a novel **S**emantic **P**robabilistic **L**ayer (SPL) for modeling intricate correlations, and logical constraints on the labels of the output space in a modular and probabilistically sound manner. It does so by leveraging recent advancements in the literature on probabilistic circuits [Vergari et al., 2020; Choi et al., 2020a]. The key features of SPL are that, on the one hand, it can be used as a *drop-in replacement* for common predictive layers of deep nets like sigmoid layers, and on the other, it *guarantees* the output's consistency with any prespecified logical constraints. Importantly, SPL also supports efficient inference and – perhaps surprisingly – does not complicate training.

3.1.1 Designing a probabilistic layer for neuro-symbolic SOP

Notation. In the following, we denote scalar constants x in lower case, random variables X in upper case, vectors of constants $\boldsymbol{x}$ in bold and vectors of random variables $\mathbf{X}$ in capital boldface. $\mathbb{1}\{\varphi\}$ denotes the indicator function that evaluates to 1 if the statement φ holds and to 0 otherwise. We denote by $\boldsymbol{x} \models \mathsf{K}$ that the value assignment $\boldsymbol{x}$ to variables $\mathbf{X}$ satisfies a logical formula K.

Neuro-symbolic SOP. We tackle SOP tasks in which a neural net classifier must learn to associate instances $\boldsymbol{x} \in \mathbb{R}^D$ to L *interdependent* labels, identified by the vector $\boldsymbol{y} \in \{0,1\}^L$. We assume that we can abstract any neural classifier into two components: a feature extractor f that maps inputs $\mathbf{X}$ to a M-dimensional embedding $\mathbf{Z} = f(\mathbf{X})$ and a predictive final layer that outputs the label distribution $p(\mathbf{Y} \mid \mathbf{Z})$. For example, the simplest, and yet widely adopted [Mullenbach et al., 2018; Xu et al., 2018a; Giunchiglia and Lukasiewicz, 2020], predictive layer in neural classifiers for SOP considers labels Y_i to be conditionally independent from each other given $\mathbf{Z}$, i.e., $p(\mathbf{Y} \mid \mathbf{Z}) = \prod_{i=1}^{L} p(Y_i \mid \mathbf{Z})$. We refer to this as *fully independent layer* (FIL). In a FIL, $p(Y_i = y_i \mid \boldsymbol{z})$ is computed as $\sigma(\mathbf{w}_i^\top \boldsymbol{z})$ where $\mathbf{w}_i \in \mathbb{R}^M$ is a vector of parameters and $\sigma(x)$ is the logistic sigmoid function $1/(1+e^{-x})$.

We are interested in dependencies between labels that can occur both as *correlations*, as is the case in MLC [Dembczyński et al., 2012], and as *logical constraints* encoded by logical formulas. For example, in a HMLC task [Giunchiglia and Lukasiewicz, 2020] one logical constraint can encode the fact that observing a label for the class cat and dog, implies observing the label for their superclass animal

$$(Y_{\mathsf{cat}} = 1 \implies Y_{\mathsf{animal}} = 1) \wedge (Y_{\mathsf{dog}} = 1 \implies Y_{\mathsf{animal}} = 1). \tag{3.1}$$

Specifically, we assume symbolic knowledge to be supplied in the form of constraints encoded as a logical formula denoted as K and defined over the labels $\mathbf{Y}$ and optionally over a subset of the discrete input variables in $\mathbf{X}$, if any (e.g., in our experiments, the predicted simple path is

Table 3.1: **SPL is the only approach to satisfy all the desiderata for neuro-symbolic SOP.** An in-depth discussion of all competitors can be found in Sec. 3.1.3.

	Losses			Layers				
Desideratum	DL2	SL	NeSyEnt	FIL	EBM	MultiplexNet	CCN	SPL *(ours)*
(D1) Probabilistic	✗	✓	✓	✓	✗	✓	✗	✓
(D2) Expressive	✗	✗	✗	✗	✓	✗	✗	✓
(D3) Consistent	✗	✗	✗	✗	✗	✓	✓	✓
(D4) General	✓	✓	✓	✗	✓	✓	✗	✓
(D5) Modular	✓	✓	✓	✓	✓	✓	✓	✓
(D6) Efficient	✓	✓	✓	✓	✗	✗	✓	✓

constrained to lie within the subset of edges appearing in the input graph, see Sec. 3.1.4). On the other hand, we expect a model to learn the label correlations from data. We call such task *neuro-symbolic SOP*.

Desiderata for neuro-symbolic SOP. To tackle this setting, we seek an algorithmic strategy for replacing the predictive layer in any neural network classifier with little effort, with the aim of injecting complex symbolic knowledge and allowing for flexible probabilistic reasoning. We formalize these observations into the following six desiderata for our predictive layer:

D1. **Probabilistic**: The layer should enjoy sound probabilistic semantics, and deliver normalized probabilistic predictions to facilitate maximum-likelihood learning and sound decision making by virtue of calibrated probabilistic predictions

D2. **Expressive**: It should be able to compactly encode intricate *correlations* between labels.

D3. **Consistent**: It should always output predictions that are consistent with the prespecified symbolic knowledge, i.e., for all $\boldsymbol{x}$ and $\boldsymbol{y}$, if $(\boldsymbol{x}, \boldsymbol{y}) \not\models \mathsf{K}$ then $p(\boldsymbol{y} \mid \boldsymbol{x}) = 0$.

D4. **General**: It should support rich *logical constraints* over the labels expressed in some formal language, e.g., propositional logic.

D5. **Modular**: It should be applicable to any off-the-shelf (and possibly pretrained) neural network in a modular fashion, enabling end-to-end learning and rapid prototyping.

D6. **Efficient**: The time required by the predictor to compute a prediction should be linear in the size of the predictor and of the hard constraint representation.

For example, FILs are clearly probabilistic (D1), modular (D5), and efficient (D6), but at the cost of being incapable of modeling intricate correlations and logical constraints and thus generating inconsistent predictions (D2–D4) (see also Figure 3.1). Table 3.1 summarizes how the other popular and effective approaches to neuro-symbolic SOP nowadays fall short of one of more desiderata as well. We discuss this in detail in Sec. 3.1.3. To the best of our knowledge, our proposed *semantic probabilistic layers* (SPLs) are the first algorithmic solution to satisfy all above desiderata.

SPL. At a high level, SPL realizes the above desiderata in a single layer that combines exact probabilistic inference with logical reasoning in a clean and modular way, learning complex distributions and restricting their support to solutions of the constraint.

Definition 2 (Semantic probabilistic layer (SPL))**.** *Given an input configuration $\boldsymbol{x}$, a SPL decomposes the computation of the probability of a label configuration as:*

$$p(\boldsymbol{y} \mid f(\boldsymbol{x})) = q_{\boldsymbol{\Theta}}(\boldsymbol{y} \mid f(\boldsymbol{x})) \cdot c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y}) / \mathcal{Z}(\boldsymbol{x}) \qquad \textit{where} \quad \mathcal{Z}(\boldsymbol{x}) = \sum\nolimits_{\boldsymbol{y}} q_{\boldsymbol{\Theta}}(\boldsymbol{y} \mid \boldsymbol{x}) \cdot c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y}). \tag{3.2}$$

Here, $q_{\boldsymbol{\Theta}}(\boldsymbol{y} \mid f(\boldsymbol{x}))$ is a module to perform probabilistic reasoning by encoding an expressive distribution over the labels parameterized by $\boldsymbol{\Theta}$; $c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$ is a module to ensure consistency of the predictions by encoding logical constraints K and being non-zero only when K is satisfied, i.e., $c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y}) = \mathbb{1}\{(\boldsymbol{x}, \boldsymbol{y}) \models \mathsf{K}\}$; and $\mathcal{Z}(\boldsymbol{x})$ is a renormalization term, also called the partition function. It is worth noting that this amounts to taking a product of experts [Hinton, 1999] which is, in general, hard.

Figure 3.2 illustrates the computational graph of our SPL at training time. In order to satisfy all D1-D6, we will realize both $q_{\boldsymbol{\Theta}}$ and c_{K} as *circuits* [Vergari et al., 2020; Choi et al., 2020a], constrained computational graphs that enable tractable computations. Differently from FILs, $q_{\boldsymbol{\Theta}}$ in SPLs can encode an expressive joint distributions over the labels and therefore attain full expressiveness by scaling the number of parameters $\boldsymbol{\Theta}$ (D2). Consistency is guaranteed by the component c_{K}: by multiplying it to the joint probability of a label configuration the resulting product distribution $r_{\boldsymbol{\Theta},\mathsf{K}}(\boldsymbol{y}, \boldsymbol{x}) = q_{\boldsymbol{\Theta}}(\boldsymbol{y} \mid f(\boldsymbol{x})) \cdot c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$ will have its support effectively cut by K, and thus cannot allocate any probability mass to inconsistent predictions (D3). Additionally, c_{K} will allow to encode general propositional logical constraints in a compact computational graph (D4). Lastly, the product $r_{\boldsymbol{\Theta},\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$ is fully differentiable and allows SPL to be an off-the-shelf replacement for other predictive layers (see Figure 3.2) and enables end-to-end learning (D5). By renormalizing $r_{\boldsymbol{\Theta},\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$ and outputting normalized probabilities, SPL enables the exact computation of gradients for $\boldsymbol{\Theta}$, which can therefore be trained by maximum likelihood (D1).

Thanks to recent advancements in the literature on circuits, we can compute the partition function $\mathcal{Z}(\boldsymbol{x})$ efficiently in time linear in the size of $r_{\boldsymbol{\Theta},\mathsf{K}}$, thus preserving efficiency (D6) and not compromising on the other desiderata. This will also yield correct (and consistent) predictions at test time, when an SPL computes the MAP state $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y}} r_{\boldsymbol{\Theta},\mathsf{K}}(\boldsymbol{y}, \boldsymbol{x}) / \sum_{\boldsymbol{y}} r_{\boldsymbol{\Theta},\mathsf{K}}(\boldsymbol{y}, \boldsymbol{x})$. The next section clarifies *how* to implement the modules of SPL as circuits while satisfying these desiderata.

3.1.2 Realizing SPLs with tractable circuit representations

The components of SPLs are *circuits*, a large class of computational graphs that can represent both functions and distributions [Choi et al., 2020a; Darwiche and Marquis, 2002]. Circuits subsume many tractable generative and discriminative probabilistic models—from Chow-Liu and latent tree models [Chow and Liu, 1968; Choi et al., 2011], to hidden Markov models (HMMs) [Rabiner and Juang, 1986], sum-product networks (SPNs) [Poon and Domingos, 2011], decision trees [Khosravi et al., 2020; Correia et al., 2020a], and deep regressors [Khosravi et al., 2019b]—as well as many

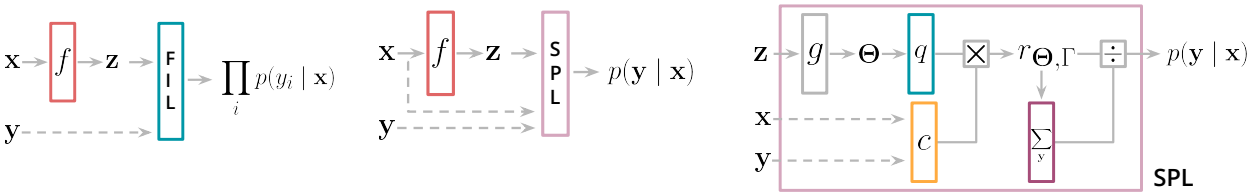


Figure 3.2: **A high level view of SPLs.** The predictive layer of a neural network for neuro-symbolic SOP, e.g., a FIL (**left**), can be readily replaced by a SPL (**middle**). SPLs are implemented (**right**) by multiplying together a probabilistic circuit $q_{\Theta}(\mathbf{Y} \mid f(\mathbf{X}))$ parameterized by (a function g of) the network's embeddings $f(\mathbf{X})$, and a constraint circuit $c_{\mathsf{K}}(\mathbf{X}, \mathbf{Y})$ embodying the symbolic knowledge. The result is normalized by efficiently marginalizing over the product circuit $r_{\Theta,\mathsf{K}}$, so as to guarantee fully probabilistic semantics and end-to-end differentiable learning by maximum likelihood.

compact representations of logical formulas, such as (ordered) binary decision diagrams [Akers, 1978], sentential decision diagrams (SDDs) [Darwiche, 2011b] and others [Darwiche and Marquis, 2002].

The key idea behind SPLs is to leverage this single formalism to represent both an expressive joint distribution for $q_{\Theta}(\boldsymbol{y} \mid f(\boldsymbol{x}))$ and a compact encoding of the logical constraints for $c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$, while ensuring the exact and efficient evaluation of Equation 3.2. This can be achieved by ensuring that these computational graphs abide certain structural properties: *smoothness*, *decomposability*, *determinism* and *compatibility* [Darwiche and Marquis, 2002; Vergari et al., 2021]. Next, we introduce *probabilistic circuits* for modeling q_{Θ} (Sec. 3.1.2.1) and *constraint circuits* for c_{K} (Sec. 3.1.2.2), while in Sec. 3.1.2.4 we propose a more efficient implementation of SPL utilizing a single circuit.

3.1.2.1 Representing expressive distributions with probabilistic circuits

We start by introducing circuits for *joint* probability distributions, and then extend the discussion to *conditional* distributions, which we use to implement $q_{\Theta}(\mathbf{Y} \mid f(\mathbf{X}))$ in SPLs.

Definition 3 (Circuits)**.** *A circuit h over variables $\mathbf{Y}$ is a computational graph encoding a parameterized function $h_{\Theta}(\mathbf{Y})$ by combining three kinds of computational units: input functional*

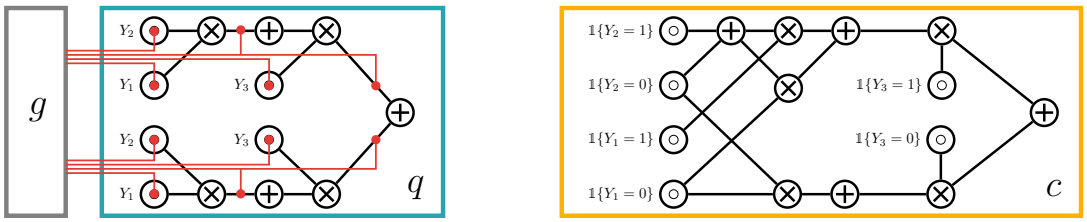


Figure 3.3: **Examples of circuits in SPL**. **Left**: a neural conditional probabilistic circuit $q_{\boldsymbol{\Theta}}$. Red lines indicate how the output of g parameterizes the input distribution parameters $\boldsymbol{\lambda}$ and the sum unit parameters $\boldsymbol{\omega}$ of q, both indicated as red dots. **Right**: constraint circuit encoding the logical constraint of Equation 3.1 where labels are $Y_i \in \{Y_{\mathsf{cat}}, Y_{\mathsf{dog}}, Y_{\mathsf{animal}}\}$. Note that q and c are smooth, decomposable (Def. 5) and compatible (Def. 9) and c is deterministic (Def. 8). By parameterizing c via g we can obtain a single-circuit SPL (Sec. 3.1.2.4). **Both:** circuits q and c are compatible, as product units with the same scope decompose in the same way. E.g., consider the first two product units of q and c, right to left and top to bottom. Both units decompose $\{Y_3, Y_2, Y_1\}$ into Y_3 and Y_2, Y_1.

units, sum units, and product units. An input functional n represents a base parametric function $h_n(\phi(n); \boldsymbol{\lambda})$ over some variables $\phi(n) \subseteq \mathbf{Y}$, called its scope, and it is parameterized by $\boldsymbol{\lambda}$. Sum and product units n elaborate the output of other units, denoted $\mathsf{ch}(n)$. Sum units are parameterized by $\boldsymbol{\omega}$ and compute the weighted sum of their inputs $\sum_{c \in \mathsf{ch}(n)} \omega_c h_c(\phi(n))$, while product units compute $\prod_{c \in \mathsf{ch}(n)} h_c(\phi(n))$. The parameters $\boldsymbol{\Theta}$ of a circuit encompass the parameters of all input functionals ($\boldsymbol{\lambda}$) and the parameters of sum units ($\boldsymbol{\omega}$).

For any input $\boldsymbol{y}$, the value of $h_{\boldsymbol{\Theta}}(\boldsymbol{y})$ can be evaluated by propagating the output of the input units through the computational graph and reading out the value of the last unit. The *support* of h is the set of all states $\boldsymbol{y}$ of $\mathbf{Y}$ for which the output is non-zero, i.e., $\mathsf{supp}(h) = \{\boldsymbol{y} \mid h(\boldsymbol{y}) \neq 0\}$.

Definition 4 (Probabilistic circuits (PCs))**.** *A circuit q is a PC if it encodes a (possibly unnormalized) probability distribution, i.e., $q_{\boldsymbol{\Theta}}(\boldsymbol{y})$ is non-negative for all configurations $\boldsymbol{y}$ of $\mathbf{Y}$.*

From here on, we will assume PCs to have positive sum parameters $\boldsymbol{\omega}$ and whose input units model valid distributions, e.g., Bernoullis, as these conditions are sufficient for satisfying Def. 4. Moreover, w.l.o.g. we will assume the sum and product units to be organized into alternating layers, and that every product unit n receives only two inputs c_1, c_2, i.e., $q_n(\mathbf{X}) = q_{c_1}(\mathbf{Y}) \cdot q_{c_2}(\mathbf{Y})$. These conditions can easily be enforced in polynomial time [Vergari et al., 2015, 2019]. We are

specifically interested in smooth and decomposable PCs, as they will be enabling efficient inference in SPL (Sec. 3.1.2.3).

Definition 5 (Smoothness & Decomposability)**.** *A circuit is smooth if for every sum unit n, its inputs depend on the same variables: $\forall\, c_1, c_2 \in \mathsf{ch}(n), \phi(c_1) = \phi(c_2)$. It is decomposable if the inputs of every product unit n depend on disjoint sets of variables: $\mathsf{ch}(n) = \{c_1, c_2\}, \phi(c_1) \cap \phi(c_2) = \emptyset$.*

Smooth and decomposable PCs are both expressive and efficient: they can encode distributions with hundred millions of parameters and be effectively learned by gradient ascent [Peharz et al., 2020b]. The structure of their computational graph can be either specified manually [Poon and Domingos, 2011; Peharz et al., 2020b,a] or acquired automatically from data [Vergari et al., 2015; Rahman et al., 2014; Dang et al., 2022b], e.g., by first learning a latent tree model and then compiling the latter into a circuit [Liu and Van den Broeck, 2021]. These circuits are competitive with intractable models such as variational autoencoders and normalizing flows scores on several benchmarks [Liu et al., 2022a].

As proposed by Shao et al. [2022], any (smooth and decomposable) PC $q_{\boldsymbol{\Theta}}(\mathbf{Y})$ encoding a *joint* distribution over the labels $\mathbf{Y}$ can be turned into a (smooth and decomposable) *conditional* circuit, conditioned by input variables $\mathbf{X}$, by letting its parameters be a function of $\mathbf{X}$.

Definition 6 (Neural conditional circuits [Shao et al., 2020])**.** *A conditional circuit $q(\mathbf{Y}; \boldsymbol{\Theta} = g(\mathbf{X}))$ models the conditional distribution $p(\mathbf{Y} \mid \mathbf{X})$ via a differentiable function g that maps every input configuration $\boldsymbol{x}$ to the set of parameters of $\boldsymbol{\Theta}$ of p, also called the gating function.*

An example of a smooth and decomposable conditional circuit is shown in Figure 3.3. This design immediately allows us to implement $q_{\boldsymbol{\Theta}}(\mathbf{Y} \mid f(\mathbf{X}))$ in SPL as a conditional PC whose gating function maps the feature embedding space $\mathbb{R}^K$ to the parameter space $\mathbb{R}_+^{|\boldsymbol{\Theta}|}$, realizing $q(\mathbf{Y}; \boldsymbol{\Theta} = g(f(\mathbf{X})))$. As such, the gating function g creates a clean interface between any pre-trained feature extractor f and the PC q (Figure 3.3). While one can devise g in several ways, we strive for simplicity in our experiments and adopt vanilla multi-layer perceptrons (MLPs) whose

final activations are either sigmoids, if they have to predict the parameters $\boldsymbol{\lambda}$ of the Bernoulli input distributions of q, or softmax, if they output the sum unit parameters $\boldsymbol{\omega}$ (Def. 3).

3.1.2.2 Encoding logical formulas with constraint circuits

The next step is to translate a logical constraint K into a smooth and decomposable circuit $c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y})$. To this end, we employ a special type of PCs, defined as follows.

Definition 7 (Constraint circuits)**.** *A PC c over variables $\mathbf{X} \cup \mathbf{Y}$ is a constraint circuit encoding prior knowledge K if it computes $\mathbb{1}\{(\boldsymbol{x}, \boldsymbol{y}) \models \mathsf{K}\}$ for every configuration $(\boldsymbol{x}, \boldsymbol{y})$.*

As a practical way to realize such a circuit, we will consider constraint circuits that have all sum unit parameters equal to 1 and input functionals that are indicator functions over their scope, e.g., $c_n(\boldsymbol{z}) = \mathbb{1}\{\boldsymbol{z} \models \varphi(n)\}$ where $\mathbf{Z}$ is the scope of the input and $\varphi(n)$ a constraint over it. Furthermore, we require each sum unit in it to be *deterministic*.

Definition 8 (Determinism)**.** *A sum unit n is deterministic if its inputs have disjoint supports, i.e.,* $\forall\, c_1, c_2 \in \mathsf{ch}(n), c_1 \neq c_2 \implies \mathsf{supp}(c_1) \cap \mathsf{supp}(c_2) = \emptyset$.

Figure 3.3 shows an example of a deterministic constraint circuit. Thanks to determinism, we can readily translate classical compact representations for logical formulas such as (ordered) binary decision diagrams [Akers, 1978; Bryant and Meinel, 2002] and sentential decision diagrams (SDDs) [Darwiche, 2011b] into constraint circuits as defined above. This becomes evident when they are written in the language of negation normal form [Darwiche and Marquis, 2002] and their *and* gates (resp. *or* gates) are replaced with product units (resp. sum units) [Choi et al., 2020a]. A logic constraint can therefore be represented as a constraint circuit for SPLs, by utilizing any of the many tools available for OBDDs [Toda and Soh, 2016] or SDDs [Choi and Darwiche, 2013; Oztok and Darwiche, 2015]. Sec. B.1.2 illustrates in detail how to compile the example constraint of Equation 3.1 into the constraint circuit of Figure 3.3 in this way.

The worst-case size of the constraint circuit depends on a) the algorithm employed for compilation and, b) the local structure of the constraints, rather than the number of labels. For example,

in our Warcraft experiment (see Sec. 3.1.4), we have a label configuration space over edges in a 12×12 grid, yielding $2^{12^2} = 2^{144} \approx 10^{43}$ states. However, only 10^{10} configurations satisfy the constraint that these edge labels form a valid path in the grid. If our compilation algorithm were to simply enumerate these configurations, putting them in a logical OR (as done in some neuro-symbolic learners such as MultiplexNet [Hoernle et al., 2022]), the size of the constraint circuit, denoted as $|c|$, would be 10^{10}. However, by using recent advancements in compiling logical formulas into constraints circuits, we can can greatly reduce the circuit size. For example, the PySDD compiler used generates circuits whose size is worst-case exponential in the treewidth of the CNF representation of the logical formula, but typically much smaller. See Sec. 3.1.4 and Sec. B.1.5 for details.

3.1.2.3 Efficient inference in SPLs

As discussed above, PCs can be expressive (D2) and are modular (D5), while constraint circuits ensure consistency (D3) for general constraints (D4). What remains to be shown to complete SPLs is that the product supports efficient normalization (D1) and inference (D6), specifically that it allows for the efficient evaluation of the normalization constant of $r_{\mathbf{\Theta},\mathsf{K}}$, and its MAP state. To this end, we need to introduce the notion of compatibility between the two circuits [Vergari et al., 2021].

Definition 9 (Compatible circuits in SPLs)**.** *A smooth and decomposable conditional PC $q(\mathbf{Y}; \mathbf{\Theta})$ is compatible over variables $\mathbf{Y}$ with a smooth and decomposable constraint circuit $c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X})$ if any pair of product units $n \in q$ and $m \in c_{\mathsf{K}}$ with the same scope over $\mathbf{Y}$ can be rearranged to be mutually compatible and decompose in the same way: $(\phi(n) = \phi(m)) \implies (\phi(n_i) = \phi(m_i),\ n_i$ and m_i are compatible$)$ for some rearrangement of the inputs of n (resp. m) into n_1, n_2 (resp. m_1, m_2). The two circuits q and c shown in Figure 3.3 are compatible.*

Theorem 1 (Efficient inference in SPLs)**.** *If $q(\mathbf{Y}; \mathbf{\Theta})$ and $c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X})$ are two smooth, decomposable and compatible circuits, then computing Equation* 3.2 *can be done in $\mathcal{O}(|q|\,|c|)$ time. Furthermore,*

if they are also deterministic, then computing the MAP state can be done in $\mathcal{O}(|q|\,|c|)$ time.

The proof can be found in Sec. B.1.1. How do we come up with compatible circuits? One option is to have a PC q that is compatible with every possible smooth and decomposable circuit c. To do so, we can represent q as a mixtures of M fully-independent models; i.e., $\sum_{i=1}^{M} \omega_i \prod_j q(Y_j; \boldsymbol{\Theta}_i)$. This additional sum unit can be enough to be more expressive than a FIL and already delivers more accurate predictions than any competitor, as our experiments in pathfinding show (Sec. 3.1.4). An example of such a circuit is shown in Figure 3.3. Another sufficient condition for compatibility is that both q and c share the exact same hierarchical scope partitioning [Vergari et al., 2021], sometimes called a vtree or variable ordering [Choi et al., 2020a; Pipatsrisawat and Darwiche, 2008].

This can be done easily if one first compiles logical constraints into OBDDs or SDDs and then uses a mechanized algorithm to build q as in [Peharz et al., 2020b] to create a compatible structure. Additionally, to ensure q is a deterministic PC, we could exploit the mechanized construction proposed in Shih and Ermon [2020]. Computing the exact MAP state, however, is of less concern as approximate inference algorithms, e.g., beam search decoding [Vijayakumar et al., 2016] or iterative pruning [Choi et al., 2022], are nowadays a commodity in deep learning frameworks. For non-deterministic PCs, we compute the MAP state with a faster approximation by replacing non-deterministic sum units with max units [Peharz et al., 2016]. This runs in time linear in the size of r, and yet delivers state-of-the-art accuracies in our experiments Sec. 3.1.4.

3.1.2.4 A single-circuit SPL

The two-circuit design we proposed for SPLs provides a clear and theoretically-backed interface between neural networks and probabilistic and symbolic reasoning. This setup, however, can sometimes be wasteful, as it requires to compute the product of two circuits and renormalize. We circumvent this issue by designing a single-circuit implementation of SPL.

Definition 10 (Single-circuit SPL)**.** *Given an input configuration $\boldsymbol{x}$, a single-circuit SPL computes $p(\boldsymbol{y} \mid \boldsymbol{x}) = c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X}; \boldsymbol{\Omega} = g(f(\mathbf{X})))$ where c_{K} is a neural conditional constraint circuit whose*

Table 3.2: SPLs outperform all loss-based competitors in the neuro-symbolic benchmarks of [Xu et al., 2018a].

	Simple Path			Preference Learning		
Architecture	Exact	Hamming	Consistent	Exact	Hamming	Consistent
MLP+FIL	5.6	85.9	7.0	1.0	**75.8**	2.7
MLP+$\mathcal{L}_{\mathsf{SL}}$	28.5	83.1	75.2	15.0	72.4	69.8
MLP+NeSyEnt	30.1	83.0	91.6	18.2	71.5	96.0
MLP+SPL (*ours*)	**37.6**	**88.5**	**100.0**	**20.8**	72.4	**100.0**

sum-unit parameters $\boldsymbol{\Omega}$ are non-unitary values parameterized via a gating function g.

In a nutshell, we can directly realize SPL by compiling a complex logical constraints (D4) into a deterministic constraint circuit c_{K}, as before, and then parameterizing it with a gating function of the network embeddings $f(\mathbf{X})$, i.e., allowing its sum units to be non-unitary and input dependent. Since the support of c_{K} is already restricted to exactly match the constraint K (D3), parameterizing $\boldsymbol{\Omega}$ induces an expressive probability distribution over the label configurations that are consistent with K (D2). We can further guarantee that the circuit's output are normalized probabilities (D1, D6) by enforcing the parameters ω of each sum unit to form a convex combination [Peharz et al., 2015]. This can be easily done by utilizing a softmax activation function for g.

One of the advantages of the two-circuit implementation of SPLs is that the size of the circuit $q_{\boldsymbol{\Theta}}$ can be easily increased to improve the capacity of the model (Sec. 3.1.2.1). The single-circuit implementation is not as flexible, as normally the number of parameters is determined by the complexity of the constraint circuit, which depends entirely on the compilation step. In this case, one option is to *overparameterize* the neural conditional circuit by introducing additional sum units, hence allowing it to capture more modes in the distribution. We detail this process is Sec. B.1.3. A side effect of overparameterization is that it relaxes determinism, meaning that MAP inference needs to be approximated, as described in Sec. 3.1.2.3. Additionally, training a gating function to map relatively small embeddings to large parameter vectors in overparameterized circuits, can slow down training. In such cases, a two-circuit implementation of SPL is to be preferred.

3.1.3 Related works

In this section, we position SPLs against state-of-the-art approaches for enforcing constraints on neural network predictions. In-depth surveys on this topic can be found in [Dash et al., 2022] and [Giunchiglia et al., 2022].

Energy-based models. Deep energy-based models (EBMs) replace FILs with an unnormalized factor graph [Koller and Friedman, 2009] that captures higher-order label dependencies [LeCun et al., 2006] (D2) but at the cost of foregoing probabilistic semantics (D1) and efficiency (D6). EBMs are typically unconcerned with hard constraints (D3). Neural approaches for segmentation [Liu et al., 2015a] and parsing [Durrett and Klein, 2015; Zhang et al., 2020a,b] remedy to this by replacing the factor graph with a full-fledged intractable (discriminative) graphical model [Koller and Friedman, 2009]. To gain efficiency, one can restrict EBMs to simpler graphical models (e.g., chains, trees), compromising expressiveness (D2) and the ability to model non-trivial logical constraints (D3, D4).

Loss-based methods. A prominent strategy consists of penalizing the network for producing inconsistent predictions using an auxiliary loss [Dash et al., 2022; Giunchiglia et al., 2022]. While popular, loss-based methods, however *cannot* guarantee that the predictions will be consistent at test time. Common losses include translating logical constraints into a differentiable fuzzy logic [Diligenti et al., 2012, 2017a], as exemplified by DL2 [Fischer et al., 2019]. Although efficient (D6), this solution is not probabilistically sound (D1) and crucially *is not syntax-invariant*: different encodings of the same formula (e.g., conjunctive vs. disjunctive normal form) yield different losses [Giannini et al., 2018; Di Liello et al., 2020]. Closer to our SPL, the Semantic Loss (SL) [Xu et al., 2018a] avoids this issue by penalizing the the probability θ_i associated to the i-th label by the neural network via the loss term

$$\mathcal{L}_{\mathsf{SL}} \propto -\sum_{y \models \mathsf{K}} \prod_{\boldsymbol{y} \models Y_i} \theta_i \prod_{\boldsymbol{y} \not\models Y_i} (1-\theta_i) = -\sum_{\boldsymbol{y} \models \mathsf{K}} \prod_i p(Y_i \mid \boldsymbol{x}) = -\sum_{\boldsymbol{y}} \prod_i q(Y_i; \theta_i) \cdot c_{\mathsf{K}}(\boldsymbol{x}, \boldsymbol{y}).$$

When K is compiled into a constraint circuit c_{K} one retrieves $-\mathcal{Z}(\boldsymbol{x})$ for a two-circuit version of

SPL that is as expressive as FIL as it assumes independent labels via a conditional PC $\prod_i q(Y_i; \theta_i)$. The neuro-symbolic entropy (NESYENT) [Ahmed et al., 2022c] extends $\mathcal{L}_{\mathsf{SL}}$by an entropy term that improves (but still does not guarantee) consistency. It still makes the same independence assumptions over labels (D2).

Consistency layers. Approaches ensuring consistency by embedding the constraints into the predictive layer as in SPLs include MultiplexNet [Hoernle et al., 2022] and HMCCN [Giunchiglia and Lukasiewicz, 2020]. MultiplexNet is able to encode only constraints in disjunctive normal form, which is problematic for generality (D4) and efficiency (D6) as neuro-symbolic SOP tasks involve an intractably large number of clauses – e.g. our pathfinding experiments involves billions of clauses. HMCCN encodes label dependencies as fuzzy relaxation and is the current state-of-the-art model for HMLC [Giunchiglia and Lukasiewicz, 2020]. HMCCN and even its recent extension [Giunchiglia and Lukasiewicz, 2021] are restricted to only certain constraints that can be exactly encoded with fuzzy logic easily. SPLs instead can express constraints encoded as arbitrary propositional logical formulas (D4).

Other approaches. Other common approaches to neuro-symbolic SOP require to invoke a solver to either obtain the MAP state or to compute (often only approximately) the gradient of the loss [Deshwal et al., 2019; Pogančić et al., 2019; Niepert et al., 2021b]. SPLs have no such requirement. Some neuro-symbolic approaches [Sarker et al., 2021] constrain the outputs of neural networks within complex logical reasoning pipelines to solve tasks harder than neuro-symbolic SOP. For instance, DeepProblog [Manhaeve et al., 2018] uses Prolog's backward chaining algorithm for first order logical rules whose probabilistic weights are predicted by the network. In modern implementations of Problog, grounding a first order program and then compiling it into constraint circuits [Dries et al., 2015] produces a conditional circuit akin to those we use in SPLs, but in which (*i*) only input distributions are parameterized and (*ii*) increasing the parameter count is not considered a straightforward operation. Scallop [Huang et al., 2021] provides a more scalable approach to deepproblog by considering only the top-k proofs. We leave to future work how we could quickly compile only a specific query as DeepProblog/Scallop do, to deal with first-order

representations efficiently.

3.1.4 Experiments

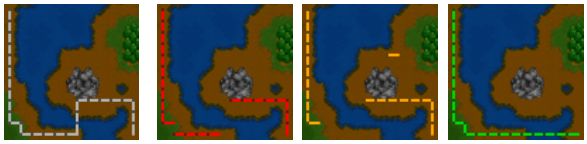
We evaluate SPLs on standard neuro-symbolic SOP benchmarks such as *simple path prediction*, *preference learning* [Xu et al., 2018a], *shortest path finding in Warcraft* [Pogančić et al., 2019] and *HMLC* [Giunchiglia and Lukasiewicz, 2020]. We compare SPLs against several state-of-the-art loss- and layer-based approaches (Sec. 3.1.3) by applying them to the same base neural network architecture as feature extractor f. As we are interested in measuring how close to the ground truth and how safe the predictions of all models are, we report the percentage of EXACT matches of the predicted labels, also called subset accuracy [Tsoumakas and Katakis, 2007], and the percentage of CONSISTENT predictions, also called "Constraint" [Xu et al., 2018a]. Note that, like other consistency layers, SPLs are guaranteed to always output 100% consistent predictions. Additionally, we report the HAMMING score [Tsoumakas and Katakis, 2007], mainly to maintain compatibility with previous experimental settings [Xu et al., 2018a; Ahmed et al., 2022c]. This metric does not consider consistency of predictions and naturally favors competitors that assume label independence and thus can minimize the per-label cross-entropy [Dembczyński et al., 2012] (Table 3.3). Sec. B.1.4 collects all experimental details such as architectures and hyperparameters used for each experiment.

In Sec. B.1.5 we provide the average timings for compiling logical formulas into circuitscarried out once, and reused in all subsequent experiments, for parameterizing the conditional circuits, computing the MAP-state of SPL and the loss function at training time (including the cost of computing the product circuit r and its normalization). All these timings, compilation excluded, are reported per batch. We compare to the timings of baselines such as semantic loss and neuro-symbolic entropy, where applicable, to which SPL is highly competitive.

Simple path prediction & preference learning. We start by comparing SPLs against loss-based approaches, reproducing the neuro-symbolic benchmarks of Xu et al. [2018a] for simple path

Table 3.3: SPLs outperform competitors in pathfinding in Warcraft. Predicted paths that do not exactly match the ground truth are still valid paths and yield very close costs to the ground truth. Competitors' predictions can have higher Hamming scores but be invalid. More examples in Sec. B.1.4.3.

ARCHITECTURE	EXACT	HAMMING	CONSISTENT
RESNET-18+FIL	55.0	**97.7**	56.9
RESNET-18+$\mathcal{L}_{\mathsf{SL}}$	59.4	**97.7**	61.2
RESNET-18+SPL (*ours*)	**78.2**	96.3	**100.0**



GROUND TRUTH FIL $\mathcal{L}_{\mathsf{SL}}$ SPL

prediction and preference learning. In the first experiment, given a source and destination node in an unweighted grid $G = (V, E)$, the neural net needs to find the shortest unweighted path connecting them. We consider a 4×4 grid. The input $(\boldsymbol{x}, \boldsymbol{y})$ is a binary vector of length $|V| + |E|$, with the first $|V|$ variables indicating the source and destination nodes, and the subsequent $|E|$ variables indicating a subgraph $G' \subseteq G$. Each label is a binary vector of length $|E|$ encoding the unique shortest path in G'. For each example, we obtain G' by dropping one third of the edges in the graph G uniformly at random, filter out the connected components with fewer than 5 nodes, to reduce degenerate cases, and then sample a source and destination node uniformly at random from G'. The dataset consists of 1600 such examples, with a $60/20/20$ train/validation/test split.

In the preference learning task, given a user's ranking over a subset of items, the network has to predict the user's ranking over the remaining items. We encode an ordering over n items as a binary matrix Y_{ij}, where for each $i, j \in 1, \ldots, n$, Y_{ij} indicates whether item i is the jth element in the ordering. The input $\boldsymbol{x}$ consist of the user's preference over 6 sushi types, and the model has to predict the users preferences (a strict total order) over the remaining 4. We use preference ranking data over 10 types of sushi for $5,000$ individuals, taken from [Mattei and Walsh, 2013b], and a $60/20/20$ split.

We employ a 5-layer and 3-layer MLP as a baseline for the simple path prediction, and preference learning, respectively, equipped with FIL layer and additionally with the Semantic Loss [Xu et al., 2018a] (MLP+$\mathcal{L}_{\mathsf{SL}}$) or its entropic extension [Ahmed et al., 2022c] (MLP+NESYENT). We compile the logical constraints into an SDD [Darwiche, 2011b] and then turn it into a the same

constraint circuit c_{K} that is used for $\mathcal{L}_{\mathsf{SL}}$, NESYENT (Sec. 3.1.3) and our 1-circuit implementation of SPLs. Table 3.2 clearly shows that the increased expressiveness of SPL, coming from overparameterizing c_{K}, allows to outperform all competitors while guaranteeing consistent predictions, as expected.

Warcraft Shortest Path. Next, we evaluate SPL on the more challenging task of predicting the minimum cost path in a weighted 12×12 grid imposed over terrain maps of Warcraft II [Pogančić et al., 2019]. Each vertex is assigned a cost corresponding to the type of the underlying terrain (e.g., earth has lower cost than water). The minimum cost path between the top left and the bottom right vertices of the grid is encoded as an indicator matrix, and serves as a label. As in [Pogančić et al., 2019] we use a ResNet18 [He et al., 2016b] with FIL optionally with $\mathcal{L}_{\mathsf{SL}}$ as a baseline. Given the largest size of the compiled constraint circuit c_{K} in this case 10^{10}, we use a two-circuit implementation of SPL. Results in Figure 3.1 and Table 3.3 are striking: not only SPL outperforms competitors by a large margin – approx. $+23\%$ over FIL and $+19\%$ over the SL – but also consistently delivers meaningful paths that are very close to the ground truth in terms of cost, even when they encode very different routes. See Sec. B.1.4.3 for a gallery of these examples. Concerning times, SPLs are able to compute the likelihood in a mere 14 seconds per batch even on a 10^{10} valid configuration space (Sec. B.1.5).

Hierarchical Multi-Label Classification. Lastly, we follow the experimental setup of Giunchiglia and Lukasiewicz [2020] and evaluate SPL on 12 real-world HMLC tasks spanning four different domains: 8 functional genomics, 2 medical images, 1 microalgea classification, and 1 text categorization. Figure 3.3 shows an example of a hierarchy of classes. These tasks are especially challenging due to the limited number of training samples, the large number of output classes, ranging from 56 to 499, as well as the sparsity of the output space. The larger datasets yield a label space of 2^{499} configurations, but we can compile them in seconds into compact constraints circuits of size ≈ 108KB (Sec. B.1.5).

For numeric features we replaced missing values by their mean, and for categorical features by

Table 3.4: Comparison between SPL and HMCNN [Giunchiglia and Lukasiewicz, 2020] on twelve HMLC datasets averaged over 10 runs. Best results for each dataset are in bold. Results which are not significantly worse than the competition, as determined using an unpaired Wilcoxon test, are marked in boldface. Consistency is always 100% for both approaches.

Dataset	Exact Match		Hamming Score	
	HMCNN	MLP+SPL	HMCNN	MLP+SPL
CellCycle	3.05 ± 0.11	$\mathbf{3.79 \pm 0.18}$	$\mathbf{98.26 \pm 0.00}$	97.84 ± 0.06
Derisi	1.39 ± 0.47	$\mathbf{2.28 \pm 0.23}$	$\mathbf{98.32 \pm 0.32}$	97.70 ± 0.07
Eisen	5.40 ± 0.15	$\mathbf{6.18 \pm 0.33}$	$\mathbf{98.09 \pm 0.01}$	97.30 ± 0.04
Expr	4.20 ± 0.21	$\mathbf{5.54 \pm 0.36}$	$\mathbf{98.29 \pm 0.01}$	97.87 ± 0.02
Gasch1	3.48 ± 0.96	$\mathbf{4.65 \pm 0.30}$	$\mathbf{98.37 \pm 0.31}$	97.59 ± 0.05
Gasch2	3.11 ± 0.08	$\mathbf{3.95 \pm 0.28}$	$\mathbf{98.27 \pm 0.00}$	97.94 ± 0.07
Seq	5.24 ± 0.27	$\mathbf{7.98 \pm 0.28}$	$\mathbf{98.31 \pm 0.01}$	97.66 ± 0.03
Spo	$\mathbf{1.97 \pm 0.06}$	$\mathbf{1.92 \pm 0.11}$	$\mathbf{98.23 \pm 0.00}$	98.17 ± 0.03
Diatoms	48.21 ± 0.57	$\mathbf{58.71 \pm 0.68}$	$\mathbf{99.75 \pm 0.00}$	99.64 ± 0.01
Enron	5.97 ± 0.56	$\mathbf{8.18 \pm 0.68}$	$\mathbf{94.10 \pm 0.04}$	93.19 ± 0.13
Imclef07a	79.75 ± 0.38	$\mathbf{86.08 \pm 0.45}$	$\mathbf{99.40 \pm 0.01}$	99.35 ± 0.03
Imclef07d	76.47 ± 0.35	$\mathbf{81.06 \pm 0.68}$	$\mathbf{98.06 \pm 0.02}$	$\mathbf{98.07 \pm 0.08}$

a vector of zeros, and standardized all features. We used the validation splits to determine the number of layers in the gating function as well as the overparameterization, keeping all other hyperparameters fixed. The final models were obtained by training using a batch size of 128 and early stopping on the validation set. We compare our single-circuit SPL against HMCNN which was shown to outperform several other state-of-the-art HMLC approaches in Giunchiglia and Lukasiewicz [2020]. We study the effect of increasing the expressivenss of SPL via overparameterization in Sec. B.1.4.4. The results in Table 3.4 highlight that SPL significantly outperforms HMCNN in terms of exact match on 11 data sets performing comparably on 1,

3.2 SIMPLE: A Gradient Estimator for k-Subset Sampling

k-subset sampling, sampling a subset of size k of n variables, is omnipresent in machine learning. It lies at the core of many fundamental problems that rely upon learning sparse features representations of input data, including stochastic high-dimensional data visualization [van der Maaten, 2009], parametric k-nearest neighbors [Grover et al., 2018], learning to explain [Chen et al., 2018], discrete variational auto-encoders [Rolfe, 2017], and sparse regression, to name a few. All such tasks involve optimizing an expectation of an objective function with respect to a latent *discrete* distribution parameterized by a neural network, which are often *assumed* intractable. Score-function estimators offer a cloyingly simple solution: rewrite the gradient of the expectation as an expectation of the gradient, which can subsequently be estimated using a finite number of samples offering an unbiased estimate of the gradient. Simple as it is, score-function estimators suffer from very high variance which can interfere with training. This provided the impetus for other, low-variance, gradient estimators, chief among them are those based on the reparameterization trick, which allows for biased, but low-variance gradient estimates. The reparameterization trick, however, does not allow for a direct application to discrete distributions thereby prompting continuous relaxations, e.g., Gumbel-softmax [Jang et al., 2017; Maddison et al., 2017], that allow for reparameterized gradients w.r.t the parameters of a *categorical* distribution. Reparameterizable subset sampling [Xie and Ermon, 2019] generalizes the Gumbel-softmax trick to k-subsets which while rendering k-subset sampling amenable to backpropagation at the cost of introducing bias in the learning by using relaxed samples.

In this work, we set out with the goal of avoiding all such relaxations. Instead, we fall back to *discrete* sampling on the forward pass. On the backward pass, we reparameterize the gradient of the loss function with respect to the samples as a function of the *exact* marginals of the k-subset distribution. Computing the exact conditional marginals is, in general, intractable [Roth, 1996]. We give an *efficient* algorithm for computing the k-subset probability, and show that the conditional marginals correspond to partial derivatives, and are therefore tractable for the k-subset distribution.

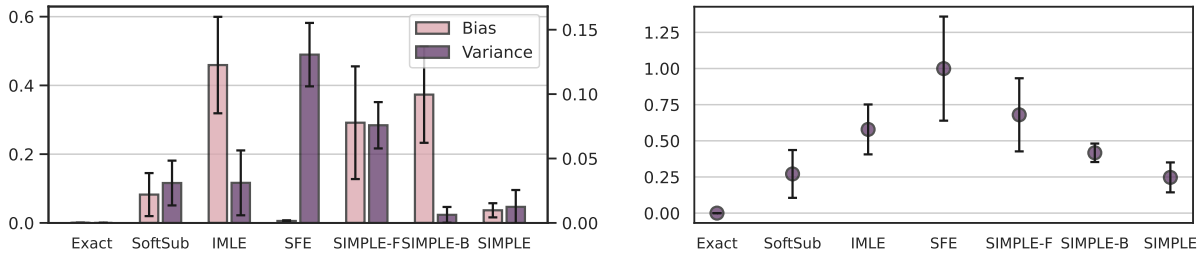


Figure 3.4: A comparison of the bias and variance of the gradient estimators (left) and the average and standard deviation of the cosine distance of a single-sample gradient estimate to the exact gradient. We used the cosine distance, defined as ($1-$ cosine similarity), in place of the euclidean distance as we only care about the direction of the gradient, not magnitude. The bias, variance and error were estimated using a sample of size 10,000. The details of this experiment are provided in Sec. 3.2.4.1.

We show that our proposed gradient estimator for the k-subset distribution, coined SIMPLE, is reminiscent of the straight-through (ST) Gumbel estimator when $k = 1$, with the gradients taken with respect to the unperturbed marginals. We empirically demonstrate that SIMPLE exhibits lower bias *and* variance compared to other known gradient estimators, including the ST Gumbel estimator in the case $k = 1$.

We include an experiment on the task of learning to explain (L2X) using the BEERADVOCATE dataset [McAuley et al., 2012], where the goal is to select the subset of words that best explains the model's classification of a user's review. We also include an experiment on the task of stochastic sparse linear regression, where the goal is to learn the best sparse model, and show that we are able to recover the KuramotoSivashinsky equation. Finally, we develop an efficient computation for the exact variational evidence lower bound (ELBO) for the k-subset distribution, which when used in conjunction with SIMPLE leads to state-of-the-art discrete sparse VAE learning.

3.2.1 Problem Statement and Motivation

We consider models described by the equations

$$\boldsymbol{\theta} = h_{\boldsymbol{v}}(\boldsymbol{x}), \qquad \boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \textstyle\sum_i z_i = k), \qquad \hat{\boldsymbol{y}} = f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x}), \tag{3.3}$$

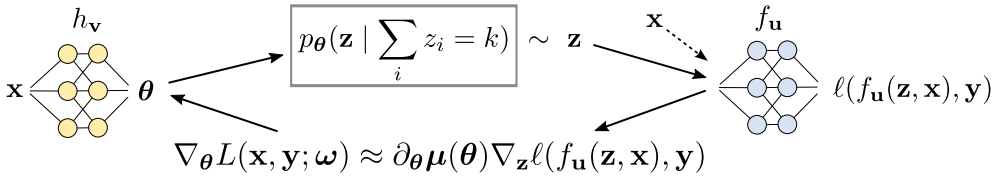


Figure 3.5: The problem setting considered in our paper. On the forward pass, a neural network h_v outputs $\boldsymbol{\theta}$ parameterizing a *discrete* distribution over subsets of size k of n items, i.e., the k-subset distribution. We sample exactly, and efficiently, from this distribution, and feed the samples to a downstream neural network. On the backward pass, we approximate the true gradient by the product of the derivative of marginals and the gradient of the sample-wise loss.

where $\boldsymbol{x} \in \mathbf{X}$ and $\hat{\boldsymbol{y}} \in \mathbf{Y}$ denote feature inputs and target outputs, respectively, $h_{\boldsymbol{v}} : \mathbf{X} \to \Theta$ and $f_{\boldsymbol{u}} : \mathbf{Z} \times \mathbf{X} \to \mathbf{Y}$ are smooth, parameterized maps and $\boldsymbol{\theta}$ are logits inducing a distribution over the latent binary vector $\boldsymbol{z}$. The induced distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ is defined as

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}) = \prod_{i=1}^{n} p_{\theta_i}(z_i), \text{ with } p_{\theta_i}(z_i = 1) = \text{sigmoid}(\theta_i) \text{ and } p_{\theta_i}(z_i = 0) = 1 - \text{sigmoid}(\theta_i). \quad (3.4)$$

The goal of our stochastic latent layer is *not* to simply sample from $p_{\boldsymbol{\theta}}(\boldsymbol{z})$, which would yield samples with a Hamming weight between 0 and n (i.e., with an arbitrary number of ones). Instead, we are interested in sampling from the distribution restricted to samples with a Hamming weight of k, for any given k. That is, we are interested in sampling from the conditional distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$.

Conditioning the distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ on this k-*subset constraint* introduces intricate dependencies between each of the z_i's. The probability of sampling any given k-subset vector $\boldsymbol{z}$, therefore, becomes

$$p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \textstyle\sum_i z_i = k) = p_{\boldsymbol{\theta}}(\boldsymbol{z}) / p_{\boldsymbol{\theta}}(\textstyle\sum_i z_i = k) \cdot [\![\textstyle\sum_i z_i = k]\!]$$

where $[\![\cdot]\!]$ denotes the indicator function. In other words, the probability of sampling each k-subset is re-normalized by $p_{\boldsymbol{\theta}}\left(\sum_i z_i = k\right)$ – the probability of sampling exactly k items from the *unconstrained* distribution induced by encoder $h_{\boldsymbol{v}}$. The quantity $p_{\boldsymbol{\theta}}(\sum_i z_i = k) = \sum_{\mathbf{z}} p_\theta(\mathbf{z}) \cdot [\![\sum_i z_i = k]\!]$ appears to be intractable. We show that not to be the case, providing a tractable

TASK	MAP $h_{\boldsymbol{v}}$	MAP $f_{\boldsymbol{u}}$	LOSS ℓ
Discrete VAE (Sec. 3.2.4.2)	Encoder	Decoder	ELBO
Learn To Explain (Sec. 3.2.4.3)	Embedding	Regression	RMSE
Sparse Regression (Sec. 3.2.4.4)	Identity	Linear Regression	RMSE

Table 3.5: Architectures of the three experiment settings.

algorithm for computing it.

Given a set of samples $\mathcal{D}$, we are concerned with learning the parameters $\boldsymbol{\theta} = (\boldsymbol{v}, \boldsymbol{u})$ of the architecture in (3.3) through minimizing the training error L, which is the expected loss:

$$L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \mathbf{E}_{\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} | \sum_i z_i = k)}[\ell(f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x}), \boldsymbol{y})] \qquad \textit{with } \boldsymbol{\theta} = h_{\boldsymbol{v}}(\boldsymbol{x}), \tag{3.5}$$

where $\ell : \mathbf{Y} \times \mathbf{Y} \to \mathbb{R}^+$ is a point-wise loss function. This formulation, illustrated in Figure 3.5, is general and subsumes many settings. Different choices of mappings $h_{\boldsymbol{v}}$ and $f_{\boldsymbol{u}}$, and sample-wise loss ℓ define various tasks. Table 3.5 presents some example settings used in our experimental evaluation. Learning then requires computing the gradient of L w.r.t. $\boldsymbol{\theta} = (\boldsymbol{v}, \boldsymbol{u})$. The gradient of L w.r.t. $\boldsymbol{u}$ is

$$\nabla_{\boldsymbol{u}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \mathbf{E}_{\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} | \sum_i z_i = k)}[\partial_{\boldsymbol{u}} f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x})^\top \nabla_{\hat{\boldsymbol{y}}} \ell(\hat{\boldsymbol{y}}, \boldsymbol{y})], \tag{3.6}$$

where $\hat{y} = f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x})$ is the decoding of a latent sample $\boldsymbol{z}$. Furthermore, the gradient of L w.r.t. $\boldsymbol{v}$ is

$$\nabla_{\boldsymbol{v}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \partial_{\boldsymbol{v}} h_{\boldsymbol{v}}(\boldsymbol{x})^\top \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}), \tag{3.7}$$

where $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) := \nabla_{\boldsymbol{\theta}} \mathbf{E}_{\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} | \sum_i z_i = k)}[\ell(f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x}), \hat{\boldsymbol{y}})]$, the loss' gradient w.r.t. the encoder.

One challenge lies in computing the expectation in (3.5) and (3.6), which has no known closed-form solution. This necessitates a Monte-Carlo estimate through sampling from $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$.

A second, and perhaps more substantial hurdle lies in computing $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ in (3.7) due to

the non-differentiable nature of discrete sampling. One could rewrite $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ as

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \mathbf{E}_{\boldsymbol{z} \sim p_\theta\left(\mathbf{z} \mid \sum_i z_i = k\right)}[\ell(f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x}), \boldsymbol{y}) \nabla_{\boldsymbol{\theta}} \log p_\theta\left(\mathbf{z} \mid \textstyle\sum_i z_i = k\right)]$$

which is known as the REINFORCE estimator [Williams, 1992], or the score function estimator (SFE). It is typically avoided due to its notoriously high variance, despite its apparent simplicity. Instead, typical approaches [Xie and Ermon, 2019; Plötz and Roth, 2018] reparameterize the samples as a deterministic transformation of the parameters, and some independent standard Gumbel noise, and relaxing the deterministic transformation, the top-k function in this case, to allow for backpropagation.

3.2.2 SIMPLE: Subset Implicit Likelihood Estimation

Our goal is to build a gradient estimator for $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$. We start by envisioning a hypothetical sampling-free architecture, where the downstream neural network $f_{\boldsymbol{u}}$ is a function of the marginals, $\boldsymbol{\mu} \coloneqq \mu(\boldsymbol{\theta}) \coloneqq \{p_{\boldsymbol{\theta}}(z_j \mid \sum_i z_i = k)\}_{j=1}^n$, instead of a discrete sample $\boldsymbol{z}$, resulting in a loss L_m s.t.

$$\nabla_{\boldsymbol{\theta}} L_m(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta})^\top \nabla_{\boldsymbol{\mu}} \ell_m(f_{\boldsymbol{u}}(\boldsymbol{\mu}, \boldsymbol{x}), \boldsymbol{y}). \tag{3.8}$$

When the marginals $\mu(\boldsymbol{\theta})$ can be efficiently computed and differentiated, such a hypothetical pipeline can be trained end-to-end. Domke [2010] observed that, for an arbitrary loss function ℓ_m defined on the marginals, the Jacobian of the marginals w.r.t. the logits is symmetric, i.e.

$$\nabla_{\boldsymbol{\theta}} L_m(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta})^\top \nabla_{\boldsymbol{\mu}} \ell_m(f_{\boldsymbol{u}}(\boldsymbol{\mu}, \boldsymbol{x}), \boldsymbol{y}) = \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta}) \nabla_{\boldsymbol{\mu}} \ell_m(f_{\boldsymbol{u}}(\boldsymbol{\mu}, \boldsymbol{x}), \boldsymbol{y}). \tag{3.9}$$

Consequently, computing the gradient of the loss w.r.t. the logits, $\nabla_{\boldsymbol{\theta}} L_m(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$, reduces to computing the *directional derivative*, or the Jacobian-vector product, of the marginals w.r.t. the logits in the direction of the gradient of the loss. This offers an alluring opportunity: the conditional marginals characterize the probability of each z_i in the sample, and could be thought of as a differ-

entiable proxy for the samples. Specifically, by reparameterizing $\boldsymbol{z}$ as a function of the conditional marginal $\boldsymbol{\mu}$ under approximation $\partial_{\boldsymbol{\mu}} \boldsymbol{z} \approx \mathbf{I}$ as proposed by Niepert et al. [2021b], and using the straight-through estimator for the gradient of the sample w.r.t. the marginals on the backward pass, we approximate our true $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta})$ as

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega}) \approx \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta}) \nabla_{\boldsymbol{z}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega}), \tag{3.10}$$

where the directional derivative of the marginals can be taken along *any downstream gradient*, rendering the whole pipeline end-to-end learnable, even in the presence of non-differentiable sampling.

Now, estimating the gradient of the loss w.r.t. the parameters can be thought of as decomposing into two sub-problems: **(P1)** Computing the derivatives of conditional marginals $\partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta})$, which requires the computation of the conditional marginals, and **(P2)** Computing the gradient of the loss w.r.t. the samples $\nabla_{\boldsymbol{z}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega})$ using sample-wise loss, which requires drawing exact samples. These two problems are complicated by conditioning on the k-subset constraint, which introduces intricate dependencies to the distribution, and is infeasible to solve naively, e.g. by enumeration. We will show simple, efficient, and exact solutions to each problem, at the heart of which is the insight that we need not care about the variables' order, only their sum, introducing symmetries that simplify the problem.

3.2.2.1 Derivatives of Conditional Marginals

In many probabilistic models, marginal inference is #P-hard [Roth, 1996; Zeng et al., 2020b]. However, we observe that it is not the case for the k-subset distribution. We notice that the conditional marginals correspond to the partial derivatives of the log-probability of the k-subset constraint. To see this, note that the derivative of a multi-linear function with respect to a single variable retains all the terms referencing that variable, and drops all other terms; this corresponds exactly to the unnormalized conditional marginals. By taking the derivative of the log-probability, this introduces

Algorithm 5 `PrExactlyk`$(\boldsymbol{\theta}, n, k)$

Input: The logits $\boldsymbol{\theta}$ of the distribution, the number of variables n, and the subset size k
Output: $p_{\boldsymbol{\theta}}(\sum_i z_i = k)$

```
// a[i, j] = p_θ(Σ_{m=1}^{i} z_m = j) for all i, j
initialize a to be 0 everywhere
a[0, 0] = 1 // p_θ(Σ_{m=1}^{0} z_m = 0) = 1
for i = 1 to n do
    for j = 0 to k do
        // cf. constructive proof of Prop. 3.1
        a[i, j] = a[i − 1, j] · p_{θ_i}(z_i = 0)
                  + a[i − 1, j − 1] · p_{θ_i}(z_i = 1)
return a[n, k]
```

Algorithm 6 `Sample`$(\boldsymbol{\theta}, n, k)$

Input: The logits $\boldsymbol{\theta}$ of the distribution, the number of variables n, and the subset size k
Output: $\boldsymbol{z} = (z_1, \ldots, z_n) \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$

```
sample = [ ], j = k
for i = n to 1 do
    // cf. proof of Prop. 3.2
    p = a[i − 1, j − 1]
    z_i ~ Bernoulli(p · p_{θ_i}(z_i = 1)/a[i, j])
    // Pick next state based on value of sample
    if z_i = 1 then j = j − 1
    sample.append(z_i)
return sample
```

the k-subset probability in the denominator, leading to the *conditional* marginals. Intuitively, the rate of change of the k-subset probability w.r.t. a variable only depends on that variable through its length-k subsets.

Theorem 2. *Let $p_{\boldsymbol{\theta}}(\sum_j z_j = k)$ be the probability of exactly-k of the unconstrained distribution parameterized by logits $\boldsymbol{\theta}$. Let $\alpha_i \coloneqq \log p_{\boldsymbol{\theta}}(z_i)$ denote the log marginals. For every variable Z_i, its conditional marginal is*

$$p_{\boldsymbol{\theta}}\Big(z_i \mid \textstyle\sum_j z_j = k\Big) = \dfrac{\partial}{\partial \alpha_i} \log p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k). \tag{3.11}$$

We refer the reader to the appendix for a detailed proof of the above theorem. To establish the tractability of the above computation of the conditional marginals, we need to show that the probability of the exactly-k constraint $p_{\boldsymbol{\theta}}(\sum_i z_i = k)$ can be obtained tractably, which we demonstrate next.

Proposition 3.1. *The probability $p_{\boldsymbol{\theta}}\left(\sum_i z_i = k\right)$ of sampling exactly k items from the* unconstrained *distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ over n items as in Equation 3.4 can be computed exactly in time $\mathcal{O}(nk)$.*

Proof. Our proof is constructive. As a base case, consider the probability of sampling $k = -1$ out of $n = 0$ items. We can see that the probability of such an event is 0. As a second base case,

consider the probability of sampling $k = 0$ out of $n = 0$ items. We can see that the probably of such an event is 1. Now assume that we are given the probability $p_{\boldsymbol{\theta}}\left(\sum_i^{n-1} z_i = k'\right)$, for $k' = 0, \ldots, k$, and we are interested in computing $p_{\boldsymbol{\theta}}\left(\sum_i^n z_i = k\right)$. By the partition theorem, we can see that

$$p_{\boldsymbol{\theta}}\left(\sum_i^n z_i = k\right) = p_{\boldsymbol{\theta}}\left(\sum_i^{n-1} z_i = k\right) \cdot p_{\theta_n}(z_n = 0) + p_{\boldsymbol{\theta}}\left(\sum_i^{n-1} z_i = k-1\right) \cdot p_{\theta_n}(z_n = 1)$$

as events $\sum_i^{n-1} z_i = k$ and $\sum_i^{n-1} z_i = k - 1$ are disjoint and, for any k, partition the sample space. Intuitively, for any k and n, we can sample k out of n items by choosing k of $n-1$ items, and not the n-th item, or choosing $k-1$ of $n-1$ items, and the n-th item. The above process gives rise to Algorithm 5, which returns $p_{\boldsymbol{\theta}}\left(\sum_i z_i = k\right)$ in time $\mathcal{O}(nk)$. □

By the construction described above, we obtain a closed-form $p_{\boldsymbol{\theta}}(\sum_i^n z_i = k)$, which allows us to compute conditional marginals $p_{\boldsymbol{\theta}}(z_i \mid \sum_j z_j = k)$ by Theorem 2 via auto-differentiation. This further allows the computation of the derivatives of conditional marginals $\partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta})_i = \partial_{\boldsymbol{\theta}}\ p_{\boldsymbol{\theta}}(z_i \mid \sum_j z_j = k)$ to be amenable to auto-differentiation, solving problem **(P1)** exactly and efficiently.

3.2.2.2 Gradients of Loss w.r.t. Samples

As alluded to in Sec. 3.2.2, we approximate $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega})$ by the directional derivative of the marginals along the gradient of the loss w.r.t. discrete samples $\boldsymbol{z}$, $\nabla_{\boldsymbol{z}} L(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\omega})$, where $\boldsymbol{z}$ is drawn from the k-subset distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$. What remains is to estimate the value of the loss, necessitating faithful sampling from the k-subset distribution, which might initially appear daunting.

Exact k-subset Sampling Next we show how to sample exactly from the k-subset distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$. We start by sampling the variables in reverse order, that is, we sample z_n through z_1. The main intuition being that, having sampled $(z_n, z_{n-1}, \cdots, z_{i+1})$ with a Hamming weight of $k - j$, we sample Z_i with a probability of choosing $k - j$ of $n - 1$ variables *and* the n-th variable *given that* we choose $k - j + 1$ of n variables. We formalize our intuition below.

Proposition 3.2. *Let* `Sample` *be defined as in Algorithm 6. Given n random variables $Z_1, \cdots, Z_n$, a subset size k, and a k-subset distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$ parameterized by log probabilities $\boldsymbol{\theta}$, Algorithm 6 draws exact samples from $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$ in time $\mathcal{O}(n)$.*

Proof. Assume that variables $Z_n, \cdots, Z_{i+1}$ are sampled and have their values to be $z_n, \cdots, z_{i+1}$ with $\sum_{m=i+1}^{n} z_m = k - j$. By Algorithm 6 we have that the probability with which to sample Z_i is

$$
\begin{aligned}
p_{\texttt{Sample}}(z_i = 1 \mid z_n, \cdots, z_{i+1}) &= \frac{p_{\boldsymbol{\theta}}(\sum_{m=i}^{n} z_m = k - j + 1 \mid \sum_m z_m = k)\, p_{\theta_i}(z_i = 1)}{p_{\boldsymbol{\theta}}(\sum_{m=i+1}^{n} z_m = k - j \mid \sum_m z_m = k)} \\
&= \frac{p_{\boldsymbol{\theta}}(\sum_{m=i+1}^{n} z_m = k - j \mid z_i = 1, \sum_m z_m = k)\, p_{\theta_i}(z_i = 1)}{p_{\boldsymbol{\theta}}(\sum_{m=i+1}^{n} z_m = k - j \mid \sum_m z_m = k)} \\
&= p_{\boldsymbol{\theta}}(z_i = 1 \mid \textstyle\sum_{m=i+1}^{n} z_m = k - j, \sum_m z_m = k) \;\; \textit{(by Bayes' theorem)}
\end{aligned}
$$

It follows that samples drawn from Algorithm 6 are distributed according to $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$. □

3.2.3 Connection to Straight-Through Gumbel-Softmax

One might wonder if our gradient estimator reduces to the Straight-Through (ST) Gumbel-Softmax estimator, or relates to it in any way when $k = 1$. On the forward pass, the ST Gumbel Softmax estimator makes use of the Gumbel-Max trick [Maddison et al., 2014], which states that we can efficiently sample from a categorical distribution by perturbing each of the logits with standard Gumbel noise, and taking the MAP, or more formally $\boldsymbol{z} = \mathrm{OneHot}(\arg\max_{i \in \{1,\ldots,k\}} \boldsymbol{\theta}_i + g_i) \sim p_{\boldsymbol{\theta}}$ where the g_i's are i.i.d Gumbel$(0, 1)$ samples, and OneHot encodes the sample as a binary vector.

Since $\arg\max$ is non-differentiable, Gumbel-Softmax uses the *perturbed* relaxed samples, $\boldsymbol{y} = \mathrm{Softmax}(\boldsymbol{\theta} + g_i)$ as a proxy for discrete samples $\boldsymbol{z}$ on the backward pass, using differentiable $\mathrm{Softmax}$ in place of the non-differentiable $\arg\max$, with the entire function returning $(\boldsymbol{z} - \boldsymbol{y}).\mathrm{detach}() + \boldsymbol{y}$ where detach ensures that the gradient flows only through the relaxed samples on the backward pass.

Algorithm 7 The proposed algorithm for the k-subset distribution

function FORWARDPASS($\boldsymbol{\theta}$)
  // $p_{\boldsymbol{\theta}}(\sum_{m=1}^{i} z_m = j)$ *for all* i, j
  $a = \texttt{PrExactlyk}(\boldsymbol{\theta}, n, k)$
  // *Sample from* $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$
  $\boldsymbol{z} = \texttt{Sample}(\boldsymbol{\theta}, n, k)$
  save a for the backward pass
  return $\boldsymbol{z}$

function BACKWARDPASS($\nabla_{\boldsymbol{z}} \ell(f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x}), \boldsymbol{y})$)
  load $\boldsymbol{\theta}$ from the forward pass
  // *derivatives of* $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$
  $\boldsymbol{\mu} = \nabla_{\boldsymbol{\theta}} \log a[n, k]$ // by auto-diff
  // *Return the directional derivative of the*
  // *marginals along the downstream gradients*
  return $\mathrm{JVP}(\boldsymbol{\mu}, \nabla_{\boldsymbol{z}} \ell(f_{\boldsymbol{u}}(\boldsymbol{z}, \boldsymbol{x})))$

That is, just like SIMPLE, ST Gumbel-Softmax returns *exact*, *discrete* samples. However, whereas SIMPLE backpropagates through the exact marginals, ST Gumbel Softmax backpropagates through the *perturbed* marginals that result from applying the Gumbel-max trick. As can be seen in Figure 3.6, such a minor difference means that, empirically, SIMPLE exhibits lower bias and variance compared to ST Gumbel Softmax while being exactly as efficient.

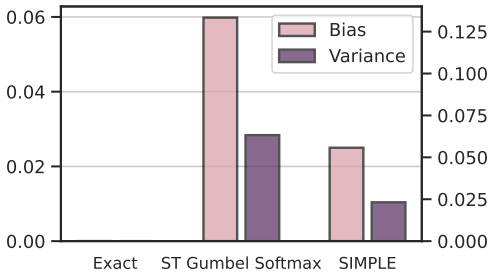


Figure 3.6: Bias and variance of SIMPLE and Gumbel Softmax over 10k samples

3.2.4 Experiments

We conduct experiments on four different tasks: 1) A synthetic experiment designed to test the bias and variance, as well as the average deviation of SIMPLE compared to a variety of well-established estimators in the literature. 2) A discrete k-subset Variational Auto-Encoder (DVAE) setting, where the latent space models a probability distribution over k-subsets. We will show that we can compute the evidence lower bound (ELBO) exactly, and that, coupled with exact sampling and our SIMPLE gradient estimator, we attain a much lower loss compared to state of the art in sparse DVAEs. 3) The learning to explain (L2X) setting, where the aim is to select the k-subset of words that best describe the classifier's prediction, where we show an improved mean-squared error, as well as precision, across the board. 4) A novel, yet simple task, sparse linear

regression, where, in a vein similar to L2X, we wish to select a k-subset of features that give rise to a linear regression model, avoiding overfitting the spurious features present in the data. Table 3.5 details the architecture with the objective functions. Our code will be made publicly available at github.com/UCLA-StarAI/SIMPLE.

3.2.4.1 Synthetic Experiments

We carried out a series of experiments with a 5-subset distribution, and a latent space of dimension 10. We set the loss to $L(\boldsymbol{\theta}) = \mathbf{E}_{z\sim p_{\boldsymbol{\theta}}\left(\mathbf{z}|\sum_i z_i=k\right)}[\|\boldsymbol{z}-\mathbf{b}\|^2]$, where $\mathbf{b}$ is the groundtruth logits sampled from $\mathcal{N}(0,\mathbf{I})$. Such a distribution is tractable: we only have $\binom{10}{5} = 252$ k-subsets, which are easily enumerable and therefore, the exact gradient, the golden standard, can be computed in closed form.

In this experiment, we are interested in three metrics: bias, variance, and the average error of each gradient estimator, where the latter is measured by averaging the deviation of each single-sample gradient estimate from the exact gradient. We used the cosine distance, defined as $1-$ cosine similarity as the measure of deviation in our calculation of the metrics above, as we only care about direction.

We compare against four different baselines: *exact*, which denotes the exact gradient; *Soft-Sub* [Xie and Ermon, 2019], which uses an extension of the Gumbel-Softmax trick to sample *relaxed* k-subsets on the forward pass; I-MLE, which denotes the IMLE gradient estimator [Niepert et al., 2021b], where *approximate* samples are obtained using perturb-and-map (PAM) on the forward pass, approximating the marginals using PAM samples on the backward pass; and score function estimator, denoted *SFE*.

We tease apart SIMPLE's improvements by comparing three different flavors: SIMPLE-F, which only uses *exact* sampling, falling back to estimating the marginals using *exact* samples; SIMPLE-B, which uses *exact* marginals on the backward pass with *approximate* PAM samples on the forward pass; and SIMPLE, coupling *exact* samples on the forward pass with *exact* marginals on the backward pass.

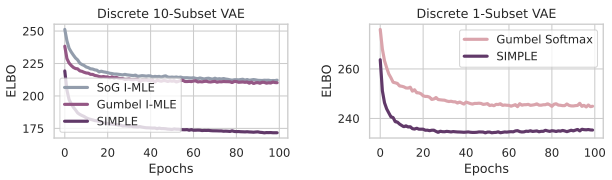


Figure 3.7: ELBO against # of epochs. (Left) Comparison of SIMPLE against variants of IMLE on the 10-subset DVAE, and (Right) against ST Gumbel Softmax on the 1-subset DVAE.

Algorithm 8 $\texttt{Entropy}(\boldsymbol{\theta}, n, k)$

Input: The logits $\boldsymbol{\theta}$ of the distribution, the number of variables n, and the subset size k
Output: $H(\boldsymbol{z}) = -\mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{\theta}}(\boldsymbol{z}|\sum_i z_i=k)}[\log p(\boldsymbol{z})]$

```
h = zeros(n, k)
for i = k to n do
    for j = 0 to k do
        // p(z_i | Σ_{m=1}^{i} z_m = j)
        p = a[i − 1, j − 1] ∗ p_{θ_i}(z_i = 1)/a[i, j]
        // cf. proof of Prop. 3.3 in Appendix
        h[i, j] = H_b(p) + p ∗ h[i − 1, j]+
                  (1 − p) ∗ h[i − 1, j + 1]
return h
```

Our results are shown in Figure 3.4 As expected, we observe that *SFE* exhibits no bias, but high variance whereas *SoftSub* suffers from both bias and variance, due to the Gumbel noise injection into the samples to make them differentiable. We observe that I-MLE exhibits very high bias, as well as very low variance. This can be attributed to the PAM sampling, which in the case of k-subset distribution does not sample faithfully from the distribution, but is instead biased to sampling only the mode of the distribution. This also means that, by approximating the marginals using PAM samples, there is a lot less variance to our gradients. On to our SIMPLE gradient estimator, we see that it exhibits *less bias as well as less variance* compared to all the other gradient estimators. We also see that each estimated gradient is, on average, much more aligned with the exact gradient. To understand why that is, we compare SIMPLE, SIMPLE-F, and SIMPLE-B. As hypothesized, we observe that *exact sampling*, SIMPLE-F, reduces the bias, but increases the variance compared to I-MLE, this is since, unlike the PAM samples, our exact sample span the entire sample space.We also observe that, even compared to I-MLE, SIMPLE-B, reduces the variance by marginalizing over all possible samples.

3.2.4.2 Discrete Variational Auto-Encoder

Next, we test our SIMPLE gradient estimator in the k-subset discrete variational auto-encoder (DVAE) setting, where the latent variables model a probability distribution over k-subsets, and has a dimensionality of 20. Similar to prior work [Jang et al., 2017; Niepert et al., 2021b], the

encoding and decoding functions of the VAE consist of three dense layers (encoding: 512-256-20x20; decoding: 256-512-784). The DVAE is trained to minimize the sum of reconstruction loss and KL-divergence of the k-subset distribution and the constrained uniform distribution, known as the ELBO, on MNIST.

In prior work, the KL-divergence was approximated using the unconditional marginals, obtained simply through a $\mathrm{Softmax}$ layer. Instead we show that the KL-divergence between the k-subset distribution and the uniform distribution can be computed exactly. First note that, through simple algebraic manipulations, the KL-divergence between the k-subset distribution and the constrained uniform distribution can be rewritten as the sum of negative entropy, $-H(\boldsymbol{z})$, where $\boldsymbol{z} \sim p_{\boldsymbol{\theta}}\left(\boldsymbol{z} \mid \sum_i z_i = k\right)$ and log the number of k-subsets, $\log \binom{n}{k}$ (see appendix for details), reducing the hardness of computing the KL-divergence, to computing the entropy of a k-subset distribution, for which Algorithm 8 gives a tractable algorithm. Intuitively, the uncertainty in the distribution over a sequence of length n, k of which are true, decomposes as the uncertainty over Z_n, and the average of the uncertainties over the remainder of the sequence. We refer the reader to the appendix for the proof of the below proposition.

Proposition 3.3. *Let* `Entropy` *be defined as in Algorithm 8. Given variables,* $Z_1, \cdots, Z_n$*, and a* k*-subset distribution* $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$*, Algorithm 8 computes entropy of* $p_{\boldsymbol{\theta}}\left(\boldsymbol{z} \mid \sum_i z_i = k\right)$.

We plot the loss ELBO against the number of epochs, as seen in Figure 3.7. We compared against I-MLE using sum-of-gamma noise as well as Gumbel noise for PAM sampling, on the 10-subset DVAE, and against ST Gumbel Softmax on the 1-subset DVAE. We observe a *significantly* lower loss on the test set on the 10-subset DVAE, partly attributable to the exact ELBO computation, but also on the 1-subset DVAE compared to ST Gumbel Softmax, where the sole difference is the backward pass.

Method	Appearance		Palate		Taste	
	Test MSE	**Precision**	**Test MSE**	**Precision**	**Test MSE**	**Precision**
SIMPLE (Ours)	**2.35 ± 0.28**	**66.81 ± 7.56**	**2.68 ± 0.06**	**44.78 ± 2.75**	**2.11 ± 0.02**	**42.31 ± 0.61**
L2X (t = 0.1)	10.70 ± 4.82	30.02 ± 15.82	6.70 ± 0.63	**50.39 ± 13.58**	6.92 ± 1.61	32.23 ± 4.92
SoftSub (t = 0.5)	**2.48 ± 0.10**	52.86 ± 7.08	2.94 ± 0.08	39.17 ± 3.17	2.18 ± 0.10	**41.98 ± 1.42**
I-MLE (τ = 30)	**2.51 ± 0.05**	**65.47 ± 4.95**	2.96 ± 0.04	40.73 ± 3.15	2.38 ± 0.04	**41.38 ± 1.55**

Table 3.6: Results for three aspects with $k = 10$: test MSE and subset precision, both $\times 100$

3.2.4.3 Learning to Explain

The BEERADVOCATE dataset [McAuley et al., 2012] consists of free-text reviews and ratings for 4 different aspects of beer: appearance, aroma, palate, and taste. The training set has 80k reviews for the aspect APPEARANCE and 70k reviews for all other aspects. In addition to the ratings for all reviews, each sentence in the test set contains annotations of the words that best describe the review score with respect to the various aspects. We address the problem introduced by the L2X paper [Chen et al., 2018] of learning a k-subset distribution over words that best explain a given rating. We follow the architecture suggested in the L2X paper, consisting of four convolutional and one dense layer.

We compare to relaxation-based baselines L2X [Chen et al., 2018] and SoftSub [Xie and Ermon, 2019] as well as to I-MLE which uses perturb-and-MAP to both compute an approximate sample in the forward pass and to estimate the marginals. Prior work has shown that the straight-through estimator (STE) did not work well and we omit it here. We used the standard hyperparameter settings of Chen et al. [2018] and choose the temperature parameter $t \in \{0.1, 0.5, 1.0, 2.0\}$ for all methods. We used the standard Adam settings and trained separate models for each aspect using MSE as point-wise loss ℓ. Table 3.7 lists results for $k \in \{5, 10, 15\}$ for the AROMA aspect. The mean-squared error (MSE) of SIMPLE is almost always lower and its subset precision never significantly exceeded by those of the baselines. Table 3.6 shows results on the remaining aspects Appearance, Palate, and Taste for $k = 10$.

Method	$k = 5$		$k = 10$		$k = 15$	
	Test MSE	**Precision**	**Test MSE**	**Precision**	**Test MSE**	**Precision**
SIMPLE (Ours)	**2.27 ± 0.05**	**57.30 ± 3.04**	**2.23 ± 0.03**	**47.17 ± 2.11**	3.20 ± 0.04	**53.18 ± 1.09**
L2X (t = 0.1)	5.75 ± 0.30	33.63 ± 6.91	6.68 ± 1.08	26.65 ± 9.39	7.71 ± 0.64	23.49 ± 10.93
SoftSub (t = 0.5)	2.57 ± 0.12	**54.06 ± 6.29**	2.67 ± 0.14	44.44 ± 2.27	**2.52 ± 0.07**	37.78 ± 1.71
I-MLE (τ = 30)	2.62 ± 0.05	**54.76 ± 2.50**	2.71 ± 0.10	**47.98 ± 2.26**	2.91 ± 0.18	39.56 ± 2.07

Table 3.7: Results for aspect Aroma: test MSE and subset precision, both $\times 100$, for $k \in \{5, 10, 15\}$.

3.2.4.4 Sparse Linear Regression

Given a library of feature functions, the task of sparse linear regression aims to learn from data which feature subset best describes the nonlinear partial differential equation (PDE) that the data are sampled from. We propose to tackle this task by learning a k-subset distribution over the feature functions. During learning, we first sample from the k-subset distribution to decide which feature function subset to choose. With k chosen features, we perform linear regression to learn the coefficients of the features from data, and then update the k-subset distribution logit parameters by minimizing RMSE.

To test our proposed approach, we follow the experimental setting in PySINDy [de Silva et al., 2020; Kaptanoglu et al., 2022] and use the dataset collected by PySINDy where the samples are collected from the KuramotoSivashinsky (KS) equation, a fourth-order nonlinear PDE known for its chaotic behavior. This PDE takes the form $v_t = -v_{xx} - v_{xxxx} - vv_x$, which can be seen as a linear combination of feature functions $\mathcal{V} = \{v_{xx}, v_{xxxx}, vv_x\}$ with the coefficients all set to a value of -1. At test time, we use the MAP estimation of the learned k-subset distribution to choose the k feature functions. For $k = 3$, our proposed method achieves the same performance as the state-of-the-art solver on this task, PySINDy. It identifies the KS PDE from data by choosing exactly the ground truth feature function subset $\mathcal{V}$, obtaining an RMSE of 0.00622 after applying linear regression on $\mathcal{V}$.

3.2.5 Complexity Analysis

In Proposition 3.1, we prove that computing the marginal probability of the exactly-k constraint can be done tractably in time $\mathcal{O}(nk)$. In the context of deep learning, we often care about vectorized complexity. We demonstrate an optimized algorithm achieving a vectorized complexity $\mathcal{O}(\log k \log n)$, assuming perfect parallelization. The optimization is possible by computing the marginal probability in a divide-and-conquer way: it partitions the variables into two subsets and compute their marginals respectively such that the complexity $\mathcal{O}(n)$ is reduced to $\mathcal{O}(\log n)$; the summation over the k terms also has its complexity reduced to $\mathcal{O}(\log k)$ in a similar manner. We refer the readers to Algorithm 12 in Appendix for the optimized algorithm. We further modify Algorithm 6 to perform divide-and-conquer such that sampling k-subsets achieves a vectorized complexity being $\mathcal{O}(\log n)$, shown as Algorithm 13 in the Appendix. As a comparison, SoftSub [Xie and Ermon, 2019] has its complexity to be $\mathcal{O}(nk)$ due to the relaxed top-k operation and its vectorized complexity to be $\mathcal{O}(k \log n)$ stemming from the fact that softmax layers need $\mathcal{O}(\log n)$ rounds of communication for normalization.

3.2.6 Related Work

There is a large body of work on gradient estimation for categorical random variables. Maddison et al. [2017]; Jang et al. [2017] propose the Gumbel-softmax distribution (named the concrete distribution by the former) to relax categorical random variables. For more complex distributions, such as the k-subset distribution which we are concerned with in this paper, existing approaches either use the straight-through and score function estimators or propose tailor-made relaxations (see for instance Kim et al. [2016]; Chen et al. [2018]; Grover et al. [2018]). We directly compare to the score function and straight-through estimator as well as the tailored relaxations of Chen et al. [2018]; Grover et al. [2018] and show that we are competitive and obtain a lower bias and/or variance than these other estimators. Tucker et al. [2017]; Grathwohl et al. [2018] develop parameterized control variates based on continuous relaxations for the score-function estimator. Lastly, Paulus et al. [2020] offers a comprehensible work on relaxed gradient estimators, deriving several

extensions of the softmax trick. All of the above works, ours included, assume the independence of the selected items, beyond there being k of them. That is with the exception of Paulus et al. [2020] which make use of a relaxation using pairwise embeddings, but do not make their code available. We leave that to future work.

A related line of work has developed and analyzed sparse variants of the softmax function, motivated by their potential computational and statistical advantages. Representative examples are Blondel et al. [2020a]; Peters et al. [2019]; Correia et al. [2019]; Martins and Astudillo [2016]. SparseMAP [Niculae et al., 2018] has been proposed in the context of structured prediction and latent variable models, also replacing the softmax with a sparser distribution. LP-SparseMAP [Niculae and Martins, 2020] is an extension that uses a relaxation of the optimization problem rather than a MAP solver. Sparsity can also be exploited for efficient marginal inference in latent variable models [Correia et al., 2020b]. Contrary to our work, they cannot control the sparsity level exactly through a k-subset constraint or guarantee a sparse output. Also, we aim at cases where samples in the forward pass are required.

Integrating specialized discrete algorithms into neural networks is growing in popularity. Examples are sorting algorithms [Cuturi et al., 2019; Blondel et al., 2020b; Grover et al., 2018], ranking [Rolinek et al., 2020; Kool et al., 2019], dynamic programming [Mensch and Blondel, 2018; Corro and Titov, 2019], and solvers for combinatorial optimization problems [Berthet et al., 2020; Rolínek et al., 2020; Shirobokov et al., 2020; Niepert et al., 2021b; Minervini et al., 2023] or even probabilistic circuits over structured output spaces [Ahmed et al., 2022b; Blondel, 2019]. There has also been work on making common programming language expression such as conditional statements, loops, and indexing differentiable through relaxations [Petersen et al., 2021]. Xie et al. [2020] propose optimal transport to obtain differentiable sorting methods for top-k classification.

CHAPTER 4

Scaling

Much of the existing work on neuro-symbolic AI offers piecemeal solutions to the problem of integrating learning and reasoning, with legacy code that does not seamlessly integrate or domain specific languages that require porting of existing code bases to fit within the design of the target framework. There is, therefore, a palpable need for a unifying framework that seamlessly integrates with existing deep learning code and allows the user to easily specify and utilize constraints. To that end, we propose Pylon, a neuro-symbolic training framework that builds on PyTorch to augment procedurally trained neural networks with declaratively specified constraints. Pylon allows users to programmatically specify constraints as PyTorch functions which are then compiled into a differentiable loss, training predictive models that fit the data and the constraint. Pylon includes exact and approximate compilers to efficiently compute the loss, ensuring scalability even to complex models and constraints. Using Pylon, an existing codebase can be extended to learn from constraints in a few lines: a function expressing the constraint and a single line of code to compile it into a loss. When the PyTorch function can be compiled into a tractable circuit, Pylon makes use of PyJuice, a framework we proposed to efficiently compute the loss. PyJuice interprets the circuit as a layerwise computational graph, leading to maximal GPU utilization and running times orders of magnitude faster than older implementations.

4.1 PYLON: A PyTorch Framework for Learning with Constraints

Deep learning models, by virtue of being universal function approximators, are able to learn even the most complex of tasks with enough available data. However, some high-level domain knowl-

edge can often be much more succinctly described *directly* in a declarative manner, such as programmatic constraints, which existing learning frameworks are not able to learn from. Instead, deep learning models attempt to extract the same knowledge from available data, leading to overfitting the spurious patterns, learning functions that are unfaithful to rules of the underlying domain.

Neuro-symbolic reasoning systems aim to straddle the line between deep learning and symbolic reasoning, combining high-level procedural knowledge with data, during learning. They aim to learn functions that fit the data while remaining faithful to the rules of the underlying domain, and empirically translates into performance improvements and more efficient learning. These systems are not without their challenges, however. Most frameworks make use of custom languages [Rajaby Faghihi et al.; Guo et al., 2020; Manhaeve et al., 2018; Stewart and Ermon, 2017] or logic [Bach et al., 2017; Diligenti et al., 2017b; Fischer et al., 2019; Hu et al., 2016; Li and Srikumar, 2019; Nandwani et al., 2019; Rocktäschel et al., 2015; Xu et al., 2018a; Zhang et al., 2016] to express the knowledge, making it unnatural, unwieldy or even impossible to express many forms of knowledge. Further, they often require porting codebases to use these systems, making them arduous to integrate into preexisting code. Final, different approaches to integrate symbolic knowledge and neural models have their own specific strengths and weaknesses, thus effective on limited set of domains and constraints; however, this is often not clear to the user.

We introduce PYLON[1], a package built on top of PyTorch that offers practitioners the ability to seamlessly integrate procedural knowledge into deep learning models. The user expresses the knowledge directly as a Python predicate function that defines the constraint on tensor variables (such as model output). PYLON compiles this user-defined function to efficiently compute a differentiable loss compatible with PyTorch trainers, providing a common interface to existing neural-symbolic approaches that integrate declarative knowledge in the learning process. With a few lines of code (defining the constraint and adding the loss), the user is able to integrate declarative knowledge into their models, testing out which of the existing approaches are most effective.

[1] PYLON website is available at `https://pylon-lib.github.io/`

4.1.1 PYLON Overview

Example Consider the code snippet in Figure 4.1, where we consider the task of entity-relation extraction. That is, given a sentence `x`, the `model` on line 14 classifies each word into a corresponding entity (e.g. person, organization), and for every pair of entities whether they are related, and if so, the type of relation that holds between them (e.g. works for)

```
1  # Only a person can live in a location
2  def check_livesin_subj(entity, relation):
3      # if word is subj of livesIn, it should be PER entity
4      return all(entity[relation==LIVESIN_SUBJ] == PER)
5
6  livesin_loss = constraint_loss(check_livesin_subj)
7
8  # there should be more non-people tokens than people
9  numppl_loss = constraint_loss(
10     lambda entity: sum(entity!=PER) > sum(entity==PER))
11
12 for i in range(train_iters):
13     ...
14     relation_logits, entity_logits = model(x)
15     loss = F.CrossEntropy(relation_logits, relation_labels)
16     loss += livesin_loss(entity_logits, relation_logits)
17     loss += numppl_loss(entity_logits)
```

Figure 4.1: Enforcing a constraint using PYLON

We wish to enforce two constraints, which stem from our knowledge of the problem domain, on the learned `model` on line 14: 1) the subject of a `lives in` relation is always a `person`, and 2) that the majority of predicted entities are not `person`. The above constraints are expressed as the PyTorch function `check_livesin_subj` defined on lines 2-4 and the lambda function on line 10, respectively.

The challenge then is, how to integrate these discrete, Boolean functions with differentiable learning. We will show how this can be achieved by interpreting the network's outputs as inducing a distribution over the output space, and reducing our problem to one of probabilistic reasoning: we wish to find the set of parameters that maximize the probability of the functions under the network's distribution.

A Probability Distribution over Structured Outputs Let θ be the parameters of a neural network defined over a set of variables $\mathbf{Y} = \{Y_1, \ldots, Y_n\}$, where each Y_i denotes a class output by the network. Let p be a vector of probabilities for the same variables $\mathbf{Y}$, where p_i denotes the predicted probability of variable Y_i and corresponds to a single output of the neural network. The

neural network's outputs *induce* a distribution $\Pr(\cdot)$ over all possible instantiations, or decodings, $\mathbf{y}$ of $\mathbf{Y}$

$$\Pr(\boldsymbol{y}) = \prod_{i:\boldsymbol{y}\models Y_i} \mathsf{p}_i \prod_{i:\boldsymbol{y}\models \neg Y_i} (1 - \mathsf{p}_i). \tag{4.1}$$

where $\boldsymbol{y} \models Y_i$ and $\boldsymbol{y} \models \neg Y_i$ denote that Y_i is true or false in the instantiation $\boldsymbol{y}$, respectively.

Training objective Having defined a distribution over all possible outputs, we now consider the problem of learning with constraints through a probabilistic lens: the problem of integrating our declaratively-defined functions into the learning process reduces to optimizing for the set of network parameters such that the probability allocated by the network to the constraint is maximized. Formally

$$\arg\min_{\theta} \mathcal{L}(\theta|\mathcal{C}, x) = \arg\min_{\theta} -\log \mathbb{E}_{\mathbf{y}\sim p_\theta(\cdot|x)}\big[\mathbf{1}\{\mathcal{C}(\mathbf{y})\}\big] \tag{4.2}$$

That is, for a given constraint $\mathcal{C}$, we penalize the network with a loss that is proportional to the extent to which the network's beliefs violate the constraint, as measured by the probability mass allocated by the network to all decodings violating the constraint $\mathcal{C}$.

Calculating the above naively requires enumerating all decodings $\mathbf{y}$ in a *brute force* manner, of which there are exponentially many, and is feasible only for simple constraints. For instance, for a classifier defined over $2n^2 - 2n$ variables – the edges in a $n \times n$ grid – and that predicts a path in the grid, a decoding is an assignment to each of the $2n^2 - 2n$ variables, and there are 2^{2n^2-2n} such decodings.

Exploiting Structure of Constraint Definition Even though the user is free to use all of PyTorch/Python to write the constraint, we parse the constraint code to see if it is expressing known structures, for example, first-order logic. When the constraints do exhibit structural properties that allow us to reuse intermediate computations, we can sidestep the intractability of eqn. 4.3 by compiling them into *logical circuits* [Xu et al., 2018a]. This does not, in general, escape the complexity of eqn. 4.3 as the compiled circuit can worst-case grow exponentially in the size of the constraint.

In such a case, we can utilize approximations based on fuzzy logic, computing differentiable probabilities of logical statements without grounding them, such as using product *T-norm* [Rocktäschel et al., 2015], or Łukasiewicz *T-norm* [Bach et al., 2017; Kimmig et al., 2012].

Black-box Optimization Alternatively, we can also approximate the loss in eqn. 4.3 by *sampling* decodings from the network's posterior. More precisely, we can use the REINFORCE gradient estimator [Glynn, 1990; Williams, 1992] to rewrite the gradient of the expectation in eqn 4.3 as the expectation of the gradient, which can be readily estimated using Monte Carlo sampling. This not only enables us to estimate the probability of otherwise-intractable constraints but also enables greater flexibility in defining our constraint functions: we can issue calls to non-differentiable resources (e.g. external APIs, database queries, etc.) and continue to yield a *differentiable* loss function, hence the moniker *black-box*.

PYLON uses implementations of these approaches that are directly compatible with PyTorch, as seen in lines 16 and 17, including ones that utilize the structure in the user-defined code for efficiency (*T-norm* and *circuit*-based losses) and ones that work for any implementation (*brute-force* and *sampling*), and is easily extensible to other techniques.

Constraint functions We encode the aforementioned declarative knowledge (read: constraints) by means of *constraint functions*. A constraint function is a Python function that accepts any number of tensor arguments, each of shape $(\texttt{batch\_size}, ...)$ and returns a Boolean tensor of shape $(\texttt{batch\_size},)$. Each argument corresponds to a (batched) *decoding* from a model. A decoding is an assignment to all variables of a model, each variable sampled with a probability corresponding to its likelihood under the model's posterior. For example, in our entity-relation extraction example, a decoding of `relation_logits` (resp. `entity_logits`) constitutes a relation (resp. entity) assigned to each word in the sentence. On the other hand, for a classifier defined over $2n^2 - 2n$ Boolean variables – the edges in a $n \times n$ grid – and that predicts a path in the grid, a decoding constitutes an assignment to each of the $2n^2 - 2n$ variables, and there are 2^{2n^2-2n}

such decodings.

A constraint function defines a *predicate* $\mathcal{C}$ on the decodings of any number of models, and returns whether or not the given decodings satisfy the constraint. For instance, lines 2-4 define a constraint function over the decodings of the entity and relations classifiers which encodes our first constraint whereas line 10 defines a lambda constraint function over the decoding of the entity classifier, and encodes our second constraint. Note that while the first constraint can be easily expressed in logic, the same does not hold true for the second constraint: we would need to conjoin all decodings satisfying the constraint, which would scale exponentially with the length of the sentence – unless we resort to introducing auxiliary variables. Using Python/PyTorch we manage to capture the constraint succinctly in a single line of code.

Constraint functions We define *constraint functions* that express this knowledge. A constraint function is a Python function that accepts any number of tensor arguments, each of shape $(\texttt{batch\_size}, ...)$ and returns a Boolean tensor of shape $(\texttt{batch\_size},)$. Each argument represents a (batched) *decoding* corresponding to the model's posterior. A constraint function represents a constraint $\mathcal{C}$ on these decodings, returning whether or not the decodings satisfy the constraint. For instance, lines 2-4 define a function corresponding to our first constraint and line 10 defines a lambda function for the second constraint. Note while the first constraint can be easily expressed in logic, the same is not true of the second constraint, which we express succinctly using Python/PyTorch.

Training objective Having defined the constraint $\mathcal{C}$, we aim to *compile* it into a loss function to encourage the model to satisfy the constraints, i.e. minimize the probability of constraint violation.

$$\arg\min_{\theta} \mathcal{L}(\theta|\mathcal{C}, x) = \arg\min_{\theta} -\log \mathbb{E}_{\mathbf{y}\sim p_\theta(\cdot|x)}\big[\mathbb{1}\{\mathcal{C}(\mathbf{y})\}\big]. \tag{4.3}$$

Calculating the above naively requires we enumerate all possible decodings $\mathbf{y}$ in a *brute force* manner, of which there are exponentially many, and is feasible only for the simplest of constraints. If the constraints have certain structure, we can sidestep the intractability of (4.3), for example, by compiling them into *circuits* [Xu et al., 2018a]. The loss can also be approximated, for example, we

can use the REINFORCE trick [Glynn, 1990; Williams, 1992] to rewrite the gradient of the expectation, estimated using Monte Carlo *sampling*. the gradient of the function, which we can estimate using Monte Carlo *sampling*. If the constraints can be expressed as logic, we can use approximations based on fuzzy logic that operate in the lifted domain, computing differentiable probabilities of logical statements without grounding them, such as using product *T-norms* [Rocktäschel et al., 2015]. PYLON contains implementations of these approaches that are directly compatible with PyTorch, as shown in lines 16 and 17, including ones that utilize the structure in the user-defined code for efficiency (*T-norm*- and *circuit*-based losses) and ones that work for any implementation (*brute-force* and *sampling*), and is easily extensible to other techniques.

4.2 Scaling Tractable Probabilistic Circuits: A Systems Perspective

Many tasks require not only precise modeling of intricate, high-dimensional data distributions but also the efficient execution of probabilistic inference on the learned model. To satisfy inference-side demands, tractable deep generative models are designed to support efficient computation of various probabilistic queries. Probabilistic Circuits (PCs) [Choi et al., 2020b; Vergari et al., 2020] are a unified framework that abstracts a myriad of tractable model families. PCs have been applied to many domains such as explainability and causality [Correia et al., 2020a; Wang and Kwiatkowska, 2023], graph link prediction [Loconte et al., 2023], and neuro-symbolic AI [Xu et al., 2018a; Manhaeve et al., 2018; Ahmed et al., 2022b]. In particular, there is a trend of using PCs' tractability to control expressive deep generative models, including (large) language models [Zhang et al., 2023b], image diffusion models [Liu et al., 2024b], and reinforcement learning models [Liu et al., 2023d].

The backbone of the application-side advancements is the recent breakthroughs on the modeling and learning side of PCs, which include designing better PC structures [Peharz et al., 2020b; Correia et al., 2023; Mathur et al., 2023; Loconte et al., 2024; Gala et al., 2024], effective structure learning algorithms [Gens and Pedro, 2013; Dang et al., 2020, 2022a; Yang et al., 2023], and

distilling from expressive deep generative models [Liu et al., 2023b]. Despite such algorithmic innovations, a fundamental obstacle to further scaling up PC learning and inference is the time and memory inefficiency of existing implementations, hindering the training of large PC models and their application to large-scale datasets.

In this work, we develop an efficient and flexible system called PyJuice that addresses various training and inference tasks for PCs. As shown in Table 4.1, PyJuice is orders of magnitude faster than previous implementations for PCs (e.g., SPFlow [Molina et al., 2019], EiNet [Peharz et al., 2020a], and Juice.jl [Dang et al., 2021]) as well as Hidden Markov Models[2] (e.g., Dynamax [Murphy et al., 2023]). Additionally, as we shall demonstrate in the experiments, PyJuice is more memory efficient than the baselines, enabling us to train larger PCs with a fixed memory quota.

Unlike other deep generative models based on neural network layers that are readily amenable to efficient systems (e.g., a fully connected layer can be emulated by a single matrix multiplication and addition kernel plus an element-wise activation kernel), PCs cannot be *efficiently* computed using well-established operands due to (i) the unique connection patterns of their computation graph,[3] and (ii) the existence of log probabilities at drastically different scales in the models, which requires to properly handle numerical underflow problems. To parallelize PCs at scale, we propose a compilation phase that converts a PC into a compact data structure amenable to block-based parallelization on modern GPUs. Further, we improve the backpropagation process by indirectly computing the parameter updates by backpropagating a quantity called PC flow [Choi et al., 2021] that is more numerically convenient yet mathematically equivalent.

In the following, we first formally define PCs and discuss common ways to parallelize their computation in Sec. 4.2.1. Sec. 4.2.2 examines the key bottlenecks in PC parallelization. Sec. 4.2.3 and 4.2.4 explains our design in details.

[2]Every HMM has an equivalent PC representation.

[3]Commonly used neural network layers mainly employ "regular" tensor operations such as matrix multiplications and tensor inner-/outer-products. In contrast, PC layers can contain nodes that are sparsely connected.

Table 4.1: **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch** of 60K samples for PyJuice and the baselines SPFlow [Molina et al., 2019], EiNet [Peharz et al., 2020a], Juice.jl [Dang et al., 2021], and Dynamax [Murphy et al., 2023]. We adopted four PC structures: PD, RAT-SPN, HCLT, and HMM. All experiments were carried out on an RTX 4090 GPU with 24GB memory. To maximize parallelism, we always use the maximum possible batch size. "OOM" denotes out-of-memory with batch size 2. The best numbers are in boldface.

	PD [Poon and Domingos, 2011]				
# nodes	172K	344K	688K	1.38M	2.06M
# edges	15.6M	56.3M	213M	829M	2.03B
SPFlow	>25000	>25000	>25000	>25000	>25000
EiNet	34.2$_{\pm 0.0}$	88.7$_{\pm 0.2}$	456.1$_{\pm 2.3}$	1534.7$_{\pm 0.5}$	OOM
Juice.jl	12.6$_{\pm 0.5}$	37.0$_{\pm 1.7}$	141.7$_{\pm 6.9}$	OOM	OOM
PyJuice	**2.0**$_{\pm 0.0}$	**5.3**$_{\pm 0.0}$	**15.4**$_{\pm 0.0}$	**57.1**$_{\pm 0.2}$	**203.7**$_{\pm 0.1}$
	RAT-SPN [Peharz et al., 2020b]				
# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
SPFlow	6372.1$_{\pm 4.2}$	>25000	>25000	>25000	>25000
EiNets	38.5$_{\pm 0.0}$	83.5$_{\pm 0.0}$	193.5$_{\pm 0.1}$	500.6$_{\pm 0.2}$	2445.1$_{\pm 2.6}$
Juice.jl	6.0$_{\pm 0.3}$	9.4$_{\pm 0.3}$	25.5$_{\pm 2.4}$	84.0$_{\pm 4.0}$	375.1$_{\pm 3.4}$
PyJuice	**0.6**$_{\pm 0.0}$	**0.9**$_{\pm 0.1}$	**1.6**$_{\pm 0.0}$	**5.8**$_{\pm 0.1}$	**13.8**$_{\pm 0.0}$
	HCLT [Liu and Van den Broeck, 2021]				
# nodes	89K	178K	355K	710K	1.42M
# edges	2.56M	10.1M	39.9M	159M	633M
SPFlow	22955.6$_{\pm 18.4}$	>25000	>25000	>25000	>25000
EiNet	52.5$_{\pm 0.3}$	77.4$_{\pm 0.4}$	233.5$_{\pm 2.8}$	1170.7$_{\pm 8.9}$	5654.3$_{\pm 17.4}$
Juice.jl	4.7$_{\pm 0.2}$	6.4$_{\pm 0.5}$	12.4$_{\pm 1.3}$	41.1$_{\pm 0.1}$	143.2$_{\pm 5.1}$
PyJuice	**0.8**$_{\pm 0.0}$	**1.3**$_{\pm 0.0}$	**2.6**$_{\pm 0.0}$	**8.8**$_{\pm 0.0}$	**24.9**$_{\pm 0.1}$
	HMM [Rabiner and Juang, 1986]				
# nodes	33K	66K	130K	259K	388K
# edges	8.16M	32.6M	130M	520M	1.17B
Dynamax	111.3$_{\pm 0.4}$	441.2$_{\pm 3.9}$	934.7$_{\pm 6.3}$	2130.5$_{\pm 19.5}$	4039.8$_{\pm 38.3}$
Juice.jl	4.6$_{\pm 0.1}$	18.8$_{\pm 0.1}$	91.6$_{\pm 0.1}$	OOM	OOM
PyJuice	**0.6**$_{\pm 0.0}$	**1.0**$_{\pm 0.0}$	**2.9**$_{\pm 0.1}$	**10.1**$_{\pm 0.2}$	**39.9**$_{\pm 0.1}$

4.2.1 Preliminaries and Related Work

Many probabilistic inference tasks can be cast into computing sums of products. By viewing them from a computation graph standpoint, PCs provide a unified perspective on many bespoke representations of tractable probability distributions, including Arithmetic Circuits [Darwiche, 2002, 2000], Sum-Product Networks [Poon and Domingos, 2011], Cutset Networks [Rahman et al., 2014], and Hidden Markov Models [Rabiner and Juang, 1986]. Specifically, PCs define distributions with

computation graphs consisting of sum and product operations, as elaborated below.

Definition 4.1 (Probabilistic Circuit)**.** *A PC defined over variables* $\mathbf{X}$ *is represented by a parameterized Directed Acyclic Graph (DAG) with a single root node* n_{r}*. Every leaf node in the DAG represents an input node that defines a primitive distribution over some variable* $X \in \mathbf{X}$*. Every inner node* n *is either a sum node or a product node, which merges the distributions encoded by its children, denoted* $\mathsf{ch}(n)$*, to construct more complex distributions. The distribution represented by every node is defined recursively as:*

$$p_n(\boldsymbol{x}) := \begin{cases} f_n(\boldsymbol{x}) & n \text{ is an input node}, \\ \prod_{c \in \mathsf{ch}(n)} p_c(\boldsymbol{x}) & n \text{ is a product node}, \\ \sum_{c \in \mathsf{ch}(n)} \theta_{n,c} \cdot p_c(\boldsymbol{x}) & n \text{ is a sum node}, \end{cases} \tag{4.4}$$

where $f_n(\boldsymbol{x})$ *is an univariate input distribution (e.g., Gaussian, Categorical), and* $\theta_{n,c}$ *denotes the parameter corresponding to edge* (n, c)*. Intuitively, sum nodes model mixtures of their input distributions, which require the mixture weights to be in the probability simplex:* $\sum_{c \in \mathsf{ch}(n)} \theta_{n,c} = 1$ *and* $\forall c \in \mathsf{ch}(n), \theta_{n,c} \geq 0$*. And product nodes build factorized distributions over their inputs. The size of a PC, denoted* $|p|$*, is the number of edges in its DAG.*

The key to guaranteeing exact and efficient computation of various probabilistic queries is to impose proper structural constraints on the DAG of the PC. As an example, with smoothness and decomposability [Poon and Domingos, 2011], computing any marginal probability amounts to a forward pass (children before parents) following Equation 4.4, with the only exception that we set the value of input nodes defined on marginalized variables to be 1. Please refer to Choi et al. [2020b] for a comprehensive overview of different structural constraints and what queries they enable.

Although different algorithms are used for different training and inference tasks, they are mostly based on (variants of) the following subroutines: a feedforward pass (Eq. (4.4)) that computes $\log p_{n_{\mathrm{r}}}(\boldsymbol{x})$, and a backward pass computing

$$\forall n, \frac{\partial \log p_{n_{\mathrm{r}}}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})} \text{ and } \forall \theta_{n,c}, \frac{\partial \log p_{n_{\mathrm{r}}}(\boldsymbol{x})}{\partial \theta_{n,c}}. \tag{4.5}$$

For example, Peharz et al. [2020a] demonstrate how the above parameter gradients can be used to apply Expectation-Maximization (EM) updates, and Vergari et al. [2021] elaborates how the forward pass can be used to compute various probabilistic and information-theoretic queries when coupled with PC structure transformation algorithms. Therefore, the speed and memory efficiency of these two procedures largely determine the overall efficiency of PCs.

Related work on accelerating PCs. There has been a great amount of effort put into speeding up training and inference for PCs. One of the initial attempts performs node-based computations on both CPUs [Lowd and Rooshenas, 2015] and GPUs [Pronobis et al., 2017; Molina et al., 2019], i.e., by computing the outputs for a mini-batch of inputs (data) recursively for every node. Despite its simplicity, it fails to fully exploit the parallel computation capability possessed by modern GPUs since it can only parallelize over a batch of samples. This problem is mitigated by also parallelizing topologically independent nodes [Peharz et al., 2020a; Dang et al., 2021]. Specifically, a PC is chunked into topological layers, where nodes in the same layer can be computed in parallel. This leads to 1-2 orders of magnitude speedup compared to node-based computation.

The regularity of edge connection patterns is another key factor influencing the design choices. Specifically, EiNets [Peharz et al., 2020a] leverage off-the-shelf Einsum operations to parallelize dense PCs where every layer contains groups of densely connected sum and product/input nodes. Mari et al. [2023] generalize the notion of dense PCs to tensorized PCs, which greatly expands the scope of EiNets. Dang et al. [2021] instead focus on speeding up sparse PCs, where different nodes could have drastically different numbers of edges. They use custom CUDA kernels to balance the workload of different GPU threads and achieve decent speedup on both sparse and dense PCs.

Another thread of work focuses on designing computation hardware that is more suitable for PCs. Specifically, Shah et al. [2021] propose DAG Processing Units (DPUs) that can efficiently traverse sparse PCs, Dadu et al. [2019] introduce an indirect read reorder-buffer to improve the efficiency of data-dependent memory accesses in PCs, and Yao et al. [2023] use addition-as-int multiplications to significantly improve the energy efficiency of PC inference algorithms.

Applications of PCs. PCs have been applied to many domains such as explainability and

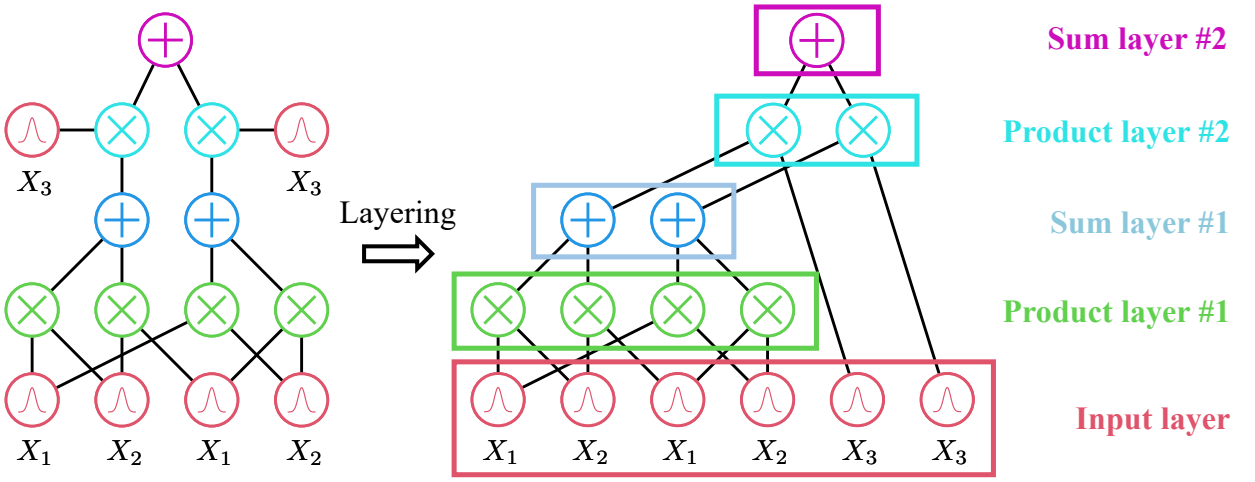


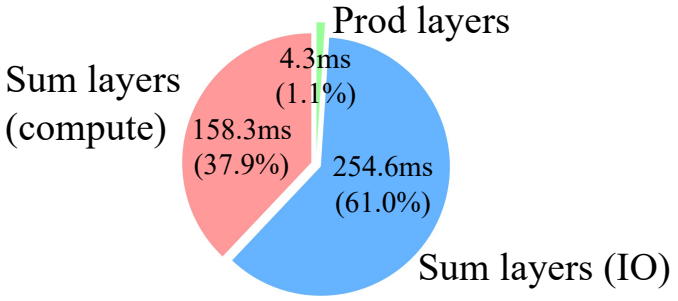
Figure 4.2: Layering a PC by grouping nodes with the same topological depth (as indicated by the colors) into disjoint subsets. Both the forward and the backward computation can be carried out independently on nodes within the same layer.

causality [Correia et al., 2020a; Wang and Kwiatkowska, 2023], graph link prediction [Loconte et al., 2023], lossless data compression [Liu et al., 2022a], neuro-symbolic AI [Xu et al., 2018a; Manhaeve et al., 2018; Ahmed et al., 2022b,c], gradient estimation [Ahmed et al., 2023c], graph neural networks rewiring [Qian et al., 2023], and even large language model detoxification [Ahmed et al., 2023b].

4.2.2 Key Bottlenecks in PC Parallelization

This section aims to lay out the key bottlenecks to efficient PC implementations. For ease of illustration, we focus solely on the forward pass, and leave the unique challenges posed by the backward pass and their solution to Sec. 4.2.4.

We start by illustrating the layering procedure deployed for PCs. Starting from the input nodes, we perform a topological sort of all nodes, clustering nodes with the same topological depth into a layer. For example, in Figure 4.2, the PC on the left side is transformed into an equivalent layered representation on the right, where nodes of the same color belong to the same layer. The forward pass proceeds by sequentially processing each layer, and finally returns the root node's output.



Summary of the PC structure

	Sum layers	Prod layers
# nodes	200K	815K
# edges	154M	1.81M
# params	154M	-

Figure 4.3: Runtime breakdown of the feedforward pass of a PC with $\sim$150M edges. Both the IO and the computation overhead of the sum layers are significantly larger than the total runtime of product layers. Detailed configurations of the PC are shown in the table.

To avoid underflow, all probabilities are stored in the logarithm space. Therefore, product layers just need to sum up the corresponding input log-probabilities, while sum layers compute weighted sums of input log-probabilities utilizing the logsumexp trick.

Assume for now that all nodes in every layer have the same number of children. A straightforward strategy is to parallelize over every node and every sample. Specifically, given a layer of size M and batch size B, we need to compute in total $M \times B$ output values, which are evenly distributed to all processors (e.g., thread-blocks in GPUs). We apply this idea to a PC with the PD structure [Poon and Domingos, 2011]. The PC has $\sim$1M nodes and $\sim$150M edges. Additionally, all nodes within a layer have the same number of children, making it an ideal testbed for the aforementioned algorithm.

Figure 4.3 illustrates the runtime breakdown of the forward pass (with batch size 512). As shown in the pie chart, both the IO and the computation overhead of the sum layers are much larger than that of the product layers. We would expect sum layers to exhibit a higher computation overhead due to (i) the number of sum edges being $\sim$85x more than the product edges (see the table in Fig. 4.3), and (ii) sum edges requiring more compute compared to product edges. However, we would not expect the gap in IO overhead to be as pronounced as indicated in the pie chart. Specifically, with batch size 512, the ideal memory read count of product layers should be roughly [batch size] $\times$ [#sum nodes] $\approx$ 102M since all children of product nodes are sum or input nodes

(the number of input nodes is an order of magnitude smaller and is omitted). Similarly, the number of memory reads required by the sum layers is approximately $[\text{batch size}] \times [\#\text{prod nodes}] + [\#\text{parameters}] \approx 571\text{M}$, which is only 5.6x compared to the product layers. The ideal memory write count of product layers should be larger since there are about 4x more product nodes compared to sum nodes.

While the ideal IO overhead of the sum layers is not much larger than that of the product layers, the drastic difference in runtime (over 50x) can be explained by the significant amount of reloads of child nodes' probabilities in the sum layers. Specifically, in the adopted PD structure, every sum node has no more than 12 parents, while most product nodes have 256 parents.[4] Recall that the parents of product nodes are sum nodes and vice versa. As a result, each sum layer needs to reload the output of every product node multiple times. Although this does not lead to 256x loads from the GPU's High-Bandwidth Memory (HBM) thanks to its caching mechanism, such excessive IO access still significantly slows down the algorithm.

The fundamental principle guiding our design is to *properly group, or allocate, sum edges to different processors to minimize the reloading of product nodes' outputs*. As an added benefit, this allows us to interpret part of the core computation as matrix multiplications, allowing us to harness Tensor Cores available in modern GPUs and resulting in a significant reduction in sum layers' computational overhead.

4.2.3 Harnessing Block-Based PC Parallelization

This section takes gradual steps toward demonstrating how we can reduce both the IO and computation overhead using block-based parallelization. Specifically, we first utilize a fully connected sum layer to sketch the high-level idea (Sec. 4.2.3.1). Consequently, we move on to the general case, providing further details of the algorithm (Secs. 4.2.3.2, 4.2.3.3).

[4] Only the children of the root sum node have 1 parent.

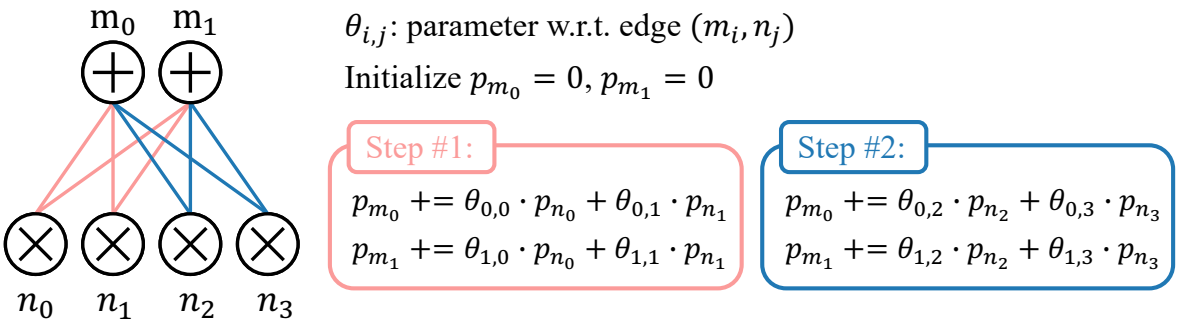


Figure 4.4: Illustration of block-based parallelization. A processor computes the output of 2 sum nodes, by iterating through blocks of 2 input product nodes and accumulating partial results.

4.2.3.1 Fully Connected Sum Layers

Consider a fully connected sum layer comprised of M sum nodes, each connected to the same set of N product nodes as inputs. Under the parallelization strategy mentioned in Sec. 4.2.2, with a single sample, we have M processors each computing the output of a sum node. Since the layer is fully connected, every processor loads all N input log-probabilities, which results in M reloads of every input.

The key to reducing excessive IO overhead is by parallelizing over blocks of nodes/edges. Specifically, we divide the M sum nodes into blocks of K_M nodes and the N product nodes into blocks of K_N nodes. We assume without loss of generality that M and N are divisible by K_M and K_N, respectively.[5] Instead of independently computing the output of every sum node, we calculate the K_M outputs of a sum node block in a single processor. To achieve this, we iterate through every product node block to compute and accumulate the partial results from the $K_M \times K_N$ edges between the corresponding sum node block and product node block.

In every step, the processor loads a block of $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ parameters and a vector of $\boldsymbol{p}_{\text{prod}} \in \mathbb{R}^{K_N}$ input probabilities, where we (temporarily) omit the fact that all probabilities are stored in the log-

[5]When the number of product and sum nodes are not divisible by the respective block size, we can add at most $K_M - 1$ (or $K_N - 1$) placeholder nodes to make them divisible by the block size. The incurred additional computation overhead can be small since we can achieve good efficiency with relatively small block sizes (e.g., 32 or 64) given that the number of nodes in a layer is typically greater than a few thousand.

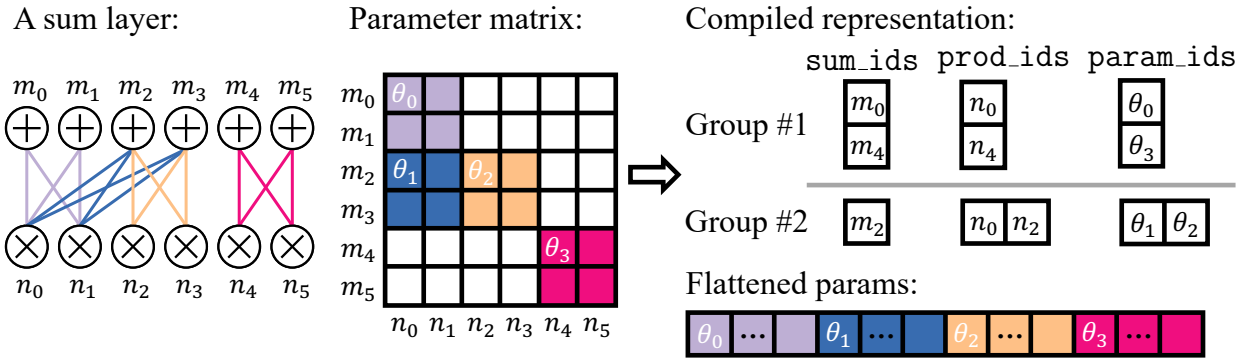


Figure 4.5: A sum layer (left) with a block-sparse parameter matrix (middle) is compiled into two kernels (right) each with a balanced workload. During execution, each kernel uses the compiled sum/prod/param indices to compute the outputs of $m_0, \ldots, m_5$.

arithm space. The partial outputs $\boldsymbol{p}_{\text{sum}} \in \mathbb{R}^{K_M}$ are computed via a matrix-vector multiplication between $\boldsymbol{\theta}$ and $\boldsymbol{p}_{\text{prod}}$. Note that if we add a second "batch" dimension to $\boldsymbol{p}_{\text{prod}}$ and $\boldsymbol{p}_{\text{sum}}$, the computation immediately becomes a matrix-matrix multiplication, which can be computed efficiently using GPU Tensor Cores.

For example, in Figure 4.4, define $K_M = K_N = 2$, we compute the output of m_0 and m_1 by first calculating the weighted sum w.r.t. the input probability of n_0 and n_1 in step #1, and then accumulate the probabilities coming from n_2 and n_3 in step #2. With the new parallelization strategy, every processor that computes K_M output values needs to load every input probability only once, and the number of reloads is reduced from M to M/K_M.

4.2.3.2 Generalizing To Practical Sum Layers

Many sum layers in practical PCs are not fully connected (e.g., in Dang et al. [2022a]; Liu et al. [2023b]). However, as we shall demonstrate, they can still harness the advantages of block-based parallelization. Specifically, consider a sum layer with M sum nodes and N product nodes as inputs. Following Sec. 4.2.3.1, we partition the sum and the product nodes into blocks of K_M and K_N nodes, respectively. For every pair of sum and product node blocks, if it is either fully connected (i.e., featuring $K_M \times K_N$ edges) or unconnected (i.e., no edge between them), we call the

layer block-sparse. In the following, we focus on efficiently parallelizing block-sparse PCs (whose sum layers all exhibit block-sparsity). We show in Sec. C.1.4.1 that many widely-adopted PCs are indeed block sparse w.r.t. large block sizes. In Sec. 4.2.3.4, we describe how our implementation can speed up sparse PCs. We also show in Sec. 4.2.5.1 that PyJuice speeds up sparse PCs.

As an example, the layer illustrated in Figure 4.5 (left) exhibits block sparsity with block sizes $K_M = K_N = 2$. This is evident as each pair of sum and product node blocks is either fully connected (e.g., $\{m_2, m_3\}$ and $\{n_0, n_1\}$) or disjoint (e.g., $\{m_4, m_5\}$ and $\{n_2, n_3\}$). In Figure 4.5 (middle), this pattern is more discernible in the parameter matrix, where *aligned* 2×2 blocks display either all non-zero parameters (indicated by the colors) or all zero parameters.

Similar to the procedure outlined in Sec. 4.2.3.1, computing the outputs of a block of K_M sum nodes involves iterating through all its connected product node blocks. This introduces two additional problems: (i) how to efficiently index the set of connected product node blocks, which may vary for each sum node block; (ii) different sum node blocks could connect to different numbers of product node blocks, which causes an imbalanced workload among processors. For instance, consider the layer in Figure 4.5. The first issue is exemplified by the two sum node blocks $\{m_0, m_1\}$ and $\{m_4, m_5\}$, both of which possess a single child node block, albeit different ones. The second issue is illustrated by the node block $\{m_2, m_3\}$, which connects to two child node blocks, while the others connect to only one.

4.2.3.3 Efficient Implementations by Compiling PC Layers

We address both problems through a compilation process, where we assign every node an index, and precompute index tensors that enable efficient block-based parallelization. The first step is to partition the sum node blocks into groups, such that every node block within a group has a similar number of connected child node blocks. We then pad the children with pseudo-product node blocks with probability 0 such that all sum node blocks in a group have the same number of children. The partition is generated by a dynamic programming algorithm that aims to divide the layer into the

Algorithm 9 Forward pass of a sum layer group

Inputs: log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, $\texttt{sum\_ids}$, $\texttt{prod\_ids}$, $\texttt{param\_ids}$
Inputs: # sum nodes: M, # product nodes: N, batch size: B
Inputs: block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
Inputs: number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
Outputs: log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$
Kernel launch: schedule to launch $C_M \times C_B$ thread-blocks with $\texttt{m}=0,\ldots,C_M-1$ and $\texttt{b}=0,\ldots,C_B-1$
$\texttt{cum} \leftarrow (-\infty)_{K_M \times K_B} \in \mathbb{R}^{K_M \times K_B}$ ▷ Scratch space on SRAM
$\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
for $\texttt{n} = 0$ **to** $C_N - 1$ **do**
  $\texttt{ps}, \texttt{ns} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{n}, \texttt{b}]$
  Load $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_{\text{flat}}[\texttt{ps}:\texttt{ps} + K_M \cdot K_N].\texttt{view}(K_M, K_N)$ to SRAM
  Load $\boldsymbol{l} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns}:\texttt{ns} + K_N, \texttt{bs}:\texttt{be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
  $\boldsymbol{l}_{\max} \leftarrow \texttt{max}(\boldsymbol{l}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
  $\boldsymbol{p}_{\text{p}} \leftarrow \texttt{exp}(\boldsymbol{l} - \boldsymbol{l}_{\max}) \in \mathbb{R}^{K_N \times K_B}$
  $\boldsymbol{p}_{\text{s}} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \boldsymbol{p}_{\text{p}}) \in \mathbb{R}^{K_M \times K_B}$ ▷ With Tensor Cores
  $\texttt{cum} \leftarrow \texttt{where}(\boldsymbol{l}_{\max} > \texttt{cum},$
    $\texttt{log}(\boldsymbol{p}_{\text{s}} + \texttt{exp}(\texttt{cum} - \boldsymbol{l}_{\max}) + \boldsymbol{l}_{\max},$
    $\texttt{log}(\texttt{exp}(\boldsymbol{l}_{\max} - \texttt{cum}) \cdot \boldsymbol{p}_{\text{s}} + 1) + \texttt{cum})$
$\boldsymbol{l}_{\text{sum}}[\texttt{ms}:\texttt{ms} + K_M, \texttt{bs}:\texttt{be}] \leftarrow \texttt{acc}$ (where $\texttt{ms} \leftarrow \texttt{sum\_ids}[\texttt{m}]$)

smallest possible number of groups while ensuring that the fraction of added pseudo-node blocks does not exceed a pre-defined threshold. Due to space constraints, we elaborate the node block partitioning algorithm in Sec. C.1.1.1. We also discuss its optimality and time/memory efficiency.

We move on to construct the index tensors for each group. In addition to assigning every node an index, we create a vector $\boldsymbol{\theta}_{\text{flat}}$, a concatentation of all the PC parameters. For every sum node block in a group with C_N child node blocks, we record (i) the starting index of the sum node block, (ii) the set of initial indices of its C_N child node blocks, and (iii) the corresponding set of C_N parameter indices (that point to the first parameter in the respective block of parameters in $\boldsymbol{\theta}_{\text{flat}}$). These parameter indices each denote the starting point for the $K_M \times K_N$ parameters of the corresponding pair of sum and product node blocks. Let C_M represent the total number of node blocks in the group. Following the indices described above, we record the following tensors: $\texttt{sum\_ids} \in \mathbb{Z}^{C_M}$ containing indices of all sum node blocks; $\texttt{prod\_ids}, \texttt{param\_ids} \in \mathbb{Z}^{C_M \times C_N}$, whose ith row represent the child indices and parameter indices of the ith sum node block (i.e., the node block with the start index $\texttt{sum\_ids}[i]$), respectively.

Figure 4.5 (right) illustrates the compiled index tensors of the sum layer shown on the left. Recall that we use the block sizes $K_M = K_N = 2$. The layer is then divided into two groups: the first group including two sum node blocks, $\{m_0, m_1\}$ and $\{m_4, m_5\}$, each having one child node block, and the second group including one sum node block, $\{m_2, m_3\}$, which has two child node blocks. Take, for instance, the first group. `sum_ids` stores the start indices (i.e., m_0 and m_4) of the two sum node blocks. `prod_ids` stores the initial indices of the child node blocks (i.e., n_0 and n_4) of the two sum node blocks, respectively. `param_ids` encodes the corresponding initial parameter indices θ_0 and θ_2.

Partitioning a layer into groups with the same number of children allows us to use different kernel launching hyperparameters according to the specific setup of every node group (e.g., number of nodes) to achieve better performance.

For every group in a sum layer, the three index tensors serve as inputs to a CUDA kernel computing the log-probabilities of the sum nodes in the group. Define $\boldsymbol{l}_{\text{prod}} \in \mathbb{R}^{N \times B}$ and $\boldsymbol{l}_{\text{sum}} \in \mathbb{R}^{M \times B}$ (B is the batch size) as the set of input and output log-probabilities, respectively. Consider a group with C_M sum node blocks and C_N child node blocks per sum node block. Algorithm 9 computes the log-probabilities of the C_M sum node blocks and stores the results in the proper locations in $\boldsymbol{l}_{\text{sum}}$. Specifically, we also divide the B samples into blocks of size K_B, leading to $C_B := B/K_B$ blocks (assume w.l.o.g. that B is divisible by K_B). Algorithm 9 schedules to launch $C_M \times C_B$ thread-blocks, each responsible for computing $K_M \times K_B$ outputs (line 6). The main loop in line 9 iterates over all C_N child node blocks. In every step, we first load the corresponding parameter matrix $\boldsymbol{\theta} \in \mathbb{R}^{K_M \times K_N}$ (line 11) and input matrix $\boldsymbol{l} \in \mathbb{R}^{K_N \times K_B}$ (line 12). Since $\boldsymbol{l}$ contains log-probabilities, we apply a variant of the logsumexp trick: we first convert $\boldsymbol{l}$ to the arithmetic space by subtracting the per-sample maximum log-probability (lines 13-14), then compute the (partial) output probabilities from the current set of $K_M \times K_N$ edges via matrix multiplication (line 15), and in line 16 aggregate the results back to the accumulator `cum` defined in line 7. Finally, we store the log-probabilities to the target locations in $\boldsymbol{l}_{\text{sum}}$ (line 17).

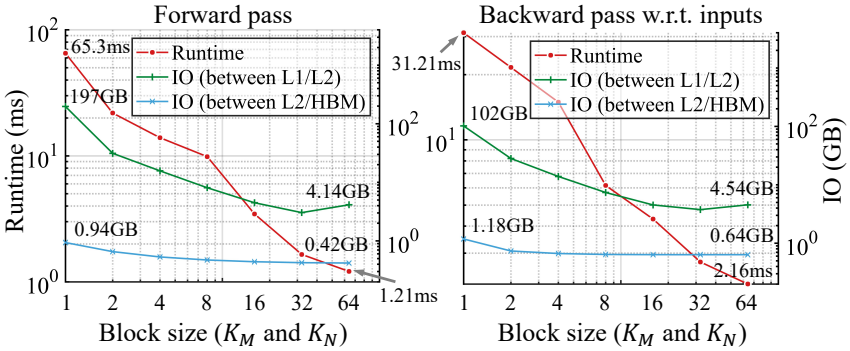


Figure 4.6: Runtime and IO overhead of a sum layer from the PD structure (with 29K nodes and 30M edges). The results demonstrate significant performance gains from our block-based parallelization, even with small block sizes.

4.2.3.4 Analysis: IO and Computation Overhead

We analyze the efficiency and IO complexity of our block-based parallelization strategy. Specifically, we benchmark on the largest sum layer in the PD structure adopted in Sec. 4.2.2. The layer consists of 29K nodes and 30M edges. In addition to the computation time, we record two types of IO overhead: (i) the IO between the L1/texture cache and the L2 cache, and (ii) the reads/writes between the L2 cache and the GPU High-Bandwidth Memory (HBM). We vary the block sizes K_M and K_N exponentially from 1 to 64. To ensure a fair comparison, we implement a dedicated kernel for $K_M = K_N = 1$, which directly parallelizes over sum node/sample pairs, allowing for better workload allocation. For other block sizes, we adjust K_B and other kernel launching hyperparameters (e.g., warps per block) and report the best runtime for every case. Results of the backward pass (w.r.t. inputs) are also reported for completeness.

Results are shown in Figure 4.6. As the block size increases, both the forward and the backward pass become significantly faster. Notably, this is accompanied by a significant drop in IO overhead. Specifically, with a large block size, the kernel consumes 2x fewer reads/writes between the L2 cache and the HBM, and 25-50x fewer IO between the L1 and L2 cache. This corroborates the hypothesis stated in Sec. 4.2.2 that the extensive value reloads significantly slow down the computation.

Additionally, we note that even with small block sizes (e.g., 2 or 4), the speedup is quite significant compared to the baseline case ($K_M = K_N = 1$), which allows us to speed up *sparse* PCs. Specifically, with the observation that every sparse PC can be viewed as a block-sparse PC with block size 1, we can transform a sparse PC into a block-sparse one, and pad zero parameters to edges belonging to the block-sparse PC but not the sparse PC. For PCs with relatively regular sparsity patterns, increasing the block sizes to even small values like 2 or 4 can lead to significant speedup even though a relatively large number of pseudo edges need to be padded.

the speedup obtained by having a larger block size outpaces the overhead caused by padded edges with zero parameters, which leads to speed-ups.

4.2.4 Optimizing Backpropagation with PC Flows

The previous section focuses on speeding up sum layers by reducing excessive memory reloads and leveraging Tensor Cores. However, when it comes to backpropagation, directly adapting Algorithm 9 by differentiating lines 13-16 would lead to poor performance due to the following. First, we need to either store some intermediate values (e.g., $\boldsymbol{l}_{\max}$ and $\boldsymbol{p}_{\mathrm{p}}$) in the forward pass or recompute them in the backward pass. Next, since different thread-blocks could access the same product node log-probabilities in line 12, they both need to write (partial) gradients of it, which introduces inter-thread-block barriers that slow down the execution.

We overcome the problems by leveraging PC flows [Choi et al., 2021], which is only a factor of $\theta_{n,c}$ away from the desired gradients (Eq. 4.5). PC flows exhibit a straightforward recursive definition, facilitating a seamless transformation into an efficient implementation for the backward pass.

Definition 4.2 (PC flows)**.** *For a PC $p_{n_r}(\mathbf{X})$ rooted at node n_r and a sample $\boldsymbol{x}$, the flow $\mathrm{F}_n(\boldsymbol{x})$ of every node n is defined recursively as follows (assume that no consecutive sum nodes or product*

nodes exist in the PC).[6]

$$\mathrm{F}_n(\boldsymbol{x}) := \begin{cases} 1 & n \textit{ is the root node}, \\ \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) & n \textit{ is input or sum}, \\ \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) & n \textit{ is a product node}, \end{cases}$$

where $\mathsf{pa}(n)$ *is the set of parents of* n*. Similarly, the edge flow* $\mathrm{F}_{n,c}(\boldsymbol{x})$ *w.r.t. the sample* $\boldsymbol{x}$ *(*$c \in \mathsf{ch}(n)$*) is defined as*

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x}) / p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

While similar results have been established in a slightly different context [Peharz et al., 2020a], we prove the following equations in Sec. C.1.4.2 for completeness:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})} \text{ and } \mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}}.$$

Following Definition 4.2, we can compute $F_n(\boldsymbol{x})$ for every node n utilizing the same set of layers created for the feedforward pass. Specifically, we first set the flow of the root node to 1 following its definition. We then iterate through the layers in reverse order (i.e., parent layers before child layers). While processing a layer, all flows of the nodes in the layer are computed by the preceding layers. And our goal is to compute the (partial) flows of the child nodes of the layer. Similar to the forward pass, we compile every layer by grouping child node blocks with a similar number of parents, and use block-based parallelization to reduce reloads of parent log-probabilities. We provide the full details of the backpropagation algorithm in Sec. C.1.2.

Another important design choice that leads to a significant reduction in memory footprint is to recompute the product nodes' probabilities in the backward pass instead of storing them all in the GPU memory during the forward pass. Specifically, we maintain a scratch space on GPU HBM that can hold the results of the largest product layer. All product layers write their outputs to this same scratch space, and the required product node probabilities are re-computed when requested by

[6] If such nodes exist, we can always collapse them into a single sum or product node.

a sum layer during backpropagation. Since product layers are extremely fast to evaluate compared to the sum layers (e.g., see the runtime breakdown in Fig. 4.3), this leads to significant memory savings at the cost of slightly increased computation time.

4.2.5 Experiments

We evaluate the impact of using PyJuice to train PC models. In Sec. 4.2.5.1, we compare PyJuice against existing implementations regarding time and memory efficiency. To demonstrate its generality and flexibility, we evaluate PyJuice on four commonly used dense PC structures as well as highly unstructured and sparse PCs. Next, we demonstrate that PyJuice can be readily used to scale up PCs for various downstream applications in Sec. 4.2.5.2. Finally, in Sec. 4.2.5.3, we benchmark existing PCs on high-resolution image datasets, hoping to incentivize future research to develop better PC structures as well as learning algorithms.

4.2.5.1 Faster Models with PyJuice

We first benchmark the runtime of PyJuice on four commonly used PC structures: PD [Poon and Domingos, 2011], RAT-SPN [Peharz et al., 2020b], HCLT [Liu and Van den Broeck, 2021], and HMM [Rabiner and Juang, 1986]. For all models, we record the runtime to process 60,000 samples (including the forward pass, the backward pass, and mini-batch EM updates). We vary their structural hyperparameters and create five PCs for every structure with sizes (i.e., number of edges) ranging from 500K to 2B. We compare against four baselines: SPFlow [Molina et al., 2019], EiNet [Peharz et al., 2020a], Juice.jl [Dang et al., 2021], and Dynamax [Murphy et al., 2023]. Dynamax is dedicated to State Space Models so it is only used to run HMMs; SPFlow and EiNet are excluded in the HMM results because we are unable to construct homogeneous HMMs with their frameworks due to the need to share the transition and emission parameters at different time steps. We describe how PyJuice implements PCs with tied parameters in Sec. C.1.3. All experiments in this subsection are carried out on an RTX 4090 GPU with 24GB memory.

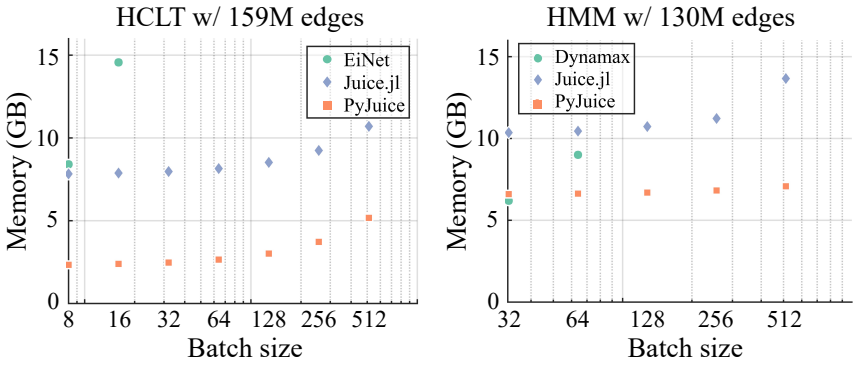


Figure 4.7: Comparison on memory efficiency. We take two PCs (i.e., an HCLT w/ 159M edges and an HMM w/ 130M edges) and record GPU memory usage under different block sizes.[7]

Table 4.1 reports the runtime in seconds per epoch with mini-batch EMs. PyJuice is orders of magnitude faster than all baselines in both small and large PCs. Further, we observe that most baselines exhaust 24GB of memory for larger PCs (indicated by "OOM" in the table), while PyJuice can still efficiently train these models. Additionally, in Sec. C.1.6.1, we show the efficiency of the compilation process. For example, it takes only ~ 8.7s to compile an HCLT with 159M edges. Note that we only compile the PC once and then reuse the compiled structure for training and inference.

In Figure 4.7, we take two PCs to show the GPU memory consumption with different batch sizes. The results demonstrate that PyJuice is more memory efficient than the baselines, especially in the case of large batch sizes (note that we always need a constant-size space to store the parameters).

We move on to benchmark PyJuice on block-sparse PCs. We create a sum layer with 209M edges (see Appx. C.1.5.1 for details). We partition the sum and input product nodes in the layer into blocks of 32 nodes respectively. We randomly discard blocks of 32×32 edges, resulting in block-sparse layers. As shown in Figure 4.8, as the fraction of removed edge blocks increases, the runtime of both the forward and the backward pass decreases significantly.

[7] In the adopted HMM, running Dynamax with batch size ≥ 128 leads to internal errors, and thus the results are not reported.

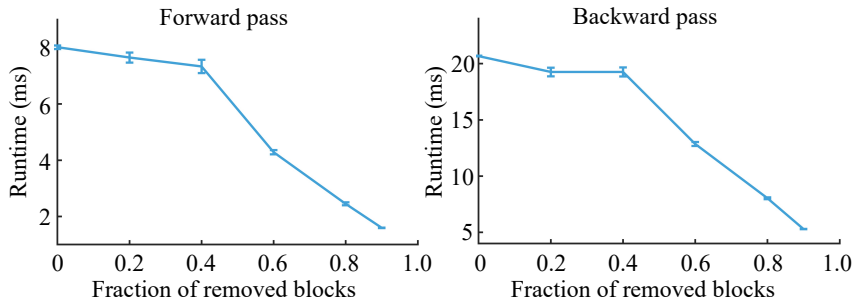


Figure 4.8: Runtime of a block-sparse sum layer as the function of the fraction of kept (non-dropped) edge blocks. The error bars represent standard deviations over 5 runs.

Finally, we proceed to evaluate the runtime of sparse PCs. We adopt the PC pruning algorithm proposed by Dang et al. [2022a] to prune two HCLTs with 10M and 40M edges, respectively. We only compare against Juice.jl since all other implementations do not support sparse PCs. As shown in Figure 4.9, PyJuice is consistently faster than Juice.jl, despite the diminishing gap when over 90% edges are pruned. Note that with sparse PCs, PyJuice cannot fully benefit from the block-based parallelization strategy described in Sec. 4.2.3, yet it can still outperform the baseline.

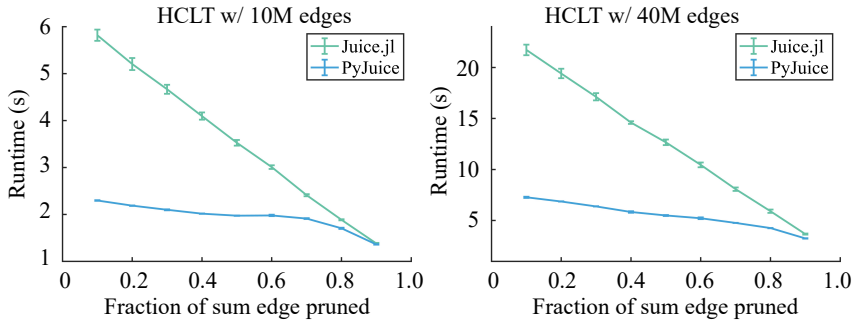


Figure 4.9: Runtime per epoch (with 60K samples) of two sparse HCLTs with different fractions of pruned edges. The error bars represent standard deviations over 5 runs.

4.2.5.2 Better PCs At Scale

This section demonstrates the ability of PyJuice to improve the state of the art by simply using larger PCs and training for more epochs thanks to its speed and memory efficiency. Specifically, we take the HMM language model proposed by Zhang et al. [2023b] and the image model introduced

Table 4.2: Perplexity of HMM language models trained on the CommonGen benchmark [Lin et al., 2020].

	Zhang et al. [2023b]	PyJuice	
# hidden states	4096	4096	8192
Perplexity	9.78	8.81	**8.65**

by Liu et al. [2023e] as two examples.

HMM language models. Zhang et al. [2023b] use the Latent Variable Distillation (LVD) [Liu et al., 2023b] technique to train an HMM with 4096 hidden states on sequences of 32 word tokens. Specifically, LVD is used to obtain a set of "good" initial parameters for the HMM from deep generative models. The HMM language model is then fine-tuned on the CommonGen dataset [Lin et al., 2020], and is subsequently used to control the generation process of (large) language models for constrained generation tasks. Following the same procedure, we use PyJuice to fine-tune two HMMs with hidden sizes 4096 and 8192, respectively.

As shown in Table 4.2, by using the same HMM with 4096 hidden states, PyJuice improved the perplexity by ~ 1.0 by running many more epochs in less time compared to the original model. We also train a larger HMM with 8192 hidden states and further improved the perplexity by a further 0.16. We refer the reader to Sec. C.1.5.2 for more details.

Sparse Image Models. Liu et al. [2023e] design a PC learning algorithm that targets image data by separately training two sets of PCs: a set of sparse patch-level PCs (e.g., 4×4 patches) and a top-level PC that aggregates outputs of the patch-level PC. In the final training step, the PCs are supposed to be assembled and jointly fine-tuned. However, due to the huge memory consumption of the PC (with over 10M nodes), only the top-level model is fine-tuned in the original paper. With PyJuice, we can fit the entire model in 24GB of memory and fine-tune the entire model. For the PC trained on the ImageNet32 dataset [Deng et al., 2009], this fine-tuning step leads to an improvement from 4.06 to 4.04 bits-per-dimension. See Sec. C.1.5.3 for more details.

Table 4.3: Density estimation performance of PCs on three natural image datasets. Reported numbers are test set bits-per-dimension.

Dataset	PD-mid	PD-large	HCLT-mid	HCLT-large
ImageNet32	5.22	5.20	4.36	4.33
ImageNet	4.98	4.95	3.57	3.53
CelebA-HQ	4.35	4.29	2.43	2.38

4.2.5.3 Benchmarking Existing PCs

We use PyJuice to benchmark the performance of the PD and the HCLT structure on three natural image datasets: ImageNet [Deng et al., 2009] and its down-sampled version ImageNet32, and CelebA-HQ [Liu et al., 2015b]. For all three datasets, we train the PCs on randomly sampled 16×16 patches, which results in a total of $16\times16\times3=768$ categorical variables each with $2^8=256$ possible values. As a preprocessing step, the image patches are converted into the YCoCg color space since it is observed that such color space transformations lead to improved density estimation performance. Note that due to the lossy transformation between the RGB space and the YCoCg space, our results are not directly comparable to the results obtained from RGB images.

We adopt two PD structures (i.e., PD-mid with 107M edges and PD-large with 405M edges) as well as two HCLT structures (i.e., HCLT-mid with 40M edges and HCLT-large with 174M edges). Details of the adopted models are described in Sec. C.1.5.4. We experiment with different optimization strategies and adopt full-batch EM as it yields consistently better performance across models and datasets. Specifically, the computed PC flows are accumulated across all samples in the training set before doing one EM step.

Results are shown in Table 4.3. Notably, we achieve *better* results compared to previous papers. For example, Liu et al. [2023b] reports 4.82 bits-per-dimension (bpd) for HCLT on ImageNet32, while we achieved 4.33 bpd. The performance improvements stem from more training epochs and the ability to do more hyperparameter search thanks to the speedup. We highlight that the goal of this section is not to set new records for tractable deep generative models, but to establish a set of baselines that can be easily reproduced to track the progress of developments in PC modeling and

learning. In Sec. C.1.5.4, we include additional benchmark results on the WikiText dataset [Merity et al., 2016].

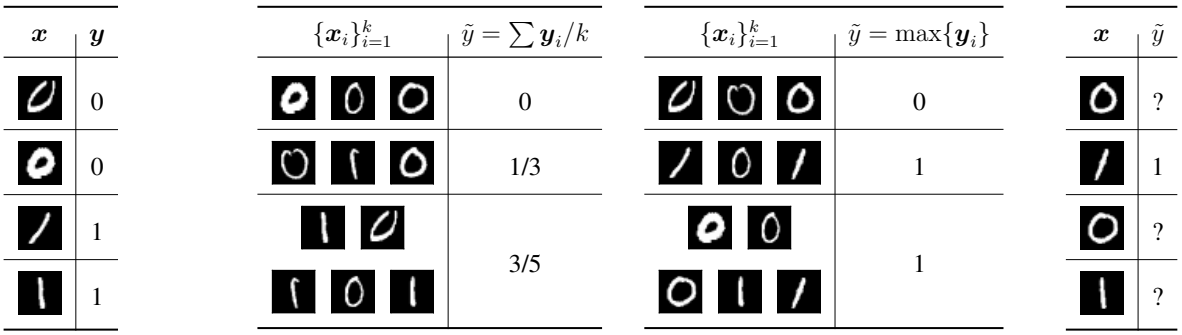
CHAPTER 5

Applications

The methods developed have seen many applications. One such application is weakly-supervised learning, where high-quality labels are often very scarce, whereas data with partial labels is more readily available due to privacy or budget constraints. These weak labels typically dictate the frequency of each respective class over a set of instances. The insight that we bring forth is that such weak supervision can very often be construed as enforcing constraints on label counts of data. At the heart of our approach is the ability to compute the probability of exactly k out of n outputs being set to true, as well as any symmetric functions thereof. Building upon the previous computation, we derive a count loss penalizing the model for deviations in its distribution from an arithmetic constraint defined over label counts. Another application is learning the structure of graph neural networks, where we proposed probabilistically rewired message-passing graph neural networks (PR-MPNNs). Building upon SIMPLE, we learn to add relevant edges while omitting less beneficial ones, sidestepping many of the pitfalls of state-of-the-art algorithms.

5.1 A Unified Approach to Count-Based Weakly-Supervised Learning

Weakly supervised learning [Zhou, 2018] enables a model to learn from data with restricted, partial or inaccurate labels, often known as *weakly-labeled data*. Weakly supervised learning fulfills a need arising in many real-world settings that are subject to privacy or budget constraints, such as privacy sensitive data [Wojtusiak et al., 2011], medical image analysis [Bortsova et al., 2018], clinical practice [Quellec et al.], personalized advertisement [Bekker and Davis, 2020] and knowledge base completion [Galárraga et al., 2015; Zupanc and Davis, 2018], to name a few. In some settings,



(a) Classical (b) LLP (c) MIL (d) PU Learning

Table 5.1: A comparison of the tasks considered in the three weakly supervised settings, LLP (cf. Section 5.1.1.1), MIL (cf. Section 5.1.1.2) and PU learning (cf. Section 5.1.1.3), against the classical fully supervised setting for binary classification, using digits from the MNIST dataset.

instance-level labels are unavailable. Instead, instances are grouped into *bags* with corresponding *bag-level labels* that are a function of the instance labels, e.g., the proportion of positive labels in a bag. A key insight that we bring forth is that such weak supervision can very often be construed as *enforcing constraints on label counts of data*.

More concretely, we consider three prominent weakly supervised learning paradigms. The first paradigm is known as *learning from label proportions* [Quadrianto et al., 2008]. Here the weak supervision consists in the *proportion* of positive labels in a given bag, which can be interpreted as *the count of positive instances* in such a bag. The second paradigm, whose supervision is strictly weaker than the former, is *multiple instance learning* [Maron and Lozano-Pérez, 1997; Dietterich et al., 2001]. Here the bag labels only indicate the *existence* of at least one positive instance in a bag, which can be recast as to whether *the count of positive instances* is greater than zero. The third paradigm, *learning from positive and unlabeled data* [De Comité et al., 1999; Letouzey et al., 2000], grants access to the ground truth labels for a subset of *only the positive instances*, providing only a class prior for what remains. We can recast the class prior as *a distribution of the count of positive labels*.

Leveraging the view of weak supervision as a constraint on label counts, we utilize a simple, efficient and probabilistically sound approach to weakly-supervised learning. More precisely, we train a neural network to make instance-level predictions that conform to the desired label counts.

To this end, we propose a *differentiable count loss* that characterizes how close the network's distribution comes to the label counts; a loss which is surprisingly tractable. Compared to prior methods, this approach does not approximate probabilities but computes them *exactly*. Our empirical evaluation demonstrates that our proposed count loss significantly boosts the classification performance on all three aforementioned settings.

5.1.1 Problem Formulations

In this section, we formally introduce the aforementioned weakly supervised learning paradigms. For notation, let $\mathcal{X} \in \mathbb{R}^d$ be the input feature space over d features and $\mathcal{Y} = \{0, 1\}$ be a binary label space. We write $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{y} \in \mathcal{Y}$ for the input and output random variables respectively. Recall that in fully-supervised binary classification, it is assumed that each feature and label pair $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X} \times \mathcal{Y}$ is sampled independently from a joint distribution $p(\boldsymbol{x}, \boldsymbol{y})$. A classifier f is learned to minimize the risk $R(f) = \mathbf{E}_{(\boldsymbol{x},\boldsymbol{y})\sim p}[\ell(f(\boldsymbol{x}), \boldsymbol{y})]$ where $\ell : [0, 1] \times \mathcal{Y} \to \mathbb{R}_{\geq 0}$ is the cross entropy loss function. Typically, the true distribution $p(\boldsymbol{x}, \boldsymbol{y})$ is implicit and cannot be observed. Therefore, a set of n training samples, $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$, is used and the empirical risk, $\hat{R}(f) = \frac{1}{n}\sum_{i=1}^n \ell(f(\boldsymbol{x}_i), \boldsymbol{y}_i)$, is minimized in practice. In the count-based weakly supervised learning settings, the supervision is given at a bag level instead of an instance level. We formally introduce these settings as below.

5.1.1.1 Learning from Label Proportions

Learning from label proportions (LLP) [Quadrianto et al., 2008] assumes that each instance in the training set is assigned to bags and only the proportion of positive instances in each bag is known. One example is in light of the coronavirus pandemic, where infection rates were typically reported based on geographical boundaries such as states and counties. Each boundary can be treated as a bag with the infection rate as the proportion annotation.

The goal of LLP is to learn an instance-level classifier $f : \mathcal{X} \to [0, 1]$ even though it is

trained on bag-level labeled data. Formally, the training dataset consists of m bags, denoted by $\mathcal{D} = \{(B_i, \tilde{y}_i)\}_{i=1}^{m}$ where each bag $B_i = \{\boldsymbol{x}_j\}_{j=1}^{k}$ consist of k instances and this k could vary among different bags. The bag proportions are defined as $\tilde{y}_i = \sum_{j=1}^{k} \boldsymbol{y}_j / k$ with $\boldsymbol{y}_j$ being the instance label that cannot be accessed and only $\tilde{y}_i$ is available during training. An example is shown in Figure 5.1b. We do not assume that the bags are non-overlapping while some existing work suffers from this limitation including Scott and Zhang [2020].

5.1.1.2 Multiple Instance Learning

Multiple instance learning (MIL) [Maron and Lozano-Pérez, 1997; Dietterich et al., 2001] refers to the scenario where the training dataset consists of bags of instances, and labels are provided at bag level. However, in MIL, the bag label is a single binary label indicating whether there is a positive instance in the bag or not as opposed to a bag proportion defined in LLP. A real-world application of MIL lies in the field of drug activity [Dietterich et al., 2001]. We can observe the effects of a group of conformations but not for any specific molecule, motivating a MIL setting. Formally, in MIL, the training dataset consists of m bags, denoted by $\mathcal{D} = \{(B_i, \tilde{y}_i)\}_{i=1}^{m}$, with a bag consisting of k instances, i.e., $B_i = \{\boldsymbol{x}_j\}_{j=1}^{k}$. The size k can vary among different bags. For each instance $\boldsymbol{x}_j$, there exists an instance-level label $\boldsymbol{y}_j$ which is not accessible. The bag-level label is defined as $\tilde{y}_i = \max_j\{\boldsymbol{y}_j\}$. An example is shown in Figure 5.1c.

The main goal of MIL is to learn a model that predicts a bag label while a more challenging goal is to learn an instance-level predictor that is able to discover positive instances in a bag. In this work, we aim to tackle both by training an instance-level classifier whose predictions can be combined into a bag-level prediction as the last step.

5.1.1.3 Learning from Positive and Unlabeled Data

Learning from positive and unlabeled data or *PU learning* [De Comité et al., 1999; Letouzey et al., 2000] refers to the setting where the training dataset consists of only positive instances

Table 5.2: A summary of the labels and objective functions for all the settings considered in the paper.

TASK	LABEL	LABEL LEVEL	OBJECTIVE
Classical Fully Supervised	Binary $\boldsymbol{y}$	Instance Level	$-\boldsymbol{y}\log p(\boldsymbol{y}) - (1-\boldsymbol{y})\log(1-p(\boldsymbol{y}))$
Learning from Label Proportion	Continuous $\tilde{y} = \sum_i \boldsymbol{y}_i/k$	Bag Level	$-\log p(\sum \hat{y}_i = k\tilde{y})$
Multiple Instance Learning	Binary $\tilde{y} = \max\{y_i\}$	Bag Level	$-\tilde{y}\log p(\sum \hat{y}_i \geq 1) - (1-\tilde{y})\log p(\sum_i \hat{y}_i = 0)$
Learning from Positive and Unlabeled Data	Binary $\tilde{y}$	Instance Level	1) $\mathbb{D}_{KL}(\texttt{Bin}(k,\beta) \parallel p(\sum_i \hat{y}_i))$ 2) $-\log p(\sum \hat{y}_i = k\beta)$

and unlabeled data, and the unlabeled data can contain both positive and negative instances. A motivation of PU learning is persistence in the case of shifts to the negative-class distribution [Plessis et al., 2015], for example, a spam filter. An attacker may alter the properties of a spam email, making a traditional classifier require a new negative dataset [Plessis et al., 2015]. We note that taking a new unlabeled sample would be more efficient, motivating PU learning. Formally, in PU learning, the training dataset $\mathcal{D} = \mathcal{D}_p \cup \mathcal{D}_u$ where $\mathcal{D}_p = \{(\boldsymbol{x}_i, \tilde{y}_i = 1)\}_{i=1}^{n_p}$ is the set of positive instances with $\boldsymbol{x}_i$ from $p(\boldsymbol{x} \mid \boldsymbol{y} = 1)$ and $\tilde{y}$ denoting whether the instance is labeled, and $\mathcal{D}_u = \{(\boldsymbol{x}_i, \tilde{y}_i = 0)\}_{i=1}^{n_u}$ the unlabeled set with $\boldsymbol{x}_i$ from

$$p_u(\boldsymbol{x}) = \beta\, p(\boldsymbol{x} \mid \boldsymbol{y} = 1) + (1-\beta)\, p(\boldsymbol{x} \mid \boldsymbol{y} = 0), \tag{5.1}$$

where the mixture proportion $\beta := p(\boldsymbol{y} = 1 \mid \tilde{y} = 0)$ is the fraction of positive instances among the unlabeled population. Although the instance label $\boldsymbol{y}$ is not accessible, its information can be inferred from the binary selection label $\tilde{y}$: if the selection label $\tilde{y} = 1$, it belongs to the positively labeled set, i.e., $p(\boldsymbol{y} = 1 \mid \tilde{y} = 1) = 1$; otherwise, the instance $\boldsymbol{x}$ can be either positive or negative. An example of such a dataset is shown in Figure 5.1d.

The goal of PU learning is to train an instance-level classifier. However, it is not straightforward to learn from PU data and it is necessary to make assumptions to enable learning with positive and unlabeled data [Bekker and Davis, 2020]. In this work, we make a commonly-used assumption for PU learning, *selected completely at random (SCAR)*, which lies at the basis of many PU learning methods.

Algorithm 10 Count Probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$

Input: A set of k log probabilities $\{t_i\}_{i=1}^{k}$ with $t_i := \log p(\hat{y}_i = 1)$, the number of instances k, and a label sum s

Output: log probabilities $\log p(\sum_{i=1}^{k} \hat{y}_i = s)$ or a set of log probability $\{\log p(\sum_{i=1}^{k} \hat{y}_i = s)\}_{s=0}^{k}$

```
// A[i, m] = log p(Σ_{j=1}^{i} y_j = m) ∀i, m
Initialize an array A to be −Inf everywhere
A[0, 0] = 0    // p(Σ_{j=1}^{0} y_j = 0) = 1
Compute t'_i ← log1mexp(t_i) // log p(y_i = 0)
for i = 1 to k do
  for m = 0 to s do
    a_+ = A[i − 1, m − 1] + t_i
    a_− = A[i − 1, m] + t'_i
    A[i, m] = logsumexp(a_+, a_−)
return A[k, s] or A[k, :]
```

Definition 11 (SCAR). *Labeled instances are selected completely at random, independent from input features, from the positive distribution* $p(\boldsymbol{x} \mid \boldsymbol{y} = 1)$*, that is,* $p(\tilde{y} = 1 \mid \boldsymbol{x}, \boldsymbol{y} = 1) = p(\tilde{y} = 1 \mid \boldsymbol{y} = 1)$.

5.1.2 A Unified Approach: Count Loss

In this section, we derive objectives for the three weakly supervised settings, LLP, MIL, and PU learning, from first principles. Our proposed objectives bridge between neural outputs, which can be observed as counts, and arithmetic constraints derived from the weakly supervised labels. The idea is to capture how close the classifier is to satisfying the arithmetic constraints on its outputs. They can be easily integrated with deep learning models, and allow them to be trained end-to-end. For the three objectives, we show that they share the same computational building block: given k instances $\{\boldsymbol{x}_i\}_{i=1}^{k}$ and an instance-level classifier f that predicts $p(\hat{y}_i \mid \boldsymbol{x}_i)$ with $\hat{y}$ denoting the prediction variable, the problem of inferring the probability of the constraint on counts $\sum_{i=1}^{k} \hat{y}_i = s$

is to compute the count probability defined below:

$$p(\sum_{i=1}^{k} \hat{y}_i = s \mid \{\boldsymbol{x}_i\}_{i=1}^{k}) := \sum_{\hat{\boldsymbol{y}} \in \mathcal{Y}^k} [\![\sum_{i=1}^{k} \hat{y}_i \; = \; s]\!] \prod_{i=1}^{k} p(\hat{y}_i \mid \boldsymbol{x}_i)$$

where $[\![\cdot]\!]$ denotes the indicator function and $\hat{\boldsymbol{y}}$ denotes the vector $(\hat{y}_1, \cdots, \hat{y}_k)$. For succinctness, we omit the dependency on the input and simply write the count probability as $p(\sum_{i=1}^{k} \hat{y}_i = s)$. Next, we show how the objectives derived from first principles can be solved by using the count probability as an oracle. We summarize all proposed objectives in Table 5.2. Later, we will show how this seemingly intractable count probability can be efficiently computed by our proposed algorithm.

LLP setting. Given a bag $B = \{\boldsymbol{x}_i\}_{i=1}^{k}$ of size k and its weakly supervised label $\tilde{y}$, by definition, it can be inferred that the number of positive instances (count) in the bag is $k\tilde{y}$. Our objective is to minimize the negative log probability $-\log p(\sum_i \hat{y}_i = k\tilde{y})$. Notice that when each bag consists of only one instance, that is, when the bag-level supervisions are reduced to instance-level ones, this objective is exactly cross-entropy loss. We further show that our method is risk-consistent, that is, the optimal classifier under our proposed loss provides predictions consistent with the underlying risk as in the supervised learning setting. Details of the risk analysis can be found in Appendix D.1.1.

MIL setting. Given a bag $B = \{\boldsymbol{x}_i\}_{i=1}^{k}$ of size k and a single binary label $\tilde{y}$ as its weakly supervised label, we propose a cross-entropy loss as below

$$\ell(B, \tilde{y}) = -\tilde{y} \log p(\sum \hat{y}_i \geq 1) - (1 - \tilde{y}) \log p(\sum \hat{y}_i = 0).$$

Notice that in the above loss, the probability term $p(\sum \hat{y}_i = 0)$ is accessible to the oracle for computing count probability, and the other probability term $p(\sum \hat{y}_i \geq 1)$ can simply be obtained from $1 - p(\sum \hat{y}_i = 0)$, i.e., the same call to the oracle since all prediction variables $\hat{y}_i$ are binary.

PU Learning setting. Recall that for the unlabeled data $\mathcal{D}_u$ in the training dataset, an unlabeled

instance $\boldsymbol{x}_i$ is drawn from a mixture distribution as shown in Equation 5.1 parameterized by a mixture proportion $\beta = p(\boldsymbol{y} = 1 \mid \tilde{y} = 0)$. Under the SCAR assumption, even though only a class prior is given, we show that the mixture proportion can be estimated from the dataset.

Proposition 5.1. *With SCAR assumption and a class prior* $\alpha := p(\boldsymbol{y} = 1)$*, the mixture proportion* $\beta := p(\boldsymbol{y} = 1 \mid \tilde{y} = 0)$ *can be estimated from dataset* $\mathcal{D}$*.*

Proof. First, the label frequency $p(\tilde{y} = 1 \mid \boldsymbol{y} = 1)$ denoted by c can be obtained by

$$c = \frac{p(\tilde{y} = 1, \boldsymbol{y} = 1)}{p(\boldsymbol{y} = 1)} = \frac{p(\tilde{y} = 1)}{p(\boldsymbol{y} = 1)} \text{ (by the definition of PU learning).}$$

that is, $c = p(\tilde{y} = 1)/\alpha$. Notice that $p(\tilde{y} = 1)$ can be estimated from the dataset $\mathcal{D}$ by counting the proportion of the labeled instances. Thus, we can estimate the mixture proportion as below,

$$\beta = \frac{p(\tilde{y} = 0 \mid \boldsymbol{y} = 1)p(\boldsymbol{y} = 1)}{p(\tilde{y} = 0)} = \frac{(1 - p(\tilde{y} = 1 \mid \boldsymbol{y} = 1))p(\boldsymbol{y} = 1)}{1 - p(\tilde{y} = 1)} = \frac{(1 - c)\alpha}{1 - \alpha c}.$$

□

The probabilistic semantic of the mixture proportion is that if we randomly draw an instance $\boldsymbol{x}_i$ from the unlabeled population, the probability that the true label $\boldsymbol{y}_i$ is positive would be β. Further, if we randomly draw k instances, the distribution of the summation of the true labels $\sum_{i=1}^{k} \boldsymbol{y}_i$ conforms to a binomial distribution $\texttt{Bin}(k, \beta)$ parameterized by the mixture proportion β, i.e.,

$$p(\sum_{i=1}^{k} \boldsymbol{y}_i = s) = \binom{k}{s} \beta^s (1 - \beta)^{k-s}. \tag{5.2}$$

Based on this observation, we propose an objective to minimize the KL divergence between the distribution of predicted label sum and the binomial distribution parameterized by the mixture proportion for a random subset drawn from the unlabeled population, that is,

$$\mathbb{D}_{KL}\left(\texttt{Bin}(k, \beta) \parallel p(\sum_{i=1}^{k} \hat{y}_i)\right) = \sum_{s=0}^{k} \texttt{Bin}(s; k, \beta) \log \frac{\texttt{Bin}(s; k, \beta)}{p(\sum_{i=1}^{k} \hat{y}_i = s)}$$

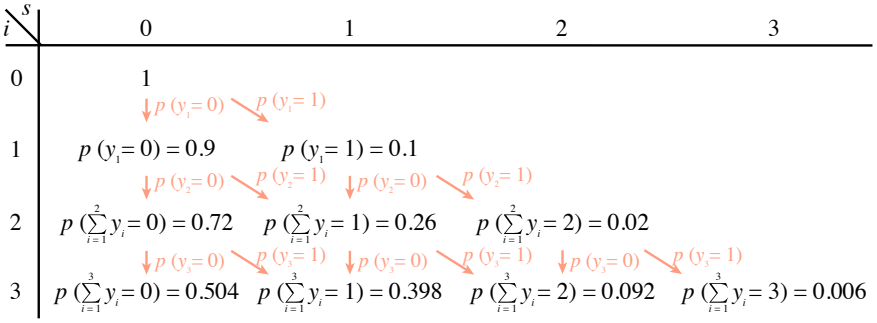


Figure 5.1: An example of how to compute the count probability in a dynamic programming manner. Assume that an instance-level classifier predicts three instances to have $p(\boldsymbol{y}_1 = 1) = 0.1$, $p(\boldsymbol{y}_2 = 1) = 0.2$, and $p(\boldsymbol{y}_3 = 1) = 0.3$ respectively. The algorithm starts from the top-left cell and propagates the results down right. A cell has its probability $p(\sum_{j=0}^{i} \boldsymbol{y}_j = s)$ computed by inputs from $p(\sum_{j=0}^{i-1} \boldsymbol{y}_j = s)$ weighted by $p(\boldsymbol{y}_i = 0)$, and $p(\sum_{j=0}^{i-1} \boldsymbol{y}_j = s - 1)$ weighted by $p(\boldsymbol{y}_i = 1)$ respectively, as indicated by the arrows.

where $\texttt{Bin}(s; k, \beta)$ denotes the probability mass function of the binomial distribution $\texttt{Bin}(k, \beta)$. Again, the KL divergence can be obtained by $k + 1$ calls to the oracle for computing count probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$. The KL divergence is further combined with a cross entropy defined over labeled data $\mathcal{D}_p$ as in the classical binary classification training as the overall objective.

As an alternative, we propose an objective for the unlabeled data that requires fewer calls to the oracle: instead of matching the distribution of the predicted label sum with the binomial distribution, this objective matches only the expectations of the two distributions, that is, to maximize $p(\sum_{i=1}^{k} \hat{y}_i = k\beta)$ where $k\beta$ is the expectation of the binomial distribution $\texttt{Bin}(k, \beta)$. We present empirical evaluations of both proposed objectives in the experimental section.

5.1.3 Tractable Computation of Count Probability

In the previous section, we show how the count probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$ serves as a computational building block for the objectives derived from first principles for the three weakly supervised learning settings. With a closer look at the count probability, we can see that given a set of instances, the classifier predicts an instance-level probability for each and it requires further manipulation to obtain count information; actually, the number of joint labelings for the set can be exponential in

the number of instances. Intractable as it seems, we show that it is indeed possible to derive a tractable computation for the count probability based on a result from Ahmed et al. [2023c].

Proposition 5.2. *The count probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$ of sampling k prediction variables that sums to s from an unconstrained distribution $p(\boldsymbol{y}) = \prod_{i=1}^{k} p(\hat{y}_i)$ can be computed exactly in time $\mathcal{O}(ks)$. Moreover, the set $\{p(\sum_{i=1}^{k} \hat{y}_i = s)\}_{s=0}^{k}$ can also be computed in time $\mathcal{O}(k^2)$.*

The above proposition can be proved in a constructive way where we show that the count probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$ can be computed in a dynamic programming manner. We provide an illustrative example of this computation in Figure 5.1. In practice, we implement this computation in log space for numeric stability which we summarized as Algorithm 10, where function `log1mexp` provides a numerically stable way to compute $\texttt{log1mexp}(x) = \log(1 - \exp(x))$ and function `logsumexp` a numerically stable way to compute $\texttt{logsumexp}(x, y) = \log(\exp(x) + \exp(y))$. Notice that since we show it is tractable to compute the set $\{p(\sum_{i=1}^{k} \hat{y}_i = s)\}_{s=0}^{k}$, for any two given label sum s_1 and s_2, a count probability $p(s_1 \leq \sum_i \hat{y}_i \leq s_2)$ where the count lies in an interval, can also be exactly and tractably computed. This implies that our tractable computation of count probabilities can potentially be leveraged by other count-based applications besides the three weakly supervised learning settings in the last section.

5.1.4 Related Work

Weakly Supervised Learning. Besides settings explored in our work there are many other weakly-supervised settings. One of which is semi-supervised learning, a close relative to PU Learning with the difference being that labeled samples can be both positive and negative [Zhu and Goldberg, 2022; Zhu, 2005]. Another is label noise learning, which occurs when our instances are mislabeled. Two common variations involve whether noise is independent or dependent on the instance [Frénay and Verleysen, 2013; Song et al., 2022]. A third setting is partial label learning, where each instance is provided a set of labels of which exactly one is true [Cour et al., 2011a]. An extension of this is partial multi-label learning, where among a set of labels, a subset is true [Xie

and Huang, 2018].

Unified Approaches. There exists some literature in regards to “general" approaches for weakly supervised learning. One example being the method proposed in Hüllermeier [2014], which provides a procedure that minimizes the empirical risk on “fuzzy” sets of data. The paper also establishes guarantees for model identification and instance-level recognition. Co-Training and Self-Training are also examples of similar techniques that are applicable to a wide variety of weakly supervised settings [Blum and Mitchell, 1998; Yarowsky, 1995]. Self-training involves progressively incorporating more unlabeled data via our models prediction (with pseudo-label) and then training a model on more data as an iterative algorithm [Karamanolakis et al., 2021]. Co-Training leverages two models that have different views of the data and iteratively augment each other’s training set with samples they deem as well-classified. They are traditionally applied to semi-supervised learning but can extend to multiple instance learning settings [Lu et al., 2011; Xu et al., 2013; Liu et al., 2023c].

LLP. Quadrianto et al. [2008] first introduced an exponential family based approach that used an estimation of mean for each class. Others seek to minimize “empirical proportion risk” or EPR as in Yu et al. [2014], which is centered around creating an instance-level classifier that is able to reproduce the label proportions of each bag. As mentioned previously, more recent methods use bag posterior approximation and neural-based approaches [Ardehaly and Culotta, 2017; Tsai and Lin, 2020]. One such method is Proportion Loss (PL) [Tsai and Lin, 2020], which we contrast to our approach. This is computed by binary cross entropy between the averaged instance-level probabilities and ground-truth bag proportion.

MIL. MIL finds its earlier approaches with SVMs, which have been used quite prolifically and still remain one of the most common baselines. We start with MI-SVM/mi-SVM [Andrews et al., 2002] which are examples of transductive SVMs [Carbonneau et al., 2018] that seek a stable instance classification through repeated retraining iterations. MI-SVM is an example of an instance space method [Carbonneau et al., 2018], which identifies methods that classify instances as a preliminary step in the problem. This is in contrast to bag-space or embedded-space methods that

omit the instance classification step. Furthermore, Wang et al. [2018] remains one of the hallmarks of the use of neural networks for Multi-Instance Learning. Ilse et al. [2018], utilize a similar approach but with attention-based mechanisms.

PU learning. Bekker and Davis [2020] groups PU Learning paradigms into three main classes: two step, biased, and class prior incorporation. Biased learning techniques train a classifier on the entire dataset with the understanding that negative samples are subject to noise [Bekker and Davis, 2020]. We will focus on a subset of biased learning techniques (Risk Estimators) as they are considered state-of-the-art and relevant to us as baselines. The Unbiased Risk Estimator (uPU) provides an alternative to the inefficiencies in manually biasing unlabeled data [du Plessis et al., 2014; Plessis et al., 2015]. Later, Non-negative Risk Estimator (nnPU) [Kiryo et al., 2017] accounted for weaknesses in the unbiased risk estimator such as overfitting.

Count Loss. To our knowledge, viewing the computation of the "bag posterior" as *probabilistic* is new. However, the prior approaches do this implicitly. Many approaches have tried to approximate the "bag posterior" by averaging the instance-level probabilities in a bag [Ardehaly and Culotta, 2017; Tsai and Lin, 2020]. In MIL settings, among instance-level approaches, the MIL-pooling is an implicit "bag posterior" computation. These include mean, max, and log-sum-exp pooling to approximate the likelihood that a bag has at least one positive instance [Wang et al., 2018]. But again, these are all approximations of what our computation does *exactly*. In PU Learning, to our best knowledge, the view of unlabeled data as a bag annotated with the mixture proportion is new.

Neuro-Symbolic Losses. In this paper, we have dealt with a specific form of distributional constraint. Conversely, there has been a plethora of work exploring the integration of *hard* symbolic constraints into the learning of neural networks. This can take the form of enforcing a hard constraint [Ahmed et al., 2022b], whereby the network's predictions are guaranteed to satisfy the pre-specified constraints. Or it can take the form of a soft constraint [Xu et al., 2018a; Manhaeve et al., 2018; Ahmed et al., 2021, 2022c,a, 2023a] whereby the network is trained with an additional loss term that penalizes the network for placing any probability mass on predictions that violate

Table 5.3: LLP results across different bag sizes. We report the mean and standard deviation of the test AUC over 5 seeds for each setting. The highest metric for each setting is shown in **boldface**.

Dataset	**Dist**	**Method**	8	32	128	512
Adult	$[0, \frac{1}{2}]$	PL	0.8889 ± 0.0024	0.8782 ± 0.0036	$\mathbf{0.8743 \pm 0.0039}$	0.8678 ± 0.0085
Adult	$[0, \frac{1}{2}]$	LMMCM	0.8728 ± 0.0019	0.8693 ± 0.0047	0.8669 ± 0.0041	0.8674 ± 0.0040
Adult	$[0, \frac{1}{2}]$	CL (Ours)	$\mathbf{0.8984 \pm 0.0013}$	$\mathbf{0.8848 \pm 0.0041}$	$\mathbf{0.8743 \pm 0.0052}$	$\mathbf{0.8703 \pm 0.0070}$
Adult	$[\frac{1}{2}, 1]$	PL	0.8781 ± 0.0038	0.8731 ± 0.0035	$\mathbf{0.8699 \pm 0.0057}$	0.8556 ± 0.0180
Adult	$[\frac{1}{2}, 1]$	LMMCM	0.8584 ± 0.0164	0.8644 ± 0.0052	0.8601 ± 0.0045	0.8500 ± 0.0186
Adult	$[\frac{1}{2}, 1]$	CL (Ours)	$\mathbf{0.8854 \pm 0.0022}$	$\mathbf{0.8738 \pm 0.0039}$	0.8675 ± 0.0043	$\mathbf{0.8607 \pm 0.0056}$
Adult	$[0, 1]$	PL	0.8884 ± 0.0030	0.8884 ± 0.0008	$\mathbf{0.8879 \pm 0.0025}$	$\mathbf{0.8828 \pm 0.0051}$
Adult	$[0, 1]$	LMMCM	0.8831 ± 0.0026	0.8819 ± 0.0006	0.8821 ± 0.0017	0.8786 ± 0.0052
Adult	$[0, 1]$	CL (Ours)	$\mathbf{0.8985 \pm 0.0010}$	$\mathbf{0.8891 \pm 0.0013}$	0.8871 ± 0.0021	0.8790 ± 0.0056
Magic	$[0, \frac{1}{2}]$	PL	0.8900 ± 0.0095	0.8510 ± 0.0032	0.8405 ± 0.0110	0.8332 ± 0.0149
Magic	$[0, \frac{1}{2}]$	LMMCM	0.8918 ± 0.0077	0.8799 ± 0.0113	0.8753 ± 0.0157	0.8734 ± 0.0092
Magic	$[0, \frac{1}{2}]$	CL (Ours)	$\mathbf{0.9088 \pm 0.0056}$	$\mathbf{0.8830 \pm 0.0097}$	$\mathbf{0.8926 \pm 0.0049}$	$\mathbf{0.8864 \pm 0.0107}$
Magic	$[\frac{1}{2}, 1]$	PL	0.9066 ± 0.0016	0.8818 ± 0.0108	0.8769 ± 0.0101	0.8429 ± 0.0443
Magic	$[\frac{1}{2}, 1]$	LMMCM	0.8911 ± 0.0083	0.8790 ± 0.0091	0.8684 ± 0.0046	0.8567 ± 0.0292
Magic	$[\frac{1}{2}, 1]$	CL (Ours)	$\mathbf{0.9105 \pm 0.0020}$	$\mathbf{0.8980 \pm 0.0059}$	$\mathbf{0.8851 \pm 0.0255}$	$\mathbf{0.8816 \pm 0.0083}$
Magic	$[0, 1]$	PL	0.9039 ± 0.0029	0.8870 ± 0.0037	0.9002 ± 0.0092	0.8807 ± 0.0200
Magic	$[0, 1]$	LMMCM	0.9070 ± 0.0026	0.9048 ± 0.0058	0.9113 ± 0.0058	0.8934 ± 0.0097
Magic	$[0, 1]$	CL (Ours)	$\mathbf{0.9173 \pm 0.0018}$	$\mathbf{0.9102 \pm 0.0057}$	$\mathbf{0.9146 \pm 0.0051}$	$\mathbf{0.9088 \pm 0.0039}$

the constraint. While in this work we focus on discrete linear inequality constraints defined over binary variables, there is existing work focusing on hybrid linear inequality constraints defined over both discrete and continuous variables and their tractability [Belle et al., 2015; Zeng et al., 2021, 2020b]. The development of inference algorithms for such constraints and their applications such as Bayesian deep learning remain an active topic [Zeng and Van den Broeck, 2019; Kolb et al., 2019; Zeng et al., 2020a; Zeng and Broeck, 2023].

5.1.5 Experiments

In this section, we present a thorough empirical evaluation of our proposed count loss on the three weakly supervised learning problems, *LLP*, *MIL*, and *PU learning*.[1] We refer the readers to the appendix for additional experimental details.

5.1.5.1 Learning from Label Proportions

We experiment on two datasets: 1) *Adult* with 8192 training samples where the task is to predict whether a person makes over $50k$ a year or not given personal information as input; 2) *Magic Gamma Ray Telescope* with 6144 training samples where the task is to predict whether the electromagnetic shower is caused by primary gammas or not given information from the atmospheric Cherenkov gamma telescope [Dua and Graff, 2017].[2]

We follow Scott and Zhang [2020] where two settings are considered: one with label proportions uniformly on $[0, \frac{1}{2}]$ and the other uniformly on $[\frac{1}{2}, 1]$. Additionally, we experiment on a third setting with label proportions distributing uniformly on $[0, 1]$ which is not considered in Scott and Zhang [2020] but is the most natural setting since the label proportion is not biased toward either 0 or 1. We experiment on four bag sizes $n \in \{8, 32, 128, 512\}$.

Count loss (CL) denotes our proposed approach using the loss objective defined in Table 5.2 for LLP. We compare our approach with a mutual contamination framework for LLP (LMMCM) [Scott and Zhang, 2020] and against Proportion Loss (PL) [Tsai and Lin, 2020].

Results and Discussions We show our results in Table 5.3. Our method showcases superior results against the baselines on both datasets and variations in bag sizes. Especially in cases with lower bag sizes, i.e., $8, 32$, CL greatly outperforms all other methodologies. Among our baselines are methods that approximate the bag posterior (PL), which we show to be less effective than

[1] Code and experiments are available at `https://github.com/UCLA-StarAI/CountLoss`

[2] Publicly available at `archive.ics.uci.edu/ml`

Table 5.4: MIL experiment on the MNIST dataset. Each block represents a different distribution from which we draw bag sizes—First Block: $\mathcal{N}(10, 2)$, Second Block: $\mathcal{N}(50, 10)$, Third Block: $\mathcal{N}(100, 20)$. We run each experiment for 3 runs and report mean test AUC with standard error. The highest metric for each setting is shown in **boldface**.

Training Bags	50	100	150	200	300	400	500
Gated Attention	0.775 ± 0.034	0.894 ± 0.012	0.935 ± 0.005	0.939 ± 0.006	$\mathbf{0.963 \pm 0.002}$	0.959 ± 0.002	$\mathbf{0.966 \pm 0.003}$
Attention	0.807 ± 0.026	$\mathbf{0.913 \pm 0.006}$	$\mathbf{0.940 \pm 0.004}$	0.942 ± 0.007	0.957 ± 0.002	0.961 ± 0.005	0.965 ± 0.004
CL (Ours)	$\mathbf{0.818 \pm 0.024}$	0.906 ± 0.009	0.929 ± 0.005	$\mathbf{0.946 \pm 0.001}$	0.952 ± 0.004	$\mathbf{0.962 \pm 0.002}$	0.963 ± 0.002
Gated Attention	$\mathbf{0.943 \pm 0.005}$	0.949 ± 0.009	$\mathbf{0.970 \pm 0.005}$	$\mathbf{0.977 \pm 0.001}$	0.983 ± 0.002	0.986 ± 0.004	$\mathbf{0.987 \pm 0.002}$
Attention	0.936 ± 0.010	$\mathbf{0.962 \pm 0.006}$	$\mathbf{0.970 \pm 0.001}$	$\mathbf{0.977 \pm 0.002}$	0.981 ± 0.002	$\mathbf{0.987 \pm 0.001}$	$\mathbf{0.987 \pm 0.002}$
CL (Ours)	0.939 ± 0.010	0.960 ± 0.002	0.964 ± 0.007	0.972 ± 0.002	$\mathbf{0.982 \pm 0.003}$	0.982 ± 0.001	$\mathbf{0.987 \pm 0.002}$
Gated Attention	0.975 ± 0.003	0.981 ± 0.004	0.992 ± 0.002	0.987 ± 0.004	$\mathbf{0.996 \pm 0.001}$	$\mathbf{0.998 \pm 0.001}$	0.990 ± 0.004
Attention	$\mathbf{0.984 \pm 0.001}$	0.982 ± 0.001	$\mathbf{0.996 \pm 0.000}$	0.987 ± 0.007	0.992 ± 0.004	0.994 ± 0.002	0.998 ± 0.000
CL (Ours)	0.981 ± 0.007	$\mathbf{0.989 \pm 0.000}$	$\mathbf{0.996 \pm 0.002}$	$\mathbf{0.995 \pm 0.001}$	$\mathbf{0.996 \pm 0.002}$	0.993 ± 0.003	$\mathbf{0.999 \pm 0.001}$

Table 5.5: MIL: We report mean test accuracy, AUC, F1, precision, and recall averaged over 5 runs with std. error on the Colon Cancer dataset. The highest value for each metric is shown in **boldface**.

Method	**Accuracy**	**AUC**	**F**1	**Precision**	**Recall**
Gated Attention	0.909 ± 0.014	0.908 ± 0.013	0.886 ± 0.021	0.916 ± 0.020	0.879 ± 0.020
Attention	0.893 ± 0.015	0.890 ± 0.008	0.876 ± 0.017	0.908 ± 0.016	0.879 ± 0.018
CL (Ours)	$\mathbf{0.915 \pm 0.008}$	$\mathbf{0.912 \pm 0.010}$	$\mathbf{0.903 \pm 0.010}$	$\mathbf{0.936 \pm 0.014}$	$\mathbf{0.898 \pm 0.007}$

optimizing the exact bag posterior with CL.

5.1.5.2 Multiple Instance Learning

We first experiment on the MNIST dataset [LeCun, 1998] and follow the MIL experimental setting in Ilse et al. [2018]: the training and test set bags are randomly sampled from the MNIST training and test set respectively; each bag can have images of digits from 0 to 9, and bags with the digit 9 are labeled positive. Moreover, the dataset is constructed in a balanced way such that there is an equal amount of positively and negatively labeled bags as in Ilse et al. [2018]. The task is to train a classifier that is able to predict bag labels; the more challenging task is to *discover key instances*, that is, to train a classifier that identifies images of digit 9. Following Ilse et al. [2018], we consider

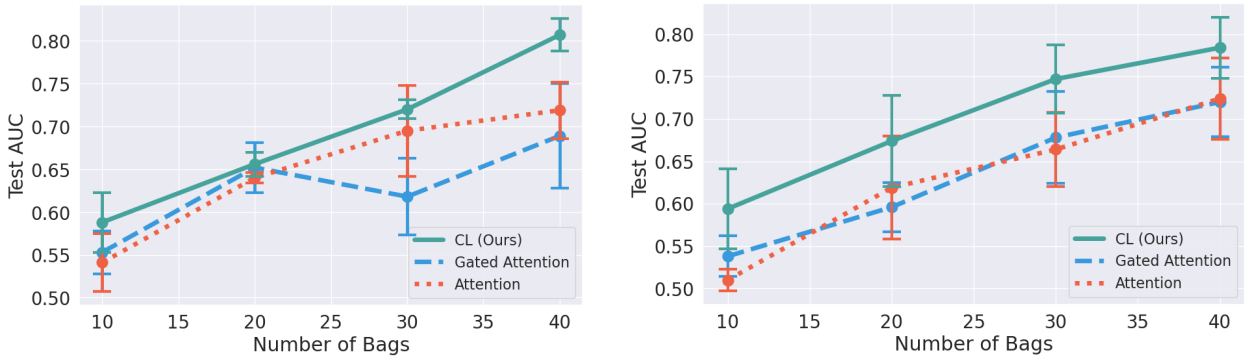


Figure 5.2: MIL MNIST dataset experiments with decreased numbers of training bags and lower bag size. Left: bag sizes sampled from $\mathcal{N}(10, 2)$; Right: bag sizes sampled from $\mathcal{N}(5, 1)$. We plot the mean test AUC (aggregated over 3 trials) with standard errors for 4 bag sizes. Best viewed in color.

three settings that vary in the bag generation process: in each setting, bags have their sizes generated from a normal distribution being $\mathcal{N}(10, 2), \mathcal{N}(50, 10), \mathcal{N}(100, 20)$ respectively. The number of bags in training set n is in $\{50, 100, 150, 200, 300, 400, 500\}$. Thus, we have $3 \times 7 = 21$ settings in total. Additionally, we introduce experimental analysis on *how the performance of the learning methods would degrade as the number of bags and total samples in training set decreases*, by modulating the number of training bags n to be $\{10, 20, 30, 40\}$ and selecting bag sizes from $\mathcal{N}(5, 1)$ and $\mathcal{N}(10, 2)$.

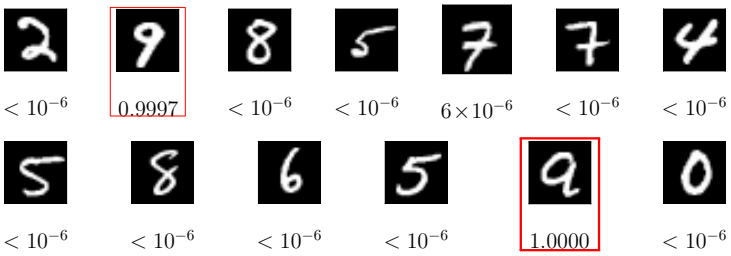


Figure 5.3: A test bag from our MIL experiments, where we set only the digit 9 as a positive instance. Highlighted in red are digits identified to be positive with corresponding probability beneath.

We also experiment on the Colon Cancer dataset [Sirinukunwattana et al., 2016] to simulate a setting where bag instances are not independent. The dataset consists of 100 total hematoxylin-eosin (H&E) stained images, each of which contains images of cell nuclei that are classified as one of: epithelial, inflammatory, fibroblast, and miscellaneous. Each image represents a bag and instances are 27×27 patches extracted from the original image. A positively labeled bag or image is one that contains the epithelial nuclei. For both datasets, we include the Attention and Gated Attention mecha-

nism [Ilse et al., 2018] as baselines. We also use the MIL objective defined in Table 5.2.

Results and Discussions For the MNIST experiments, CL is able to outperform all other baselines or exhibit highly comparable performance for bag-level predictions as shown in Table 5.4. A more interesting setting is to compare how robust the learning methods are if the number of training bags decreases. Wang et al. [2018] claim that instance-level classifiers tend to lose against embedding-based methods. However, we show in our experiment that this is not true in all cases as seen in Figure 5.2. While Attention and Gated Attention are based on embedding, they suffer from a more severe drop in predictive performance than CL when the number of training bags drops from 40 to 10; our method shows great robustness and consistently outperforms all baselines. The rationale we provide is that with a lower number of training instances, we need more supervision over the limited samples we have. Our constraint provides this additional supervision, which accounts for the difference in performance.

We provide an additional investigation in Figure 5.3 to show that our approach learns effectively and delivers accurate instance-level predictions under bag-level supervision. In Figure 5.3, we can see that even though the classifier is trained on feedback about whether a bag contains the digit 9 or not, it accurately discovers all images of digit 9. To reinforce this, Table D.1 and Table D.2, in Appendix D.1.2, show that our approach outperforms existing instance-space methods on instance-level classification.

Our experimental results on the Colon Cancer dataset are shown in Table 5.5. We show that both our proposed objectives are able to consistently outperform baseline methods on all metrics. Interestingly, we do not expect CL to perform well when instances in a bag are dependent; however, the results indicate that our count loss is robust to these settings.

5.1.5.3 Learning from Positive and Unlabeled Data

Table 5.6: PU Learning: We report accuracy and standard deviation on a test set of unlabeled data, which is aggregated over 3 runs. The results from CVIR, nnPU, and uPU are aggregated over 10 epochs, as in Garg et al. [2021], while we choose the single best epoch based on validation for our approaches. The highest metric for each setting is shown in **boldface**.

Dataset	Network	CL-expect (Ours)	CL (Ours)	CVIR	nnPU	nPU
Binarized MNIST	MLP	95.9 ± 0.15	$\mathbf{96.4 \pm 0.01}$	96.3 ± 0.07	96.1 ± 0.14	95.2 ± 0.19
MNIST17	MLP	98.7 ± 0.17	$\mathbf{99.0 \pm 0.19}$	98.7 ± 0.09	98.4 ± 0.20	98.4 ± 0.09
Binarized CIFAR	ResNet	79.2 ± 0.27	80.1 ± 0.34	$\mathbf{82.3 \pm 0.18}$	77.2 ± 1.03	76.7 ± 0.74
CIFAR Cat vs. Dog	ResNet	$\mathbf{76.5 \pm 1.86}$	74.8 ± 1.64	73.3 ± 0.94	71.8 ± 0.33	68.8 ± 0.53

We experiment on dataset MNIST and CIFAR-10 [Krizhevsky and Hinton, 2009], following the four simulated settings from Garg et al. [2021]: 1) Binarized MNIST: the training set consist of images of digits $0-9$ and images with digits in range $[0, 4]$ are positive instances while others as negative; 2) MNIST17: the training set consist of images of digits 1 and 7 and images with digit 1 are defined as positive while 7 as negative; 3) Binarized CIFAR: the training set consists of images from ten classes and images from the first five classes is defined as positive instances while others as negative; 4) CIFAR Cat vs. Dog: the training set consist of images of cats and dogs and images of cats are defined as positive while dogs as negative. The mixture proportion is 0.5 in all experiments. The performance is evaluated using the accuracy on a test set of unlabeled data.

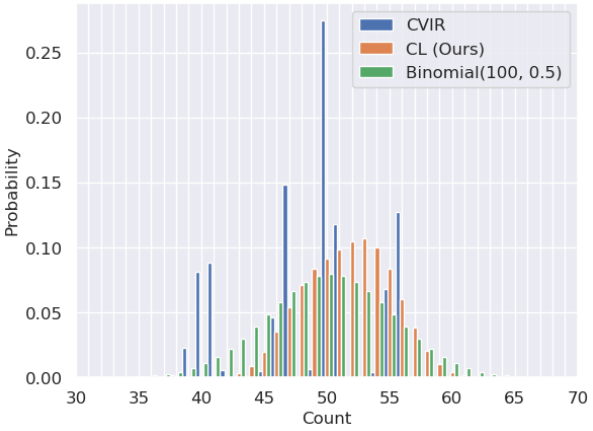


Figure 5.4: MNIST17 setting for PU Learning: We compute the average discrete distribution for CL and CVIR, over 5 test bags, each of which contain 100 instances. A ground truth binomial distribution of counts is also shown.

As shown in Table 5.2, we propose two objectives for PU learning. Our first objective is denoted by CL whereas the second approach is denoted by CL-expect. We compare against the Conditional Value Ignoring Risk approach (CVIR) [Garg et al., 2021], nnPU [Kiryo et al., 2017], and uPU [Plessis et al., 2015].

Results and Discussions Accuracy results are presented in Table 5.6 where we can see that our proposed methods perform better than baselines on 3 out of the 4 simulated PU learning settings. CL-expect builds off a similar "exactly-k" count approach, which we have shown to work well in the label proportion setting. The more interesting results are from CL where we fully leverage the information from a distribution as supervision instead of simply using the expectation. We think of this as applying a loss on each count weighted by their probabilities from the binomial distribution. We provide further evidence that our proposed count loss effectively guides the classifier towards predicting a binomial distribution as shown in Figure 5.4: we plot the count distributions predicted by CL and CVIR as well as the ground-truth binomial distribution. We can see that CL is able to generate the expected distribution, proving the efficacy of our approach.

5.2 Probabilistically Rewired Message-Passing Neural Networks

Graph-structured data is prevalent across various application domains, including fields like chemo- and bioinformatics [Barabasi and Oltvai, 2004; Jumper et al., 2021; Reiser et al., 2022], combinatorial optimization [Cappart et al., 2023], and social-network analysis [Easley et al., 2012], highlighting the need for machine learning techniques designed explicitly for graphs. In recent years, message-passing graph neural networks (MPNNs) [Kipf and Welling, 2017; Gilmer et al., 2017; Scarselli et al., 2008b; Veličković et al., 2018] have become the dominant approach in this area, showing promising performance in tasks such as predicting molecular properties [Klicpera et al., 2020; Jumper et al., 2021] or enhancing combinatorial solvers [Cappart et al., 2023].

However, MPNNs have a limitation due to their local aggregation mechanism. They focus on encoding local structures, severely limiting their expressive power [Morris et al., 2019, 2021; Xu et al., 2019]. In addition, MPNNs struggle to capture global or long-range information, possibly leading to phenomena like under-reaching [Barcelo et al., 2020] or over-squashing [Alon and Yahav, 2021]. Over-squashing, as explained by Alon and Yahav [2021], refers to excessive information compression from distant nodes due to a source node's extensive receptive field, occurring when too many layers are stacked.

Topping et al. [2021]; Bober et al. [2022] investigated over-squashing from the perspective of Ricci and Forman curvature. Refining Topping et al. [2021], Di Giovanni et al. [2023] analyzed how the architectures' width and graph structure contribute to the over-squashing problem, showing that over-squashing happens among nodes with high commute time, stressing the importance of graph rewiring techniques, i.e., adding edges between distant nodes to make the exchange of information more accessible. In addition, Deac et al. [2022]; Shirzad et al. [2023] utilized expander graphs to enhance message passing and connectivity, while Karhadkar et al. [2022] resort to spectral techniques, and Banerjee et al. [2022] proposed a greedy random edge flip approach to overcome over-squashing. Recent work [Gutteridge et al., 2023] aims to alleviate over-squashing by again resorting to graph rewiring. In addition, many studies have suggested different versions

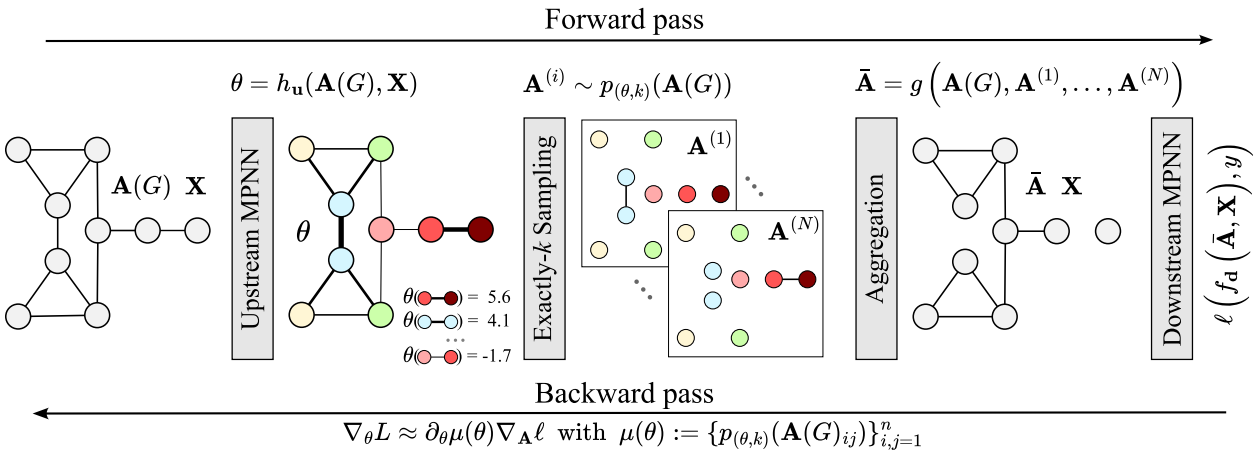


Figure 5.5: Overview of the probabilistically rewired MPNN framework. PR-MPNNs use an upstream model to learn priors $\boldsymbol{\theta}$ for candidate edges, parameterizing a probability mass function conditioned on exactly-k constraints. Subsequently, we sample multiple k-edge adjacency matrices (here: $k = 1$) from this distribution, aggregate these matrices (here: subtraction), and use the resulting adjacency matrix as input to a downstream model, typically an MPNN, for the final predictions task. On the backward pass, the gradients of the loss ℓ regarding the parameters $\boldsymbol{\theta}$ are approximated through the derivative of the exactly-k marginals in the direction of the gradients of the point-wise loss ℓ regarding the sampled adjacency matrix. We use recent work to make the computation of these marginals exact and differentiable, reducing both bias and variance.

of multi-hop-neighbor-based message passing to maintain long-range dependencies [Abboud et al., 2022; Abu-El-Haija et al., 2019; Gasteiger et al., 2019; Xue et al., 2023], which can also be interpreted as a heuristic rewiring scheme. The above works indicate that graph rewiring is an effective strategy to mitigate over-squashing. However, most existing graph rewiring approaches rely on heuristic methods to add edges, potentially not adapting well to the specific data distribution or introducing edges randomly. Furthermore, there is limited understanding to what extent probabilistic rewiring, i.e., adding or removing edges based on the prediction task, impacts the expressive power of a model. In contrast to the above lines of work, graph transformers [Chen et al., 2022; Dwivedi et al., 2022b; He et al., 2023; Müller et al., 2023; Rampášek et al., 2022] and similar global attention mechanisms [Liu et al., 2021; Wu et al., 2021] marked a shift from local to global message passing, aggregating over all nodes. While not understood in a principled way, empirical studies indicate that graph transformers possibly alleviate over-squashing; see, e.g., Müller et al. [2023]. However, due to their global aggregation mode, computing an attention matrix with n^2 entries for an n-order graph makes them applicable only to small or mid-sized graphs. Further, to capture non-trivial graph structure, they must resort to hand-engineered positional or structural

encodings.

Overall, current strategies to mitigate over-squashing rely on heuristic rewiring methods that may not adapt well to a prediction task or employ computationally intensive global attention mechanisms. Furthermore, the impact of probabilistic rewiring on a model's expressive power remains unclear.

Present work By leveraging recent progress in differentiable k-subset sampling [Ahmed et al., 2023c], we derive probabilistically rewired MPNNs (PR-MPNNs). Concretely, we utilize an upstream model to learn prior weights for candidate edges. We then utilize the weights to parameterize a probability distribution constrained by so-called k-subset constraints. Subsequently, we sample multiple k-edge adjacency matrices from this distribution and process them using a downstream model, typically an MPNN, for the final predictions task. To make this pipeline trainable via gradient descent, we adapt recently proposed discrete gradient estimation and tractable sampling techniques [Ahmed et al., 2023c; Niepert et al., 2021a; Xie and Ermon, 2019]; see Figure 5.5 for an overview of our architecture. Our theoretical analysis explores how PR-MPNNs overcome MPNNs' inherent limitations in expressive power and identifies precise conditions under which they outperform purely randomized approaches. Empirically, we demonstrate that our approach effectively mitigates issues like over-squashing and under-reaching. In addition, on established real-world datasets, our method exhibits competitive or superior predictive performance compared to traditional MPNN models and graph transformer architectures.

Overall, PR-MPNNs pave the way for the principled design of more flexible MPNNs, making them less vulnerable to potential noise and missing information.

5.2.1 Related Work

MPNNs are inherently biased towards encoding local structures, limiting their expressive power [Morris et al., 2019, 2021; Xu et al., 2019]. Specifically, they are at most as powerful as distinguishing non-isomorphic graphs or nodes with different structural roles as the 1-dimensional Weisfeiler–

Leman algorithm [Weisfeiler and Leman, 1968], a simple heuristic for the graph isomorphism problem; see Sec. 5.2.2. Additionally, they cannot capture global or long-range information, often linked to phenomena such as under-reaching [Barcelo et al., 2020] or over-squashing [Alon and Yahav, 2021], with the latter being heavily investigated in recent works.

Graph rewiring Several recent works aim to circumvent over-squashing via graph rewiring. Perhaps the most straightforward way of graph rewiring is incorporating multi-hop neighbors. For example, Brüel-Gabrielsson et al. [2022] rewires the graphs with k-hop neighbors and virtual nodes and also augments them with positional encodings. MixHop [Abu-El-Haija et al., 2019], SIGN [Frasca et al., 2020], DIGL [Gasteiger et al., 2019], and SP-MPNN [Abboud et al., 2022] can also be considered as graph rewiring as they can reach further-away neighbors in a single layer. Particularly, Gutteridge et al. [2023] rewires the graph similarly to Abboud et al. [2022] but with a novel delay mechanism, showcasing promising empirical results. Several rewiring methods depend on particular metrics, e.g., Ricci or Forman curvature [Bober et al., 2022] and balanced Forman curvature [Topping et al., 2021]. In addition, Deac et al. [2022]; Shirzad et al. [2023] utilize expander graphs to enhance message passing and connectivity, while Karhadkar et al. [2022] resort to spectral techniques, and Banerjee et al. [2022] propose a greedy random edge flip approach to overcome over-squashing. Refining Topping et al. [2021], Di Giovanni et al. [2023] analyzed how the architectures' width and graph structure contribute to the over-squashing problem, showing that over-squashing happens among nodes with high commute time, stressing the importance of rewiring techniques. Contrary to our proposed method, these strategies to mitigate over-squashing either rely on heuristic rewiring methods or use purely randomized approaches that may not adapt well to a given prediction task. Furthermore, the impact of existing rewiring methods on a model's expressive power remains unclear and we close this gap with our work.

There also exists a large set of works from the field of graph structure learning proposing heuristical graph rewiring approaches; see Sec. D.2.1 for details.

Graph transformers Different from the above, graph transformers [Dwivedi et al., 2022b; He et al., 2023; Müller et al., 2023; Rampášek et al., 2022; Chen et al., 2022] and similar global atten-

tion mechanisms [Liu et al., 2021; Wu et al., 2021] marked a shift from local to global message passing, aggregating over all nodes. While not understood in a principled way, empirical studies indicate that graph transformers possibly alleviate over-squashing; see, e.g., Müller et al. [2023]. However, all transformers suffer from their quadratic space and memory requirements due to computing an attention matrix.

5.2.2 Background

In the following, we provide the necessary background.

Notations Let $\mathbb{N} \coloneqq \{1, 2, 3, \dots\}$. For $n \geq 1$, let $[n] \coloneqq \{1, \dots, n\} \subset \mathbb{N}$. We use $\{\!\{\dots\}\!\}$ to denote multisets, i.e., the generalization of sets allowing for multiple instances for each of its elements. A graph G is a pair $(V(G), E(G))$ with *finite* sets of vertices or nodes $V(G)$ and edges $E(G) \subseteq \{\{u, v\} \subseteq V(G) \mid u \neq v\}$. If not otherwise stated, we set $n \coloneqq |V(G)|$, and the graph is of order n. We also call the graph G an n-order graph. For ease of notation, we denote the edge $\{u, v\}$ in $E(G)$ by (u, v) or (v, u). Throughout the paper, we use standard notations, e.g., we denote the neighborhood of a vertex v by $N(v)$ and $\ell(v)$ denotes its discrete vertex label, and so on; see Sec. D.2.2 for details.

1-dimensional Weisfeiler–Leman algorithm The 1-WL or color refinement is a well-studied heuristic for the graph isomorphism problem, originally proposed by Weisfeiler and Leman [1968]. Formally, let $G = (V(G), E(G), \ell)$ be a labeled graph. In each iteration, $t > 0$, the 1-WL computes a node coloring $C^1_t \colon V(G) \to \mathbb{N}$, depending on the coloring of the neighbors. That is, in iteration $t > 0$, we set

$$C^1_t(v) \coloneqq \mathsf{RELABEL}\Big(\big(C^1_{t-1}(v), \{\!\{C^1_{t-1}(u) \mid u \in N(v)\}\!\}\big)\Big),$$

for all nodes $v \in V(G)$, where $\mathsf{RELABEL}$ injectively maps the above pair to a unique natural number, which has not been used in previous iterations. In iteration 0, the coloring $C^1_0 \coloneqq \ell$. To test if two graphs G and H are non-isomorphic, we run the above algorithm in "parallel" on both

graphs. If the two graphs have a different number of nodes colored $c \in \mathbb{N}$ at some iteration, the 1-WL distinguishes the graphs as non-isomorphic. Moreover, if the number of colors between two iterations, t and $(t+1)$, does not change, i.e., the cardinalities of the images of C^1_t and C^1_{i+t} are equal, or, equivalently,

$$C^1_t(v) = C^1_t(w) \iff C^1_{t+1}(v) = C^1_{t+1}(w),$$

for all nodes v and w in $V(G)$, the algorithm terminates. For such t, we define the stable coloring $C^1_\infty(v) = C^1_t(v)$, for v in $V(G)$. The stable coloring is reached after at most $\max\{|V(G)|, |V(H)|\}$ iterations [Grohe, 2017]. It is easy to see that the algorithm cannot distinguish all non-isomorphic graphs [Cai et al., 1992]. Nonetheless, it is a powerful heuristic that can successfully test isomorphism for a broad class of graphs [Babai and Kucera, 1979]. A function $f\colon V(G) \to {R_2}^d$, for $d > 0$, is 1-WL-equivalent if $f \equiv C^1_\infty$; see Sec. D.2.2 for details.

Message-passing graph neural networks Intuitively, MPNNs learn a vectorial representation, i.e., a d-dimensional real-valued vector, representing each vertex in a graph by aggregating information from neighboring vertices. Let $\mathbf{G} = (G, \mathbf{L})$ be an attributed graph, following, Gilmer et al. [2017] and Scarselli et al. [2008a], in each layer, $t > 0$, we compute vertex features

$$\mathbf{h}_v^{(t)} \coloneqq \mathsf{UPD}^{(t)}\Big(\mathbf{h}_v^{(t-1)}, \mathsf{AGG}^{(t)}\big(\{\!\!\{\mathbf{h}_u^{(t-1)} \mid u \in N(v)\}\!\!\}\big)\Big) \in {R_2}^d,$$

where $\mathsf{UPD}^{(t)}$ and $\mathsf{AGG}^{(t)}$ may be differentiable parameterized functions, e.g., neural networks, and $\mathbf{h}_v^{(t)} = \vec{L}_v$. In the case of graph-level tasks, e.g., graph classification, one uses

$$\mathbf{h}_G \coloneqq \mathsf{READOUT}\big(\{\!\!\{\mathbf{h}_v^{(T)} \mid v \in V(G)\}\!\!\}\big) \in {R_2}^d,$$

to compute a single vectorial representation based on learned vertex features after iteration T. Again, $\mathsf{READOUT}$ may be a differentiable parameterized function, e.g., a neural network. To adapt the parameters of the above three functions, they are optimized end-to-end, usually through

a variant of stochastic gradient descent, e.g., Kingma and Ba [2015], together with the parameters of a neural network used for classification or regression.

5.2.3 Probalistically rewired MPNNs

Here, we outline probabilistically rewired MPNNs (PR-MPNNs) based on recent advancements in discrete gradient estimation and tractable sampling techniques [Ahmed et al., 2023c]. Let $\mathfrak{A}_n$ denote the set of adjacency matrices of n-order graphs. Further, let $(G, \vec{X})$ be a n-order attributed graph with an adjacency matrix $\vec{A}(G) \in \mathfrak{A}_n$ and node attribute matrix $\vec{X} \in \mathbb{R}^{n \times d}$, for $d > 0$. A PR-MPNN maintains a (parameterized) *upstream model* $h_{\boldsymbol{v}} \colon \mathfrak{A}_n \times \mathbb{R}^{n \times d} \to \Theta$, typically a neural network, parameterized by $\boldsymbol{v}$, mapping an adjacency matrix and corresponding node attributes to unnormalized edge priors $\boldsymbol{\theta} \in \Theta \subseteq \mathbb{R}^{n \times n}$.

In the following, we use the *priors* $\boldsymbol{\theta}$ as parameters of a (conditional) probability mass function,

$$p_{\boldsymbol{\theta}}(\vec{A}(H)) \coloneqq \prod_{i,j=1}^{n} p_{\theta_{ij}}(\vec{A}(H)_{ij}),$$

assigning a probability to each adjacency matrix in $\mathfrak{A}_n$, where $p_{\theta_{ij}}(\vec{A}(H)_{ij} = 1) = \operatorname{sigmoid}(\theta_{ij})$ and $p_{\theta_{ij}}(\vec{A}(H)_{ij} = 0) = 1 - \operatorname{sigmoid}(\theta_{ij})$. Since the parameters $\boldsymbol{\theta}$ depend on the input graph G, we can view the above probability as a conditional probability mass function conditioned on the graph G.

Unlike previous probabilistic rewiring approaches, e.g., [Franceschi et al., 2019], we introduce dependencies between the graph's edges by conditioning the probability mass function $p_{\theta_{ij}}(\vec{A}(H))$ on a *k-subset constraint*. That is, the probability of sampling any given k-edge adjacency matrix $\vec{A}(H)$, becomes

$$p_{(\boldsymbol{\theta},k)}(\vec{A}(H)) \coloneqq \begin{cases} {}^{p_{\boldsymbol{\theta}}(\vec{A}(H))}/_{Z} & \text{if } \|\vec{A}(H)\|_1 = k, \\ 0 & \text{otherwise,} \end{cases} \quad \text{with} \quad Z \coloneqq \sum_{\vec{B} \in \mathfrak{A}_n \colon \|\vec{B}\|_1 = k} p_{\boldsymbol{\theta}}(\vec{B}). \tag{5.3}$$

The original graph G is now rewired into a new adjacency matrix $\bar{\vec{A}}$ by combining N samples $\vec{A}^{(i)} \sim p_{(\boldsymbol{\theta},k)}(\vec{A}(G))$ for $i \in [N]$ together with the original adjacency matrix $\vec{A}(G)$ using a differentiable aggregation function $g\colon \mathfrak{A}_n^{(N+1)} \to \mathfrak{A}_n$, i.e., $\bar{\vec{A}} \coloneqq g(\vec{A}(G), \vec{A}^{(1)}, \ldots, \vec{A}^{(N)}) \in \mathfrak{A}_n$. Subsequently, we use the resulting adjacency matrix as input to a downstream model $f_{\vec{d}}$, parameterized by $\vec{d}$, typically an MPNN, for the final predictions task.

We have so far assumed that the upstream MPNN computes one set of priors $h_{\boldsymbol{v}}\colon \mathfrak{A}_n \times \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times n}$ which we use to generate a new adjacency matrix $\bar{\vec{A}}$ through sampling and then aggregating the adjacency matrices $\vec{A}^{(1)}, \ldots, \vec{A}^{(N)}$. In Sec. 5.2.5, we show empirically that having multiple sets of priors from which we sample is beneficial. Multiple sets of priors mean that we learn an upstream model $h_{\boldsymbol{v}}\colon \mathfrak{A}_n \times \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times n\times M}$ where M is the number of priors. We can then sample and aggregate the adjacency matrices from these multiple sets of priors.

Learning to sample To learn the parameters of the up- and downstream model $\boldsymbol{\omega} = (\boldsymbol{v}, \boldsymbol{u})$ of the PR-MPNN architecture, we minimize the expected loss

$$L(\vec{A}(G), \vec{X}, y; \boldsymbol{\omega}) \coloneqq \mathbb{E}_{\vec{A}^{(i)} \sim p_{(\boldsymbol{\theta},k)}(\vec{A}(G))} \left[\ell \left(f_{\boldsymbol{u}} \left(g \left(\vec{A}(G), \vec{A}^{(1)}, \ldots, \vec{A}^{(N)} \right), \vec{X} \right), y \right) \right],$$

with $y \in \mathcal{Y}$, the targets, ℓ a point-wise loss such as the cross-entropy or MSE, and $\boldsymbol{\theta} = h_{\boldsymbol{v}}(\vec{A}(G), \vec{X})$. To minimize the above expectation using gradient descent and backpropagation, we need to efficiently draw Monte-Carlo samples from $p_{(\boldsymbol{\theta},k)}(\vec{A}(G))$ and estimate $\nabla_{\boldsymbol{\theta}} L$ the gradients of an expectation regarding the parameters $\boldsymbol{\theta}$ of the distribution $p_{(\boldsymbol{\theta},k)}$.

Sampling To sample an adjacency matrix $\vec{A}^{(i)}$ from $p_{(\boldsymbol{\theta},k)}(\vec{A}(G))$ conditioned on k-edge constraints, and to allow PR-MPNNs to be trained end-to-end, we use SIMPLE [Ahmed et al., 2023c], a recently proposed gradient estimator. Concretely, we can use SIMPLE to sample *exactly* from the k-edge adjacency matrix distribution $p_{(\boldsymbol{\theta},k)}(\vec{A}(G))$ on the forward pass. On the backward pass, we compute the approximate gradients of the loss (which is an expectation over a discrete probability

mass function) regarding the prior weights $\boldsymbol{\theta}$ using

$$\nabla_{\boldsymbol{\theta}} L \approx \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta}) \nabla_{\vec{A}} \ell \;\text{ with }\; \mu(\boldsymbol{\theta}) \coloneqq \{p_{(\boldsymbol{\theta},k)}(\vec{A}(G)_{ij})\}_{i,j=1}^{n} \in \mathbb{R}^{n \times n},$$

with an exact and efficient computation of the marginals $\mu(\boldsymbol{\theta})$ that is differentiable on the backward pass, achieving lower bias and variance. We show empirically that SIMPLE [Ahmed et al., 2023c] outperforms other sampling and gradient approximation methods such as GUMBEL SOFTSUB-ST [Xie and Ermon, 2019] and I-MLE [Niepert et al., 2021a], improving accuracy without incurring a computational overhead.

Computational complexity The vectorized complexity of the exact sampling and marginal inference step is $\mathcal{O}(\log k \log l)$, where k is from our k-subset constraint, and l is the maximum number of edges that we can sample. Assuming a constant number of layers, PR-MPNN's worst-case training complexity is $\mathcal{O}(l)$ for both the upstream and downstream models. Let n be the number of nodes in the initial graph, and $l = \max(\{l_{\text{add}}, l_{\text{rm}}\})$, with l_{add} and l_{rm} being the maximum number of added and deleted edges. If we consider all of the possible edges for l_{add}, the worst-case complexity becomes $\mathcal{O}(n^2)$. Therefore, to reduce the complexity in practice, we select a subset of the possible edges using simple heuristics, such as considering the top l_{add} edges of the most distant nodes. During inference, since we do not need gradients for edges not sampled in the forward pass, the complexity is $\mathcal{O}(l)$ for the upstream model and $\mathcal{O}(L)$ for the downstream model, with L being the number of edges in the rewired graph.

5.2.4 Expressive Power of Probabilistically Rewired MPNNs

We now, for the first time, explore the extent to which probabilistic MPNNs overcome the inherent limitations of MPNNs in expressive power caused by the equivalence to 1-WL in distinguishing non-isomorphic graphs [Xu et al., 2018b; Morris et al., 2019]. Moreover, we identify formal conditions under which PR-MPNNs outperform popular randomized approaches such as those dropping nodes and edges uniformly at random. We first make precise what we mean by probabilistically

separating graphs by introducing a probabilistic and generally applicable notion of graph separation.

Let us assume a conditional probability mass function $p\colon \mathfrak{A}_n \to [0,1]$ conditioned on a given n-order graph, defined over the set of adjacency matrices of n-order graphs. In the context of PR-MPNNs, p is the probability mass function defined in Sec. 5.2.3 but it could also be any other conditional probability mass function over graphs. Moreover, let $f\colon \mathfrak{A}_n \to {R_2}^d$, for $d > 0$, be a permutation-invariant, parameterized function mapping a sampled graph's adjacency matrix to a vector in ${R_2}^d$. The function f could be the composition of an aggregation function g that removes the sampled edges from the input graph G and of a downstream MPNN. Now, the conditional probability mass function p *separates* two graphs G and H with probability ρ with respect to f if

$$\mathbb{E}_{\bar{G}\sim p(\cdot|G),\bar{H}\sim p(\cdot|H)}\left[f(\vec{A}(\bar{G})) \neq f(\vec{A}(\bar{H}))\right] = \rho,$$

that is, if in expectation over the conditional probability distribution, the vectors $f(\vec{A}(\bar{G}))$ and $f(\vec{A}(\bar{H}))$ are distinct with probability ρ.

In what follows, we analyze the case of p being the exactly-k probability distribution defined in Equation 5.3 and f being the aggregation function removing edges and a downstream MPNN. However, our framework readily generalizes to the case of node removal, and we provide these theoretical results in the appendix. Following Sec. 5.2.3, we sample adjacency matrices with exactly k edges and use them to remove edges from the original graph. We aim to understand the separation properties of the probability mass function $p_{(k,\boldsymbol{\theta})}$ in this setting and for various types of graph structures. Most obviously, we do not want to separate isomorphic graphs and, therefore, remain isomorphism invariant, a desirable property of MPNNs.

Theorem 3. *For sufficiently large n, for every $\varepsilon \in (0,1)$ and $k > 0$, we have that for almost all pairs, in the sense of Babai et al. [1980], of isomorphic n-order graphs G and H and all permutation-invariant, 1-*WL*-equivalent functions $f\colon \mathfrak{A}_n \to {R_2}^d$, $d > 0$, there exists a probability mass function $p_{(\boldsymbol{\theta},k)}$ that separates the graph G and H with probability at most ε with respect to f.*

Theorem 3 relies on the fact that most graphs have a discrete 1-WL coloring. For graphs where the 1-WL stable coloring consists of a discrete and non-discrete part, the following result shows that there exist distributions $p_{(\boldsymbol{\theta},k)}$ not separating the graphs based on the partial isomorphism corresponding to the discrete coloring.

Proposition 5.3. *Let $\varepsilon \in (0,1)$, $k > 0$, and let G and H be graphs with identical 1-WL stable colorings. Let V_G and V_H be the subset of nodes of G and H that are in color classes of cardinality 1. Then, for all choices of 1-WL-equivalent functions f, there exists a conditional probability distribution $p_{(\boldsymbol{\theta},k)}$ that separates the graphs $G[V_G]$ and $H[V_H]$ with probability at most ε with respect to f.*

Existing methods such as DropGNN [Papp et al., 2021] or DropEdge [Rong et al., 2020] are more likely to separate two (partially) isomorphic graphs by removing different nodes or edges between discrete color classes, i.e., on their (partially) isomorphic subgraphs. For instance, in the appendix, we prove that pairs of graphs with m edges exist where the probability of non-separation under uniform edge sampling is at most $1/m$. This is undesirable as it breaks the MPNNs' permutation-invariance in these parts.

Now that we have established that distributions with priors from upstream MPNNs are more likely to preserve (partial) isomorphism between graphs, we turn to analyze their behavior in separating the non-discrete parts of the coloring. The following theorem shows that PR-MPNNs are more likely to separate non-isomorphic graphs than probability mass functions that remove edges or nodes uniformly at random.

Theorem 4. *For every $\varepsilon \in (0,1)$ and every $k > 0$, there exists a pair of non-isomorphic graphs G and H with identical and non-discrete 1-WL stable colorings such that for every 1-WL-equivalent function f,*

(1) there exists a probability mass function $p_{(k,\boldsymbol{\theta})}$ that separates G and H with probability at least $(1-\varepsilon)$ with respect to f;

Table 5.7: Comparison between PR-MPNN and baselines on three molecular property prediction datasets. We report results for PR-MPNN with different gradient estimators for k-subset sampling: GUMBEL SOFTSUB-ST [Maddison et al., 2017; Jang et al., 2017; Xie and Ermon, 2019], I-MLE [Niepert et al., 2021a], and SIMPLE [Ahmed et al., 2023c] and compare them with the base downstream model, and two graph transformer architectures. The variant using SIMPLE consistently outperforms the base models and is competitive or better than the two graph transformers. We use **green** for the best model, **blue** for the second-best, and **red** for third. We note with + EDGE the instances where edge features are provided and with - EDGE when they are not.

		ZINC		OGBG-MOLHIV	ALCHEMY
		- EDGE ↓	+ EDGE ↓	+ EDGE ↑	+ EDGE ↓
GIN BACKBONE	K-ST SAT	0.166±0.007	0.115±0.005	0.625±0.039	N/A
	K-SG SAT	0.162±0.013	**0.095**±0.002	0.613±0.010	N/A
	BASE	0.258±0.006	0.207±0.006	**0.775**±0.011	11.12±0.690
	BASE W. PE	0.162±0.001	0.101±0.004	0.764±0.018	7.197±0.094
	PR-MPNN$_{\text{GMB}}$ (OURS)	0.153±0.003	0.103±0.008	0.760±0.025	**6.858**±0.090
	PR-MPNN$_{\text{IMLE}}$ (OURS)	**0.151**±0.001	0.104±0.008	**0.774**±0.015	**6.692**±0.061
	PR-MPNN$_{\text{SIM}}$ (OURS)	**0.139**±0.001	**0.085**±0.002	**0.795**±0.009	**6.447**±0.057
PNA	GPS	N/A	**0.070**±0.004	**0.788**±0.010	N/A
	K-ST SAT	0.164±0.007	0.102±0.005	0.625±0.039	N/A
	K-SG SAT	**0.131**±0.002	**0.094**±0.008	0.613±0.010	N/A

(2) removing edges uniformly at random separates G and H with probability at most ε with respect to f.

Finally, we can also show a negative result, namely that there exist classes of graphs for which PR-MPNNs cannot do better than random sampling.

Proposition 5.4. *For every $k > 0$, there exist non-isomorphic graphs G and H with identical 1-WL colorings such that every probability mass function $p_{(\boldsymbol{\theta},k)}$ separates the two graphs with the same probability as the distribution that samples edges uniformly at random.*

5.2.5 Experimental Evaluation

Here, we explore to what extent our probabilistic graph rewiring leads to improved predictive performance on synthetic and real-world datasets. Concretely, we answer the following questions.

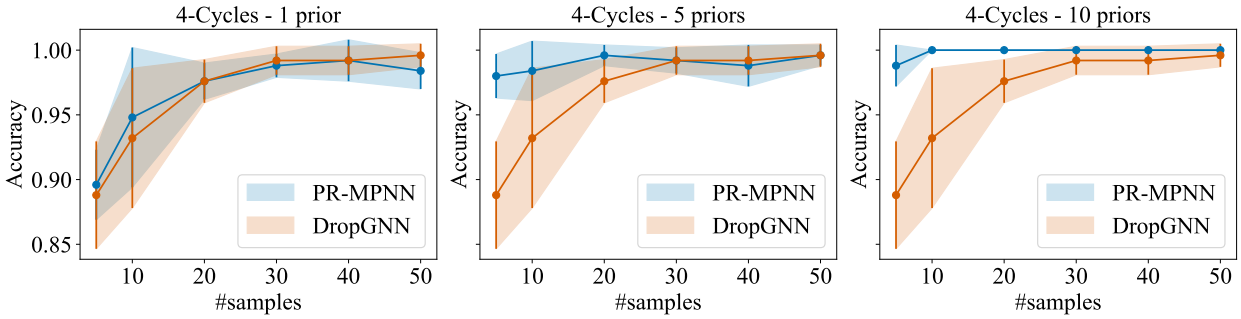


Figure 5.6: Comparison between PR-MPNN and DropGNN on the 4-CYCLES dataset. PR-MPNN rewiring is almost always better than randomly dropping nodes, and is always better with 10 priors.

Q1 Can probabilistic graph rewiring mitigate the problems of over-squashing and under-reaching in synthetic datasets?

Q2 Is the expressive power of standard MPNNs enhanced through probabilistic graph rewiring? That is, can we verify empirically that the separating probability mass function of Sec. 5.2.4 can be learned with PR-MPNNs and that multiple priors help?

Q3 Does the increase in predictive performance due to probabilistic rewiring apply to (a) graph-level molecular prediction tasks and (b) node-level prediction tasks involving heterophilic data?

An anonymized repository of our code can be accessed at `https://anonymous.4open.science/r/PR-MPNN`.

Datasets To answer **Q1**, we utilized the TREES-NEIGHBORSMATCH dataset [Alon and Yahav, 2021]. Additionally, we created the TREES-LEAFCOUNT dataset to investigate whether our method could mitigate under-reaching issues; see Sec. D.2.5 for details. To tackle **Q2**, we performed experiments with the EXP [Abboud et al., 2020] and CSL datasets [Murphy et al., 2019] to assess how much probabilistic graph rewiring can enhance the models' expressivity. In addition, we utilized the 4-CYCLES dataset from Loukas [2020]; Papp et al. [2021] and set it against a standard DropGNN model [Papp et al., 2021] for comparison while also ablating the performance difference concerning the number of priors and samples per prior. To answer **Q3** (a), we used the established

molecular graph-level regression datasets ALCHEMY [Chen et al., 2019], ZINC [Jin et al., 2017; Dwivedi et al., 2020], OGBG-MOLHIV [Hu et al., 2020a], QM9 [Hamilton et al., 2017], LRGB [Dwivedi et al., 2022b] and five datasets from the TUDATASET repository [Morris et al., 2020]. To answer **Q3** (b), we used the CORNELL, WISCONSIN, TEXAS node-level classification datasets [Pei et al., 2020].

Baseline and model configurations For our upstream model $h_{\boldsymbol{v}}$, we use an MPNN, specifically the GIN layer [Xu et al., 2019]. For an edge $(v, w) \in E(G)$, we compute $\boldsymbol{\theta}_{vw} = \phi([\mathbf{h}_v^T || \mathbf{h}_w^T]) \in \mathbb{R}$, where $[\cdot||\cdot]$ is the concatenation operator and ϕ is an MLP. After obtaining the prior $\boldsymbol{\theta}$, we rewire our graphs by sampling two adjacency matrices for deleting edges and adding new edges, i.e., $g(\vec{A}(G), \vec{A}^{(1)}, \vec{A}^{(2)}) \coloneqq (\vec{A}(G) - \vec{A}^{(1)}) + \vec{A}^{(2)}$ where $\vec{A}^{(1)}$ and $\vec{A}^{(2)}$ are two sampled adjacency matrices with a possibly different number of edges, respectively. Finally, the rewired adjacency matrix (or multiple adjacency matrices) is used in a *downstream model* $f_{\boldsymbol{u}} \colon \mathfrak{A}_n \times \mathbb{R}^{n \times d} \to \mathcal{Y}$, typically an MPNN, with parameters $\boldsymbol{u}$ and $\mathcal{Y}$ the prediction target set. For the instance where we have multiple priors, as described in Sec. 5.2.3, we can either aggregate the sampled adjacency matrices $\vec{A}^{(1)}, \ldots, \vec{A}^{(N)}$ into a single adjacency matrix $\bar{\mathbf{A}}$ that we send to a downstream model as described in Figure 5.5, or construct a downstream ensemble with multiple aggregated matrices $\bar{\mathbf{A}}_1, \ldots, \bar{\mathbf{A}}_M$. In practice, we always use a downstream ensemble before the final projection layer when we rewire with more than one adjacency matrix, and we do rewiring by both adding and deleting edges, please consult Table D.13 in the Appendix for more details.

All of our downstream models $f_{\vec{d}}$ and base models are MPNNs with GIN layers. When we have access to edge features, we use the GINE variant [Hu et al., 2020b] for edge feature processing. For graph-level tasks, we use mean pooling, while for node-level tasks, we take the node embedding $\vec{h}_v^T$ for a node v. The final embeddings are then processed and projected to the target space by an MLP.

For ZINC, ALCHEMY, and OGBG-MOLHIV, we compare our rewiring approaches with the base downstream model, both with and without positional embeddings. Further, we compare to GPS [Rampášek et al., 2022] and SAT [Chen et al., 2022], two state-of-the-art graph transformers. For

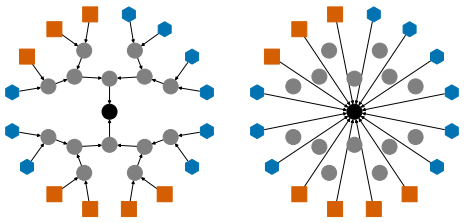


Figure 5.7: Example graph from the TREES-LEAFCOUNT test dataset with radius 4 (left). PR-MPNN rewires the graph, allowing the downstream MPNN to obtain the label information from the leaves in one massage-passing step (right).

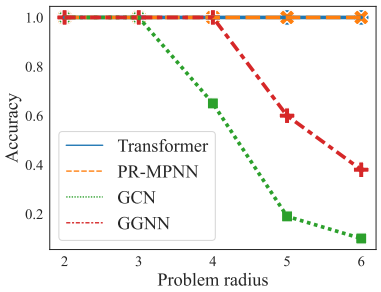


Figure 5.8: Test accuracy of our rewiring method on the TREES-NEIGHBORSMATCH [Alon and Yahav, 2021] dataset, compared to the reported accuracies from Müller et al. [2023].

the TUDATASET, we compare with the reported scores from Giusti et al. [2023b] and use the same evaluation strategy as in Xu et al. [2019]; Giusti et al. [2023b], i.e., running 10-fold cross-validation and reporting the maximum average validation accuracy. For different tasks, we search for the best hyperparameters for sampling and our upstream and downstream models. See Table D.13 in the appendix for the complete description. For ZINC, ALCHEMY, and OGBG-MOLHIV, we evaluate multiple gradient estimators in terms of predictive power and computation time. Specifically, we compare GUMBEL SOFTSUB-ST [Maddison et al., 2017; Jang et al., 2017; Xie and Ermon, 2019], I-MLE [Niepert et al., 2021a], and SIMPLE [Ahmed et al., 2023c]. The results in terms of predictive power are detailed in Table 5.7, and the computation time comparisons can be found in Table D.14 in the appendix. Further experimental results on QM9 and LRGB are included in Sec. D.2.7 in the appendix.

Experimental results and discussion Concerning **Q1**, our rewiring method achieves perfect test accuracy up to a problem radius of 6 on both TREES-NEIGHBORSMATCH and TREES-LEAFCOUNT, demonstrating that it can successfully alleviate over-squashing and under-reaching, see Figure 5.8. For TREES-LEAFCOUNT, our model can create connections directly from the leaves to the root, achieving perfect accuracy with a downstream model containing a single MPNN layer. We provide a qualitative result in Figure 5.7 and a detailed discussion in Sec. D.2.5. Concerning **Q2**, on the 4-CYCLES dataset, our probabilistic rewiring method matches or outperforms DropGNN. This advantage is most pronounced with 5 and 10 priors, where we achieve 100% task accuracy using 20

Table 5.8: Comparison between the base GIN model, PR-MPNN, and other more expressive models on the EXP dataset.

MODEL	ACCURACY ↑
GIN	0.511±0.021
GIN + ID-GNN	1.000±0.000
OSAN	1.000±0.000
PR-MPNN (OURS)	1.000±0.000

Table 5.9: Comparison between the base GIN model and probabilistic rewiring model on CSL dataset, w/o positional encodings.

MODEL	ACCURACY ↑
GIN	0.100±0.000
GIN + POSENC	1.000±0.000
PR-MPNN (OURS)	0.998±0.008
PR-MPNN + POSENC (OURS)	1.000±0.000

samples, as detailed in Figure 5.6. On the EXP dataset, we showcase the expressive power of probabilistic rewiring by achieving perfect accuracy, see Table 5.8. Besides, our rewiring approach can distinguish the regular graphs from the CSL dataset without any positional encodings, whereas the 1-WL-equivalent GIN obtains only random accuracy. Concerning **Q3** (a), the results in Table 5.7 show that our rewiring methods consistently outperform the base models on ZINC, ALCHEMY, and OGBG-MOLHIV and are competitive or better than the state-of-the-art GPS and SAT graph transformer methods. On TUDATASET, see Table 5.10, our probabilistic rewiring method outperforms existing approaches and obtains lower variance on most of the datasets, with the exception being NCI1, where our method ranks second, after the WL kernel. Hence, our results indicate that probabilistic graph rewiring can improve performance for molecular prediction tasks. Concerning **Q3** (b), we obtain performance gains over the base model and other existing MPNNs, see Table D.16 in the appendix, indicating that data-driven rewiring has the potential of alleviating the *effects* of over-smoothing by removing undesirable edges and making new ones between nodes with similar features. The graph transformer methods outperform the rewiring approach and the base models, except on the TEXAS dataset, where our method gets the best result. We speculate that GIN's aggregation mechanism for the downstream models is a limiting factor on heterophilic data. We leave the analysis of combining probabilistic graph rewiring with downstream models that address over-smoothing for future investigations.

Table 5.10: Comparison between PR-MPNN and other approaches as reported in Giusti et al. [2023b]; Karhadkar et al. [2022]; Papp et al. [2021]. Our model outperforms existing approaches while keeping a lower variance in most of the cases, except for NCI1, where the WL Kernel is the best. We use **green** for the best model, **blue** for the second-best, and **red** for third.

MODEL	MUTAG	PTC_MR	PROTEINS	NCI1	NCI109
GK ($k = 3$) [SHERVASHIDZE ET AL., 2009]	81.4±1.7	55.7±0.5	71.4±0.3	62.5±0.3	62.4±0.3
PK [NEUMANN ET AL., 2016]	76.0±2.7	59.5±2.4	73.7±0.7	82.5±0.5	N/A
WL KERNEL [SHERVASHIDZE ET AL., 2011]	90.4±5.7	59.9±4.3	75.0±3.1	**86.0**±1.8	N/A
DGCNN [ZHANG ET AL., 2018]	85.8±1.8	58.6±2.5	75.5±0.9	74.4±0.5	N/A
IGN [MARON ET AL., 2019B]	83.9±13.0	58.5±6.9	76.6±5.5	74.3±2.7	72.8±1.5
GIN [XU ET AL., 2019]	89.4±5.6	64.6±7.0	76.2±2.8	82.7±1.7	N/A
PPGNs [MARON ET AL., 2019A]	90.6±8.7	66.2±6.6	77.2±4.7	83.2±1.1	82.2±1.4
NATURAL GN [DE HAAN ET AL., 2020]	89.4±1.6	66.8±1.7	71.7±1.0	82.4±1.3	83.0±1.9
GSN [BOURITSAS ET AL., 2022]	92.2±7.5	68.2±7.2	76.6±5.0	83.5±2.0	83.5±2.3
CIN [BODNAR ET AL., 2021]	92.7±6.1	68.2±5.6	77.0±4.3	83.6±1.4	**84.0**±1.6
CAN [GIUSTI ET AL., 2023A]	**94.1**±4.8	**72.8**±8.3	**78.2**±2.0	84.5±1.6	83.6±1.2
CIN++ [GIUSTI ET AL., 2023B]	**94.4**±3.7	**73.2**±6.4	**80.5**±3.9	**85.3**±1.2	**84.5**±2.4
FOSR [KARHADKAR ET AL., 2022]	86.2±1.5	58.5±1.7	75.1±0.8	72.9±0.6	71.1±0.6
DROPGNN [PAPP ET AL., 2021]	90.4±7.0	66.3±8.6	76.3±6.1	81.6±1.8	80.8±2.6
PR-MPNN (10-FOLD CV)	**98.4**±2.4	**74.3**±3.9	**80.7**±3.9	**85.6**±0.8	**84.6**±1.2

CHAPTER 6

Conclusion and Future Directions

This dissertation has demonstrated unmistakable strides in neuro-symbolic AI, by viewing ML systems as inducing a distribution and reasoning about it. Still, much remains to be done.

Toward Real-Time Reasoning Modern neural network architectures have attained unprecedented expressivity while retaining efficiency, enabling self-driving cars to perform real-time perception. Exact neuro-symbolic AI makes use of tractable representations of constraints that make it possible to efficiently perform reasoning on the learned distribution by means of posing and answering probabilistic queries. These remain quite limited both in terms of the class of functions that can be efficiently represented and the speed with which we could reason about them. I plan to investigate exact and efficient representations when possible, developing *approximate representations with guarantees* otherwise. I will also intensify systems development efforts, collaborating with systems and software engineering colleagues to develop GPU-utilizing, sound and user-friendly systems.

Towards Ever-Changing Environments Constraints are often assumed to be given, an assumption that holds true in many applications of interest, but in many other cases does not. Take for instance a self-driving car trained on data from one city and deployed in another governed by unknown, possibly differing traffic rules which has to learn the rules underlying the world from raw data. This is a vital direction that should receive significant attention if we are to attain truly intelligent systems. I plan to develop approaches for extracting knowledge in the form of symbolic rules from unstructured data, collaborating with colleagues working on learning and perception.

From Single Queries to Logical Reasoning The existing literature on neuro-symbolic AI often concerns itself with posing a single query to a probability distribution to solve a specific problem.

There, the amount of "reasoning" involved is restricted to one-shot inference. This, however, admits a rather limited view of neuro-symbolic AI: A self-driving car should have a belief as to the current state of the world, and that belief should be continually updated as new information is made available, and as more reasoning is performed. I aim to move beyond answering single queries and towards logical reasoning in a sense that involves a chain of queries interleaved with observations, as well as actively seeking out more data to update the models beliefs.

Appendix A

Learning with Constraints

A.1 Neuro-Symbolic Entropy Regularization

A.1.1 Compiling Logical Formulas into Tractable Circuits

At a high level, there exist off-the-shelf compilers [Choi and Darwiche, 2013; Oztok and Darwiche, 2015; Darwiche, 2004; Muise et al., 2012; Lagniez and Marquis, 2017; Toda and Soh, 2016] utilizing SAT solvers, essentially through case analysis, to compile a logical formula into a tractable logical circuit. NeSy Entropy is agnostic to the exact flavor of circuit so long as the properties outlined in Section 2.1.2.2 are respected. In our experiments, we use PySDD[1] a python SDD compiler [Darwiche, 2011a; Choi and Darwiche, 2013]. We will now step through an example of compiling a logical formula. Consider the circuit in Figure 2.2 encoding constraint

$$(A \wedge B) \implies C,$$

to be construed as encoding, $\mathsf{animal} \wedge \mathsf{barks} \implies \mathsf{dog}$.

Intuitively, our aim is to transform the above logical formula into a *compact* target form representing all possible assignments to A, B and C satisfying the logical formula. We compile such a constraint by proceeding in a bottom up fashion, where bottom-up compilation can be seen as composing Boolean sub-functions whose domain is determined by a variable ordering. Concretely, starting from circuits for literals A and B, we compile a circuit $\beta = A \wedge B$. We compose the

[1] https://github.com/wannesm/PySDD

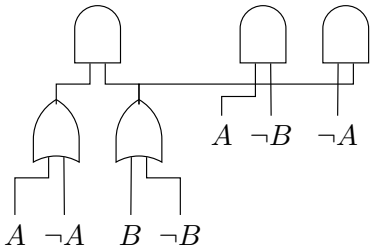
previously compiled circuit β with the circuit for literal C. We point out that this is achieved using a couple of simple API calls to a bottom-up compiler. We will now step through the actual construction of the circuit. We introduce logical circuits representing the literals

$$A \qquad \neg A \qquad B \qquad \neg B \qquad C \qquad \neg C$$

The compiler disjoins literals A with $\neg A$, and B with $\neg B$, introducing deterministic and smooth OR nodes.

An OR node represents *disjoint solutions* to the logical formula, meaning there exists distinct assignments, characterized by the children, satisfying the constraint e.g. $a, \neg a, b$ and $\neg b$ all occur as part of distinct solutions to the constraint.

Compilation proceeds by conjoining constraint circuits for $A \vee \neg A$ with $B \vee \neg B$, $\neg A$ with $B \vee \neg B$ and A with $\neg B$.



Decomposable AND nodes *compose* functions over *disjoint sets of variables*. These AND nodes represent Boolean functions $(A \vee \neg A) \wedge (B \vee \neg B)$, $\neg A \wedge (B \vee \neg B)$, and $A \wedge \neg B$.

The compiler disjoins $\neg A \wedge (B \vee \neg B)$, with $A \wedge \neg B$ and $(A \vee \neg A) \wedge (B \vee \neg B)$ with true, the multiplicative identity, guaranteeing alternating AND and OR nodes, for convenience. It is worth reiterating that every child of an OR node encodes disjoint solutions over the same set of variables.

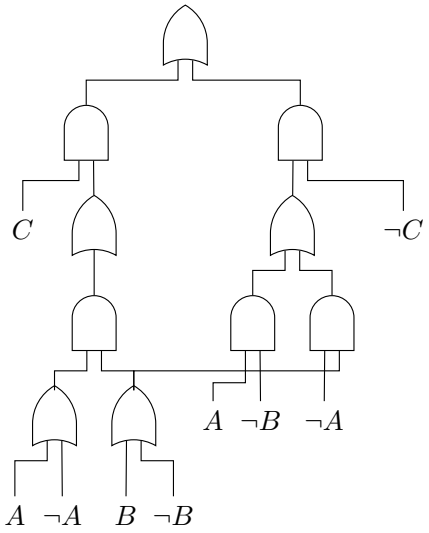
So far, we have compiled logical circuits for the formula

$$(\neg A \wedge (B \vee \neg B)) \vee (A \wedge \neg B) \tag{A.1}$$

as well as for the fomula

$$(A \vee \neg A) \wedge (B \vee \neg B) \tag{A.2}$$

What remains is to conjoin Equation A.1 with C, and Equation A.2 with $\neg C$, and disjoin the resulting circuits. What we get is a disjunction over the possible solutions of the constraint: predicting the presence of a barking animal implies the presence of a dog. Otherwise, there might or not be a dog.



Compilation techniques like the one we illustrated do not, however, escape the hardness of the problem: the compiled circuit can be exponential in the size of the constraint, *in the worst case*. *In practice*, however, we can obtain compact circuits because real-life logical constraints exhibit enough structure (e.g., repeated sub-problems) that can be easily exploited by a compiler [Darwiche and Marquis, 2002].

A.2 A Pseudo-Semantic Loss for Autoregressive Models with Logical Constraints

A.2.1 Circuit Construction

Any logical formula can be compiled into a smooth, deterministic and decomposable logical circuit: every disjunction factorizes the solution space into mutually exclusive events whereas every conjunction factorizes the function into two sub-functions over disjoint sets of variables. Here is a simple albeit potentially sub-optimal recipe: order variables lexicographically. Alternate OR and AND nodes. An OR node branches on the current variable being true or false, and has two children: a left (right) AND node whose children are the positive (negative) literal and the subtree corresponding to substituting the positive (negative) literal into the formula. Repeat while variables remain. We use the PySDD compiler which outputs circuits satisfying the above properties, in addition to structured-decomposability, which asserts that functions, or constraints, over the same variables decompose in the same manner. We say the above recipe is potentially sub-optimal as we use a fixed variable order. In general, there can be an exponential gap in the size of the logical circuit obtained using the worst and best variable order. Finding the best such order is, in general, NP-hard. However, in practice, compilers (PySDD included) use search heuristics that yield demonstrably-good orders.

A.2.2 Language Detoxification

The experiments were run on a server with an AMD EPYC 7313P 16-Core Processor @ 3.7GHz, 2 NVIDIA RTX A6000, and 252 GB RAM. Our LLM detoxification experiments utilized both GPUs using the Huggingface Accelerate [Gugger et al., 2022] library.

In order to construct our constraint, we start with the list of bad words[2] and their space-prefixed

[2]List downloaded from here.

variants[3]. We then tokenize this list of augment bad words, yielding 871 unique possibly-bad tokens (some tokens are only bad when considered in context with other tokens), in addition to an extra catch-all good token to which remaining tokens map to. Our constraint then disallows all sentences containing any of the words on the augmented list, starting at any of the sentence locations 0 through len(sentence) - len(word). The code to process the list of words, the code to create the constraint as well as the constraint itself will be released as part of our code.

Similar to SGEAT [Wang et al., 2022], the SoTA domain-adaptive training approach to detoxification, we finetune our model on self-generations as opposed to any external dataset. More specifically, we unpromptedly generate 100k samples using GPT-2 through Hugging Face [Wolf et al., 2020], which are then filtered through Perspective API, keeping only the 50% most nontoxic portion of the generations. We leverage the curated nontoxic corpus to further fine-tune the pretrained LLM with standard log-likelihood loss and adapt it to the nontoxic data domain. Unlike the two other tasks where we use model samples, we use the toxic portion of the corpus to which we apply our newly proposed pseudo-semantic loss. The intuition here is that the local perturbations of a toxic sentence are also toxic, and these are exactly the assignments whose probability we would like to penalize.

Our training script is adapted from that provided by Hugging Face[4]. We use a batch size of 16, a learning rate of 1e-5 with the AdamW optimizer [Loshchilov and Hutter, 2017] with otherwise default parameters. We did a grid search over the pseudo-semantic loss weight in the values $\{0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 2, 4, 8\}$. All other hyperparameters were left unchanged. Similar to [Wang et al., 2022], we use use nucleus sampling with $p = 0.9$ and a temperature of 1 during generation. A randomized 10k portion of the RealToxicityPrompts dataset was used to determine early stopping.

For only this task, our implementation of the pseudo-semantic loss makes use of top-k to construct the pseudo-likelihood distribution (lines 7-12 in Algorithm 4) due to the lack of computa-

[3]A word will be encoded differently whether it is space-prefixed or not.

[4]Downloaded from here.

tional resources. We constructed our distribution using only the top-10 good words and the top-470 toxic words.

A.2.3 Sudoku

The experiments were run on a server with an AMD EPYC 7313P 16-Core Processor @ 3.7GHz, 2 NVIDIA RTX A6000, and 252 GB RAM. Training utilized only one of the two GPUs.

We follow the experimental setting and dataset provided by Wang et al. [2019], consisting of 10K Sudoku puzzles, split into 9K training examples, and 1K test samples, all puzzles having 10 missing entries. Our model consists of an RNN with an input size of 9, a hidden dimension of 128, 5 layers, a tanh nonlinearity and a dropout of 0.2. We used Adam with default PyTorch parameters and a learning rate of 3e-4. We did a grid search over the pseudo-semantic loss weight in the values $\{0.01, 0.05\}$. Our constraint disallows any solution in which the rows, columns and square are not unique.

A.2.4 Warcraft Shortest Path

The experiments were run on a server with an AMD EPYC 7313P 16-Core Processor @ 3.7GHz, 2 NVIDIA RTX A6000, and 252 GB RAM. Training utilized only one of the two GPUs. We follow the experimental setting and dataset provided by Pogančić et al. [2019]. Our training set consists of $10,000$ terrain maps curated using Warcraft II tileset. We use a CNN-LSTM model for this task. Precisely, a ResNet-18 encodes the map to an embedding of dimension 128. An LSTM with 1 layer, and a hidden size of 512 then predicts the next edge in the shortest path conditioned on the input map and all previous edges. We used Adam with the default PyTorch parameters and a learning rate of 5e-4. We did a grid search over the pseudo-semantic loss weight in the values $\{0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1\}$. Our constraint disallows any prediction not a valid path connecting the upper left and lower right vertices.

Appendix B

Guarantees Within and Without Neural Networks

B.1 Semantic Probabilistic Layers for Neuro-Symbolic Learning

B.1.1 Proofs

Theorem 3.1 (Efficient inference in SPLs). *If $q(\mathbf{Y}; \boldsymbol{\Theta})$ and $c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X})$ are two smooth, decomposable and compatible circuits, then computing Equation* 3.2 *can be done in $\mathcal{O}(|q|\,|c|)$ time, where $|\cdot|$ denotes the circuit size. Furthermore, if they are also deterministic, then computing the MAP state can be done in $\mathcal{O}(|q|\,|c|)$ time. .*

We prove the first statement by first showing that the partition function $\mathcal{Z}(\boldsymbol{x})$ in Equation 3.2 can solved exactly in time $\mathcal{O}(|q|\,|c|)$. It will then follow from it that computing Equation 3.2 can be done in $\mathcal{O}(|q|\,|c| + |q| + |c|) \approx \mathcal{O}(|q|\,|c|)$ where the last two additive factors derive from evaluating q and c for an input configuration $(\boldsymbol{x}, \boldsymbol{y})$.

To do so, we will exploit two ingredients: i) the product of q and c can be represented as a smooth and decomposable circuit in time $\mathcal{O}(|q|\,|c|)$ [Vergari et al., 2021] and ii) any smooth and decomposable circuit guarantees tractable marginalization in time linear in its size [Choi et al., 2020a]. The next two propositions formalize these statements.

Proposition B.1 (Tractable product of circuits)**.** *Let $q(\mathbf{Y}; \boldsymbol{\Theta})$ and $c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X})$ be two smooth, decomposable circuits that are compatible over $\mathbf{Y}$ then computing their product as a circuit $r_{\boldsymbol{\Theta},\mathsf{K}}(\mathbf{X}, \mathbf{Y}) = q(\mathbf{Y}; \boldsymbol{\Theta}) \cdot c_{\mathsf{K}}(\mathbf{Y}, \mathbf{X})$ that is decomposable over $\mathbf{Y}$ can be done in $\mathcal{O}(|q|\,|c|)$. If both q and c are also deterministic, then r is as well.*

Proof. The proof directly follows from Theorem 3.2 from Vergari et al. [2021]. □

Note that $\mathcal{O}(|q|\,|c|)$ is a loose upperbound and the size of r is in practice smaller [Vergari et al., 2021].

Proposition B.2 (Tractable marginalization of circuits)**.** *Let $r(\mathbf{X}, \mathbf{Y})$ be a circuit that is smooth and decomposable over $\mathbf{Y}$ with input functions over $\mathbf{Y}$ that can be tractably marginalized out. Then for any variables $\mathbf{Y}' \subseteq \mathbf{Y}$ and their assignment $\boldsymbol{y}'$, the marginalization $\sum_{\boldsymbol{y}'} r(\boldsymbol{y}', \boldsymbol{y}'', \boldsymbol{x})$ can be computed exactly in time linear in the size of r, where $\mathbf{Y}'' = \mathbf{Y} \setminus \mathbf{Y}'$.*

Proof. The proof follows by considering that i) the input functionals in SPLs are simple distributions such as Bernoullis and indicators and can be easily marginalized in $\mathcal{O}(1)$ and ii) that for every configuration $\boldsymbol{x}$ of variables $\mathbf{X}$, $r(\mathbf{Y}, \boldsymbol{x})$ is a circuit only over $\mathbf{Y}$ and therefore Proposition 2.1 from Vergari et al. [2021] can be directly applied. □

Analogously, the second statement of Theorem 1 follows from Proposition B.1 and by recalling that the MAP state of a deterministic circuit can be computed in time linear in its size.

Proposition B.3 (Tractable MAP state of circuits (Choi et al. [2020a]))**.** *Let $r(\mathbf{X}, \mathbf{Y})$ be a circuit that is smooth and decomposable and deterministic over $\mathbf{Y}$ then for a configuration $\boldsymbol{x}$ its MAP state $\arg\max_{\boldsymbol{y}} r(\boldsymbol{x}, \boldsymbol{y})$ can be computed in time $\mathcal{O}(|r|)$.*

B.1.2 Compiling logical formulas into circuits

For our experiments we use standard compilation tools to obtain a constraint circuit starting from a propositional logical formula in conjunctive normal form. Specifically, we use Graphillion[1] to compile the constraints in the Warcraft pathfinding experiment into an SDD. For all other experiments, we use PySDD[2] [pys, 2017] a python SDD compiler [Darwiche, 2011b; Choi and Darwiche, 2013].

[1] https://github.com/takemaru/graphillion

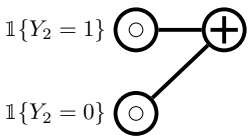
[2] https://github.com/wannesm/PySDD

We now illustrate step-by-step one example of such a compilation for a simple logical formula. Consider the constraint circuit c in Figure 3.3 encoding the constraint

$$(Y_{\mathsf{cat}} \implies Y_{\mathsf{animal}}) \wedge (Y_{\mathsf{dog}} \implies Y_{\mathsf{animal}}). \tag{B.1}$$

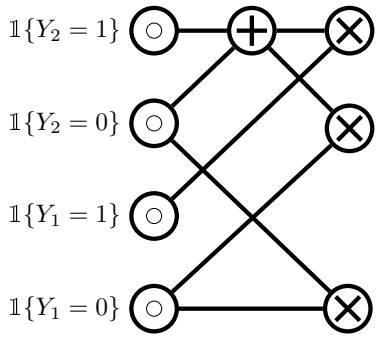
Intuitively, our aim is to compile the above logical formula into a *compact* form representing all possible assignments to $Y_{\mathsf{cat}}, Y_{\mathsf{dog}}, Y_{\mathsf{animal}}$ satisfying the above constraint. We compile such a constraint by proceeding in a bottom up fashion, where bottom-up compilation can be seen as composing Boolean sub-functions whose domain is determined by a variable ordering, also called vtree (see Sec. 3.1.2.3). In this example, we assume the function $f(Y_{\mathsf{animal}}, Y_{\mathsf{cat}}, Y_{\mathsf{dog}})$ decomposes as $f_1(Y_{\mathsf{animal}}) \cdot f_2(Y_{\mathsf{dog}}) \cdot f_3(Y_{\mathsf{cat}})$ We therefore start by compiling a constraint circuit that is a function of Y_{cat} and Y_{dog}, and compose it with a constraint circuit that is a function of Y_{animal} We first introduce input functionals representing indicators associated with $Y_{\mathsf{cat}}, Y_{\mathsf{dog}}, Y_{\mathsf{animal}}$. We will denote by Y_i the indicator $\mathbb{1}\{Y_i = 1\}$ and by $\neg Y_i$ the indicator $\mathbb{1}\{Y_i = 0\}$.

We start by disjoining the indicator Y_{cat} with $\neg Y_{\mathsf{cat}}$. This corresponds to introducing deterministic and smooth sum units in our circuits.



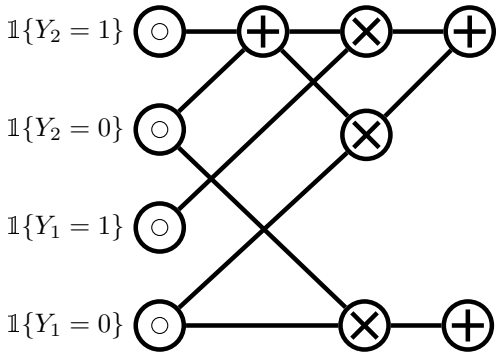
Deterministic sum units represent *disjoint solutions* to the logical formula, meaning there exists distinct assignments, characterized by the children, that satisfy the logical constraint e.g. $Y_{\mathsf{cat}}, Y_{\mathsf{dog}}, Y_{\mathsf{animal}}$ and $\neg Y_{\mathsf{cat}}, Y_{\mathsf{dog}}, Y_{\mathsf{animal}}$ are two distinct assignments which satisfy the constraint.

The compilation process proceeds by conjoining the constraint circuits for $Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}$ with Y_{dog}, $Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}$ with $\neg Y_{\mathsf{dog}}$, and $\neg Y_{\mathsf{cat}}$ with $\neg Y_{\mathsf{dog}}$.



A decomposable product unit *decomposes* functions over disjoint sets of variables. The above products represent the Boolean functions $(Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge Y_{\mathsf{dog}}$, $(Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge \neg Y_{\mathsf{dog}}$, and $\neg Y_{\mathsf{dog}} \wedge \neg Y_{\mathsf{cat}}$.

We disjoin $(Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge Y_{\mathsf{dog}}$ with $(Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge \neg Y_{\mathsf{dog}}$, and $\neg Y_{\mathsf{dog}} \wedge \neg Y_{\mathsf{cat}}$ with true, the logical multiplicative identity, guaranteeing alternating sum and product nodes, as mentioned in Sec. 3.1.2.1.



So far, we have compiled constraint circuits for the logical formulas

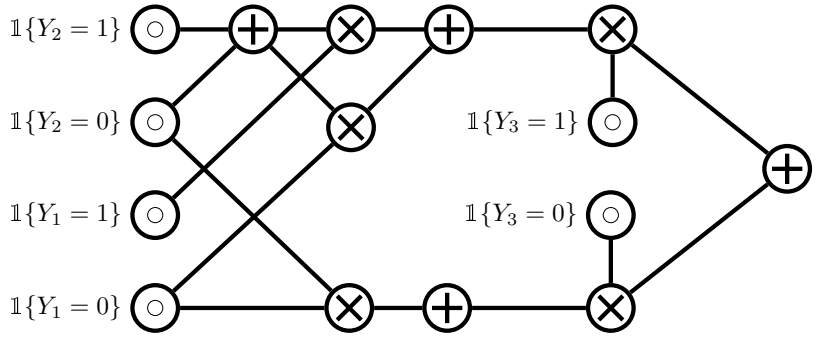
$$((Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge Y_{\mathsf{dog}}) \vee ((Y_{\mathsf{cat}} \vee \neg Y_{\mathsf{cat}}) \wedge \neg Y_{\mathsf{dog}})) \tag{B.2}$$

and

$$\neg Y_{\mathsf{dog}} \wedge \neg Y_{\mathsf{cat}}. \tag{B.3}$$

What remains is to conjoin Equation B.2 with Y_{animal}, and Equation B.3 with $\neg Y_{\mathsf{animal}}$, and disjoin the resulting constraint circuits. What we get is a mixture over the possible solutions: If we

predict there is a dog or a cat, or both, in e.g., an image, we better predict that there's an animal. Conversely, the absence of a dog and a cat from an image implies nothing as to the presence of an animal in the image.



Compilation techniques like the one we illustrated do not, however, escape the hardness of the problem: the compiled circuit can be exponential in the size of the constraint, *in the worst case*. *In practice*, nevertheless, we can obtain compact circuits because real-life logical constraints exhibit enough structure (e.g., they encode repeated sub-problems) that can be easily exploited by a compiler. We refer to the literature of compilation for details on this [Darwiche and Marquis, 2002].

B.1.3 Overparameterizing the single-circuit SPL

As mentioned in Def. 10, SPLs can be realized as a single circuit by first compiling a complex logical constraint into a deterministic constraint circuit, and then parameterizing it using a gating function of the network embeddings. Intuitively, this parameterization induces a probability distribution over the possible solutions of a logical formula encoded in the constraint circuit. The expressiveness of this distribution depends on the number of parameters of the constraint circuit, i.e., the number of weighted edges associated to sum units. As we would like to endow our single-circuit SPL with the ability to induce complex distributions, we devise *two strategies* to introduce more parameters than what the constraint circuit alone can offer: *replication* and *mixture multiplication*.

Replication works by maintaining m copies of the circuit, and taking their weighted average,

Algorithm 11 OVERPARAMETERIZE(c, k, cache, first_call)

Input: a smooth, deterministic, and structured-decomposable circuit c over variables $\mathbf{X}$, an overparameterization factor k, and a cache for memoization, and a flag to denote the first call
Output: an overparameterized, smooth, and structured-decomposable circuit c over $\mathbf{X}$
if $\boldsymbol{q} \in$ cache **then**
    return cache $[\boldsymbol{q}]$
if c is an input unit **then**
    nodes $\leftarrow [c]$
else if c is a sum unit **then**
    elements $\leftarrow [\,]$
    //For every product unit that is an input of c
    //recursively overparameterize its inputs,
    //which are sum units, and take their cross (cartesian) product
    for $(c_L, c_R) \in \mathsf{ch}(c)$ **do**
        left $\leftarrow$ OVERPARAMETERIZE(c_L, k)
        right $\leftarrow$ OVERPARAMETERIZE(c_R, k)
        elements.APPEND([CROSSPRODUCT(left, right)]
    $\mathsf{ch}(c) \leftarrow$ elements
    nodes $= [c] + [\text{COPY}(c)$ **for** $i = 1$ **to** $k]$
if first_call **then**
    //Create a sum unit whose inputs are nodes
    //and whose parameters are 1s.
    nodes $\leftarrow$ SUM(nodes, $\{1\}_{i=1}^{|\text{nodes}|}$)
cache$(c) \leftarrow$ nodes
return nodes

i.e., introducing a sum unit that mixes them [Peharz et al., 2020b]. Mixture multiplication, instead, substitutes a single local marginal distribution encoded by a sub-circuit rooted into a sum unit with k mixture models over the same scope. In practice, we create $k - 1$ copies of each sum units and rewire them by computing a cross product of their inputs as in Peharz et al. [2020b]. Algorithm 11 formalizes this process.

As mentioned in Def. 10, both strategies relax determinism. However, note that *they do not alter the support of the underlying distribution*. This guarantees that all the predictions will be consistent with the encoded constraint (D3) (Sec. 3.1.1).

B.1.4 Additional experimental details

B.1.4.1 Simple path prediction and preference learning

In the simple path prediction task, given a source and destination node in an unweighted grid $G = (V, E)$, the neural net needs to find the shortest unweighted path connecting them. We consider a 4×4 grid. The input $(\boldsymbol{x}, \boldsymbol{y})$ is a binary vector of length $|V| + |E|$, with the first $|V|$ variables indicating the source and destination nodes, and the subsequent $|E|$ variables indicating a subgraph $G' \subseteq G$. Each label is a binary vector of length $|E|$ encoding the unique shortest path in G'. For each example, we obtain G' by dropping one third of the edges in the graph G uniformly at random, filtering out the connected components with fewer than 5 nodes, to reduce degenerate cases, and then sample a source and destination node uniformly at random from G'. The dataset consists of 1600 such examples, with a $60/20/20$ train/validation/test split.

In the preference learning task, given a user's ranking over a subset of items, the network has to predict the user's ranking over the remaining items. We encode an ordering over n items as a binary matrix Y_{ij}, where for each $i, j \in 1, \ldots, n$, Y_{ij} indicates whether item i is the jth element in the ordering. The input $\boldsymbol{x}$ consist of the user's preference over 6 sushi types, and the model has to predict the users preferences (a strict total order) over the remaining 4. We use preference ranking data over 10 types of sushi for $5,000$ individuals, taken from [Mattei and Walsh, 2013b], and a 60/20/20 split.

We follow Xu et al. [2018a] in employing a 5-layer with 50 hidden units each and sigmoid activation functions, and 3-layer MLP with 50 hidden units each as a baseline for the simple path prediction, and preference learning, respectively. We equip this baselines with a FIL and additionally with the Semantic Loss [Xu et al., 2018a] (MLP+$\mathcal{L}_{\mathsf{SL}}$) or its entropic extension [Ahmed et al., 2022c] (MLP+NESYENT).

We compile the logical constraints into an SDD [Darwiche, 2011b] and then turn it into a constraint circuit c_{K} that is used for $\mathcal{L}_{\mathsf{SL}}$, NESYENT (Sec. 3.1.3) and our 1-circuit implementation of SPLs. To obtain the results for SPL in Table 3.2, we perform a grid search using the validation

set for a maximum of 2000 iterations, similar to Xu et al. [2018a]. We search over the learning rates in the range $\{1 \times 10^{-3}, 5 \times 10^{-3}, 1 \times 10^{-4}, 5 \times 10^{-4}\}$, the overparameterization factor k in the range $\{2, 4, 8\}$, as well as the number of circuit mixtures m in the range $\{2, 4, 8\}$, evaluating the model with the best performance on the validation set.

B.1.4.2 Hierarchical Multi-Label Classification

We follow the experimental setup of Giunchiglia and Lukasiewicz [2020] and evaluate SPL on 12 real-world HMLC tasks spanning four different domains: 8 functional genomics, 2 medical images, 1 microalgea classification, and 1 text categorization. These tasks are especially challenging due to the limited number of training samples, the large number of output classes, ranging from 56 to 4130, as well as the sparsity of the output space. We used the same train-validation-test splits and experimental setup as [Giunchiglia and Lukasiewicz, 2020]. For numeric features we replaced missing values by their mean, and for categorical features by a vector of zeros, and standardized all features. We used the validation splits to determine the number of layers in the gating function in the range $\{2, 4, 8\}$, the overparameterization factor in the range $\{2, 4, 8\}$, and the number of mixtures in the range $\{2, 4, 8\}$, keeping all other hyperparameters fixed. The final models were obtained by training using a batch size of 128 and early stopping with a patience of 20 on the validation set.

B.1.4.3 Warcraft pathfinding

We evaluate SPL on the more challenging task of predicting the minimum cost path in a weighted 12×12 grid imposed over terrain maps of Warcraft II [Pogančić et al., 2019]. Our setting differs from the one proposed by Pogančić et al. [2019] in two ways: i) a node only neighbors four nodes as instead of eight, excluding the diagonals; ii) the neural network predicts the edges in the path, as opposed to the vertices, resolving ambiguities in the previous task (note that a set of vertices can *might* ambiguously encode more than one path). Each vertex is assigned a cost corresponding to

the type of the underlying terrain (e.g., earth has lower cost than water). The minimum cost path between the top left and the bottom right vertices of the grid is encoded as an indicator matrix, and serves as a label.

We use Graphillion[3] to compile the path constraint, limiting our constraint to the set of paths whose length is less than 29, as determined on the training set.

As in [Pogančić et al., 2019] we use a ResNet18 [He et al., 2016b] with FIL optionally with $\mathcal{L}_{\mathsf{SL}}$ as a baseline. Given the largest size of the compiled constraint circuit c_{K} in this case 10^{10}, we use a two-circuit implementation of SPL. We use the identity function as our gating function and do a grid search over only the number of mixtures in the range $\{2, 4, 8\}$ in our model, keeping all other hyperparameters as proposed in [Pogančić et al., 2019].

B.1.4.4 A study on the effect of overparameterization in SPL

We now illustrate the effect that overparameterization has on the performance of the single-circuit SPL. To that end, we performed an ablation study, comparing single-circuit SPLs comprising a different number of circuit copies m for our replication strategy, a different number of layers in the gating function, denoted by *Gates*, and the overparameterization factor k as used in Algorithm 11 in our mixture multiplication strategy.

We report the exact match percentage of the predicted labels on the validation set of the 12 HMLC datasets in Table B.1. As a general trend, we can see that our overparameterization strategies pay off and in general more mixture nodes help ($k = 4$) as well as using more replicas ($m \geq 4$). The effect of employing a deeper gating function is less striking instead, with a two-layer gating function achieving highest performances on 9 datasets.

[3]https://github.com/takemaru/graphillion

Table B.1: A comparison of the performance of single-circuit SPL with different parameters: m, the number of circuit copies in our replication strategy; *gates*, the number of layers in the gating function; and k the overparameterization factor in the mixture multiplication strategy (Algorithm 11). We report the percentage of exact matches of the predicted labels on the validation set of the *HMLC* dataset, highlighting the best numbers in **boldface**. As can be seen, all datasets benefit from overparameterization.

DATASET	m: 2				m: 4				m: 8			
	GATES: 2		GATES: 4		GATES: 2		GATES: 4		GATES: 2		GATES: 4	
	k: 2	k: 4	k: 2	k: 4	k: 2	k: 4	k: 2	k: 4	k: 2	k: 4	k: 2	k: 4
CELLCYCLE	4.25	4.48	4.48	4.01	4.60	**4.83**	4.25	4.48	4.36	4.13	4.36	4.13
DERISI	2.26	2.02	2.14	2.26	**2.49**	2.26	2.38	2.38	**2.49**	2.38	2.26	**2.49**
EISEN	6.05	6.05	6.05	6.05	5.86	6.43	**6.81**	6.24	6.43	6.43	6.05	6.43
EXPR	5.42	4.83	5.18	5.30	4.83	**5.54**	**5.54**	5.18	**5.54**	5.42	5.18	5.42
GASCH1	5.56	5.79	5.67	5.91	5.44	5.67	6.03	**6.26**	5.79	5.79	**6.26**	6.03
GASCH2	4.00	4.24	4.83	**4.95**	4.12	4.00	4.12	4.36	4.24	3.53	4.24	4.59
SEQ	7.74	7.74	7.51	7.85	8.19	7.28	7.96	7.17	7.96	7.39	7.51	**8.42**
SPO	2.27	2.15	2.15	2.51	2.39	2.27	2.51	2.51	**2.87**	2.27	2.39	2.63
DIATOMS	53.71	**54.68**	50.16	51.29	53.23	52.10	49.35	48.23	52.90	52.58	46.61	47.26
ENRON	19.53	18.52	17.85	19.87	19.87	20.20	**20.54**	20.20	19.53	20.20	19.53	19.87
IMCLEF07A	86.97	87.03	86.27	86.60	87.00	**87.33**	86.50	86.70	87.07	86.90	87.00	86.83
IMCLEF07D	85.93	85.80	85.87	85.73	85.60	**86.50**	85.87	85.90	85.87	85.83	86.10	85.50

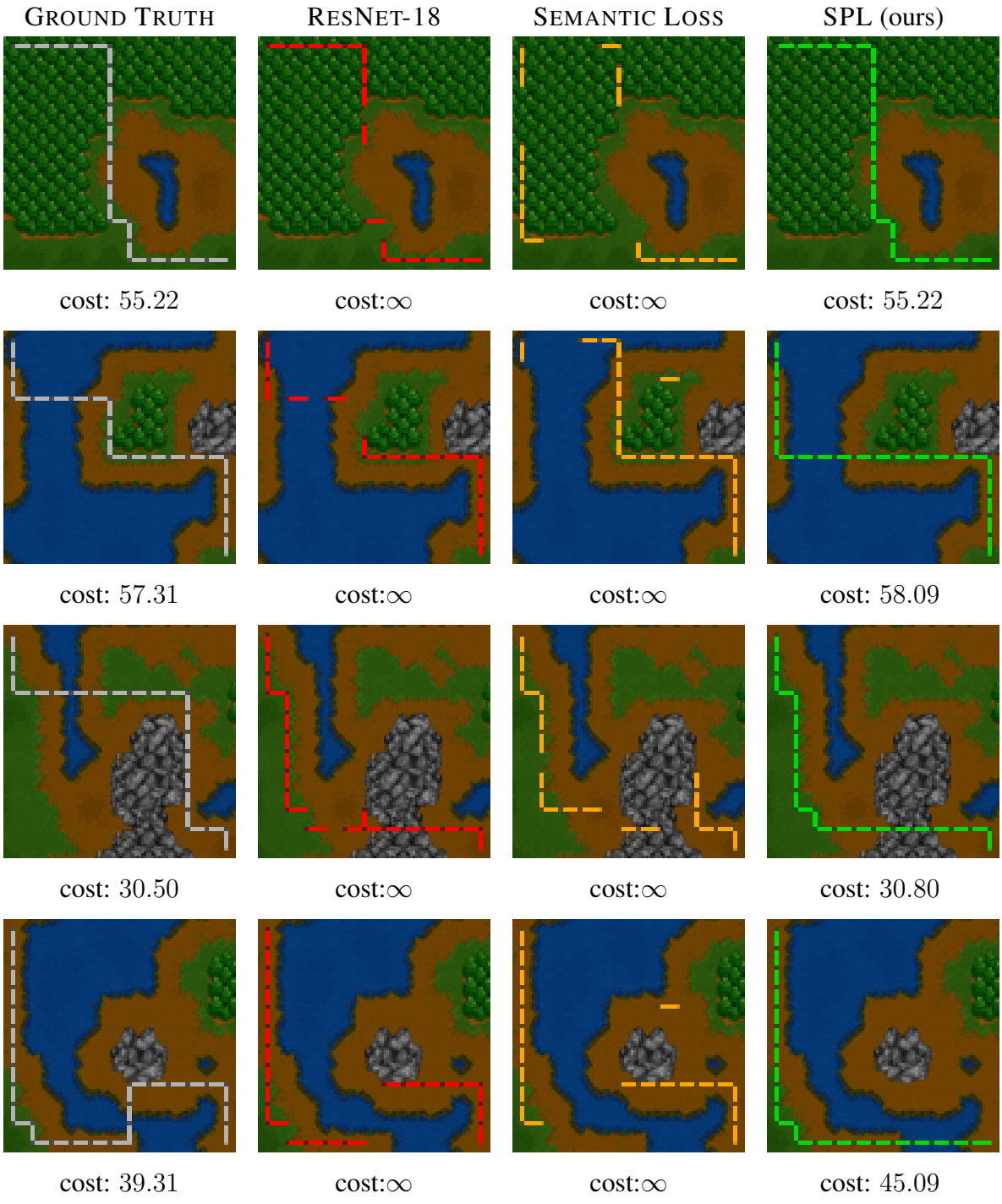


Figure B.1: **More examples of shortest path predictions in SPLs and competitors.** SPLs always deliver valid paths and even when these do not exact match the ground truth, they are very close in terms of their global cost. Paths from the baselines might yield a higher Hamming score (as they have more overlapping edges with the ground truth) but are invalid.

B.1.5 Timings

Table B.2: A comparison of the timings of the different methods used throughout our experiments. All timings are in seconds. The timings for HMLC datasets are obtained by averaging over the timings of an entire epoch. All other timings are the average over three function calls. An empty cell, denoted by a dash, indicates the method was not used for that dataset, and therefore its timing is unavailable.

DATASET	COMPILATION	$\mathcal{L}_{\mathsf{SL}}$	NESYENT	SPLS		
				PARAMETERIZE	CROSS-ENTROPY	MAP
CELLCYCLE	68	-	-	0.03	0.41	0.74
DERISI	68	-	-	0.01	0.21	0.37
EISEN	29	-	-	0.01	0.16	0.28
EXPR	68	-	-	0.00	0.11	0.19
GASCH1	68	-	-	0.02	0.42	0.77
GASCH2	68	-	-	0.03	0.40	0.74
SEQ	66	-	-	0.01	0.22	0.36
SPO	67	-	-	0.03	0.40	0.74
DIATOMS	8	-	-	0.00	0.09	0.14
ENRON	0.04	-	-	0.01	0.16	0.28
IMCLEF07A	0.35	-	-	0.00	0.06	0.11
IMCLEF07D	0.08	-	-	0.00	0.05	0.10
WARCRAFT	457	16.30	-	0.21	14.11	15.59
PREFERENCE [XU ET AL., 2018A]	0.024	0.035		0.00	0.00	0.01
SIMPLE PATH [XU ET AL., 2018A]	0.34	0.49		0.00	0.13	0.19

B.2 SIMPLE: A Gradient Estimator for k-subset sampling

B.2.1 Proofs

Theorem 1. *Let $p_{\boldsymbol{\theta}}(\sum_j z_j = k)$ be the probability of exactly-k of the unconstrained distribution parameterized by logits $\boldsymbol{\theta}$. Let $\alpha_i \coloneqq \log p_{\boldsymbol{\theta}}(z_i)$ denote the log marginals. For every variable Z_i,*

its conditional marginal is

$$p_{\boldsymbol{\theta}}\Big(z_i \mid \textstyle\sum_j z_j = k\Big) = \dfrac{\partial}{\partial \alpha_i} \log p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k). \tag{B.4}$$

Proof. We first rewrite the marginal $p_{\boldsymbol{\theta}}(\sum_i z_i = k)$ into a summation as the probability for all possible events by definition as follows.

$$p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k) = \sum_{\boldsymbol{z}:\sum_j z_j = k} \prod_{j:z_j=1} \exp(\alpha_j) \prod_{j:z_j=0} (1 - \exp(\bar{\alpha}_j)) \tag{B.5}$$

Here we assume that the probability of $z_j = 0$ is a constant term w.r.t. parameter α_j, i.e., $\frac{\partial}{\partial \alpha_j}(1 - \exp(\bar{\alpha}_j)) = 0$.[4] Further, the derivative of $p_{\boldsymbol{\theta}}(\sum_j z_j = k)$ w.r.t. α_i is as follows,

$$\begin{aligned}
\frac{\partial}{\partial \alpha_i} p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k) &= \frac{\partial}{\partial \alpha_i} \sum_{\boldsymbol{z}:\sum_j z_j = k \wedge z_i = 1} \prod_{j:z_j=1} \exp(\alpha_j) \prod_{j:z_j=0} (1 - \exp(\bar{\alpha}_j)) \\
&= \frac{\partial}{\partial \alpha_i} \exp(\alpha_i) \sum_{\boldsymbol{z}:\sum_j z_j = k \wedge z_i = 1} \prod_{j:z_j=1, j\neq i} \exp(\alpha_j) \prod_{j:z_j=0} (1 - \exp(\bar{\alpha}_j)) \\
&= \exp(\alpha_i) \sum_{\boldsymbol{z}:\sum_j z_j = k \wedge z_i = 1} \prod_{j:z_j=1, j\neq i} \exp(\alpha_j) \prod_{j:z_j=0} (1 - \exp(\bar{\alpha}_j)) \\
&= p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k \wedge z_i = 1),
\end{aligned}$$

where the first equality holds since terms corresponding to $z_i \neq 1$ has their derivative to be zero w.r.t. α_i. It further holds that

$$\begin{aligned}
\frac{\partial}{\partial \alpha_i} \log p_{\boldsymbol{\theta}}(\textstyle\sum_j z_j = k) &= \frac{\frac{\partial}{\partial \alpha_i} p_{\boldsymbol{\theta}}(\sum_j z_j = k)}{p_{\boldsymbol{\theta}}(\sum_j z_j = k)} \\
&= \frac{p_{\boldsymbol{\theta}}(\sum_j z_j = k \wedge z_i = 1)}{p_{\boldsymbol{\theta}}(\sum_j z_j = k)} \\
&= p_{\boldsymbol{\theta}}\Big(z_i \mid \textstyle\sum_j z_j = k\Big)
\end{aligned}$$

[4] In practice, this can be easily implemented. For example, in framework `Tensorflow`, it can be done by setting `tf.stop_gradients`.

which finishes our proof. □

Proposition 3. *Let* `Entropy` *be defined as in Algorithm 8. Given variables* $Z_1, \cdots, Z_n$ *and a* k*-subset distribution* $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$ *parameterized by* $\boldsymbol{\theta}$*, Algorithm 8 computes entropy of* $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)$.

Proof. In a slight abuse of notation, let z_n denote $z_n = 1$, and let $\bar{z}_n$ denote $z_n = 0$. Furthermore, we denote by $\sigma_n^k, \sigma_{n-1}^k$ and σ_{n-1}^{k-1} the events $\sum_{i=0}^{n} = k$, $\sum_{i=0}^{n-1} = k$ and $\sum_{i=0}^{n-1} = k-1$, respectively.

The entropy of the k-subset distribution is given by

$$H(\boldsymbol{Z}) = -\mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sigma_n^k)} \left[\log p(\boldsymbol{z})\right] = -\sum_{\boldsymbol{z}:\sigma_n^k} p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sigma_n^k) \log p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sigma_n^k)$$

We start by simplifying the expression for $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sigma_n^k)$, where, by the chain rule , the above is

$$\sum_{\boldsymbol{z}:\sigma_n^k} p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) + p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n)$$

Plugging the above in the expression for the entropy, distributing the sum over the product, we get

$$\begin{aligned}
&= -\sum_{\boldsymbol{z}:\sigma_n^k} p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) \\
&\qquad \cdot \log \left[p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) + p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n)\right] \\
&+ p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n) \\
&\qquad \cdot \log \left[p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) + p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n)\right],
\end{aligned}$$

where, since the two events $\bar{z}_n$ and z_n are mutually exclusive, we can simplify the above to

$$-\sum_{\boldsymbol{z}:\sigma_n^k} p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) \cdot \log \left[p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n)\right]$$

$$+ p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n) \cdot \log \left[p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n) \right].$$

Expanding the logarithms, rearranging terms, and using that conditional probabilities sum to 1 we get

$$\begin{aligned}
&p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \log p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) + p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \log p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \\
&+ p_{\boldsymbol{\theta}}(\bar{z}_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) \log p_{\boldsymbol{\theta}}(\sigma_{n-1}^k \mid \sigma_n^k, \bar{z}_n) \\
&+ p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \cdot p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n) \log p_{\boldsymbol{\theta}}(\sigma_{n-1}^{k-1} \mid \sigma_n^k, z_n) \\
&= -\mathbb{E}_{z_n \sim p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k)} \left[-\log p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k) \right] + \mathbb{E}_{z_n \sim p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k)} \left[H(\boldsymbol{Z}_{:n-1} | \sigma_n^k, z_n) \right] \\
&= H_b(Z_n \mid \sigma_n^k) + \mathbb{E}_{z_n \sim p_{\boldsymbol{\theta}}(z_n \mid \sigma_n^k)} \left[H(\boldsymbol{Z}_{:n-1} | \sigma_n^k, z_n) \right].
\end{aligned}$$

That is, simply stated, the entropy of the k-subset distribution decomposes as the entropy of the *constrained* distribution over Z_n, and average entropy of the distribution on the remaining variables.

As the base case, the entropy of the k-subset distribution when $k = n$ is 0; there is only one way in which to pick to choose n of n variables, and the k-subset distribution is therefore deterministic. □

B.2.2 Optimized Algorithms

Algorithm 12 is the optimized version of Algorithm 5, both of which compute the marginal probability of the exactly-k constraint. Algorithm 13 is the optimized version of Algorithm 6, both of which sample faithfully from the k-subset distribution.

Algorithm 12 $\texttt{PrExactlyk}(\boldsymbol{\theta}, l, u, k)$

Input: The logits $\boldsymbol{\theta}$ of the distribution, range of variable indices $[l, u]$, and the subset size k

Output: The exact marginal probability of variables summing up to k, $P(\sum_{i=l}^{u} X_i = k)$

if $l > u$ **then return** 0

if $l = u$ **then return** $p_\theta(X_l = k)$

for $m = 0$ to k **do**

$p_m = \texttt{PrExactlyk}(\boldsymbol{\theta}, l, \lfloor u/2 \rfloor, m) *$
$\texttt{PrExactlyk}(\boldsymbol{\theta}, \lfloor u/2 \rfloor + 1, u, k - m)$

return $\sum_{m=0}^{k} p_m$

Algorithm 13 $\texttt{Sample}(\boldsymbol{\theta}, l, u, k)$

Input: The logits $\boldsymbol{\theta}$ of the distribution, range of variable indices $[l, u]$, and the subset size k

Output: A sample $\boldsymbol{z} = (z_1, \ldots, z_n)$ from $p_{\boldsymbol{\theta}}(\boldsymbol{z} \mid \sum_i z_i = k)$

define $p(x = m) = p_m$, $m = 0, \cdots, k$

// with p_m as defined in Algorithm 12

sample m^* from p

$\boldsymbol{z}_{l:\lfloor u/2 \rfloor} = \texttt{Sample}(\boldsymbol{\theta}, l, \lfloor u/2 \rfloor, m^*)$

$\boldsymbol{z}_{\lfloor u/2 \rfloor + 1:u} = \texttt{Sample}(\boldsymbol{\theta}, \lfloor u/2 \rfloor + 1, u, k - m^*)$

return $\texttt{Concat}(\boldsymbol{z}_{l:\lfloor u/2 \rfloor}, \boldsymbol{z}_{\lfloor u/2 \rfloor + 1:u})$

B.2.3 Experimental Details

B.2.3.1 Synthetic Experiments

In this experiment we analyzed the behavior of various discrete gradient estimators for the k-subset distribution. We were interested in three different metrics: the bias of the the gradients estimators, the variance of the gradient estimators, as well as the average deviation of each estimated gradient from the exact gradient. We used cosine distance, defined as $1-$ cosine similarity as our measure of distance, as we typically care about the direction, not the magnitude of the gradient; the latter can be recovered using an appropriate learning rate. Following Niepert et al. [2021b], we chose a tractable 5-subset distribution, where $n = 10$, and were therefore limited to $\binom{10}{5} = 252$ possible subsets. We set the loss to $L(\boldsymbol{\theta}) = \mathbf{E}_{z \sim p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)}[\|\boldsymbol{z} - \mathbf{b}\|^2]$, where $\mathbf{b}$ is the groundtruth logits sampled from $\mathcal{N}(0, \mathbf{I})$. We used a sample size of 10000 to estimate each of our metrics.

B.2.3.2 Discrete Variational Auto-Encoder

We tested our SIMPLE gradient estimator in the discrete k-subset Variational Auto-Encoder (VAE) setting, where the latent variables model a probability distribution over k-subsets, and has a dimensionality of 20. The experimental setup is similar to those used in prior work on the Gumbel softmax tricks [Jang et al., 2017] and IMLE [Niepert et al., 2021b] The encoding and decoding functions of the VAE consist of three dense layers (encoding: 512-256-20x20; decoding: 256-512-784). As is commonplace in discrete VAEs, the loss is the sum of the reconstruction loss (binary cross-entropy loss on output pixels) and KL divergence of the $k-$subset distribution and the uniform distribution, known as the evidence lower bound, or the ELBO. The task being to learn a *sparse* generative model of MNIST. As in prior work, we use a batch size of 100 and train for 100 epochs, plotting the test loss after each epoch. We use the standard Adam settings in Tensorflow 2.x, and do not employ any learning rate scheduling. The encoder network consists of an input layer with dimension 784 (we flatten the images), a dense layer with dimension 512 and ReLu activation, a dense layer with dimension 256 and ReLu activation, and a dense layer with dimension $400(20 \times 20)$ which outputs $\boldsymbol{\theta}$ and no non-linearity SIMPLE takes $\boldsymbol{\theta}$ as input and outputs a discrete latent code of size 20×20. The decoder network, which takes this discrete latent code as input, consists of a dense layer with dimension 256 and ReLu activation, a dense layer with dimension 512 and ReLu activation, and finally a dense layer with dimension 784 returning the logits for the output pixels. Sigmoids are applied to these logits and the binary cross-entropy loss is computed. To obtain the best performing model of each of the compared methods, we performed a grid search over the learning rate in the range $[1 \times 10^{-3}, 5 \times 10^{-4}]$, λ in the range $[1\times10^{-3}, 1\times10^{-2}, 1\times10^{-1}, 1\times10^{0}, 1\times10^{1}, 1\times10^{2}, 1\times10^{3}]$, and for SoG I-MLE, the temparature τ in the range $[1 \times 10^{-1}, 1 \times 10^{0}, 1 \times 10^{1}, 1 \times 10^{2}]$

We will now present a formal proof on how to compute the KL-divergence between the k-subset distribution and a uniform distribution tractably and exactly.

Proposition B.4. *Let $p_{\boldsymbol{\theta}}\left(\mathbf{z} \mid \sum_i z_i = k\right)$ be a k-subset distribution parameterized by $\boldsymbol{\theta}$ and $\mathcal{U}(\boldsymbol{z})$*

be a uniform distribution on the constrained space $\mathcal{C} = \{\boldsymbol{z} \mid \sum_i z_i = k\}$. *Then the KL-divergence between distribution* $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)$ *and* $\mathcal{U}(\boldsymbol{z})$ *can be computed by*

$$D_{KL}(p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k) \mid\mid \mathcal{U}(\boldsymbol{z})) = -H(\boldsymbol{z}) + \log \binom{n}{k},$$

where H *denote the entropy of distribution* $p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)$.

Proof. By the definition of KL divergence, it holds that

$$\begin{aligned}
&D_{\mathrm{KL}}(p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k) \mid\mid \mathcal{U}(\boldsymbol{z})) \\
&= \sum_{\boldsymbol{z} \in \mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k) \cdot \log \frac{p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)}{U(\boldsymbol{z})} \\
&= (\sum_{\boldsymbol{z} \in \mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k) \ \log p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k)) - \sum_{\boldsymbol{z} \in \mathcal{C}} p_{\boldsymbol{\theta}}(\mathbf{z} \mid \sum_i z_i = k) \log U(\boldsymbol{z}) \\
&= -H(\boldsymbol{z}) + \log \binom{n}{k}.
\end{aligned}$$

The last equality holds since $U(\boldsymbol{z}) \equiv 1/\binom{n}{k}$. □

B.2.3.3 Learning to Explain

The BEERADVOCATE dataset [McAuley et al., 2012] consists of free-text reviews and ratings for 4 different aspects of beer: appearance, aroma, palate, and taste. The training set has 80k reviews for the aspect APPEARANCE and 70k reviews for all other aspects. The maximum review length is 350 tokens. We follow Niepert et al. [2021b] in computing 10 different evenly sized validation/test splits of the 10k held out set and compute mean and standard deviation over 10 models, each trained on one split. In addition to the ratings for all reviews, each sentence in the test set contains annotations of the words that best describe the review score with respect to the various aspects. Following the experimental setup of recent work [Paulus et al., 2020; Niepert et al., 2021b], we address the problem introduced by the L2X paper [Chen et al., 2018] of learning

a k-subset distribution over words that best explain a given aspect rating. Subset precision was computed using a set of 993 annotated reviews. We use pre-trained word embeddings from Lei et al. [2016][5] We use the standard neural network architecture from prior work Chen et al. [2018]; Paulus et al. [2020] with 4 convolutional and one dense layer. This neural network outputs the parameters $\boldsymbol{\theta}$ of the k-subset distribution over k-hot binary latent masks with $k \in \{5, 10, 15\}$. We train for 20 epochs using the standard Adam settings in Tensorflow 2.x, and no learning rate schedule. We always evaluate the model with the best validation MSE among the 20 epochs.

[5]http://people.csail.mit.edu/taolei/beer/.

Appendix C

Scaling

C.1 Scaling Tractable Probabilistic Circuits: A Systems Perspective

C.1.1 Algorithm Details

In this section, we provide additional details of the design of PyJuice. Specifically, we introduce the layer partitioning algorithm that divides a layer into groups of node blocks with a similar number of children in Sec. C.1.1.1, and describe the details of the backpropagation algorithm in Sec. C.1.2.

C.1.1.1 The Layer Partitioning Algorithm

The layer partitioning algorithm receives as input a vector of integers `nchs` where each number denotes the number of child node blocks connected to a node block in the layer. It also receives as input the maximum number of groups to be considered (denoted `G`) and a sparsity tolerance threshold $\texttt{tol} \in (0, 1]$. Our goal is to search for a set of n (at most `G`) groups with capacities $g_1, \ldots, g_n$, respectively. Every number in `nchs` is then placed into the group with the smallest capacity it can fit in. Every number in `nchs` must fit in a group. Assume there are k_i numbers assigned to group i, the overhead/cost w.r.t. a partitioning $\{g_1, \ldots, g_n\}$ is defined as $\sum_{i \in [n]} k_i \cdot g_i$. Our goal is to find a partitioning with overhead smaller than $\texttt{sum}(\texttt{nchs}) \cdot (1 + \texttt{tol})$.

Algorithm 14 Partition a layer into groups

Inputs: a list of child node (block) counts of the current layer $\texttt{nchs} \in \mathbb{Z}^N$ (N is the number of node blocks in the layer)
Inputs: the maximum number of groups $\texttt{G}$, the sparsity tolerance threshold $\texttt{tol} \in (0, 1]$
$\texttt{uni\_nchs}, \texttt{counts} \leftarrow \texttt{unique}(\texttt{nchs}, \texttt{sorted} = \texttt{True})$ (get the unique values and their appearance counts; we require the numbers in `uni_nchs` to be sorted in ascending order)
$\texttt{L} \leftarrow \texttt{length}(\texttt{uni\_nchs})$
$\texttt{target\_overhead} \leftarrow \lceil \texttt{sum}(\texttt{uni\_nchs} * \texttt{counts}) * (1.0 + \texttt{tol}) \rceil$ (get the target overhead)
$\texttt{cum\_counts} \leftarrow \texttt{cumsum}(\texttt{counts})$
$\texttt{dp}, \texttt{backtrace} \leftarrow (0)_{\texttt{L}\times\texttt{G}+1} \in \mathbb{R}^{\texttt{L}\times\texttt{G}+1}, (0)_{\texttt{L}\times\texttt{G}+1} \in \mathbb{Z}^{\texttt{L}\times\texttt{G}+1}$
for $i = 0$ **to** $\texttt{L} - 1$ **do**
    $\texttt{dp}[i, 1] \leftarrow \texttt{uni\_nchs}[i] * \texttt{cum\_counts}[i]$
Main DP algorithm
$\texttt{target\_n\_group} \leftarrow \texttt{G}$
for $\texttt{n\_group} = 2$ **to** $\texttt{G}$ **do**
    $\texttt{dp}[0, \texttt{n\_group}] \leftarrow \texttt{uni\_nchs}[0] * \texttt{cum\_counts}[0]$
    $\texttt{backtrace}[0, \texttt{n\_group}] \leftarrow 0$
    for $\texttt{i} = 1$ **to** $\texttt{L} - 1$ **do**
        $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{inf}, -1$
        for $\texttt{j} = 0$ **to** $\texttt{i} - 1$ **do**
            $\texttt{curr\_overhead} \leftarrow \texttt{dp}[\texttt{j}, \texttt{n\_group} - 1] + \texttt{uni\_nchs}[\texttt{i}] * (\texttt{cum\_counts}[\texttt{i}] - \texttt{cum\_counts}[\texttt{j}])$
            if $\texttt{curr\_overhead} < \texttt{min\_overhead}$ **then**
                $\texttt{min\_overhead}, \texttt{best\_idx} \leftarrow \texttt{curr\_overhead}, \texttt{j}$
        $\texttt{dp}[\texttt{i}, \texttt{n\_group}], \texttt{backtrace}[\texttt{i}, \texttt{n\_group}] \leftarrow \texttt{min\_overhead}, \texttt{best\_idx}$
    if $\texttt{dp}[-1, \texttt{n\_group}] <= \texttt{target\_overhead}$ **then**
        $\texttt{target\_n\_group} \leftarrow \texttt{n\_group}$
Backtrace
$\texttt{group\_sizes} \leftarrow (0)_{\texttt{target\_n\_group}} \in \mathbb{Z}^{\texttt{target\_n\_group}}$
$\texttt{i} \leftarrow \texttt{L} - 1$
for $n = \texttt{target\_n\_group}$ **to** 1 **do**
    $\texttt{group\_sizes}[n - 1] \leftarrow \texttt{i}$
    $\texttt{i} \leftarrow \texttt{backtrace}[\texttt{i}, \texttt{target\_n\_group}]$
return `group_sizes`

We use a dynamic programming algorithm that is based on the following main idea. We first sort the numbers in `nchs` in ascending order. Denote `L` as the size of `nchs`, we maintain a scratch table of size $\texttt{L} \times \texttt{G}$ whose ith row and jth column indicates the best possible overhead achieved by the first i numbers in `nchs` when having in total at most j partitions. The update formula of the DP

table is

$$\texttt{dp}[i,j] \leftarrow \min_{k \in [i-1]} \texttt{dp}[k, j-1] + \texttt{nchs}[i] \cdot (i-k), \tag{C.1}$$

where we try to find the best place (k) to put a new group/partition. By simultaneously maintaining a matrix for backtracking, we can retrieve the best partition found by the algorithm.

The algorithm is shown in Algorithm 14. A practical trick to speed it up is to coalesce the identical values in `nchs` as done in line 3. Lines 7-9 initialize the buffers, and lines 11-23 are the main loop of the DP algorithm. Finally, the result partitioning is retrieved using lines 25-29.

Theoretical guarantee. Algorithm 14 is guaranteed to find an optimal grouping given a pre-specified number of groups, and is fairly efficient in practice. We formally state the problem in the following and provide the proof and analysis as follows.

As described in Appendix A.1, the grouping algorithm essentially takes as input a list of "# child node blocks" for each parent node block in a layer, and the goal is to partition all parent node blocks into K groups such that we minimize the following cost: the sum of the cost of each group, where the cost of a group is the maximum "# child node blocks" in the group times the number of parent node blocks in the group. In the following, we first demonstrate that the proposed dynamic programming (DP) algorithm (Algorithm 2) can retain the optimal cost for every K. We then proceed to analyze the time and space complexity of the algorithm.

To simplify notations, we assume the input is a vector of integers $[n_1, \ldots, n_N]$. We assume without loss of generality that the numbers are sorted because if not, we can apply any sorting algorithm. The main idea of the DP algorithm is to maintain a table termed dp of size N times K, where $\texttt{dp}[i,j]$ indicates the optimal cost when partitioning the first i integers into j groups. For the base cases, we can set $\texttt{dp}[i,1] = n_i(\forall i)$ and $\texttt{dp}[1,j] = n_1(\forall j)$. For the inductive case, we have Equation C.1. It is straightforward to verify that when $\texttt{dp}[k, j-1](\forall k \in [1, i-1])$ are optimal, $\texttt{dp}[i,j]$ is also optimal. Therefore, for any K, Algorithm 14 computes the optimal grouping strategy for K groups.

Efficiency. We then focus on the runtime. Given N and K, Algorithm 14 requires $\mathcal{O}(KN^2)$ runtime and $\mathcal{O}(KN)$ memory, which is undesired for large N (in practice, we set K to be smaller than 10). However, as demonstrated in Algorithm 14 (line 3), we only need to enumerate through the unique values in $[n_1, \ldots, n_N]$, which could potentially lower the computation cost significantly. Even when we are dealing with highly non-structured PCs, we can always round the numbers up to a minimum integer that is divisible by a small integer such as 10. This allows us to achieve a decent approximated solution with much less computation time.

C.1.2 Details of the Backpropagation Algorithm for Sum Layers

Algorithm 15 Backward pass of a sum layer group w.r.t. parameters

1: **Inputs:** log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, $\texttt{sum\_ids}$, $\texttt{prod\_ids}$, $\texttt{param\_ids}$
2: **Inputs:** # sum nodes: M, # product nodes: N, batch size: B
3: **Inputs:** block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
4: **Inputs:** number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
5: **Outputs:** flows of params $\texttt{f}_{\text{params}}$
6: **Kernel launch:** schedule to launch $C_M \times C_N$ thread-blocks with $\texttt{m}=0, \ldots, C_M-1$ and $\texttt{n}=0, \ldots, C_N-1$
7: $\texttt{cum} \leftarrow (0)_{K_M \times K_N} \in \mathbb{R}^{K_M \times K_N}$ ▷ Scratch space on SRAM
8: $\texttt{ms}, \texttt{me} \leftarrow \texttt{sum\_ids}[\texttt{m}], \texttt{sum\_ids}[\texttt{m}] + K_M$
9: $\texttt{ns}, \texttt{ne} \leftarrow \texttt{prod\_ids}[\texttt{m}, \texttt{n}], \texttt{prod\_ids}[\texttt{m}, \texttt{n}] + K_N$
10: **for** $\texttt{b} = 0$ **to** $C_B - 1$ **do**
11: $\quad \texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b}+1) \cdot K_B$ ▷ Start and end batch index
12: $\quad$ Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
13: $\quad$ Load $\boldsymbol{l}_{\text{p}} \leftarrow \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \in \mathbb{R}^{K_N \times K_B}$ to SRAM
14: $\quad \texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
15: $\quad \texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim}=0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
16: $\quad \texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
17: $\quad \texttt{scaled\_emars} \leftarrow \texttt{transpose}(\texttt{exp}(\boldsymbol{p}_{\text{p}} + \texttt{log\_nf\_max})) \in \mathbb{R}^{K_B \times K_N}$
18: $\quad \texttt{partial\_flows} \leftarrow \texttt{matmul}(\texttt{log\_nf\_sub}, \texttt{scaled\_emars}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
19: $\quad \texttt{cum} \leftarrow \texttt{cum} + \texttt{partial\_flows}$
20: $\texttt{ps}, \texttt{pe} \leftarrow \texttt{param\_ids}[\texttt{m}, \texttt{n}], \texttt{param\_ids}[\texttt{m}, \texttt{n}] + K_M \cdot K_N$
21: $\texttt{f}_{\text{params}}[\texttt{ps:pe}] \leftarrow \texttt{f}_{\text{params}}[\texttt{ps:pe}] + \boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}] * \texttt{cum.view}(K_M * K_N)$

We compute the backward pass with respect to the inputs and the parameters of the sum layer in two different kernels as we need two different layer partitioning strategies to improve efficiency. In the following, we first introduce the backpropagation algorithm for the parameters since it reuses the index tensors compiled for the forward pass (i.e., `sum_ids`, `prod_ids`, and `param_ids`).

The algorithm is shown in Algorithm 15. In addition to the log-probabilities of the product nodes (i.e., $\boldsymbol{l}_{\text{prod}}$), the log-probabilities of the sum nodes (i.e., $\boldsymbol{l}_{\text{sum}}$), and the flattened parameters (i.e., $\boldsymbol{\theta}_{\text{flat}}$), the algorithm takes as input the flows $\texttt{f}_{\text{sum}}$ computed for the sum nodes. Following Definition 4.2, we can compute the flow w.r.t. the sum parameters as

$$\mathrm{F}_{n,c}(\boldsymbol{x}) := \theta_{n,c} \cdot p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x}).$$

Similar to Algorithm 9, we partition the sum nodes, product nodes, and samples into blocks of size K_M, K_N, and K_B, respectively. We schedule to launch $C_M \times C_N$ thread-blocks, each responsible for computing the parameter flows for a block of $K_M \times K_N$ parameter flows. The main loop (line 10) iterates through blocks of K_B samples. In every iteration, we first load the log-probabilities (i.e., $\boldsymbol{l}_{\text{s}}$ and $\boldsymbol{l}_{\text{p}}$) and the sum node flows (i.e., $\boldsymbol{f}_{\text{s}}$) to compute the partial flow $p_c(\boldsymbol{x})/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ for the block of samples (note that this equals $\mathrm{F}_{n,c}(\boldsymbol{x})/\theta_{n,c}$. The partial flows are accumulated in the matrix `cum` initialized in line 7. After processing all blocks of samples, we add back the parameter flows by accumulating $\texttt{cum} * [\text{the corresponding parameters}]$ in line 21.

As elaborated in Sec. 4.2.4, if we use the same set of index tensors used in the forward pass, we have the problem of different thread-blocks needing to write (partial) flows to the same input product node blocks. Therefore, we do a separate compilation step for the backward pass. Consider a sum layer with sum node blocks of size K_M and child product node blocks of size K_N. We first partition the C_N children into groups such that every child node block in a group has a similar number of parents. This is done by the dynamic programming algorithm described in Sec. C.1.1.1.

Similar to the compilation procedure of the forward pass, for a group with C_N child node blocks (assume every block has C_M blocks of parents), we generate three index tensors: $\texttt{ch\_ids} \in \mathbb{Z}^{C_N}$

and $\texttt{par\_ids}, \texttt{par\_param\_ids} \in \mathbb{Z}^{C_N \times C_M}$. $\texttt{ch\_ids}$ contains the initial index of all C_N child node blocks belonging to the group. For the ith node block in the group (i.e., the product node block with the initial index $\texttt{ch\_ids}[i]$), $\texttt{par\_ids}[i, :]$ encode the start indices of its parent sum node blocks, and $\texttt{par\_param\_ids}[i, :]$ represent the corresponding initial parameter indices.

The main algorithmic procedure is very similar to $Algorithm\ 9$. Specifically, the kernel schedules to launch $C_N \times C_B$ thread-blocks each computing a block of $K_N \times K_B$ product node flows. In the main loop (line 9), we iterate through all C_M parent node blocks. In lines 13-16, we are essentially computing $\theta_{n,c}/p_n(\boldsymbol{x}) \cdot \mathrm{F}_n(\boldsymbol{x})$ (notations inherited from Definition 4.2) for the block of $K_N \times K_B$ values using the logsumexp trick. Finally, we store the results back to $\texttt{f}_{\text{prod}}$.

Algorithm 16 Backward pass of a sum layer group w.r.t. inputs

1: **Inputs:** log-probs of product nodes $\boldsymbol{l}_{\text{prod}}$, log-probs of sum nodes $\boldsymbol{l}_{\text{sum}}$, flows of sum nodes $\texttt{f}_{\text{sum}}$, flattened parameter vector $\boldsymbol{\theta}_{\text{flat}}$, $\texttt{ch\_ids}, \texttt{par\_ids}, \texttt{par\_param\_ids}$
2: **Inputs:** # sum nodes: M, # product nodes: N, batch size: B
3: **Inputs:** block sizes K_M, K_N, K_B for the sum node, product node, and batch dimensions, respectively
4: **Inputs:** number of sum node blocks C_M; number of product node blocks C_N; number of batch blocks C_B
5: **Outputs:** flows of inputs $\texttt{f}_{\text{prod}}$
6: **Kernel launch:** schedule to launch $C_N \times C_B$ thread-blocks with $\texttt{n} = 0, \ldots, C_N - 1$ and $\texttt{b} = 0, \ldots, C_B - 1$
7: $\texttt{cum} \leftarrow (-\infty)_{K_N \times K_B} \in \mathbb{R}^{K_N \times K_B}$ ▷ Scratch space on SRAM
8: $\texttt{bs}, \texttt{be} \leftarrow \texttt{b} \cdot K_B, (\texttt{b} + 1) \cdot K_B$
9: **for** $\texttt{m} = 0$ **to** $C_M - 1$ **do**
10: $\quad \texttt{ps}, \texttt{pe} \leftarrow \texttt{par\_param\_ids}[\texttt{n}, \texttt{m}]$
11: $\quad$ Load $\boldsymbol{f}_{\text{s}} \leftarrow \texttt{f}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ and $\boldsymbol{l}_{\text{s}} \leftarrow \boldsymbol{l}_{\text{sum}}[\texttt{ms:me}, \texttt{bs:be}] \in \mathbb{R}^{K_M \times K_B}$ to SRAM
12: $\quad$ Load $\boldsymbol{\theta} \leftarrow \texttt{transpose}(\boldsymbol{\theta}_{\text{flat}}[\texttt{ps:pe}].\texttt{view}(K_M, K_N)) \in \mathbb{R}^{K_N \times K_M}$ to SRAM
13: $\quad \texttt{log\_nf} \leftarrow \log(\boldsymbol{f}_{\text{s}}) - \boldsymbol{l}_{\text{s}}$
14: $\quad \texttt{log\_nf\_max} \leftarrow \texttt{max}(\texttt{log\_nf}, \texttt{dim} = 0) \in \mathbb{R}^{1 \times K_B}$ ▷ Compute on chip
15: $\quad \texttt{log\_nf\_sub} \leftarrow \texttt{exp}(\texttt{log\_nf} - \texttt{log\_nf\_max}) \in \mathbb{R}^{K_M \times K_B}$
16: $\quad \texttt{partial\_flows} \leftarrow \texttt{matmul}(\boldsymbol{\theta}, \texttt{log\_nf\_sub}) \in \mathbb{R}^{K_M \times K_N}$ ▷ With Tensor Cores
$\quad \texttt{cum} \leftarrow \texttt{where}(\texttt{log\_nf\_max} > \texttt{cum},$
$\qquad \texttt{log}(\texttt{partial\_flows} + \texttt{exp}(\texttt{cum} - \texttt{log\_nf\_max}) + \texttt{log\_nf\_max},$
17: $\qquad \texttt{log}(\texttt{exp}(\texttt{log\_nf\_max} - \texttt{cum}) \cdot \texttt{partial\_flows} + 1) + \texttt{cum})$
18: $\texttt{ns}, \texttt{ne} \leftarrow \texttt{ch\_ids}[\texttt{n}], \texttt{ch\_ids}[\texttt{n}] + K_N$
19: $\texttt{f}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}] \leftarrow \exp(\texttt{cum} + \boldsymbol{l}_{\text{prod}}[\texttt{ns:ne}, \texttt{bs:be}])$

C.1.3 PCs with Tied Parameters

Formally, PCs with tied parameters are PCs containing same sub-structures in different parts of its DAG. Although the nodes in these sub-structures could have different semantics, they can have shared/tied parameters. For example, in homogeneous HMMs, although the transition probabilities

between different pairs of consecutive latent variables are represented by different sets of nodes and edges in the PC, they all have the *same* set of probability parameters.

PyJuice can be readily adapted to PCs with tied parameters. For the forward pass, we just need the compiler to assign the same parameter indices in `param_ids`. Similarly, we only need to slightly change the compilation procedure of `par_param_ids`. One notable difference is that in the backward pass w.r.t. the parameters, multiple thread-blocks would need to write partial flows to the same memory addresses, which leads to inter-thread-block barriers. We implemented a memory-IO tradeoff by letting the compiler create new sets of memory addresses to store the parameter flows when the number of thread-blocks writing to the same address is greater than a predefined threshold (by default set to 4).

C.1.4 Additional Technical Details

C.1.4.1 Block-Sparsity of Common PC Structures

Most commonly-adopted PC structures such as PD [Poon and Domingos, 2011], RAT-SPN [Peharz et al., 2020b], and HCLT [Liu and Van den Broeck, 2021] have block-sparse sum layers because one of the key building blocks of the structure is a set of sum nodes fully connected to their inputs. Therefore, every sum layer must contain multiple fully-connected blocks of sum and product nodes, and hence they are block sparse.

C.1.4.2 Relation Between PC Flows and Gradients

We first show the equality for the node flows:

$$\mathrm{F}_n(\boldsymbol{x}) = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{C.2}$$

We do the proof by induction. As a base case, we have by definition that $\mathrm{F}_{n_r}(\boldsymbol{x}) = \partial \log p_{n_r}(\boldsymbol{x}) / \partial \log p_{n_r}(\boldsymbol{x}) = 1$.

Next, suppose n is a sum or an input node, and for all its parents m, we have Equation C.2 is satisfied by induction. Since all parents of n are product nodes, we have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})},$$

where $p_{n \to m}(\boldsymbol{x})$ denotes the probability carried by the edge from n to m.

Finally, suppose n is a product node and thus all its parents are sum nodes. We have

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \mathrm{F}_m(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} \frac{\theta_{m,n} \cdot p_n(\boldsymbol{x})}{p_m(\boldsymbol{x})} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_m(\boldsymbol{x})}, \tag{C.3}$$

$$= \sum_{m \in \mathsf{pa}(n)} \theta_{m,n} \cdot p_n(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})}. \tag{C.4}$$

Denote $p_{n \to m}(\boldsymbol{x}) = \theta_{m,n} \cdot p_n(\boldsymbol{x})$ as the probability carried on the edge (m, n). Since $p_m(\boldsymbol{x}) = \sum_{n' \in \mathsf{ch}(m)} p_{n' \to m}(\boldsymbol{x})$, we have

$$\forall n \in \mathsf{ch}(m), \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_m(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})}.$$

Plug in the above equation on $\mathrm{F}_n(\boldsymbol{x})$, this results in

$$\mathrm{F}_n(\boldsymbol{x}) = \sum_{m \in \mathsf{pa}(n)} p_{n \to m}(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_{n \to m}(\boldsymbol{x})} = \sum_{m \in \mathsf{pa}(n)} \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{n \to m}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_n(\boldsymbol{x})}. \tag{C.5}$$

We move on to demonstrate the following relation:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \theta_{n,c} \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \theta_{n,c}} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}},$$

where n is a sum node and c is one of its children. We reuse the results derived in Equations C.4

and C.5, where we replace n with c and m with n:

$$\mathrm{F}_{n,c}(\boldsymbol{x}) = \frac{\theta_{n,c} \cdot p_c(\boldsymbol{x})}{p_n(\boldsymbol{x})} \cdot \mathrm{F}_n(\boldsymbol{x}) = \theta_{n,c} \cdot p_c(\boldsymbol{x}) \cdot \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial p_n(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log p_{c \to n}(\boldsymbol{x})} = \frac{\partial \log p_{n_r}(\boldsymbol{x})}{\partial \log \theta_{n,c}}.$$

C.1.5 Experimental Details

C.1.5.1 The Adopted Block-Sparse PC Layer

The PC layer contains 200 independent fully-connected sets of nodes. Every connected subset consists of 1024 sum nodes and 1024 product nodes. When compiling the layer, we divide the layer into blocks of size 32. When dropping 32×32 edge blocks from the layer, we ensure that every sum node has at least one child.

C.1.5.2 Details of Training the HMM Language Model

Following Zhang et al. [2023b], we first fine-tune a GPT-2 model with the CommonGen dataset. We then sample 8M sequences of length 32 from the fine-tuned GPT-2. After initializing the HMM parameters with latent variable distillation, we fine-tune the HMM with the sampled data. Specifically, following Zhang et al. [2023b], we divide the 8M samples into 40 equally-sized subsets, and run full-batch EM on the 40 subsets repeatedly. Another set of 800K samples is drawn from the fine-tuned GPT as the validation set.

C.1.5.3 Details of Training the Sparse Image Model

Following Liu et al. [2023e], we fine-tune the model with an equivalent batch size of 6400 and a step size of 0.01 in the mini-batch EM algorithm. Specifically, suppose $\boldsymbol{\theta}$ are the current parameters, $\boldsymbol{\theta}^{\text{new}}$ are the new set of parameters computed by the EM update. Given step size α, the update formula is $\boldsymbol{\theta} \leftarrow (1-\alpha)\boldsymbol{\theta} + \alpha\boldsymbol{\theta}^{\text{new}}$.

C.1.5.4 Additional Benchmark Results

Hyperparameters of the adopted HCLTs. We adopt two HCLTs [Liu and Van den Broeck, 2021] with hidden sizes 256 and 512, respectively. The backbone CLT structure is constructed using 20,000 randomly selected training samples.

Hyperparameters of the adopted PDs. Starting from the set of all random variables, the PD structure recursively splits the subset with product nodes. Specifically, consider an image represented as a $H \times W \times C$ (H is the hight; W is the width; C is the number of channels), the PD structure recursively splits over both the height and the width coordinates, where every coordinate has a set of pre-defined split points. For both the height and the width coordinates, we add split points with interval 2. PD-mid has a hidden dimension of 128 and PD-large has 256.

Benchmark results on WikiText-103. Table C.1 illustrates results on WikiText-103. We train the model on sequences with 64 tokens. We adopt two (homogeneous) HMM models, HMM-mid and HMM-large with hidden sizes 2048 and 4096, respectively.

Table C.1: Density estimation performance of PCs on the WikiText-103 dataset. Reported numbers are test set perplexity.

Dataset	HMM-mid	HMM-large
WikiText-103	146.59	167.65

C.1.6 Additional Experiments

C.1.6.1 Speed of the Compilation Process

In Table C.2, we show the compilation speed of PCs with different structures and different sizes. Experiments are conducted on a server with an AMD EPYC 7763 64-Core Processor and 8 RTX 4090 GPUs (we only use one GPU). The results demonstrate the efficiency of the compilation process, where even the PD model with close to 1B parameters can be compiled in around 30 seconds.

Table C.2: Average ($\pm$ standard deviation of 3 runs) runtime (in seconds) of the compilation process of four PCs.

Structure	HMM	PD	HCLT	RAT-SPN
# nodes	130K	1.38M	710K	465K
# edges	130M	829M	159M	33.4M
Compilation time (s)	$1.50_{\pm 0.02}$	$30.57_{\pm 0.86}$	$8.70_{\pm 0.32}$	$4.72_{\pm 0.16}$

C.1.6.2 Runtime on Different GPUs

In addition to the RTX 4090 GPU adopted in the experiments in Table 4.1, we compare the runtime of PyJuice with the baselines on an NVIDIA A40 GPU. As shown in the following table, PyJuice is still significantly faster than all baselines for PCs of different sizes.

Table C.3: **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch** of 60K samples for PyJuice and the baselines on five RAT-SPNs [Peharz et al., 2020b] with different sizes. All other settings are the same as described in Sec. 4.2.5.1.

# nodes	58K	116K	232K	465K	930K
# edges	616K	2.2M	8.6M	33.4M	132M
EiNet	$60.29_{\pm 0.30}$	$136.85_{\pm 0.13}$	$282.58_{\pm 0.27}$	$690.73_{\pm 0.08}$	$1936.28_{\pm 0.26}$
Juice.jl	$4.41_{\pm 0.21}$	$11.57_{\pm 0.07}$	$32.74_{\pm 1.86}$	$121.25_{\pm 0.43}$	$331.98_{\pm 2.87}$
PyJuice	$\mathbf{1.53}_{\pm 0.07}$	$\mathbf{3.11}_{\pm 0.07}$	$\mathbf{6.47}_{\pm 0.08}$	$\mathbf{13.62}_{\pm 0.37}$	$\mathbf{30.69}_{\pm 0.19}$

C.1.6.3 Runtime on Different Batch Sizes

As a supplement to Table 4.1, we report the runtime for a RAT-SPN [Peharz et al., 2020b] with $465K$ nodes and $33.4M$ edges using batch sizes $\{8, 16, 32, 64, 128, 256, 512\}$. To minimize distractions, we only record the time to compute the forward and backward process, but not the time used for EM updates. Results are shown in the table below.

Table C.4: **Average ($\pm$ standard deviation of 5 runs) runtime (in seconds) per training epoch (excluding EM updates)** of 60K samples for PyJuice and the baselines on a RAT-SPNs [Peharz et al., 2020b] with $465K$ nodes and $33.4M$ edges. All other settings are the same as described in Sec. 4.2.5.1. OOM denotes out-of-memory.

Batch size	8	16	32	64	128	256	512
EiNet	$332.87_{\pm 0.21}$	OOM	OOM	OOM	OOM	OOM	OOM
Juice.jl	$1045.04_{\pm 0.06}$	$853.15_{\pm 0.03}$	$775.87_{\pm 0.02}$	$642.54_{\pm 0.04}$	$324.23_{\pm 0.02}$	$163.68_{\pm 0.02}$	$80.57_{\pm 0.01}$
PyJuice	$\mathbf{43.09}_{\pm 0.04}$	$\mathbf{18.63}_{\pm 0.02}$	$\mathbf{7.38}_{\pm 0.01}$	$\mathbf{4.58}_{\pm 0.01}$	$\mathbf{3.50}_{\pm 0.01}$	$\mathbf{3.04}_{\pm 0.01}$	$\mathbf{2.76}_{\pm 0.03}$

Appendix D

Applications

D.1 A Unified Approach to Count-Based Weakly-Supervised Learning

D.1.1 Proofs

Lemma 5. *Let R_{llp} be our risk estimator defined over $p(\boldsymbol{x}, \tilde{y})$ as $R_{llp}(f) = \frac{1}{k(k+1)}\mathbf{E}_{p(\boldsymbol{x}^k,\tilde{y})}[\ell(f(\boldsymbol{x}), \boldsymbol{y})]$. Following the assumptions in Section 3.1 from Kobayashi et al. [2022], our proposed method is risk-consistent.*

Proof. In Kobayashi et al. [2022], it is shown that the risk R in classical multi-class classification can be reduced to a risk R_{rc} over $p(\boldsymbol{x}^k, \tilde{y}^k)$ as shown in Equation 1 in Kobayashi et al. [2022] under certain assumptions.

Consider binary classification and follow our notations, we rewrite the Equation 1 in Kobayashi et al. [2022] as below,

$$R_{rc}(f) = \frac{1}{k(k+1)}\mathbf{E}_{p(\boldsymbol{x}^k,\tilde{y})} \sum_{\boldsymbol{y}\in\mathcal{Y}^k} \frac{\prod_{j=1}^{k} p(\boldsymbol{y}_j \mid \boldsymbol{x}_j)}{\sum_{\boldsymbol{y}'\in\mathcal{Y}^k, \sum_j \boldsymbol{y}'_j=\tilde{y}} \prod_{j=1}^{k} p(\boldsymbol{y}'_j \mid \boldsymbol{x}_j)} \ell(f(\boldsymbol{x}^k), \boldsymbol{y})$$

We notice that the weight term attached to the loss can be further rewritten as a constrained proba-

bility as follows,

$$\frac{\prod_{j=1}^{k} p(\boldsymbol{y}_j \mid \boldsymbol{x}_j)}{\sum\limits_{\boldsymbol{y}' \in \mathcal{Y}^k, \sum_j \boldsymbol{y}'_j = \tilde{y}} \prod_{j=1}^{k} p(\boldsymbol{y}'_j \mid \boldsymbol{x}_j)} = p(\boldsymbol{y} \mid \sum_{j=1}^{k} \boldsymbol{y}_j = \tilde{y}, \boldsymbol{x}^k)$$

This allows us to further rewrite the risk R_{rc} with likelihood loss being $\ell(f(\boldsymbol{x}^k), \boldsymbol{y}) = -p(\sum_{j=1}^{k} \boldsymbol{y}_j = k\tilde{y} \mid \boldsymbol{x}^k)$:

$$\begin{aligned}
R_{rc}(f) &= \frac{1}{k(k+1)} \mathbf{E}_{p(\boldsymbol{x}^k, \tilde{y})} \\
&\left[- \sum_{\boldsymbol{y} \in \mathcal{Y}^k} p(\boldsymbol{y} \mid \sum_{j=1}^{k} \boldsymbol{y}_j = k\tilde{y}, \boldsymbol{x}^k) p(\sum_{j=1}^{k} \boldsymbol{y}_j = k\tilde{y} \mid \boldsymbol{x}^k) \right] \\
&= \frac{1}{k(k+1)} \mathbf{E}_{p(\boldsymbol{x}^k, \tilde{y})} \left[- \sum_{\boldsymbol{y} \in \mathcal{Y}^k} p(\boldsymbol{y}, \sum_{j=1}^{k} \boldsymbol{y}_j = k\tilde{y} \mid \boldsymbol{x}^k) \right] \\
&= \frac{1}{k(k+1)} \mathbf{E}_{p(\boldsymbol{x}^k, \tilde{y})} \left[-p(\sum_{j=1}^{k} \boldsymbol{y}_j = k\tilde{y} \mid \boldsymbol{x}^k) \right] \\
&= \frac{1}{k(k+1)} \mathbf{E}_{p(\boldsymbol{x}^k, \tilde{y})} [\ell(f(\boldsymbol{x}^k), \boldsymbol{y})] = R_{llp}(f)
\end{aligned}$$

The last few lines follow from the definition of conditional probabilities. This shows that the risk $R_{rc}(f) = R_{llp}(f)$, meaning that the reduction from risk $R_{rc}(f)$ to the classical risk $R(f)$ in Kobayashi et al. [2022] is applicable to our risk estimator R_{llp}, which proves that our learning method is risk-consistent. □

Proposition D.1. Assume that the loss function $\ell(f(\boldsymbol{x}), \boldsymbol{y})$ is ρ-Lipschitz with respect to $f(\boldsymbol{x})$ for any $\boldsymbol{y} \in \mathcal{Y}$ bounded by some constant. Let f_{llp} be the hypothesis that minimizes the empirical risk, and f^*_{llp} is the hypothesis that minimizes the true risk, then f_{llp} converges to f^*_{llp} as $m \to \infty$.

Proof. This claim immediately follows Lemma 5, where we shows that $R_{rc}(f) = R_{llp}(f)$. Therefore, it holds that $R_{llp}(\hat{f}) - R_{llp}(f^*) = R_{(sc)}(\hat{f}) - R_{(sc)}(f^*)$, where the latter term, an always positive term, is shown in Theorem 3.1 in Kobayashi et al. [2022] that it converges to 0 at rate

$\sqrt{m}$. □

Proposition 5.2 *The count probability $p(\sum_{i=1}^{k} \hat{y}_i = s)$ of sampling k prediction variables with summation being s from an unconstrained distribution $p(\boldsymbol{y}) = \prod_{i=1}^{k} p(\hat{y}_i)$ can be computed exactly in time $\mathcal{O}(ks)$. Moreover, the set $\{p(\sum_{i=1}^{k} \hat{y}_i = s)\}_{s=0}^{k}$ can also be computed in time $\mathcal{O}(k^2)$.*

Proof. The claim that $p(\sum_{i=1}^{k} \hat{y}_i = s)$ can be computed exactly in time $\mathcal{O}(ks)$ follows immediately from Proposition 1 in Ahmed et al. [2023c]: in Ahmed et al. [2023c], the unconstrained distribution is a factorized distribution obtained from k outputs from a single neural network model while in our case, the unconstrained distribution $p(\boldsymbol{y})$ is obtained from applying a classifier that gives a single output $p(\boldsymbol{y}_i)$ on k inputs; the constructive proof of Proposition 1 in Ahmed et al. [2023c] still applies in our case. Moreover, the computation of $p(\sum_{i=1}^{k} \hat{y}_i = k)$ is done in a dynamic programming manner in the sense that for any $s < k$, $p(\sum_{i=1}^{k} \hat{y}_i = s)$ is an intermediate result for computing $p(\sum_{i=1}^{k} \hat{y}_i = k)$. By caching the intermediate result, the set $\{p(\sum_{i=1}^{k} \hat{y}_i = s)\}_{s=0}^{k}$ can be obtained by the time $p(\sum_{i=1}^{k} \hat{y}_i = k)$ is computed, which finishes our proof. □

D.1.2 Instance MIL Experimental Results

In this section, we provide results for instance level feedback in the MIL setting. The baselines that we used in our experiments, Gated-Attention and Attention are both examples of embedding based approaches and do not make instance-level predictions. We compare against one baseline approach, which is based on Instance-Max from Ilse et al. [2018]. This uses the maximum instance probability as an approximation for the "positiveness" of a bag. We then train it with a binary cross entropy. Note that max pooling is stated in the literature as the best performing option and makes the most sense in the MIL setting [Ilse et al., 2018; Wang et al., 2018].

Our results show that for bags of size less than or equal to 150, our method greatly improves upon the baseline and is better for bag sizes greater than or equal to 200. We notice that across both methods, performance goes down as bag size increases; we expect this because we have less supervision on positive bags (at least 1 label is less meaningful for bigger bags). However,

Table D.1: MIL experiment on MNIST dataset on instance-level classification. Each block represents a different distribution from which we draw bag sizes—First Block: $\mathcal{N}(10, 2)$, Second Block: $\mathcal{N}(50, 10)$, Third Block: $\mathcal{N}(100, 20)$. We run each experiment for 3 runs and report mean test accuracy with standard error. We bold the highest value and both if the standard-errors overlap.

Training Bags	50	100	150	200	300	400	500
Instance-Max	0.8714 ± 0.0015	0.9577 ± 0.0096	0.9494 ± 0.0232	$\mathbf{0.9845 \pm 0.0009}$	0.9885 ± 0.0004	$\mathbf{0.9903 \pm 0.0008}$	0.9908 ± 0.0004
CL (Ours)	$\mathbf{0.9551 \pm 0.0055}$	$\mathbf{0.9780 \pm 0.0015}$	$\mathbf{0.9826 \pm 0.0014}$	$\mathbf{0.9864 \pm 0.0005}$	$\mathbf{0.9906 \pm 0.0001}$	$\mathbf{0.9905 \pm 0.0007}$	$\mathbf{0.9916 \pm 0.0003}$
Instance-Max	0.9398 ± 0.0010	0.9415 ± 0.0008	0.9513 ± 0.0113	0.9686 ± 0.0123	$\mathbf{0.9849 \pm 0.0010}$	0.9848 ± 0.0008	$\mathbf{0.9867 \pm 0.0008}$
CL (Ours)	$\mathbf{0.9732 \pm 0.0009}$	$\mathbf{0.9776 \pm 0.0009}$	$\mathbf{0.9799 \pm 0.0010}$	$\mathbf{0.9816 \pm 0.0005}$	$\mathbf{0.9839 \pm 0.0013}$	$\mathbf{0.9864 \pm 0.0006}$	$\mathbf{0.9865 \pm 0.0014}$
Instance-Max	0.9446 ± 0.0007	0.9462 ± 0.0005	0.9583 ± 0.0076	0.9700 ± 0.0035	0.9750 ± 0.0017	0.9776 ± 0.0008	0.9695 ± 0.0097
CL (Ours)	$\mathbf{0.9695 \pm 0.0010}$	$\mathbf{0.9717 \pm 0.0011}$	$\mathbf{0.9759 \pm 0.0013}$	$\mathbf{0.9764 \pm 0.0006}$	$\mathbf{0.9780 \pm 0.0001}$	$\mathbf{0.9805 \pm 0.0008}$	$\mathbf{0.9798 \pm 0.0003}$

Table D.2: MIL experiment on MNIST dataset on instance-level classification. Each block represents a different distribution from which we draw bag sizes—First Block: $\mathcal{N}(10, 2)$, Second Block: $\mathcal{N}(50, 10)$, Third Block: $\mathcal{N}(100, 20)$. We run each experiment for 3 runs and report mean test AUC with standard error. We bold the highest value and both if the standard-errors overlap.

Training Bags	50	100	150	200	300	400	500
Instance-Max	0.4904 ± 0.0054	0.8171 ± 0.0465	0.7740 ± 0.1072	0.9288 ± 0.0064	0.9460 ± 0.0022	$\mathbf{0.9562 \pm 0.0037}$	0.9603 ± 0.0016
CL (Ours)	$\mathbf{0.8341 \pm 0.0135}$	$\mathbf{0.9040 \pm 0.0146}$	$\mathbf{0.9291 \pm 0.0070}$	$\mathbf{0.9394 \pm 0.0005}$	$\mathbf{0.9571 \pm 0.0021}$	$\mathbf{0.9592 \pm 0.0029}$	$\mathbf{0.9647 \pm 0.0012}$
Instance-Max	0.4956 ± 0.0007	0.4965 ± 0.0003	0.5960 ± 0.0821	$\mathbf{0.7297 \pm 0.0959}$	$\mathbf{0.8566 \pm 0.0088}$	0.8554 ± 0.0080	$\mathbf{0.8733 \pm 0.0048}$
CL (Ours)	$\mathbf{0.7518 \pm 0.0090}$	$\mathbf{0.7900 \pm 0.0081}$	$\mathbf{0.8125 \pm 0.0106}$	$\mathbf{0.8261 \pm 0.0064}$	$\mathbf{0.8473 \pm 0.0064}$	$\mathbf{0.8717 \pm 0.0063}$	$\mathbf{0.8709 \pm 0.0120}$
Instance-Max	0.4974 ± 0.0002	0.5007 ± 0.0016	0.6170 ± 0.0571	0.7099 ± 0.0311	0.7546 ± 0.0164	0.7792 ± 0.0080	0.7102 ± 0.0867
CL (Ours)	$\mathbf{0.7008 \pm 0.0077}$	$\mathbf{0.7214 \pm 0.0102}$	$\mathbf{0.7617 \pm 0.0130}$	$\mathbf{0.7673 \pm 0.0059}$	$\mathbf{0.7832 \pm 0.0011}$	$\mathbf{0.8085 \pm 0.0084}$	$\mathbf{0.8007 \pm 0.0032}$

our approach is able to recover this gap compared to the baseline methodology. In the case of less overall training bags, less than 150 training bags, we find that Instance-max really suffers on AUC while our objective guides the model to learning something more meaningfulshowcasing the robustness of our methodology.

D.1.3 Experimental Details

In this section, we will provide relevant training details as it relates to each of our settings including hyperparameters and dataset details.

Table D.3: Illustration of Adult and Magic datasets showing the number of training bags for each bag size. Note that we test on the same number of instances in all variations of bag size for both experiments: 16280 for Adult and 3804 for Magic. The breakdown of training bags is the same across all distributions of label proportion as well, i.e., $[0, \frac{1}{2}]$, $[\frac{1}{2}, 1]$, $[0, 1]$.

Bag Size	Training Bags Adult	Training Bags Magic
8	1024	768
32	256	192
128	64	48
512	16	12

D.1.3.1 Label Proportion

Adult Dataset

Hyperparameters. We use a learning rate of 0.00001 with the Adam Optimizer and $\beta_1 = 0.9, \beta_2 = 0.999$. The weight decay value is set to 0.001. We also notice that adding in $L1$ regularization of 0.001 improved the performance of our method. We train for 10000 epochs and use a set number of warm epochs for our experiments. All parameters were obtained by using a holdout of 12.5% of training data for validation on the $[0, 1]$ uniform setting. The network shown in Table D.4 was also obtained grid search on this same validation set.

Table D.4: Network used for Adult dataset in LLP Experiments.

Layer	Type
1	fc - 2048 + ReLU
2	fc - 64 + ReLU
3	fc - 1 + logsigmoid

Training Procedure. For CL, we use the parameters and network described in the previous paragraph and early stopping criterion based on validation loss from a held out validation set (12.5% of training data). For PL, we use the parameters and network except that we do not use $L1$ as we found this improves performance. We also use an early stopping criterion based on validation loss from a held out validation set (12.5% of training data).

Computing Resources. Trained on Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHzU and AMD EPYC 7313P 16-Core Processor CPU.

Magic Dataset

Hyperparameters. We use a learning rate of 0.0001 with the Adam Optimizer and $\beta_1 = 0.9, \beta_2 = 0.999$. The weight decay value is set to 0.001. We also notice that adding in $L1$ regularization of 0.001 improved the performance of our method. We train for 10000 epochs and use a set number of warm epochs for our experiments. All parameters were obtained by using a holdout of 12.5% of training data for validation on the $[0, 1]$ uniform setting. The network shown in Table D.5 was also obtained grid search on this same validation set.

Table D.5: Network used for Magic dataset in LLP Experiments.

Layer	Type
1	fc - 2048 + ReLU
2	fc - 1 + logsigmoid

Training Procedure. For CL, we use the parameters and network described in the previous paragraph and early stopping criterion based on validation loss from a held out validation set (12.5% of training data). For PL, we use the parameters and network except that we do not use $L1$ regularization as we found this improves performance. We also use an early stopping criterion based on validation loss from a held out validation set (12.5% of training data). In Table 5.3, there are two instances where we reran our method with no validation set, i.e. Magic $[0, \frac{1}{2}]$ and Magic $[\frac{1}{2}, 1]$ because early stopping proved to be unstable with a small amount of validation samples. In these experiments, we only use 87.5% of training data and ran for a fixed number of epochs: 2000. This is because with only one validation bag, we can find ourselves with some instability in the training procedure. Note that PL did not benefit from rerunning with this method.

Computing Resources. Trained on Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHzU and AMD EPYC 7313P 16-Core Processor CPU.

D.1.3.2 Multi-Instance Learning

MNIST-Bags

Dataset Details. We experiment on various modulations of training bag size and number of training bags. In the main experiment, we draw bag size from: $\{\mathcal{N}(10, 2), \mathcal{N}(50, 10), \mathcal{N}(100, 20)\}$ and modulate number of training bags from $\{50, 100, 150, 200, 300, 400, 500\}$. In total, this makes 21 different settings. In our follow up experiment where we limit the number of training bags and overall bag size, we draw bag size from: $\{\mathcal{N}(5, 1), \mathcal{N}(10, 2)\}$. For each experiment, we sample 1000 test bags with size coorelating to the normal distribution associated.

Hyperparameters. All of our hyperparameters derive from Ilse et al. [2018]. This includes using the Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.999$, a learning rate of 0.0005, weight decay of 0.0001, and max epochs of 200. For the main experiment, we use a validation holdout of 20% to find a class weight for balancing the loss on positive bags versus negative bags. (We omit this step for our limited data experiments.)

Table D.6: Network used for all MNIST experiments in MIL settings. Derived from the same network shown in Ilse et al. [2018].

Layer	Type
1	conv(5, 1, 0) - 20 + ReLU
2	maxpool(2, 2)
3	conv(5, 1, 0) - 50 + ReLU
4	maxpool(2, 2)
5	fc-500 + ReLU
6	fc-1 + logsigmoid

Training Procedure. For CL, we train on all the training data for the maximum number of iterations: 200. We also use all of the hyperparameters described in the last paragraph and Ilse et al. [2018]. Because we were unable to reproduce the values in Ilse et al. [2018] for the Attention and Gated Attention mechanisms, we reran their experiments with our own implementation. To try and reproduce their results, we follow their optimization procedure. Specifically, we use a holdout of training data (20%) and validation loss + error for early stopping. We found that doing so provided the best values for Attention and Gated Attention.

Instance Pooling. To pool together instance level classification at the final stage, there are several operations that have been considered in the literature. Some include using the max and mean operator [Wang et al., 2018]. We propose a new method based on our constraint. We compute the relevant probabilities defined in 5.1.2 for the MIL setting. More specifically, we compute the probability that a bag has at least one positive instance. We then round the probability of at least one positive instance to obtain our bag level classification.

Computing Resources. Trained on AMD EPYC 7313P 16-Core Processor CPU.

Colon Cancer Dataset

Dataset Details. The dataset consists of 100 H&E images of which we use 99 of them. There are a total of 51 positive bags and 48 negative bags. We use a series of data augmentations including flipping, cropping, and rotation[1]. Note that these data augmentations do not align with those in the original paper by Ilse et al. [2018], so we reran their baseline methods.

Hyperparameters. We derive our set of hyperparameters from Ilse et al. [2018]. We use the Adam optimizer for all experiments with $\beta_1 = 0.9, \beta_2 = 0.999$. This includes weight decay of 0.0005, learning rate of 0.0001, and a maximum of 100 epochs.

[1]Refer to https://github.com/utayao/Atten_Deep_MIL for the preprocessed data generation code

Table D.7: MIL: Network used for CL in colon cancer dataset. Derived from the same network shown in Ilse et al. [2018].

Layer	Type
1	conv(4, 1, 0) - 36 + ReLU
2	maxpool(2, 2)
3	conv(3, 1, 0) - 48 + ReLU
4	maxpool(2, 2)
5	fc-512 + ReLU
6	dropout
7	fc - 512 + ReLU
8	dropout
9	fc-2 + logsigmoid

Training Procedure. We perform 10-fold cross-validation and average the mean value of each metric over 5 seeds. For CL, we do not use early stopping and train on all data for the maximum number of epochs using the hyperparameters mentioned in the previous paragraph. For our baselines, Attention and Gated-Attention, we use the same hyperparameters as mentioned above. However, we follow the optimization procedure detailed in Ilse et al. [2018] to give try and reproduce the results given in the paper. This involves using a held out validation set for early stopping with validation loss + error as the stopping criteria. For this experiment, this validation set is assumed to be the size of 1 fold or one-ninth of the training data. (We find that including early stopping helps increase performance for both baselines.)

Computing Resources. Trained on NVIDIA RTX A6000 GPU.

D.1.3.3 PU Learning

MNIST Dataset

Dataset Details. Our settings derive from Garg et al. [2021]. We construct two main datasets from the original MNIST dataset. This includes the Binarized MNIST and MNIST-17 as detailed in Table D.9. In the Binarized MNIST setting, we assign digits $[0-4]$ as positive and $[5-9]$ as negative. In the MNIST-17 setting, we assign digit 1 as positive and 7 as negative. The test set for both settings are chosen from a set of unlabeled data.

Table D.8: Network used for MNIST data in PU Learning experiments. Resembles the network in Garg et al. [2021] except we replace the last layer with a single output and logsigmoid instead of softmax.

Layer	Type
1	fc - 5000 + ReLU
2	fc - 5000 + ReLU
3	fc - 50 + ReLU
4	fc-1 + logsigmoid

Hyperparameters. We fix weight decay to be 0.0005 and Adam optimizer for all experiments with $\beta_1 = 0.9, \beta_2 = 0.999$. We use a learning rate of 0.0001 and train for a maximum of 2000 epochs in all experiments for both CL and CL-expect. We use a validation set with size equal to 10% of training data in order to weigh the loss on positive data versus loss on unlabeled data.

Training Procedure. For MNIST dataset experiments, we use a fully connected multi-layer perceptron (MLP) defined in Table D.8. We train CL and CL-expect with the hyperparameters defined in the previous paragraph. Furthermore, we use a held out validation set, equivalent to 10% of training data, for early stopping. While as results in Garg et al. [2021] are aggregated over 10 epochs, we choose to pick a single epoch based on our early stopping as this makes the most sense for our optimization technique.

Table D.9: Table taken almost directly from Garg et al. [2021]. Table shows the break down of the various simulated PU datasets that we train on.

Dataset	Simulated PU Dataset	P vs N	Training		Test
			Positive	Unlabeled	Unlabeled
CIFAR	Binarized CIFAR	$[0-4]$ vs. $[5-9]$	12500	12500	2500
	CIFAR Cat vs. Dog	3 vs. 5	3000	3000	500
MNIST	Binarized MNIST	$[0-4]$ vs. $[5-9]$	15000	15000	2500
	MNIST-17	1 vs. 7	3000	3000	500

Computing Resources. Trained on a singular NVIDIA RTX 2080-Ti GPU.

CIFAR Dataset.

Dataset Details. Our settings derive from Garg et al. [2021]. We construct two main datasets from the original CIFAR dataset. This includes the Binarized CIFAR and CIFAR Cat vs. Dog as detailed in Table D.9. In the Binarized CIFAR setting, we assign classes $[0-4]$ as positive and classes $[5-9]$ as negative. In the CIFAR Cat vs. Dog setting, we assign Cats (class 3) as positive and Dogs (class 5) as negative. The test set for both settings are chosen from a set of unlabeled data.

Hyperparameters. We fix weight decay to be 0.0005 and Adam optimizer for all experiments with $\beta_1 = 0.9, \beta_2 = 0.999$. We use a learning rate of 0.0001 for all experiments except for CL-expect in the CIFAR Cat vs. Dog setting where we use 0.001. We use a validation set with size equal to 10% of training data in order to weigh the loss on positive data versus loss on unlabeled data.

Training Procedure. We use a ResNet-18 architecture for all CIFAR experiments. We train CL and CL-expect with the hyperparameters defined in the previous paragraph. Furthermore, we use a held out validation set, equivalent to 10% of training data, for early stopping. While as results in

Garg et al. [2021] are aggregated over 10 epochs, we choose to pick a single epoch as this makes the most sense for our optimization technique.

Computing Resources. Trained on a singular NVIDIA 2080-Ti GPU.

Early Stopping

The early stopping procedure that we used in our experiments was a bit unique. Using our holdout of validation data, we do early stopping using the proximity to the class prior and validation loss to break ties. We can imagine that if we perfectly identify all positive and unlabeled samples and then calculate accuracy against the actually provided labels, we would get an accuracy equivalent to the class prior. This is because all the positive samples in the unlabeled set would be labeled incorrect.

D.2 Probabilistically Rewired Message-Passing Neural Networks

D.2.1 Additional related work

In the following, we discuss additional related work.

Graph structure learning The field of graph structure learning (GSL) is a topic related to graph rewiring. Motivated by robustness and more general purposes, several GSL works have been proposed. Jin et al. [2020] optimizes a graph structure from scratch with some loss function as bias. More generally, an edge scorer function is learned, and modifications are made to the original graph structure [Chen et al., 2020; Yu et al., 2021; Zhao et al., 2021]. To introduce discreteness and sparsity, Kazi et al. [2022]; Franceschi et al. [2019]; Zhao et al. [2021] leverage Gumbel and Bernoulli discrete sampling, respectively. Saha et al. [2023] incorporates end-to-end differentiable discrete sampling through the smoothed-Heaviside function. Moreover, GSL also benefits from self-supervised or unsupervised learning approaches; see, e.g., Zou et al. [2023]; Fatemi et al.

[2021]; Liu et al. [2022b,c]. For a comprehensive survey of GSL. see Fatemi et al. [2023]; Zhou et al. [2023]. In the context of node classification, there has been recent progress in understanding the interplay between graph structure and features [Castellana and Errica, 2023].

The main differences to the proposed PR-MPNN framework are as follows: (a) for sparsification, existing GSL approaches typically use a k-NN algorithm, a simple randomized version of k-NN, or model edges with independent Bernoulli random variables. In contrast, PR-MPNN uses a proper probability mass function derived from exactly-k constraints. Hence, we introduce complex dependencies between the edge random variables and trade-off exploration and exploitation during training, and (b) GSL approaches do not use exact sampling of the exactly-k distribution and recent sophisticated gradient estimation techniques. However, the theoretical insights we provide in this paper also largely translate to GSL approaches with the difference that sampling is replaced with an $\arg\max$ operation and, therefore, should be of independent interest to the GSL community.

D.2.2 Extended notation

A graph G is a pair $(V(G), E(G))$ with *finite* sets of vertices or nodes $V(G)$ and edges $E(G) \subseteq \{\{u, v\} \subseteq V(G) \mid u \neq v\}$. If not otherwise stated, we set $n \coloneqq |V(G)|$, and the graph is of order n. We also call the graph G an n-order graph. For ease of notation, we denote the edge $\{u, v\}$ in $E(G)$ by (u, v) or (v, u). A (vertex-)labeled graph G is a triple $(V(G), E(G), \ell)$ with a (vertex-)label function $\ell \colon V(G) \to \mathbb{N}$. Then $\ell(v)$ is a label of v, for v in $V(G)$. An attributed graph G is a triple $(V(G), E(G), a)$ with a graph $(V(G), E(G))$ and (vertex-)attribute function $a \colon V(G) \to {R_2}^{1\times d}$, for some $d > 0$. That is, contrary to labeled graphs, we allow for vertex annotations from an uncountable set. Then $a(v)$ is an attribute or feature of v, for v in $V(G)$. Equivalently, we define an n-order attributed graph $G \coloneqq (V(G), E(G), a)$ as a pair $\mathbf{G} = (G, \mathbf{L})$, where $G = (V(G), E(G))$ and $\mathbf{L}$ in ${R_2}^{n\times d}$ is a node attribute matrix. Here, we identify $V(G)$ with $[n]$. For a matrix $\mathbf{L}$ in ${R_2}^{n\times d}$ and v in $[n]$, we denote by $\mathbf{L}_{v\cdot}$ in ${R_2}^{1\times d}$ the vth row of $\mathbf{L}$ such that $\mathbf{L}_{v\cdot} \coloneqq a(v)$. Furthermore, we can encode an n-order graph G via an adjacency matrix $\vec{A}(G) \in \{0, 1\}^{n\times n}$, where $A_{ij} = 1$ if, and only, if $(i, j) \in E(G)$. We also write ${R_2}^{d}$ for ${R_2}^{1\times d}$.

The neighborhood of v in $V(G)$ is denoted by $N(v) \coloneqq \{u \in V(G) \mid (v,u) \in E(G)\}$ and the degree of a vertex v is $|N(v)|$. Two graphs G and H are isomorphic and we write $G \simeq H$ if there exists a bijection $\varphi\colon V(G) \to V(H)$ preserving the adjacency relation, i.e., (u,v) is in $E(G)$ if and only if $(\varphi(u),\varphi(v))$ is in $E(H)$. Then φ is an isomorphism between G and H. In the case of labeled graphs, we additionally require that $l(v) = l(\varphi(v))$ for v in $V(G)$, and similarly for attributed graphs. Further, we call the equivalence classes induced by $\simeq$ isomorphism types.

A node coloring is a function $c\colon V(G) \to {R_2}^d$, $d > 0$, and we say that $c(v)$ is the color of $v \in V(G)$. A node coloring induces an edge coloring $e_c\colon E(G) \to \mathbb{N}$, where $(u,v) \mapsto \{c(u), c(v)\}$ for $(u,v) \in E(G)$. A node coloring (edge coloring) c refines a node coloring (edge coloring) d, written $c \sqsubseteq d$ if $c(v) = c(w)$ implies $d(v) = d(w)$ for every $v, w \in V(G)$ ($v, w \in E(G)$). Two colorings are equivalent if $c \sqsubseteq d$ and $d \sqsubseteq c$, in which case we write $c \equiv d$. A color class $Q \subseteq V(G)$ of a node coloring c is a maximal set of nodes with $c(v) = c(w)$ for every $v, w \in Q$. A node coloring is called discrete if all color classes have cardinality 1.

D.2.3 Missing proofs

In the following, we outline missing proofs from the main paper.

Theorem 6 (Theorem 3 in the main paper)**.** *For sufficiently large n, for every $\varepsilon \in (0,1)$ and $k > 0$, we have that for almost all pairs, in the sense of Babai et al. [1980], of isomorphic n-order graphs G and H and all permutation-invariant, 1-WL-equivalent functions $f\colon \mathfrak{A}_n \to {R_2}^d$, $d > 0$, there exists a conditional probability mass function $p_{(\boldsymbol{\theta},k)}$ that separates the graph G and H with probability at most ε regarding f.*

Before proving the above result, we first need three auxiliary results. The first one is the well-known universal approximation theorem for multi-layer perceptrons.

Theorem 7 (Cybenko [1992]; Leshno et al. [1993])**.** *Let $\sigma\colon R_2 \to R_2$ be continuous and not polynomial. Then for every continuous function $f\colon K \to {R_2}^n$, where $K \subseteq {R_2}^m$ is a compact set, and every $\varepsilon > 0$ there is a depth-two multi-layer perceptron N with activation function $\sigma^{(1)} = \sigma$*

on layer 1 *and no activation function on layer 2 (i.e.,* $\sigma^{(2)}$ *is the identity function) computing a function* f_N *such that*

$$\sup_{\vec{x} \in K} \|f(\vec{x}) - f_N(\vec{x})\| < \varepsilon.$$

Building on the first, the second result shows that an MPNN can approximate real-valued node colorings of a given finite graph arbitrarily close.

Lemma 8. *Let* G *be an* n*-order graph and let* $c \colon V(G) \to \mathbb{R}^d$*,* $d > 0$*, be a* 1*-*WL*-equivalent node coloring. Then, for all* $\varepsilon > 0$*, there exists a (permutation-equivariant) MPNN* $f \colon V(G) \to {R_2}^d$*, such that*

$$\max_{v \in V(G)} \|f(v) - c(v)\| < \varepsilon.$$

Proof sketch. First, by [Morris et al., 2019, Theorem 2], there exists an 1-WL-equivalent MPNN $m \colon V(G) \to {R_2}^d$ such that

$$c \equiv m.$$

Since the graph's number of vertices, by assumption, is finite, the cardinality of the image $K \coloneqq m^{-1}$ is also finite. Hence, we can find a continuous function $g \colon K \to \mathbb{R}^d$ such that $(g \circ m)(v) = c(v)$ for $v \in V(G)$. Since K is finite and hence compact and g is continuous, by Theorem 7, we can approximate it arbitrarily close with a two-layer multi-layer perceptron, implying the existence of the MPNN f. □

The third result lifts the previous result to edge colorings.

Lemma 9. *Let* G *be an* n*-order graph and let* $c \colon E(G) \to \mathbb{R}^d$*,* $d > 0$*, be a* 1*-*WL*-equivalent edge coloring. Then, for all* $\varepsilon > 0$*, there exists a (permutation-equivariant) MPNN* $f \colon E(G) \to {R_2}^d$*,*

such that

$$\max_{e \in E(G)} \|f(e) - c(e)\| < \varepsilon.$$

Proof sketch. The proof is analogous to the proof of Lemma 8. □

We note here that we can extend the above results to any finite subset of n-order graphs. We are now ready to prove Theorem 6.

Proof sketch. Following Babai et al. [1980], for a sufficiently large order n, the 1-WL will compute a discrete coloring for almost any n-order graph. Concretely, they showed that an algorithm equivalent to the 1-WL computes a discrete coloring of graphs sampled from the ErdsRényi random graph model $G(n, 1/2)$ with the probability of failure bounded by $\mathcal{O}(n^{-1/7})$. Since the $G(n, 1/2)$ model assigns a uniform distribution over all graphs, the 1-WL succeeds on "almost all" graphs.

By the above, and due to Lemmas 8 and 9, every node, and thereby any edge, can be assigned a distinct arbitrary prior weight with an upstream MPNN. Consequently, there exists an upstream MPNN that returns a high prior $\boldsymbol{\theta}_{ij}$ for exactly k edges ($\boldsymbol{\theta}_i$ for exactly k nodes) such that sampling from the exactly-k distribution returns these k edges (nodes) with probability at least $\sqrt{1-\varepsilon}$. Specifically, we know that the upstream MPNN can return arbitrary priors $\boldsymbol{\theta}_{ij}$, and we want to show that, given some $\delta \in (0, 1)$, there exists at least one $\boldsymbol{\theta}$ such that for a set S of k edges (nodes) of G, we have $p_{\boldsymbol{\theta},k}(S) \geq \delta$.

Let m be the number of edges in the graph G That is, from our probability definition, we obtain $p_{\boldsymbol{\theta}}(S) \geq \delta Z$. Without losing generality, let w_1 and w_2 be two prior weights with $w_1 > w_2$ such that $\theta_i = w_1$ for the edges (nodes) in S and $\theta_i = w_2$ otherwise. Then $p_{\boldsymbol{\theta}}(S) \geq \delta Z$ becomes $w_1^k \geq \delta(\sum_{i=0}^{k} \binom{k}{i}\binom{m-k}{k-i} w_1^i w_2^{k-i})$. We use the upper bound $Z \leq w_1^k + (\binom{m}{k} - 1) w_2 w_1^{k-1}$ and obtain $w_2 \leq (1-\delta) w_1 \delta^{-1} (\binom{m}{k} - 1)^{-1}$. Therefore, a prior $\boldsymbol{\theta}$ exists, and we can obtain it by using the derived inequality. Now, we can set $\delta = \sqrt{1-\varepsilon}$. The sampled k edges (nodes) are then identical for both graphs with probability at least $\sqrt{1-\varepsilon}^2 = 1 - \varepsilon$ and, therefore, the edges (nodes) that

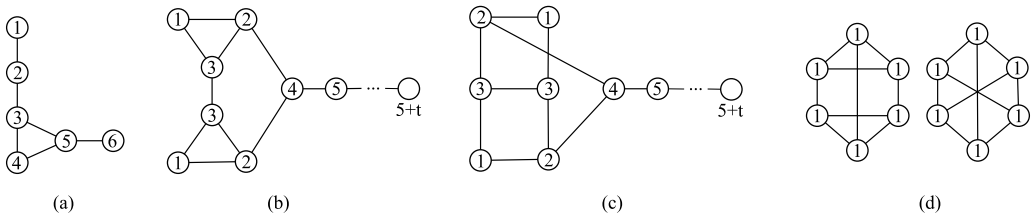


Figure D.1: Example graphs used in the theoretical analysis.

are removed are isomorphic edges (nodes) with probability at least $1 - \varepsilon$. When we remove pairs of edges (nodes) from two isomorphic graphs that are mapped to each other via an isomorphism, the graphs remain isomorphic and, therefore, must have the same 1-WL coloring. Since the two graphs have the same 1-WL coloring, an MPNN downstream model f cannot separate them. □

Proposition D.2 (Proposition 5.3 in the main paper)**.** *Let $\varepsilon \in (0, 1)$, $k > 0$, and let G and H be graphs with identical 1-WL stable colorings. Let V_G and V_H be the subset of nodes of G and H that are in color classes of cardinality 1. Then, for all choices of 1-WL-equivalent functions f, there exists a conditional probability mass function $p_{(\boldsymbol{\theta},k)}$ that separates the graphs $G[V_G]$ and $H[V_H]$ with probability at most ε regarding f.*

Proof sketch. Since the graphs $G[V_G]$ and $H[V_H]$ have a discrete coloring under the 1-WL and the graphs G and H have identical 1-WL colorings, it follows that there exists an isomorphism $\varphi\colon G[V_G] \to H[V_H]$.

Analogous to the proof of Theorem 3, we can now show that there exists a set of prior weights that ensures that the exactly-k sample selects the same subset of edges (nodes) from, respectively, the same subset of edges from $G_1[V_1]$ and $G_2[V_2]$ (the same subset of nodes from V_G and V_H) with probability at least $\sqrt{1-\epsilon}$. Note that the cardinality of the sampled subsets could also be empty since the priors could be putting a higher weight on nodes (edges) with non-discrete color classes. □

Regarding uniform edge removal, consider the two graphs in Figure D.1 (b) and (c). With a distribution based on an MPNN upstream model, the probability of separating the graphs by

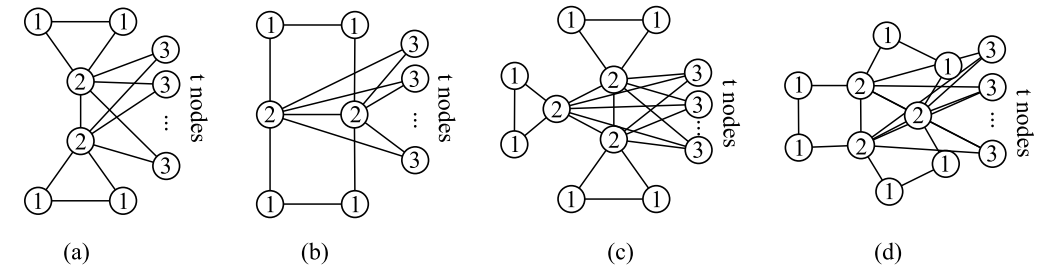


Figure D.2: The graphs used in the proof of Theorem 10 for node sampling.

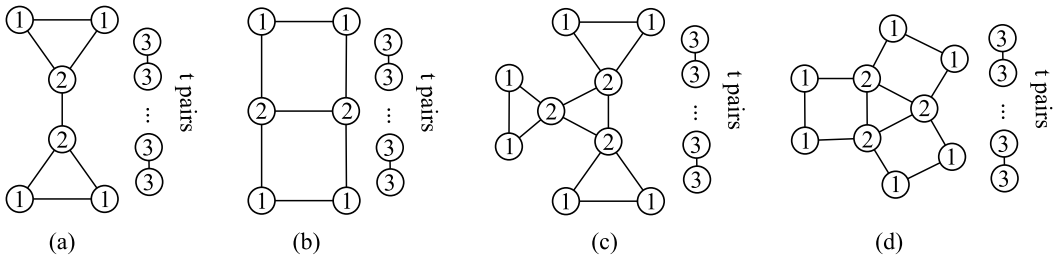


Figure D.3: The graphs used in the proof of Theorem 10 for edge sampling.

removing edges in the isomorphic subgraphs induced by the nodes with colors 4 to $5 + t$ can be made arbitrarily small for any t. However, the graphs would still be separated through samples in the parts whose coloring is non-discrete. In contrast, sampling uniformly at random separates the graphs in these isomorphic subgraphs with probability converging towards 1 with increasing t.

Theorem 10. *For every $\varepsilon \in (0, 1)$ and every $k > 0$, there exists a pair of non-isomorphic graphs G and H with identical and non-discrete 1-WL stable colorings such that for every 1-WL-equivalent function f*

(1) there exists a probability mass function $p_{(k,\boldsymbol{\theta})}$ that separates G and H with probability at least $(1 - \varepsilon)$ with respect to f;

(2) removing edges uniformly at random separates G and H with probability at most ε with respect to f.

Proof. We distinguish the two cases: (1) sampling nodes to be removed and (2) sampling edges to be removed from the original graphs.

For case (1), where we sample nodes to be removed, consider the graphs in Figure D.2. For $k = 1$, we take the graphs (a) and (b). Both of these graphs have the same 1-WL coloring, indicated by the color numbers of the nodes. To separate the graphs, we need to sample and remove one of the nodes with color 1 or 2. Removing a node with color 3 would lead again to an identical color partition for the two graphs. Removing nodes with color 1 or 2 is achievable by placing a high prior weight $\boldsymbol{\theta}_u$ on nodes of *one* of the corresponding color classes; see Lemma 8. Without loss of generality, we choose all nodes u in color class 2 and set the prior weight $\boldsymbol{\theta}_u$ such that a node in this color class is sampled with probability $\sqrt{1-\varepsilon}$. Since a random sampler would uniformly sample any of the nodes, we simply have to increase the number t of nodes with color 3 such that the probability of randomly sampling a node of the color classes 1 or 2 is smaller than or equal to $\sqrt{\epsilon}$.

For $k > 1$, we construct the graphs depicted in Figure D.2 (c) and (d) for $k = 2$. These are constructed by first taking a $(k+1)$-cycle and connecting to each node of the cycle the nodes with color class 1 in the two ways shown in Figure D.2 (c) and (d). Finally, we connect t nodes of color class 3 to each of the nodes in the cycle. These graphs can be separated by sampling k nodes from either the color class 1 or 2. For instance, removing k nodes from color class 2 always creates k disconnected subgraphs of size 2 in the first parameterized graph but not the second. By Lemma 8, we know that we can find an upstream model that leads to prior weights $\boldsymbol{\theta}_u$ such that sampling k nodes from a color class has probability at least $\sqrt{1-\epsilon}$; see the proof of Theorem 6. As argued before, by increasing the number of nodes with color class 3, we can make the probability that a uniform sampler picks a node with color classes 1 or 2 to be less than or equal to $\sqrt{\epsilon}$.

For case (2), where we sample edges to be removed from the original graph, consider the graphs in Figure D.3. For $k = 1$, we take the graphs (a) and (b). Both of these graphs have the same 1-WL coloring, indicated by the color numbers of the nodes. To separate the graphs, we need to sample from each graph an edge (u, v) such that either $C^1_\infty(u) = C^1_\infty(v) = 1$ or $C^1_\infty(u) = 1$ and $C^1_\infty(v) = 2$. Removing an edge between two nodes with color class 3 in both graphs would lead again to an identical color partition of the two graphs. Removing an edge between the color

classes $(1,1)$ and $(1,2)$ is possible by Lemma 9 and choosing a prior weight large enough such that the probability of sampling an edge between these color classes is at least $\sqrt{1-\epsilon}$; see the proof of Theorem 6. Since a random sampler would sample an edge uniformly at random, we simply have to increase the number of nodes with color class 3 such that the probability of sampling an edge between the color classes $(1,2)$ or $(1,2)$ is smaller than $\sqrt{\epsilon}$.

For $k > 1$, we construct the graphs depicted in Figure D.3(c) and (d) for $k = 2$. We first take a $(k+1)$-cycle and connect to each node of the said cycle the nodes with color classes 1 in the two different ways shown in Figure D.3(c) and (d). Finally, we again add pairs of connected nodes with color class 3. The two graphs can be separated by sampling k edges between the color classes $(1,1), (1,2)$, and $(2,2)$. For instance, sampling k edges between nodes in color class 2 leads to a disconnected subgraph of size 3 in the first graph but not the second. By Lemma 9, we know that we can learn an upstream MPNN that results in prior edge weights $\boldsymbol{\theta}_{uv}$ for all edges (u,v) where both u and v are in color class 2, such that sampling k of these edges has probability at least $\sqrt{1-\epsilon}$; see the proof of Theorem 6. Again, by increasing the number of nodes with color class 3, we can make the probability that a uniform sampler picks an edge between the color classes $(1,1), (1,2)$ or $(2,2)$ to be less than or equal to $\sqrt{\epsilon}$. □

Finally, we can also show a negative result, i.e., graphs exist such that PR-MPNNs cannot do better than random sampling.

Proposition D.3 (Proposition 5.4 in the main paper)**.** *For every $k > 0$, there exist non-isomorphic graphs H and H with identical 1-WL colorings such that every probability mass function $p_{(\boldsymbol{\theta},k)}$ separates the two graphs with the same probability as the distribution that samples nodes (edges) uniformly at random.*

Proof. Any pair of graphs where the 1-WL coloring consists of a single color class suffices to show the result. For instance, consider the graphs in Figure D.1(d), where all nodes have the same color. In fact, any pair of non-isomorphic d-regular graphs for $d > 0$ works here. An MPNN

upstream model cannot separate the prior weights of the nodes and, therefore, behaves as a uniform sampler. □

D.2.4 SIMPLE: Subset Implicit Likelihood

In this section, we introduce SIMPLE, which is a main component of our work. The goal of SIMPLE is to build a gradient estimator for $\nabla_{\boldsymbol{\theta}} L(\vec{X}, y; \boldsymbol{\omega})$. It is inspired by a hypothetical sampling-free architecture, where the downstream neural network $f_{\vec{d}}$ is a function of the marginals, $\boldsymbol{\mu} \coloneqq \mu(\boldsymbol{\theta}) \coloneqq \{p_{\boldsymbol{\theta}}(z_j \mid \sum_i z_i = k)\}_{j=1}^n$, instead of a discrete sample $\boldsymbol{z}$, resulting in a loss L_m s.t.

$$\nabla_{\boldsymbol{\theta}} L_m(\vec{X}, y; \boldsymbol{\omega}) = \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta})^\top \nabla_{\boldsymbol{\mu}} \ell_m(f_{\boldsymbol{u}}(\boldsymbol{\mu}, \vec{X}), y).$$

When the marginals $\mu(\boldsymbol{\theta})$ can be efficiently computed and differentiated, such a hypothetical pipeline can be trained end-to-end. Furthermore, Domke [2010] observed that, for an arbitrary loss function ℓ_m defined on the marginals, the Jacobian of the marginals w.r.t. the logits is symmetric. Consequently, computing the gradient of the loss w.r.t. the logits, $\nabla_{\boldsymbol{\theta}} L_m(\vec{X}, y; \boldsymbol{\omega})$, reduces to computing the *directional derivative*, or the Jacobian-vector product, of the marginals w.r.t. the logits in the direction of the gradient of the loss. This offers an alluring opportunity, i.e., the conditional marginals characterize the probability of each z_i in the sample, and could be thought of as a differentiable proxy for the samples. Specifically, by reparameterizing $\boldsymbol{z}$ as a function of the conditional marginal $\boldsymbol{\mu}$ under approximation $\partial_{\boldsymbol{\mu}} \boldsymbol{z} \approx \mathbf{I}$ as proposed by Niepert et al. [2021a], and using the straight-through estimator for the gradient of the sample w.r.t. the marginals on the backward pass, SIMPLE approximate

$$\nabla_{\boldsymbol{\theta}} L(\vec{X}, y; \boldsymbol{\omega}) \approx \partial_{\boldsymbol{\theta}} \mu(\boldsymbol{\theta}) \nabla_{\boldsymbol{z}} L(\vec{X}, y; \boldsymbol{\omega}),$$

where the directional derivative of the marginals can be taken along *any downstream gradient*, rendering the whole pipeline end-to-end learnable despite the presence of sampling.

Now, estimating the gradient of the loss w.r.t. the parameters can be thought of as decomposing into two sub-problems: **(P1)** Computing the derivatives of conditional marginals $\partial_{\boldsymbol{\theta}}\mu(\boldsymbol{\theta})$, which requires the computation of the conditional marginals, and **(P2)** Computing the gradient of the loss w.r.t. the samples $\nabla_{\boldsymbol{z}} L(\vec{X}, y; \boldsymbol{\omega})$ using sample-wise loss, which requires drawing exact samples. These two problems are complicated by conditioning on the k-subset constraint, which introduces intricate dependencies to the distribution and is infeasible to solve naively, e.g., by enumeration. Next, we will show the solutions that SIMPLE provide to each problem, at the heart of which is the insight that we need not care about the variables' order, only their sum, introducing symmetries that simplify the problem.

Derivatives of conditional marginals In many probabilistic models, the marginal inference is a #P-hard problem [Roth, 1996], and this is not the case for the k-subset distribution. Theorem 1 in Ahmed et al. [2023c] shows that the conditional marginals correspond to the partial derivatives of the log probability of the k-subset constraint. To see this, note that the derivative of a multi-linear function regarding a single variable retains all the terms referencing that variable and drops all other terms; this corresponds exactly to the unnormalized conditional marginals. By taking the derivative of the log probability, this introduces the k-subset probability in the denominator, leading to *conditional* marginals. Intuitively, the rate of change of the k-subset probability w.r.t. a variable only depends on that variable through its length-k subsets. They further show in Proposition 1 in Ahmed et al. [2023c] that the log probability of the exactly-k constraint $p_{\boldsymbol{\theta}}(\sum_j z_j = k)$ is tractable is tractable as well as amenable to auto-differentiation, solving problem **(P1)** exactly and efficiently.

Gradients of loss w.r.t. samples What remains is estimating the loss value, requiring faithful sampling from the k-subset distribution.To perform exact sampling from the k-subset distribution, SIMPLE starts by sampling the variables in reverse order, that is, it samples z_n through z_1. The intuition is that, having sampled $(z_n, z_{n-1}, \cdots, z_{i+1})$ with a Hamming weight of $k - j$, it samples Z_i with a probability of choosing $k-j$ of $n-1$ variables *and* the nth variable *given that* we choose

$k - j + 1$ of n variables, providing an exact and efficient solution to problem **(P2)**.

By combining the use of conditional marginal derivatives in the backward pass and the exact sampling in the forward pass, SIMPLE can achieve both low bias and low variance in its gradient estimation. We refer the readers to SIMPLE [Ahmed et al., 2023c] for the proofs and full details of the approach.

D.2.5 Datasets

Here, we give additional information regarding the datasets. The statistics of the datasets in our paper can be found in D.10. Among them, ZINC, ALCHEMY, MUTAG, PTC_MR, NCI1, NCI109, PROTEINS, IMDB-B, and IMDB-M are from TUDatasets [Morris et al., 2020]. Whereas PEPTIDES-FUNC and PEPTIDES-STRUCT are featured in Dwivedi et al. [2022b]. Besides, CORNELL, TEXAS and WISCONSIN are WebKB datasets [Craven et al., 1998] also used in Pei et al. [2020]. The OGB datasets are credited to Hu et al. [2020a]. Moreover, we also incorporate synthetic datasets from the literature. EXP dataset consists of partially isomorphic graphs as described in Abboud et al. [2020], while the graphs in the CSL dataset are synthetic regular graphs proposed in Murphy et al. [2019]. The construction of TREES-NEIGHBORSMATCH dataset is introduced in Alon and Yahav [2021].

Similar to the TREES-NEIGHBORSMATCH dataset, we propose our own TREES-LEAFCOUNT dataset. We fix a problem radius $R > 0$ and retrieve the binary representation of all numbers fitting into 2^R bits. This construction allows us to create 2^R unique binary trees by labeling the leaves with "0" and "1" corresponding to the binary equivalents of the numbers. A label is then assigned to the root node, reflecting the count of leaves tagged with "1". From the resulting graphs, we sample to ensure an equal class distribution. The task requires a model to predict the root label, thereby requiring a strategy capable of conveying information from the leaves to the root.

We aim to have a controlled environment to observe if our upstream model $h_{\vec{u}}$ can sample meaningful edges for the new graph configuration. Conventionally, a minimum of R message-passing

Table D.10: Dataset statistics and properties for graph-level prediction tasks, †—Continuous vertex labels following Gilmer et al. [2017], the last three components encode 3D coordinates.

Dataset	Properties						
	Number of graphs	Number of targets	Loss	∅ Number of vertices	∅ Number of edges	Vertex labels	Edge labels
Alchemy	202 579	12	MAE	10.1	10.4	✓	✓
Qm9	129 433	13	MAE	18.0	18.6	✓(13+3D)†	✓(4)
Zinc	249 456	1	MAE	23.1	24.9	✓	✓
Exp	1 200	2	ACC	44.5	55.2	✓	✗
CSL	150	10	ACC	41.0	82.0	✗	✗
ogbg-molhiv	41 127	2	ROCAUC	25.5	27.5	✓	✓
Cornell	1	5	ACC	183.0	298.0	✓	✗
Texas	1	5	ACC	183.0	325.0	✓	✗
Wisconsin	1	5	ACC	251.0	515.0	✓	✗
Trees-Leafcount($R=4$)	16 000	16	ACC	31	61	✓	✗
Trees-Neighborsmatch($R=4$)	14 000	7	ACC	31	61	✓	✗
Peptides-func	15 535	10	AP	150.9	153.7	✓	✓
Peptides-struct	15 535	11	MAE	150.9	153.7	✓	✓
MUTAG	188	2	ACC	17.9	19.8	✓	✓
PTC_MR	344	2	ACC	14.3	14.7	✓	✓
NCI1	4 110	2	ACC	29.9	32.3	✓	✗
NCI109	4 127	2	ACC	29.7	32.1	✓	✗
Proteins	1 113	2	ACC	39.1	72.8	✓	✗
IMDB-M	1 500	3	ACC	13.0	65.9	✗	✗
IMDB-B	1 000	2	ACC	19.7	96.5	✗	✗

layers is required to accomplish both tasks [Barcelo et al., 2020; Alon and Yahav, 2021]. However, a single-layer upstream MPNN could trivially resolve both datasets, provided the rewired graphs embed direct pathways from the root node to the leaf nodes containing the label information. To circumvent any potential bias within the sampling procedure, we utilize the self-attention mechanism described in Sec. 5.2.3 as our upstream model $h_{\vec{u}}$, along with a single-layer GIN architecture serving as the downstream model $f_{\vec{d}}$. For each problem radius, we sample exactly $k = 2^D$ edges. Indeed, our method consistently succeeded in correctly rewiring the graphs in all tested scenarios, extending up to a problem radius of $R = 6$, and achieved perfect test accuracy on both datasets. Figure 5.7 presents a qualitative result from the Trees-LeafCount dataset, further illustrating the capabilities of our approach.

D.2.6 Hyperparameter and Training Details

Experimental Protocol Table D.13 lists our hyperparameters choices. For all our experiments, we use early stopping with an initial learning rate of 0.001 that we decay by half on a plateau.

We compute each experiment's mean and standard deviation with different random seeds over a minimum of three runs. We take the best results from the literature for the other models, ex-

cept for SAT on the OGBG-MOLHIV, where we use the same hyperparameters as the authors use on ZINC. We evaluate test predictive performance based on validation performance. In the case of the WEBKB datasets, we employ a 10-fold cross-validation with the provided data splits. For PEPTIDES, OGBG-MOLHIV, ALCHEMY, and ZINC, our models use positional and structural embeddings concatenated to the initial node features. Specifically, we add both RWSE and LAPPE [Dwivedi et al., 2022a]. We use the same downstream model as the base model for the rewiring models.

Our code can be accessed at `https://github.com/chendiqian/PR-MPNN/`.

D.2.7 Additional Experimental Results

Here, we report on the computation times of different variants of our probabilistic graph rewiring schemes and results on synthetic datasets.

Training times We report the average training time per epoch in Table D.14. The RANDOM entry refers to using random adjacency matrices as rewired graphs.

Extended TUDatasets In addition to Table 5.10 in the main paper, we report the results of IMDB-B and IMDB-M datasets in Table D.12. We also propose a proper train/validation/test splitting and show the results in Table D.11.

QM9 We compare our PR-MPNN with multiple current methods on QM9 dataset, see Table D.15. The baselines are R-GNN in Alon and Yahav [2021], GNN-FiLM [Brockschmidt, 2020], SPN [Abboud et al., 2022] and the recent DRew paper [Gutteridge et al., 2023]. Following the settings of Abboud et al. [2022] and Gutteridge et al. [2023], we train the network on each task separately. We use the normalized regression labels for training and report the de-normalized numbers. Similar to Abboud et al. [2022] and Gutteridge et al. [2023], we also exclude the 3D coordinates of the datasets. It is worth noting that our PR-MPNN reaches the overall lowest mean absolute error on HOMO, LUMO, gap, and Omega tasks while gaining at most $14.13\times$ better performance compared with a base GIN model.

Table D.11: Extended results between our probabilistic rewiring method and the other approaches reported in Giusti et al. [2023b]. Besides the 10-fold cross-validation, as in Giusti et al. [2023b]; Xu et al. [2019], we provide a train/validation/test split. In addition, we also provide results on the IMDB-B and IMDB-M datasets. We use **green** for the best model, **blue** for the second-best, and **red** for third.

MODEL	MUTAG	PTC_MR	PROTEINS	NCI1	NCI109	IMDB-B	IMDB-M
PR-MPNN (10-FOLD CV)	**98.4**$\pm$2.4	**74.3**$\pm$3.9	**80.7**$\pm$3.9	**85.6**$\pm$0.8	**84.6**$\pm$1.2	**75.2**$\pm$3.2	**52.9**$\pm$3.2
PR-MPNN (TRAIN/VAL/TEST)	91.0$\pm$3.7	58.9$\pm$5.0	**79.1**$\pm$2.8	81.5$\pm$1.6	81.8$\pm$1.5	71.6$\pm$1.2	45.8$\pm$0.8

WebKB To show PR-MPNNs's capability on heterophilic graphs, we carry out experiments on the three WebKB datasets, namely CORNELL, TEXAS, and WISCONSIN, Table D.16. We compare with diffusion-based GNN [Gasteiger et al., 2019], Geom-GCN [Pei et al., 2020], and the recent graph rewiring work SDRF [Topping et al., 2021]. Besides the MPNN baselines above, we also compare them against graph transformers. PR-MPNNs consistently outperform the other MPNN methods and are even better than graph transformers on the TEXAS dataset.

LRGB We apply PR-MPNNs on the two Long Range Graph Benchmark tasks [Dwivedi et al., 2022b], PEPTIDES-FUNC and PEPTIDES-STRUCT, which are graph classification and regression tasks, respectively. The baseline methods are also reported in Gutteridge et al. [2023]. Notably, on PEPTIDES-STRUCT, PR-MPNNs reach the overall lowest mean absolute error.

D.2.8 Robustness analysis

A beneficial side-effect of training PR-MPNNs is the enhanced robustness of the downstream model to graph structure perturbations. This is because PR-MPNNs generate multiple adjacency matrices for the same graph during training, akin to augmenting training data by randomly dropping or adding edges, but with a parametrized "drop/add" distribution rather than a uniform one.

To observe the performance degradation when testing on noisy data, we conduct an experiment on the PROTEINS dataset. After training our models on clean data, we compared the models test accuracy on the clean and corrupted graphs. Corrupted graphs were generated by either deleting or adding a certain percentage of their edges. We report the change in the average test accuracy over

Table D.12: Extended comparison on the IMDB-B and IMDB-M datasets from the TUDATASET collection. We use **green** for the best model, **blue** for the second-best, and **red** for third.

MODEL	IMDB-B	IMDB-M
DGCNN [ZHANG ET AL., 2018]	70.0±0.9	47.8±0.9
IGN [MARON ET AL., 2019B]	71.3±4.5	48.6±3.9
GIN [XU ET AL., 2019]	75.1±5.1	52.3±2.8
PPGNS [MARON ET AL., 2019A]	73.0±5.7	50.4±3.6
NATURAL GN [DE HAAN ET AL., 2020]	74.8±2.0	51.2±1.5
GSN [BOURITSAS ET AL., 2022]	**77.8±3.3**	**54.3±3.3**
CIN [BODNAR ET AL., 2021]	**75.6±3.2**	**52.5±3.0**
PR-MPNN (10-FOLD CV)	**75.2±3.2**	**52.9±3.2**
PR-MPNN (TRAIN/VAL/TEST)	71.6±1.2	45.8±0.8

5 runs, comparing the base model with variants of PR-MPNN in Table D.18.

Table D.13: Overview of used hyperparameters.

DATASET	$\text{HIDDEN}_{\text{UPSTREAM}}$	$\text{HIDDEN}_{\text{DOWNSTREAM}}$	$\text{LAYERS}_{\text{UPSTREAM}}$	$\text{LAYERS}_{\text{DOWNSTREAM}}$	K_{ADD}	K_{RM}	L_{ADD}	HEUR	DROPOUT	N_{PRIORS}	$\text{SAMPLES}_{\text{TRAIN/TEST}}$
ZINC	{32, 64, 96}	256	{2,4,8}	4	[1, 128]	[1, 128]	256	DISTANCE	.0	1	5
QM9	64	{196, 256}	8	4	{5, 20, 35, 50, 80}	5	{100, 350}	DISTANCE	{.1, .5}	{1, 2}	{2, 3, 5}
PEPTIDES-FUNC	{128, 256}	{128, 256}	{4,8}	{4, 8}	{16, 64, 256, 512}	{16, 64, 256, 512}	{128, 256, 512, 2048}	DISTANCE	{.0, .1}	{1, 2, 5}	{1, 2, 5}
PEPTIDES-STRUCT	{128, 256}	{128, 256}	{4,8}	{4, 8}	{16, 64, 256, 512}	{16, 64, 256, 512}	{128, 256, 512, 2048}	DISTANCE	{.0, .1}	{1, 2, 5}	{1, 2, 5}
MUTAG	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
PTC_MR	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
NCI1	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
NCI109	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
PROTEINS	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
IMDB-M	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
IMDB-B	{32, 64}	{32, 64, 96}	{4, 8, 16}	{4,8}	{0, 5, 10, 25}	{0, 5, 10, 25}	256	DISTANCE	{.0, .1, .2}	{2, 3}	{2, 5}
CORNELL	[64, 256]	192	{2, 3, 4}	3	[1024, 2048]	[256, 512]	4096	SIMILARITY	.0	[1, 5]	{1, 5}
WISCONSIN	[64, 256]	192	{2, 3, 4}	3	[1024, 2048]	[256, 512]	4096	SIMILARITY	.0	[1, 5]	{1, 5}
TEXAS	[64, 256]	192	{2, 3, 4}	3	[1024, 2048]	[256, 512]	4096	SIMILARITY	.0	[1, 5]	{1, 5}
TREES-LEAFCOUNT	32	32	1	1	N_{LEAFS}	ALL	-	ALL	.0	1	1
TREES-NEIGHBORSMATCH	32	32	2	DEPTH+1	{20, 32, 64, 128, 256}	0	-	ALL	.0	1	1
CSL	32	128	{4, 8, 12}	2	1	0	1	DISTANCE	.0	{1, 3, 5}	{1, 10}
4-CYCLES	16	16	4	4	2	2	-	ALL	.0	{1, 5, 10}	{5, 10, 20, 30, 40, 50}
EXP	64	32	8	6	{1, 5, 10, 15, 20, 25, 50, 100}	{1, 5, 10, 15, 20, 25, 50, 100}	350	DISTANCE	.0	{1, 5, 10, 25 }	{1, 5, 10}

Table D.14: Train and validation time per epoch in seconds for a GINE model, the SAT Graph Transformer, and PR-MPNN using different gradient estimators on OGBG-MOLHIV. The time is averaged over five epochs. PR-MPNN is approximately 5 times slower than a GINE model with a similar parameter count, while SAT is approximately 30 times slower than the GINE, and 6 times slower than the PR-MPNN models. Experiments performed on a machine with a single Nvidia RTX A5000 GPU and a Intel i9-11900K CPU.

MODEL	#PARAMS	TOTAL SAMPLED EDGES	TRAIN TIME/EP (S)	VAL TIME/EP (S)
GINE	$502k$	-	3.19 $\pm$0.03	0.20 $\pm$0.01
K-ST SAT$_{GINE}$	$506k$	-	86.54 $\pm$0.13	4.78 $\pm$0.01
K-SG SAT$_{GINE}$	$481k$	-	97.94 $\pm$0.31	5.57 $\pm$0.01
K-ST SAT$_{PNA}$	$534k$	-	90.34 $\pm$0.29	4.85 $\pm$0.01
K-SG SAT$_{PNA}$	$509k$	-	118.75 $\pm$0.50	5.84 $\pm$0.04
PR-MPNN$_{Gmb}$	$582k$	20	15.20 $\pm$0.08	1.01 $\pm$0.01
PR-MPNN$_{Gmb}$	$582k$	100	18.18 $\pm$0.08	1.08 $\pm$0.01
PR-MPNN$_{Imle}$	$582k$	20	15.01 $\pm$0.22	1.08 $\pm$0.06
PR-MPNN$_{Imle}$	$582k$	100	15.13 $\pm$0.17	1.13 $\pm$0.06
PR-MPNN$_{Sim}$	$582k$	20	15.98 $\pm$0.13	1.07 $\pm$0.01
PR-MPNN$_{Sim}$	$582k$	100	17.23 $\pm$0.15	1.15 $\pm$0.01

Table D.15: Performance of PR-MPNN on QM9, in comparison with the base downstrem model (Base-GIN) and other competing methods. The relative improvement of PR-MPNN over the base downstream model is reported in the paranthesis. The metric used is MAE, lower scores are better. We note the best performing method with **green**, the second-best with **blue**, and third with **orange**.

PROPERTY	R-GIN+FA	GNN-FILM	SPN	DRew-GIN	BASE-GIN	PR-MPNN
MU	2.54±0.09	2.38±0.13	**2.32**±0.28	**1.93**±0.06	2.64±0.01	**1.99**±0.02 (1.33×)
ALPHA	**2.28**±0.04	3.75±0.11	**1.77**±0.09	**1.63**±0.03	7.67±0.16	**2.28**±0.06 (3.36×)
HOMO	1.26±0.02	**1.22**±0.07	1.26±0.09	**1.16**±0.01	1.70±0.02	**1.14**±0.01 (1.49×)
LUMO	1.34±0.04	**1.30**±0.05	**1.19**±0.05	**1.13**±0.02	3.05±0.01	**1.12**±0.01 (2.72×)
GAP	1.96±0.04	1.96±0.06	**1.89**±0.11	**1.74**±0.02	3.37±0.03	**1.70**±0.01 (1.98×)
R2	12.61±0.37	15.59±1.38	**10.66**±0.40	**9.39**±0.13	23.35±1.08	**10.41**±0.35 (2.24×)
ZPVE	5.03±0.36	11.00±0.74	**2.77**±0.17	**2.73**±0.19	66.87±1.45	**4.73**±0.08 (14.13×)
U0	**2.21**±0.12	5.43±0.96	**1.12**±0.13	**1.01**±0.09	21.48±0.17	**2.23**±0.13 (9.38×)
U	**2.32**±0.18	5.95±0.46	**1.03**±0.09	**0.99**±0.08	21.59±0.30	**2.31**±0.06 (9.35×)
H	**2.26**±0.19	5.59±0.57	**1.05**±0.04	**1.06**±0.09	21.96±1.24	2.66 ±0.01 (8.26×)
G	**2.04**±0.24	5.17±1.13	**0.97**±0.06	**1.06**±0.14	19.53±0.47	2.24±0.01 (8.24×)
CV	1.86±0.03	3.46±0.21	**1.36**±0.06	**1.24**±0.02	7.34±0.06	**1.44**±0.01 (5.10×)
OMEGA	0.80±0.04	0.98±0.06	**0.57**±0.04	**0.55**±0.01	0.60±0.03	**0.48**±0.00 (1.25×)

Table D.16: Quantitative results on the heterophilic and transductive WEBKB datasets. **Best overall**; **Second best**; **Third best**. Rewiring outperforms the base models on all of the datasets. Graph transformers have an advantage over both the base models and the ones employing rewiring.

		HETEROPHILIC & TRANSDUCTIVE		
		CORNELL ↑	TEXAS ↑	WISCONSIN ↑
MPNNs	BASE	0.574±0.006	0.674±0.010	0.697±0.013
	BASE W. PE	0.540±0.043	0.654±0.010	0.649±0.018
	DIGL [GASTEIGER ET AL., 2019]	0.582±0.005	0.620±0.003	0.495±0.003
	DIGL + UNDIRECTED [GASTEIGER ET AL., 2019]	0.595±0.006	0.635±0.004	0.522±0.005
	GEOM-GCN [PEI ET AL., 2020]	0.608± N/A	0.676± N/A	0.641± N/A
	SDRF [TOPPING ET AL., 2021]	0.546±0.004	0.644±0.004	0.555±0.003
	SDRF + UNDIRECTED [TOPPING ET AL., 2021]	0.575±0.003	0.703±0.006	0.615±0.008
	PR-MPNN (OURS)	0.659±0.040	**0.827±0.032**	0.750±0.015
GTs	GPS (LAPPE)	0.662±0.038	**0.778±0.010**	0.747±0.029
	GPS (RWSE)	**0.708±0.020**	**0.775±0.012**	**0.802±0.022**
	GPS (DEG)	**0.718±0.024**	0.773±0.013	**0.798±0.090**
	GRAPHORMER (DEG)	**0.683±0.017**	0.767±0.017	**0.770±0.019**
	GRAPHORMER (DEG + ATTN BIAS)	**0.683±0.017**	0.767±0.017	**0.770±0.019**

Table D.17: Comparison between PR-MPNN and other methods as reported in Gutteridge et al. [2023]. **Best overall**; **Second best**; **Third best**. PR-MPNN obtains the best score on the PEPTIDES-STRUCT dataset from the LRGB collection, but ranks below Drew on the PEPTIDES-FUNC dataset.

MODEL	PEPTIDES-FUNC AP ↑	PEPTIDES-STRUCT MAE ↓
GCN	0.5930±0.0023	0.3496±0.0013
GINE	0.5498±0.0079	0.3547±0.0045
GATEDGCN	0.5864±0.0077	0.3420±0.0013
GATEDGCN+PE	0.6069±0.0035	0.3357±0.0006
DIGL+MPNN	0.6469±0.0019	0.3173±0.0007
DIGL+MPNN+LAPPE	0.6830±0.0026	0.2616±0.0018
MIXHOP-GCN	0.6592±0.0036	0.2921±0.0023
MIXHOP-GCN+LAPPE	0.6843±0.0049	0.2614±0.0023
TRANSFORMER+LAPPE	0.6326±0.0126	**0.2529±0.0016**
SAN+LAPPE	0.6384±0.0121	0.2683±0.0043
GRAPHGPS+LAPPE	0.6535±0.0041	**0.2500±0.0005**
DREW-GCN	0.6996±0.0076	0.2781±0.0028
DREW-GCN+LAPPE	**0.7150±0.0044**	0.2536±0.0015
DREW-GIN	0.6940±0.0074	0.2799±0.0016
DREW-GIN+LAPPE	**0.7126±0.0045**	0.2606±0.0014
DREW-GATEDGCN	0.6733±0.0094	0.2699±0.0018
DREW-GATEDGCN+LAPPE	**0.6977±0.0026**	0.2539±0.0007
PR-MPNN	0.6825±0.0086	**0.2477±0.0005**

Table D.18: Robustness results on the PROTEINS dataset when testing on various levels of noise, obtained by removing or adding random edges. The percentages indicate the change in the average test accuracy over 5 runs. PR-MPNNs consistently obtain the best results for both removing $k = 25$ and $k = 50$ edges.

NOISE \ MODEL	GINE	PR-MPNN$_{k=25}$	PR-MPNN$_{k=50}$
RM 10%	-7.68%	+0.04%	+0.34%
RM 30%	-11.87%	-2.56%	-0.56%
RM 50%	-9.18%	-3.50%	-0.21%
ADD 10%	-5.93%	+0.28%	+0.80%
ADD 30%	-11.14%	-1.27%	+0.28%
ADD 50%	-21.75%	-1.99%	-0.43%

Bibliography

Pysdd. In *Recent Trends in Knowledge Compilation, Report from Dagstuhl Seminar 17381*, sep 2017.

Ralph Abboud, Ismail Ilkan Ceylan, Martin Grohe, and Thomas Lukasiewicz. The surprising power of graph neural networks with random node initialization. *arXiv preprint arXiv:2010.01179*, 2020.

Ralph Abboud, Radoslav Dimitrov, and Ismail Ilkan Ceylan. Shortest path networks for graph property prediction. In *Learning on Graphs Conference*, pages 5–1. PMLR, 2022.

Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *International Conference on Machine Learning*, pages 21–29. PMLR, 2019.

Kareem Ahmed, Eric Wang, Kai-Wei Chang, and Guy Van den Broeck. Leveraging unlabeled data for entity-relation extraction through probabilistic constraint satisfaction, mar 2021.

Kareem Ahmed, Tao Li, Thy Ton, Quan Guo, Kai-Wei Chang, Parisa Kordjamshidi, Vivek Srikumar, Guy Van den Broeck, and Sameer Singh. Pylon: A pytorch framework for learning with constraints. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (Demo Track)*, feb 2022a.

Kareem Ahmed, Stefano Teso, Kai-Wei Chang, Guy Van den Broeck, and Antonio Vergari. Semantic probabilistic layers for neuro-symbolic learning. In *NeurIPS*, 2022b.

Kareem Ahmed, Eric Wang, Kai-Wei Chang, and Guy Van den Broeck. Neuro-symbolic entropy regularization. In *The 38th Conference on Uncertainty in Artificial Intelligence*, 2022c.

Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. Semantic strengthening of neuro-symbolic learning. In *Proceedings of the 26th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2023a.

Kareem Ahmed, Kai-Wei Chang, and Guy Van den Broeck. A pseudo-semantic loss for deep autoregressive models with logical constraints. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023b.

Kareem Ahmed, Zhe Zeng, Mathias Niepert, and Guy Van den Broeck. Simple: A gradient estimator for k-subset sampling. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023c.

Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, 27(06):509–516, 1978.

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *International Conference on Learning Representations*, 2021.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*, 2016.

Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. Support vector machines for multiple-instance learning. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2002.

Eric Arazo, Diego Ortego, Paul Albert, Noel E. O'Connor, and Kevin McGuinness. Pseudo-labeling and confirmation bias in deep semi-supervised learning. *arXiv preprint arXiv:1908.02983*, 2019.

Ehsan Mohammady Ardehaly and Aron Culotta. Co-training for demographic classification using deep learning from label proportions. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 1017–1024. IEEE, 2017.

Akari Asai and Hannaneh Hajishirzi. Logic-Guided Data Augmentation and Regularization for Consistent Question Answering. In *ACL*, 2020.

Laszlo Babai and Ludik Kucera. Canonical labelling of graphs in linear average time. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 39–46. IEEE, 1979.

László Babai, Paul Erdos, and Stanley M Selkow. Random graph isomorphism. *SIaM Journal on computing*, 9(3):628–635, 1980.

Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *Journal of Machine Learning Research*, 18(109):1–67, 2017. URL `http://jmlr.org/papers/v18/15-631.html`.

Pradeep Kr Banerjee, Kedar Karhadkar, Yu Guang Wang, Uri Alon, and Guido Montúfar. Oversquashing in gnns through the lens of information contraction and graph expansion. In *2022 58th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1–8. IEEE, 2022.

Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: Understanding the cell's functional organization. *Nature reviews genetics*, 5(2):101–113, 2004.

Pablo Barcelo, Egor V. Kostylev, Mikael Monet, Jorge Pérez, Juan Reutter, and Juan Pablo Silva. The logical expressiveness of graph neural networks. In *International Conference on Learning Representations*, 2020.

Jessa Bekker and Jesse Davis. Learning from positive and unlabeled data: A survey. *Machine Learning*, 109:719–760, 2020.

Vaishak Belle, Andrea Passerini, and Guy Van den Broeck. Probabilistic inference in hybrid domains by weighted model integration. In *Proceedings of IJCAI*, pages 2770–2776, 2015.

David Berthelot, Nicholas Carlini, Ian Goodfellow, Nicolas Papernot, Avital Oliver, and Colin A Raffel. Mixmatch: A holistic approach to semi-supervised learning. In *Advances in Neural Information Processing Systems 32*. 2019.

Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis R. Bach. Learning with differentiable pertubed optimizers. In *NeurIPS*, 2020.

Julian Besag. Statistical analysis of non-lattice data. *Journal of the Royal Statistical Society. Series D (The Statistician)*, pages pp. 179–195, 1975.

Concha Bielza, Guangdi Li, and Pedro Larranaga. Multi-dimensional classification with bayesian networks. *International Journal of Approximate Reasoning*, 52(6):705–727, 2011.

Mathieu Blondel. Structured prediction with projection oracles. *Advances in neural information processing systems*, 32, 2019.

Mathieu Blondel, André FT Martins, and Vlad Niculae. Learning with fenchel-young losses. *J. Mach. Learn. Res.*, 21(35):1–69, 2020a.

Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, pages 950–959. PMLR, 2020b.

Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 92–100, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130570. doi: 10.1145/279943.279962. URL https://doi.org/10.1145/279943.279962.

Jakub Bober, Anthea Monod, Emil Saucan, and Kevin N Webster. Rewiring networks for graph neural network training using discrete geometry. *arXiv preprint arXiv:2207.08026*, 2022.

Cristian Bodnar, Fabrizio Frasca, Yuguang Wang, Nina Otter, Guido F Montufar, Pietro Lio, and Michael Bronstein. Weisfeiler and lehman go topological: Message passing simplicial networks. In *International Conference on Machine Learning*, pages 1026–1037. PMLR, 2021.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Hanen Borchani, Gherardo Varando, Concha Bielza, and Pedro Larranaga. A survey on multi-output regression. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 5 (5):216–233, 2015.

Gerda Bortsova, Florian Dubost, Silas Ørting, Ioannis Katramados, Laurens Hogeweg, Laura Thomsen, Mathilde Wille, and Marleen de Bruijne. Deep learning from label proportions for emphysema quantification. In *Medical Image Computing and Computer Assisted Intervention–MICCAI 2018: 21st International Conference, Granada, Spain, September 16-20, 2018, Proceedings, Part II 11*, pages 768–776. Springer, 2018.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *Proceedings of the 34th ICML*, 2017.

Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):657–668, 2022.

Marc Brockschmidt. Gnn-film: Graph neural networks with feature-wise linear modulation. In *International Conference on Machine Learning*, pages 1144–1152. PMLR, 2020.

Rickard Brüel-Gabrielsson, Mikhail Yurochkin, and Justin Solomon. Rewiring with positional encodings for graph neural networks. *arXiv preprint arXiv:2201.12674*, 2022.

Randal E Bryant and Christoph Meinel. Ordered binary decision diagrams. In *Logic synthesis and verification*, pages 285–307. Springer, 2002.

Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

Marc-André Carbonneau, Veronika Cheplygina, Eric Granger, and Ghyslain Gagnon. Multiple instance learning: A survey of problem characteristics and applications. *Pattern Recognition*, 2018.

Daniele Castellana and Federico Errica. Investigating the interplay between features and structures in graph learning, 2023.

Ming-Wei Chang, Lev Ratinov, and Dan Roth. Guiding semi-supervision with constraint-driven learning. In *Proceedings of the 45th ACL*, 2007.

Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. The MIT Press, 1st edition, 2010. ISBN 0262514125.

Dexiong Chen, Leslie O'Bray, and Karsten Borgwardt. Structure-aware transformer for graph representation learning. In *International Conference on Machine Learning*, pages 3469–3489. PMLR, 2022.

Guangyong Chen, Pengfei Chen, Chang-Yu Hsieh, Chee-Kong Lee, Benben Liao, Renjie Liao, Weiwen Liu, Jiezhong Qiu, Qiming Sun, Jie Tang, et al. Alchemy: A quantum chemistry dataset for benchmarking ai models. *arXiv preprint arXiv:1906.09427*, 2019.

Jianbo Chen, Le Song, Martin Wainwright, and Michael Jordan. Learning to explain: An information-theoretic perspective on model interpretation. In Jennifer Dy and Andreas Krause,

editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 883–892. PMLR, 10–15 Jul 2018.

Yu Chen, Lingfei Wu, and Mohammed Zaki. Iterative deep graph learning for graph neural networks: Better and robust node embeddings. *Advances in Neural Information Processing Systems*, 33:19314–19326, 2020.

Arthur Choi and Adnan Darwiche. Dynamic minimization of sentential decision diagrams. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 187–194. AAAI Press, 2013.

Myung Jin Choi, Vincent YF Tan, Animashree Anandkumar, and Alan S Willsky. Learning latent tree graphical models. *Journal of Machine Learning Research*, 12:1771–1812, 2011.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic modeling. 2020a.

YooJung Choi, Antonio Vergari, and Guy Van den Broeck. Probabilistic circuits: A unifying framework for tractable probabilistic models. *techreport*, 2020b. URL `http://starai.cs.ucla.edu/papers/ProbCirc20.pdf`.

YooJung Choi, Meihua Dang, and Guy Van den Broeck. Group fairness by probabilistic modeling with latent fair decisions. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, 2021.

YooJung Choi, Tal Friedman, and Guy Van den Broeck. Solving marginal map exactly by probabilistic circuit transformations. In *International Conference on Artificial Intelligence and Statistics*, pages 10196–10208. PMLR, 2022.

C Chow and Cong Liu. Approximating discrete probability distributions with dependence trees. *IEEE transactions on Information Theory*, 14(3):462–467, 1968.

Alvaro H. C. Correia, Robert Peharz, and Cassio P. de Campos. Joints in random forests. In *NeurIPS*, 2020a.

Alvaro HC Correia, Gennaro Gala, Erik Quaeghebeur, Cassio de Campos, and Robert Peharz. Continuous mixtures of tractable probabilistic models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7244–7252, 2023.

Gonçalo M Correia, Vlad Niculae, and André FT Martins. Adaptively sparse transformers. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2174–2184, 2019.

Gonçalo M Correia, Vlad Niculae, Wilker Aziz, and André FT Martins. Efficient marginalization of discrete and structured latent variables via sparsity. *Advances in Neural Information Processing Systems*, 2020b.

Caio Corro and Ivan Titov. Differentiable perturb-and-parse: Semi-supervised parsing with a structured variational auto-encoder. In *International Conference on Learning Representations*, 2019.

Timothee Cour, Ben Sapp, and Ben Taskar. Learning from partial labels. *The Journal of Machine Learning Research*, 12:1501–1536, 2011a.

Timothee Cour, Ben Sapp, and Ben Taskar. Learning from partial labels. *J. Mach. Learn. Res.*, 12 (null):1501–1536, jul 2011b. ISSN 1532-4435.

Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Seán Slattery. Learning to extract symbolic knowledge from the world wide web. *AAAI/IAAI*, 3(3.6):2, 1998.

Marco Cuturi, Olivier Teboul, and Jean-Philippe Vert. Differentiable ranking and sorting using optimal transport. *Advances in neural information processing systems*, 32, 2019.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Math. Control. Signals Syst.*, 5(4):455, 1992.

Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.

Wang-Zhou Dai, Qiu-Ling Xu, Yang Yu, and Zhi-Hua Zhou. Tunneling neural perception and logic reasoning through abductive learning, 2018.

Meihua Dang, Antonio Vergari, and Guy Van den Broeck. Strudel: Learning structured-decomposable probabilistic circuits. In *International Conference on Probabilistic Graphical Models*, pages 137–148. PMLR, 2020.

Meihua Dang, Pasha Khosravi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. Juice: A julia package for logic and probabilistic circuits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 16020–16023, 2021.

Meihua Dang, Anji Liu, and Guy Van den Broeck. Sparse probabilistic circuits via pruning and growing. *Advances in Neural Information Processing Systems*, 35:28374–28385, 2022a.

Meihua Dang, Antonio Vergari, and Guy Van den Broeck. Strudel: A fast and accurate learner of structured-decomposable probabilistic circuits. *International Journal of Approximate Reasoning*, 140:92–115, 2022b.

Alessandro Daniele, Emile van Krieken, Luciano Serafini, and F. V. Harmelen. Refining neural network predictions using background knowledge. *ArXiv*, abs/2206.04976, 2022.

Adnan Darwiche. A differential approach to inference in bayesian networks. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 123–132, 2000.

Adnan Darwiche. A logical approach to factoring belief networks. *KR*, 2:409–420, 2002.

Adnan Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 318–322, NLD, 2004. IOS Press. ISBN 9781586034528.

Adnan Darwiche. Sdd: A new canonical representation of propositional knowledge bases. In *IJCAI*, 2011a.

Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011b.

Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

Tirtharaj Dash, Sharad Chitlangia, Aditya Ahuja, and Ashwin Srinivasan. A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Scientific Reports*, 12(1): 1–15, 2022.

N. De Cao and T. Kipf. MolGAN: An implicit generative model for small molecular graphs. *International Conference on Machine Learning 2018 Workshop*, 2018.

Francesco De Comité, François Denis, Rémi Gilleron, and Fabien Letouzey. Positive and unlabeled examples help learning. pages 219–230, 12 1999. ISBN 978-3-540-66748-3. doi: 10.1007/3-540-46769-6_18.

Pim de Haan, Taco S. Cohen, and Max Welling. Natural graph networks. *Advances in Neural Information Processing Systems*, 33:3636–3646, 2020.

Luc De Raedt, Andrea Passerini, and Stefano Teso. Learning constraints from examples. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Luc De Raedt, Sebastijan Dumančić, Robin Manhaeve, and Giuseppe Marra. From statistical relational to neuro-symbolic artificial intelligence. In *IJCAI*, 2020.

Brian de Silva, Kathleen Champion, Markus Quade, Jean-Christophe Loiseau, J. Kutz, and Steven Brunton. Pysindy: A python package for the sparse identification of nonlinear dynamical systems from data. *Journal of Open Source Software*, 5(49):2104, 2020. doi: 10.21105/joss.02104. URL `https://doi.org/10.21105/joss.02104`.

Andreea Deac, Marc Lackenby, and Petar Veličković. Expander graph propagation. In *Learning on Graphs Conference*, pages 38–1. PMLR, 2022.

Krzysztof Dembczyński, Willem Waegeman, Weiwei Cheng, and Eyke Hüllermeier. On label dependence and loss minimization in multi-label classification. *Machine Learning*, 88(1-2): 5–45, 2012.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Aryan Deshwal, Janardhan Rao Doppa, and Dan Roth. Learning and inference for structured prediction: A unifying perspective. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)*, 2019.

Francesco Di Giovanni, Lorenzo Giusti, Federico Barbero, Giulia Luise, Pietro Lio, and Michael M Bronstein. On over-squashing in message passing neural networks: The impact of width, depth, and topology. In *International Conference on Machine Learning*, pages 7865–7885. PMLR, 2023.

Luca Di Liello, Pierfrancesco Ardino, Jacopo Gobbi, Paolo Morettin, Stefano Teso, and Andrea Passerini. Efficient generation of structured objects with constrained adversarial networks. *Advances in neural information processing systems*, 33:14663–14674, 2020.

Thomas Dietterich, Richard Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 03 2001. doi: 10.1016/S0004-3702(96)00034-3.

Michelangelo Diligenti, Marco Gori, Marco Maggini, and Leonardo Rigutini. Bridging logic and kernel machines. *Machine learning*, 86(1):57–88, 2012.

Michelangelo Diligenti, Marco Gori, and Claudio Sacca. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244:143–165, 2017a.

Michelangelo Diligenti, Soumali Roychowdhury, and Marco Gori. Integrating prior knowledge into deep learning. In *ICMLA*, 2017b.

Justin Domke. Implicit differentiation by perturbation. In *Advances in Neural Information Processing Systems 23*, pages 523–531. 2010.

Ivan Donadello, Luciano Serafini, and Artur d'Avila Garcez. Logic tensor networks for semantic image interpretation. In *IJCAI*, 2017.

Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *Joint european conference on machine learning and knowledge discovery in databases*, pages 312–315. Springer, 2015.

Marthinus C du Plessis, Gang Niu, and Masashi Sugiyama. Analysis of learning from positive and unlabeled data. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

Greg Durrett and Dan Klein. Neural CRF parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages

302–312. The Association for Computer Linguistics, 2015. doi: 10.3115/v1/p15-1030. URL https://doi.org/10.3115/v1/p15-1030.

Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

Vijay Prakash Dwivedi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Graph neural networks with learnable structural and positional representations. In *International Conference on Learning Representations*, 2022a.

Vijay Prakash Dwivedi, Ladislav Rampášek, Michael Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. Long range graph benchmark. *Advances in Neural Information Processing Systems*, 35:22326–22340, 2022b.

David Easley, Jon Kleinberg, et al. Networks, crowds, and markets. *Cambridge Books*, 2012.

Bahare Fatemi, Layla El Asri, and Seyed Mehran Kazemi. Slaps: Self-supervision improves structure learning for graph neural networks. *Advances in Neural Information Processing Systems*, 34:22667–22681, 2021.

Bahare Fatemi, Sami Abu-El-Haija, Anton Tsitsulin, Mehran Kazemi, Dustin Zelle, Neslihan Bulut, Jonathan Halcrow, and Bryan Perozzi. Ugsl: A unified framework for benchmarking graph structure learning. *arXiv preprint arXiv:2308.10737*, 2023.

Bertram Felgenhauer and Frazer Jarvis. Enumerating possible sudoku grids. 2005.

Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. DL2: Training and querying neural networks with logic. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1931–1941. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/fischer19a.html.

Luca Franceschi, Mathias Niepert, Massimiliano Pontil, and Xiao He. Learning discrete structures for graph neural networks. In *International Conference on Machine Learning*, pages 1972–1982. PMLR, 2019.

Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Ben Chamberlain, Michael Bronstein, and Federico Monti. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*, 2020.

Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: a survey. *IEEE transactions on neural networks and learning systems*, 25(5):845–869, 2013.

Gennaro Gala, Cassio de Campos, Robert Peharz, Antonio Vergari, and Erik Quaeghebeur. Probabilistic integral circuits. In *International Conference on Artificial Intelligence and Statistics*, pages 2143–2151. PMLR, 2024.

Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek. Fast rule mining in ontological knowledge bases with amie++. *The VLDB Journal*, 24(6):707–730, 2015.

Kuzman Ganchev, Joao Graça, Jennifer Gillenwater, and Ben Taskar. Posterior regularization for structured latent variable models. *Journal of Machine Learning Research*, 2010.

Saurabh Garg, Yifan Wu, Alexander J Smola, Sivaraman Balakrishnan, and Zachary Lipton. Mixture proportion estimation and pu learning: a modern approach. *Advances in Neural Information Processing Systems*, 34:8532–8544, 2021.

Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. Diffusion improves graph learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Samuel Gehman, Suchin Gururangan, Maarten Sap, Yejin Choi, and Noah A. Smith. Realtoxicityprompts: Evaluating neural toxic degeneration in language models. *ArXiv*, abs/2009.11462, 2020.

Robert Gens and Domingos Pedro. Learning the structure of sum-product networks. In *International conference on machine learning*, pages 873–880. PMLR, 2013.

Mor Geva, Ankit Gupta, and Jonathan Berant. Injecting numerical reasoning skills into language models. *arXiv preprint arXiv:2004.04487*, 2020.

Francesco Giannini, Michelangelo Diligenti, Marco Gori, and Marco Maggini. On a convex logic fragment for learning and reasoning. *IEEE Transactions on Fuzzy Systems*, 27(7):1407–1416, 2018.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

Eleonora Giunchiglia and Thomas Lukasiewicz. Coherent hierarchical multi-label classification networks. *Advances in Neural Information Processing Systems*, 33:9662–9673, 2020.

Eleonora Giunchiglia and Thomas Lukasiewicz. Multi-label classification neural networks with hard logical constraints. *Journal of Artificial Intelligence Research*, 72:759–818, 2021.

Eleonora Giunchiglia, Mihaela Catalina Stoian, and Thomas Lukasiewicz. Deep learning with logical constraints. *arXiv preprint arXiv:2205.00523*, 2022.

Lorenzo Giusti, Claudio Battiloro, Lucia Testa, Paolo Di Lorenzo, Stefania Sardellitti, and Sergio Barbarossa. Cell attention networks. In *2023 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2023a.

Lorenzo Giusti, Teodora Reu, Francesco Ceccarelli, Cristian Bodnar, and Pietro Liò. Cin++: Enhancing topological message passing. *arXiv preprint arXiv:2306.03561*, 2023b.

Peter W. Glynn. Likelihood ratio gradient estimation for stochastic systems. *Commun. ACM*, 33(10):75–84, October 1990. ISSN 0001-0782. doi: 10.1145/84537.84552. URL `https://doi.org/10.1145/84537.84552`.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Yves Grandvalet and Yoshua Bengio. Semi-supervised learning by entropy minimization. In *NeurIPS*, 2005.

Will Grathwohl, Dami Choi, Yuhuai Wu, Geoffrey Roeder, and David Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. *ICLR*, 2018.

Martin Grohe. *Descriptive complexity, canonisation, and definable graph structure theory*, volume 47. Cambridge University Press, 2017.

Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations*, 2018.

Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, and Sourab Mangrulkar. Accelerate: Training and inference at scale made simple, efficient and adaptable., 2022.

Quan Guo, Hossein Rajaby Faghihi, Yue Zhang, Andrzej Uszok, and Parisa Kordjamshidi. Inference-masked loss for deep structured output learning. In *IJCAI*, 2020.

Benjamin Gutteridge, Xiaowen Dong, Michael M Bronstein, and Francesco Di Giovanni. Drew: dynamically rewired message passing with delay. In *International Conference on Machine Learning*, pages 12252–12267. PMLR, 2023.

William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, June 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016b.

Xiaoxin He, Bryan Hooi, Thomas Laurent, Adam Perold, Yann LeCun, and Xavier Bresson. A generalization of vit/mlp-mixer to graphs. In *International Conference on Machine Learning*, pages 12724–12745. PMLR, 2023.

Geoffrey E. Hinton. Products of experts. In *Proceedings of the Ninth International Conference on Artificial Neural Networks*, 1999.

Nicholas Hoernle, Rafael-Michael Karampatsis, Vaishak Belle, and Ya'akov Gal. Multiplexnet: Towards fully satisfied logical constraints in neural networks. In *AAAI*, 2022.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in Neural Information Processing Systems*, 33:22118–22133, 2020a.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations*, 2020b.

Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. In *ACL*, 2016.

Zhiting Hu, Zichao Yang, Russ R Salakhutdinov, LIANHUI Qin, Xiaodan Liang, Haoye Dong, and Eric P Xing. Deep generative models with learnable knowledge constraints. In *Advances in Neural Information Processing Systems*, 2018.

Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances*

in Neural Information Processing Systems, volume 34, pages 25134–25145. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/d367eef13f90793bd8121e2f675f0dc2-Paper.pdf.

Eyke Hüllermeier. Learning from imprecise and fuzzy observations: Data disambiguation through generalized loss minimization. *International Journal of Approximate Reasoning*, 55(7):1519–1534, 2014.

Maximilian Ilse, Jakub Tomczak, and Max Welling. Attention-based deep multiple instance learning. In *International conference on machine learning*, pages 2127–2136. PMLR, 2018.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Wei Jin, Yao Ma, Xiaorui Liu, Xianfeng Tang, Suhang Wang, and Jiliang Tang. Graph structure learning for robust graph neural networks. *arXiv preprint arXiv:2005.10203*, 2020.

Wengong Jin, Connor Coley, Regina Barzilay, and Tommi Jaakkola. Predicting organic reaction outcomes with weisfeiler-lehman network. *Advances in Neural Information Processing Systems*, 30, 2017.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

Alan A. Kaptanoglu, Brian M. de Silva, Urban Fasel, Kadierdan Kaheman, Andy J. Goldschmidt, Jared Callaham, Charles B. Delahunt, Zachary G. Nicolaou, Kathleen Champion, Jean-Christophe Loiseau, J. Nathan Kutz, and Steven L. Brunton. Pysindy: A comprehensive python package for robust sparse system identification. *Journal of Open Source Software*, 7(69):3994, 2022. doi: 10.21105/joss.03994. URL https://doi.org/10.21105/joss.03994.

Giannis Karamanolakis, Subhabrata Mukherjee, Guoqing Zheng, and Ahmed Hassan Awadallah. Self-training with weak supervision. *arXiv preprint arXiv:2104.05514*, 2021.

Kedar Karhadkar, Pradeep Kr Banerjee, and Guido Montúfar. Fosr: First-order spectral rewiring for addressing oversquashing in gnns. *arXiv preprint arXiv:2210.11790*, 2022.

Anees Kazi, Luca Cosmo, Seyed-Ahmad Ahmadi, Nassir Navab, and Michael M Bronstein. Differentiable graph module (dgm) for graph convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):1606–1617, 2022.

Pasha Khosravi, YooJung Choi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. On tractable computation of expected predictions. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, dec 2019a. URL `http://starai.cs.ucla.edu/papers/KhosraviNeurIPS19.pdf`.

Pasha Khosravi, YooJung Choi, Yitao Liang, Antonio Vergari, and Guy Van den Broeck. On tractable computation of expected predictions. In *Advances in Neural Information Processing Systems*, pages 11169–11180, 2019b.

Pasha Khosravi, Antonio Vergari, YooJung Choi, Yitao Liang, and Guy Van den Broeck. Handling missing data in decision trees: A probabilistic approach. In *Proceedings of The Art of Learning with Missing Values, Workshop at ICML*, 2020.

Carolyn Kim, Ashish Sabharwal, and Stefano Ermon. Exact sampling with integer linear programs and random perturbations. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Ryuichi Kiryo, Gang Niu, Marthinus C Du Plessis, and Masashi Sugiyama. Positive-unlabeled learning with non-negative risk estimator. *Advances in neural information processing systems*, 30, 2017.

Johannes Klicpera, Janek Groß, and Stephan Günnemann. Directional message passing for molecular graphs. In *International Conference on Learning Representations*, 2020.

Ryoma Kobayashi, Yusuke Mukuta, and Tatsuya Harada. Learning from label proportions with instance-wise consistency. *arXiv preprint arXiv:2203.12836*, 2022.

Samuel Kolb, Paolo Morettin, Pedro Zuidberg Dos Martires, Francesco Sommavilla, Andrea Passerini, Roberto Sebastiani, and Luc De Raedt. The pywmi framework and toolbox for probabilistic inference using weighted model integration. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*, pages 6530–6532, 7 2019. doi: 10.24963/ijcai.2019/946.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Vladimir Kolmogorov. Blossom v: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1:43–67, 2009.

Wouter Kool, Herke Van Hoof, and Max Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, pages 3499–3508. PMLR, 2019.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 667–673, 2017. doi: 10.24963/ijcai.2017/93. URL https://doi.org/10.24963/ijcai.2017/93.

Yann LeCun. The mnist database of handwritten digits. 1998.

Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.

Dong-Hyun Lee. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *ICML Workshop on Challenges in Representation Learning*, 2013.

Tao Lei, R. Barzilay, and T. Jaakkola. Rationalizing neural predictions. In *EMNLP*, 2016.

Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

Fabien Letouzey, François Denis, and Rémi Gilleron. Learning From Positive and Unlabeled examples. In *Proceedings of the 11th International Conference on Algorithmic Learning Theory, ALT'00*, pages 71–85, Sydney, Australia, 2000. Springer Verlag.

Jurica Levatić, Dragi Kocev, and Sašo Džeroski. The importance of the label hierarchy in hierarchical multi-label classification. *Journal of Intelligent Information Systems*, 45(2):247–271, 2015.

Tao Li and Vivek Srikumar. Augmenting neural networks with first-order logic. In *ACL*, 2019.

Tao Li, Vivek Gupta, Maitrey Mehta, and Vivek Srikumar. A logic-driven framework for consistency of neural models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019.

Bill Yuchen Lin, Wangchunshu Zhou, Ming Shen, Pei Zhou, Chandra Bhagavatula, Yejin Choi, and Xiang Ren. Commongen: A constrained text generation challenge for generative commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1823–1840, 2020.

Anji Liu and Guy Van den Broeck. Tractable regularization of probabilistic circuits. *Advances in Neural Information Processing Systems*, 34, 2021.

Anji Liu, Stephan Mandt, and Guy Van den Broeck. Lossless compression with probabilistic circuits. *International Conference of Learning Representations*, 2022a.

Anji Liu, Hongming Xu, Guy Van den Broeck, and Yitao Liang. Out-of-distribution generalization by neural-symbolic joint training. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, feb 2023a.

Anji Liu, Honghua Zhang, and Guy Van den Broeck. Scaling up probabilistic circuits by latent variable distillation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023b.

Anji Liu, Kareem Ahmed, and Guy Van den Broeck. Scaling tractable probabilistic circuits: A systems perspective. In *Proceedings of the 41st International Conference on Machine Learning ICML*, 2024a.

Anji Liu, Mathias Niepert, and Guy Van den Broeck. Image inpainting via tractable steering of diffusion models. 2024b.

Fayao Liu, Guosheng Lin, and Chunhua Shen. Crf learning with cnn features for image segmentation. *Pattern Recognition*, 48(10):2983–2992, 2015a.

Kangning Liu, Weicheng Zhu, Yiqiu Shen, Sheng Liu, Narges Razavian, Krzysztof J Geras, and Carlos Fernandez-Granda. Multiple instance learning via iterative self-paced supervised con-

trastive learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3355–3365, 2023c.

Meng Liu, Zhengyang Wang, and Shuiwang Ji. Non-local graph neural networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(12):10270–10276, 2021.

Nian Liu, Xiao Wang, Lingfei Wu, Yu Chen, Xiaojie Guo, and Chuan Shi. Compact graph structure learning via mutual information compression. In *Proceedings of the ACM Web Conference 2022*, pages 1601–1610, 2022b.

Xuejie Liu, Anji Liu, Guy Van den Broeck, and Yitao Liang. Expressive modeling is insufficient for offline rl: A tractable inference perspective. *arXiv preprint arXiv:2311.00094*, 2023d.

Xuejie Liu, Anji Liu, Guy Van den Broeck, and Yitao Liang. Understanding the distillation process from deep generative models to tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 21825–21838. PMLR, 2023e.

Yixin Liu, Yu Zheng, Daokun Zhang, Hongxu Chen, Hao Peng, and Shirui Pan. Towards unsupervised deep graph structure learning. In *Proceedings of the ACM Web Conference 2022*, pages 1392–1403, 2022c.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015b.

Lorenzo Loconte, Nicola Di Mauro, Robert Peharz, and Antonio Vergari. How to turn your knowledge graph embeddings into generative models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Lorenzo Loconte, Aleksanteri M Sladek, Stefan Mengel, Martin Trapp, Arno Solin, Nicolas Gillis, and Antonio Vergari. Subtractive mixture models via squaring: Representation and learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

Andreas Loukas. What graph neural networks cannot learn: depth vs width. In *International Conference on Learning Representations*, 2020.

Daniel Lowd and Amirmohammad Rooshenas. The libra toolkit for probabilistic models. *Journal of Machine Learning Research*, 16:2459–2463, 2015.

Huchuan Lu, Qiuhong Zhou, Dong Wang, and Ruan Xiang. A co-training framework for visual tracking with multiple instance learning. In *2011 IEEE International Conference on Automatic Face & Gesture Recognition (FG)*, pages 539–544, 2011. doi: 10.1109/FG.2011.5771455.

Sidi Lu, Tao Meng, and Nanyun Peng. Insnet: An efficient, flexible, and performant insertion-based text generation model. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022a.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Neurologic decoding:(un) supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2021.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2022b.

Yi Luan, Luheng He, Mari Ostendorf, and Hannaneh Hajishirzi. Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction. In *EMNLP*, 2018.

Chris J Maddison, Daniel Tarlow, and Tom Minka. A$*$ sampling. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/dc5d637ed5e62c36ecb73b654b05ba2a-Paper.pdf.

Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. Approximate Inference for Neural Probabilistic Logic Programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486, 11 2021. doi: 10.24963/kr.2021/45. URL https://doi.org/10.24963/kr.2021/45.

Antonio Mari, Gennaro Vessio, and Antonio Vergari. Unifying and understanding overparameterized circuit representations via low-rank tensor decompositions. In *The 6th Workshop on Tractable Probabilistic Modeling*, 2023.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In *Advances in Neural Information Processing Systems*, pages 2153–2164, 2019a.

Haggai Maron, Heli Ben-Hamu, Nadav Shamir, and Yaron Lipman. Invariant and equivariant graph networks. In *International Conference on Learning Representations*, 2019b.

Oded Maron and Tomás Lozano-Pérez. A framework for multiple-instance learning. In M. Jordan, M. Kearns, and S. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1997.

Andre Martins and Ramon Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. In *International conference on machine learning*, pages 1614–1623. PMLR, 2016.

Saurabh Mathur, Vibhav Gogate, and Sriraam Natarajan. Knowledge intensive learning of cutset networks. In *Uncertainty in Artificial Intelligence*, pages 1380–1389. PMLR, 2023.

Nicholas Mattei and Toby Walsh. Preflib: A library of preference data HTTP://PREFLIB.ORG. In *ADT*, 2013a.

Nicholas Mattei and Toby Walsh. PrefLib: A library for preferences. In *International conference on algorithmic decision theory*, pages 259–270. Springer, 2013b.

Julian McAuley, J. Leskovec, and Dan Jurafsky. Learning attitudes and attributes from multi-aspect reviews. *2012 IEEE 12th International Conference on Data Mining*, pages 1020–1025, 2012.

David McClosky, Eugene Charniak, and Mark Johnson. Effective self-training for parsing. In *Proceedings of the main conference on human language technology conference of the North American Chapter of the Association of Computational Linguistics*. Association for Computational Linguistics, 2006.

Geoffrey J. McLachlan. Iterative reclassification procedure for constructing an asymptotically optimal rule of allocation in discriminant analysis. *Journal of the American Statistical Association*, 70(350):365–369, 1975.

Mattia Medina Grespan, Ashim Gupta, and Vivek Srikumar. Evaluating relaxations of logic for neural networks: A comprehensive study. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 2812–2818, 8 2021.

Tao Meng, Sidi Lu, Nanyun Peng, and Kai-Wei Chang. Controllable text generation with neurally-decomposed oracle. In *Advances in Neural Information Processing Systems 35 (NeurIPS)*, 2022.

Arthur Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. In *International Conference on Machine Learning*, pages 3462–3471. PMLR, 2018.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Octavio César Mesner and Cosma Rohilla Shalizi. Conditional mutual information estimation for mixed discrete and continuous variables with nearest neighbors. *arXiv: Statistics Theory*, 2019.

Pasquale Minervini, Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Adversarial sets for regularising neural link predictors. In *UAI*, 2017.

Pasquale Minervini, Luca Franceschi, and Mathias Niepert. Adaptive perturbation-based gradient estimation for discrete latent variable models. In *AAAI*, 2023.

Takeru Miyato, Shin-ichi Maeda, Shin Ishii, and Masanori Koyama. Virtual adversarial training: a regularization method for supervised and semi-supervised learning. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. *arXiv preprint arXiv:1901.03704*, 2019.

Paolo Morettin, Samuel Kolb, Stefano Teso, and Andrea Passerini. Learning weighted model integration distributions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5224–5231, 2020.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

Christopher Morris, Nils M Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*, 2020.

Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M Kriege, Martin Grohe, Matthias Fey, and Karsten Borgwardt. Weisfeiler and leman go machine learning: The story so far. *arXiv preprint arXiv:2112.09992*, 2021.

Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-dnnf compilation with sharpsat. Canadian AI'12, pages 356–361, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642303524. doi: 10.1007/978-3-642-30353-1_36. URL https://doi.org/10.1007/978-3-642-30353-1_36.

James Mullenbach, Sarah Wiegreffe, Jon Duke, Jimeng Sun, and Jacob Eisenstein. Explainable prediction of medical codes from clinical text. *arXiv preprint arXiv:1802.05695*, 2018.

Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampášek. Attending to graph transformers. *arXiv preprint arXiv:2302.04181*, 2023.

Kevin Murphy, Scott Linderman, Peter G Chang, Xinglong Li, Aleyna Kara, Giles Harper-Donnelly, and Gerardo Duran-Martin. Dynamax, 2023. URL https://github.com/probml/dynamax.

Ryan Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Relational pooling for graph representations. In *International Conference on Machine Learning*, pages 4663–4673. PMLR, 2019.

Yatin Nandwani, Abhishek Pathak, Mausam, and Parag Singla. *A Primal-Dual Formulation for Deep Learning with Constraints*. 2019.

Nagarajan Natarajan, Inderjit S Dhillon, Pradeep K Ravikumar, and Ambuj Tewari. Learning with noisy labels. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. Propagation kernels: efficient graph kernels from propagated information. *Machine learning*, 102:209–245, 2016.

Vlad Niculae and André F. T. Martins. Lp-sparsemap: Differentiable relaxed optimization for sparse structured prediction. In *ICML*, 2020.

Vlad Niculae, André F. T. Martins, Mathieu Blondel, and Claire Cardie. Sparsemap: Differentiable sparse structured inference. In *ICML*, 2018.

Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit mle: backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34:14567–14579, 2021a.

Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit mle: Backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34, 2021b.

Masaaki Nishino, Norihito Yasuda, Shin ichi Minato, and Masaaki Nagata. Compiling graph substructures into sentential decision diagrams. In *AAAI*, 2017.

Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3141–3148. AAAI Press, 2015. ISBN 9781577357384.

Pál András Papp, Karolis Martinkus, Lukas Faber, and Roger Wattenhofer. Dropgnn: Random dropouts increase the expressiveness of graph neural networks. *Advances in Neural Information Processing Systems*, 34:21997–22009, 2021.

Kyubyong Park. Can convolutional neural networks crack sudoku puzzles? `https://github.com/Kyubyong/sudoku`, 2018.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Max B Paulus, Dami Choi, Daniel Tarlow, Andreas Krause, and Chris J Maddison. Gradient estimation with stochastic softmax tricks. *arXiv preprint arXiv:2006.08063*, 2020.

Robert Peharz, Sebastian Tschiatschek, Franz Pernkopf, and Pedro Domingos. On theoretical properties of sum-product networks. In *Artificial Intelligence and Statistics*, pages 744–752. PMLR, 2015.

Robert Peharz, Robert Gens, Franz Pernkopf, and Pedro Domingos. On the latent variable interpretation in sum-product networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(10):2030–2044, 2016.

Robert Peharz, Steven Lang, Antonio Vergari, Karl Stelzner, Alejandro Molina, Martin Trapp, Guy Van den Broeck, Kristian Kersting, and Zoubin Ghahramani. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *International Conference on Machine Learning*, pages 7563–7574. PMLR, 2020a.

Robert Peharz, Antonio Vergari, Karl Stelzner, Alejandro Molina, Xiaoting Shao, Martin Trapp, Kristian Kersting, and Zoubin Ghahramani. Random sum-product networks: A simple and effective approach to probabilistic deep learning. In *Uncertainty in Artificial Intelligence*, pages 334–344. PMLR, 2020b.

Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. Geom-gcn: Geometric graph convolutional networks. In *International Conference on Learning Representations*, 2020.

Ben Peters, Vlad Niculae, and André FT Martins. Sparse sequence-to-sequence models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1504–1519, 2019.

Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Learning with algorithmic supervision via continuous relaxations. *Advances in Neural Information Processing Systems*, 34, 2021.

Hieu Pham and Quoc V Le. Semi-supervised learning by coaching. *Submitted to the 8th International Conference on Learning Representations*, 2019. `https://openreview.net/forum?id=rJe04p4YDB`.

Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In *AAAI*, volume 8, pages 517–522, 2008.

Marthinus Du Plessis, Gang Niu, and Masashi Sugiyama. Convex formulation for learning from positive and unlabeled data. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37, 2015.

Tobias Plötz and Stefan Roth. Neural nearest neighbors networks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Informa-*

tion Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, pages 1095–1106, 2018.

Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL) (Long Papers)*, 2018.

Andrzej Pronobis, Avinash Ranganath, and Rajesh PN Rao. Libspn: A library for learning and inference with sum-product networks and tensorflow. In *Principled Approaches to Deep Learning Workshop*, 2017.

Connor Pryor, Charles Dickens, Eriq Augustine, Alon Albalak, William Yang Wang, and Lise Getoor. Neupsl: Neural probabilistic soft logic. 2022.

Vasin Punyakanok, Dan Roth, and Wen-tau Yih. The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics*, 34(2):257–287, 2008. doi: 10.1162/coli.2008.34.2.257. URL `https://aclanthology.org/J08-2005`.

Chendi Qian, Andrei Manolache, Kareem Ahmed, Zhe Zeng, Guy Van den Broeck, Mathias Niepert, and Christopher Morris. Probabilistic task-adaptive graph rewiring. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

Novi Quadrianto, Alex Smola, Tibério Caetano, and Quoc Le. Estimating labels from label proportions. 2008.

Gwenolé Quellec, Guy Cazuguel, Béatrice Cochener, and Mathieu Lamard. Multiple-instance learning for medical image and video analysis. *IEEE reviews in biomedical engineering*.

Lawrence Rabiner and Biinghwang Juang. An introduction to hidden markov models. *ieee assp magazine*, 3(1):4–16, 1986.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Predrag Radivojac, Wyatt T Clark, Tal Ronnen Oron, Alexandra M Schnoes, Tobias Wittkop, Artem Sokolov, Kiley Graim, Christopher Funk, Karin Verspoor, Asa Ben-Hur, et al. A large-scale evaluation of computational protein function prediction. *Nature methods*, 10(3):221–227, 2013.

Tahrima Rahman, Prasanna Kothalkar, and Vibhav Gogate. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2014, Nancy, France, September 15-19, 2014. Proceedings, Part II 14*, pages 630–645. Springer, 2014.

Hossein Rajaby Faghihi, Quan Guo, Andrzej Uszok, Aliakbar Nafar, and Parisa Kordjamshidi. DomiKnowS: A library for integration of symbolic domain knowledge in deep learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 231–241, Online and Punta Cana, Dominican Republic, November . Association for Computational Linguistics. URL `https://aclanthology.org/2021.emnlp-demo.27`.

Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35:14501–14515, 2022.

Patrick Reiser, Marlen Neubert, André Eberhard, Luca Torresi, Chen Zhou, Chen Shao, Houssam Metni, Clint van Hoesel, Henrik Schopmans, Timo Sommer, et al. Graph neural networks for materials science and chemistry. *Communications Materials*, 3(1):93, 2022.

Tim Rocktäschel, Sameer Singh, and Sebastian Riedel. Injecting logical background knowledge into embeddings for relation extraction. In *Proceedings of the 2015 Conference of the NAACL*, 2015.

Jason Tyler Rolfe. Discrete variational autoencoders. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

Michal Rolinek, Vít Musil, Anselm Paulus, Marin Vlastelica, Claudio Michaelis, and Georg Martius. Optimizing rank-based metrics with blackbox differentiation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7620–7630, 2020.

Michal Rolínek, Paul Swoboda, Dominik Zietlow, Anselm Paulus, Vít Musil, and Georg Martius. Deep graph matching via blackbox differentiation of combinatorial solvers. In *ECCV*, 2020.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2020.

Chuck Rosenberg, Martial Hebert, and Henry Schneiderman. Semi-supervised self-training of object detection models. In *Proceedings of the Seventh IEEE Workshops on Application of Computer Vision*, 2005.

Dan Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302, 1996. doi: 10.1016/0004-3702(94)00092-1. URL `https://doi.org/10.1016/0004-3702(94)00092-1`.

Claudio Sacca, Stefano Teso, Michelangelo Diligenti, and Andrea Passerini. Improved multi-level protein–protein interaction prediction with semantic-based regularization. *BMC bioinformatics*, 15(1):1–18, 2014.

Avishkar Saha, Oscar Mendez, Chris Russell, and Richard Bowden. Learning adaptive neighborhoods for graph neural networks. *arXiv preprint arXiv:2307.09065*, 2023.

Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. Neuro-symbolic artificial intelligence: Current trends. *arXiv preprint arXiv:2105.05330*, 2021.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008a.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20(1):81–102, 2008b.

Clayton Scott and Jianxin Zhang. Learning from label proportions: A mutual contamination framework. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22256–22267. Curran Associates, Inc., 2020.

H Scudder. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3), 1965.

Nimish Shah, Laura Isabel Galindez Olascoaga, Shirui Zhao, Wannes Meert, and Marian Verhelst. Dpu: Dag processing unit for irregular graphs with precision-scalable posit arithmetic in 28 nm. *IEEE Journal of Solid-State Circuits*, 57(8):2586–2596, 2021.

Xiaoting Shao, Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Thomas Liebig, and Kristian Kersting. Conditional sum-product networks: Imposing structure on deep prob-

abilistic architectures. In *International Conference on Probabilistic Graphical Models*, pages 401–412. PMLR, 2020.

Xiaoting Shao, Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Thomas Liebig, and Kristian Kersting. Conditional sum-product networks: Modular probabilistic circuits via gate functions. *International Journal of Approximate Reasoning*, 140:298–313, 2022.

Yujia Shen, Arthur Choi, and Adnan Darwiche. Tractable operations for arithmetic circuits of probabilistic models. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper/2016/file/5a7f963e5e0504740c3a6b10bb6d4fa5-Paper.pdf.

Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495. PMLR, 2009.

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

Andy Shih and Stefano Ermon. Probabilistic circuits for variational inference in discrete graphical models. *Advances in Neural Information Processing Systems*, 33:4635–4646, 2020.

Sergey Shirobokov, Vladislav Belavin, Michael Kagan, Andrei Ustyuzhanin, and Atilim Gunes Baydin. Black-box optimization with local generative surrogates. *Advances in Neural Information Processing Systems*, 33:14650–14662, 2020.

Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J Sutherland, and Ali Kemal Sinop. Exphormer: Sparse transformers for graphs. *arXiv preprint arXiv:2303.06147*, 2023.

Vinay Shukla, Zhe Zeng, Kareem Ahmed, and Guy Van den Broeck. A unified approach to count-based weakly-supervised learning. In *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023.

Korsuk Sirinukunwattana, Shan e Ahmed Raza, Yee Tsang, David Snead, Ian Cree, and Nasir Rajpoot. Locality sensitive deep learning for detection and classification of nuclei in routine colon cancer histology images. *IEEE Transactions on Medical Imaging*, 35:1–1, 02 2016. doi: 10.1109/TMI.2016.2525803.

Aishwarya Sivaraman, Golnoosh Farnadi, Todd Millstein, and Guy Van den Broeck. Counterexample-guided learning of monotonic neural networks. *Advances in Neural Information Processing Systems*, 33:11936–11948, 2020.

Hwanjun Song, Minseok Kim, Dongmin Park, Yooju Shin, and Jae-Gil Lee. Learning from noisy labels with deep neural networks: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

Mohammad S Sorower. A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis*, 18:1–25, 2010.

Russell Stewart and Stefano Ermon. Label-free supervision of neural networks with physics and domain knowledge. *AAAI Conference on Artificial Intelligence*, 31(1), Feb. 2017. URL `https://ojs.aaai.org/index.php/AAAI/article/view/10934`.

Volker Strehl. Counting domino tilings of rectangles via resultants. *Advances in Applied Mathematics*, 27(2):597–626, 2001. ISSN 0196-8858. doi: https://doi.org/10.1006/aama.2001.0752. URL `https://www.sciencedirect.com/science/article/pii/S0196885801907523`.

Raymond Hendy Susanto, Shamil Chollampatt, and Liling Tan. Lexically constrained neural machine translation with levenshtein transformer. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2020.

Taro Tezuka and Shizuma Namekawa. Information bottleneck analysis by a conditional mutual information bound. *Entropy*, 23, 2021.

Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *ACM J. Exp. Algorithmics*, 21(1):1.12:1–1.12:44, 2016. doi: 10.1145/2975585. URL `https://doi.org/10.1145/2975585`.

Jake Topping, Francesco Di Giovanni, Benjamin Paul Chamberlain, Xiaowen Dong, and Michael M Bronstein. Understanding over-squashing and bottlenecks on graphs via curvature. *arXiv preprint arXiv:2111.14522*, 2021.

Marc Toussaint. Robot trajectory optimization using approximate inference. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161.

Kuen-Han Tsai and Hsuan-Tien Lin. Learning from label proportions with consistency regularization. In *Asian Conference on Machine Learning*, pages 513–528. PMLR, 2020.

Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *Proceedings of the twenty-first international conference on Machine learning*, page 104, 2004.

Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)*, 3(3):1–13, 2007.

George Tucker, Andriy Mnih, Chris J Maddison, Dieterich Lawson, and Jascha Sohl-Dickstein. Rebar: Low-variance, unbiased gradient estimates for discrete latent variable models. *Advances in Neural Information Processing Systems*, 2017.

Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 1979a.

L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 1979b.

Laurens van der Maaten. Learning a parametric embedding by preserving local structure. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 384–391, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

Emile van Krieken, Erman Acar, and F. V. Harmelen. Analyzing differentiable fuzzy logic operators. *ArXiv*, abs/2002.06100, 2020.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Simplifying, regularizing and strengthening sum-product network structure learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2015.

Antonio Vergari, Nicola Di Mauro, and Floriana Esposito. Visualizing and understanding sum-product networks. *Machine Learning*, 108(4):551–573, 2019.

Antonio Vergari, YooJung Choi, Robert Peharz, and Guy Van den Broeck. Probabilistic circuits: Representations, inference, learning and applications. In *Tutorial at the The 34th AAAI Conference on Artificial Intelligence*, 2020.

Antonio Vergari, YooJung Choi, Anji Liu, Stefano Teso, and Guy Van den Broeck. A compositional atlas of tractable circuit operations for probabilistic inference. In *NeurIPS*, 2021.

Ashwin K Vijayakumar, Michael Cogswell, Ramprasath R Selvaraju, Qing Sun, Stefan Lee, David Crandall, and Dhruv Batra. Diverse beam search: Decoding diverse solutions from neural sequence models. *arXiv preprint arXiv:1610.02424*, 2016.

Christopher Walker, Stephanie Strassel, Julie Medero, and Kazuaki Maeda. Ace 2005 multilingual training corpus. *LDC*, 2006.

Benjie Wang and Marta Kwiatkowska. Compositional probabilistic and causal inference using tractable circuit models. In *International Conference on Artificial Intelligence and Statistics*, pages 9488–9498. PMLR, 2023.

Boxin Wang, Wei Ping, Chaowei Xiao, Peng Xu, Mostofa Patwary, Mohammad Shoeybi, Bo Li, Anima Anandkumar, and Bryan Catanzaro. Exploring the limits of domain-adaptive training for detoxifying large-scale language models. In *Neurips*, 2022.

Po-Wei Wang, Priya L. Donti, Bryan Wilder, and J. Zico Kolter. Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 6545–6554. PMLR, 2019.

Xinggang Wang, Yongluan Yan, Peng Tang, Xiang Bai, and Wenyu Liu. Revisiting multiple instance neural networks. *Pattern Recognition*, 74:15–24, 2018.

Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series*, 2(9):12–16, 1968.

Johannes Welbl, Amelia Glaese, Jonathan Uesato, Sumanth Dathathri, John F. J. Mellor, Lisa Anne Hendricks, Kirsty Anderson, Pushmeet Kohli, Ben Coppin, and Po-Sen Huang. Challenges in detoxifying language models. *ArXiv*, abs/2109.07445, 2021.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992. doi: 10.1007/BF00992696.

Janusz Wojtusiak, Katherine Irvin, Aybike Birerdinc, and Ancha V Baranova. Using published medical results and non-homogenous data in rule learning. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, volume 2, pages 84–89. IEEE, 2011.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2020.

Zhanghao Wu, Paras Jain, Matthew Wright, Azalia Mirhoseini, Joseph E Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. *Advances in Neural Information Processing Systems*, 34:13266–13279, 2021.

Ming-Kun Xie and Sheng-Jun Huang. Partial multi-label learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Qizhe Xie, Eduard Hovy, Minh-Thang Luong, and Quoc V. Le. Self-training with noisy student improves ImageNet classification. *arXiv preprint arXiv:1911.04252*, 2019.

Sang Michael Xie and Stefano Ermon. Reparameterizable subset sampling via continuous relaxations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3919–3925. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/544.

Yujia Xie, Hanjun Dai, Minshuo Chen, Bo Dai, Tuo Zhao, Hongyuan Zha, Wei Wei, and Tomas Pfister. Differentiable top-k with optimal transport. *Advances in Neural Information Processing Systems*, 33:20520–20531, 2020.

Chang Xu, Dacheng Tao, and Chao Xu. A survey on multi-view learning. *arXiv preprint arXiv:1304.5634*, 2013.

Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Van den Broeck. A semantic loss

function for deep learning with symbolic knowledge. In *Proceedings of the 35th ICML 2018*, 2018a.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5453–5462. PMLR, 2018b.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Rui Xue, Haoyu Han, MohamadAli Torkamani, Jian Pei, and Xiaorui Liu. Lazygnn: Large-scale graph neural networks via lazy propagation. *arXiv preprint arXiv:2302.01503*, 2023.

Kevin Yang and Dan Klein. Fudge: Controlled text generation with future discriminators. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, 2021.

Yang Yang, Gennaro Gala, and Robert Peharz. Bayesian structure scores for probabilistic circuits. In *International Conference on Artificial Intelligence and Statistics*, pages 563–575. PMLR, 2023.

Lingyun Yao, Martin Trapp, Karthekeyan Periasamy, Jelin Leslin, Gaurav Singh, and Martin Andraud. Logarithm-approximate floating-point multiplier for hardware-efficient inference in probabilistic circuits. In *The 6th Workshop on Tractable Probabilistic Modeling*, 2023.

David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd Annual Meeting on Association for Computational Linguistics*, ACL '95, pages 189–196, USA, 1995. Association for Computational Linguistics. doi: 10.3115/981658.981684. URL https://doi.org/10.3115/981658.981684.

Donghan Yu, Ruohong Zhang, Zhengbao Jiang, Yuexin Wu, and Yiming Yang. Graph-revised con-

volutional network. In *Machine Learning and Knowledge Discovery in Databases: European Conference*, pages 378–393, 2021.

Felix X Yu, Krzysztof Choromanski, Sanjiv Kumar, Tony Jebara, and Shih-Fu Chang. On learning from label proportions. *arXiv preprint arXiv:1402.5902*, 2014.

Wenhao Yu, Chenguang Zhu, Zaitang Li, Zhiting Hu, Qingyun Wang, Heng Ji, and Meng Jiang. A survey of knowledge-enhanced text generation. *ACM Comput. Surv.*, 2022.

Zhe Zeng and Guy Van den Broeck. Collapsed inference for bayesian deep learning. *arXiv preprint arXiv:2306.09686*, 2023.

Zhe Zeng and Guy Van den Broeck. Efficient search-based weighted model integration. *Proceedings of UAI*, 2019.

Zhe Zeng, Paolo Morettin, Fanqi Yan, Antonio Vergari, and Guy Van den Broeck. Scaling up hybrid probabilistic inference with logical and arithmetic constraints via message passing. In *Proceedings of the International Conference of Machine Learning (ICML)*, 2020a.

Zhe Zeng, Paolo Morettin, Fanqi Yan, Antonio Vergari, and Guy Van den Broeck. Probabilistic inference with algebraic constraints: Theoretical limits and practical approximations. *Advances in Neural Information Processing Systems*, 33:11564–11575, 2020b.

Zhe Zeng, Paolo Morettin, Fanqi Yan, Antonio Vergari, and Guy Van den Broeck. Is parameter learning via weighted model integration tractable? In *Proceedings of the UAI Workshop on Tractable Probabilistic Modeling (TPM)*, jul 2021.

Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, jul 2023a.

Honghua Zhang, Meihua Dang, Nanyun Peng, and Guy Van den Broeck. Tractable control for autoregressive language generation. In *International Conference on Machine Learning*, pages 40932–40945. PMLR, 2023b.

Jiacheng Zhang, Yang Liu, Huanbo Luan, Jingfang Xu, and Maosong Sun. Prior knowledge integration for neural machine translation using posterior regularization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Xiao Zhang, Maria Leonor Pacheco, Chang Li, and Dan Goldwasser. Introducing DRAIL – a step towards declarative deep relational learning. In *Proceedings of the Workshop on Structured Prediction for NLP*, 2016.

Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. Greaselm: Graph reasoning enhanced language models. In *International Conference on Learning Representations*, 2021.

Yu Zhang, Zhenghua Li, and Min Zhang. Efficient Second-Order TreeCRF for Neural Dependency Parsing. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 3295–3305. Association for Computational Linguistics, 2020a. doi: 10.18653/v1/2020.acl-main.302.

Yu Zhang, Houquan Zhou, and Zhenghua Li. Fast and accurate neural CRF constituency parsing. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4046–4053, 2020b. doi: 10.24963/ijcai.2020/560. URL `https://doi.org/10.24963/ijcai.2020/560`.

Tong Zhao, Yozen Liu, Leonardo Neves, Oliver Woodford, Meng Jiang, and Neil Shah. Data augmentation for graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11015–11023, 2021.

Zexuan Zhong and Danqi Chen. A frustratingly easy approach for joint entity and relation extraction. *CoRR*, 2020.

Zhi-Hua Zhou. A brief introduction to weakly supervised learning. *National science review*, 5(1): 44–53, 2018.

Zhiyao Zhou, Sheng Zhou, Bochao Mao, Xuanyi Zhou, Jiawei Chen, Qiaoyu Tan, Daochen Zha, Can Wang, Yan Feng, and Chun Chen. Opengsl: A comprehensive benchmark for graph structure learning. *arXiv preprint arXiv:2306.10280*, 2023.

Xiaojin Zhu and Andrew B Goldberg. *Introduction to semi-supervised learning*. Springer Nature, 2022.

Xiaojin Jerry Zhu. Semi-supervised learning literature survey. 2005.

Dongcheng Zou, Hao Peng, Xiang Huang, Renyu Yang, Jianxin Li, Jia Wu, Chunyang Liu, and Philip S Yu. Se-gsl: A general and effective graph structure learning framework through structural entropy optimization. *arXiv preprint arXiv:2303.09778*, 2023.

Yang Zou, Zhiding Yu, BVK Vijaya Kumar, and Jinsong Wang. Unsupervised domain adaptation for semantic segmentation via class-balanced self-training. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 289–305, 2018.

Kaja Zupanc and Jesse Davis. Estimating rule quality for knowledge base completion with the relationship between coverage assumption. In *Proceedings of the 2018 World Wide Web Conference*, pages 1073–1081, 2018.