Learning to Reason: Leveraging Neural Networks for Approximate DNF Counting

Ralph Abboud, İsmail İlkan Ceylan, Thomas Lukasiewicz
Department of Computer Science
University of Oxford, UK
{ralph.abboud, ismail.ceylan, thomas.lukasiewicz}@cs.ox.ac.uk

Abstract

Weighted model counting (WMC) has emerged as a prevalent approach for probabilistic inference. In its most general form, WMC is #P-hard. Weighted DNF counting (weighted #DNF) is a special case, where approximations with probabilistic guarantees are obtained in $O(nm)$, where n denotes the number of variables, and m the number of clauses of the input DNF, but this is not scalable in practice. In this paper, we propose a *neural model counting* approach for weighted #DNF that combines approximate model counting with deep learning, and accurately approximates model counts in linear time when width is bounded. We conduct experiments to validate our method, and show that our model learns and generalizes very well to large-scale #DNF instances.

1 Introduction

Propositional *model counting (MC)*, or *#SAT*, is the task of counting the number of satisfying assignments for a given propositional formula (Gomes, Sabharwal, and Selman 2009). *Weighted model counting (WMC), or weighted #SAT,* additionally incorporates a *weight function* over the set of all possible assignments. Offering an elegant formalism for encoding various probabilistic inference problems, WMC is a unifying approach for inference in a wide range of probabilistic models. In particular, *probabilistic graphical models* (Koller and Friedman 2009), *probabilistic planning* (Domshlak and Hoffmann 2007), *probabilistic logic programming* (De Raedt, Kimmig, and Toivonen 2007), *probabilistic databases* (Suciu et al. 2011), and *probabilistic ontologies* (Borgwardt, Ceylan, and Lukasiewicz 2017) can greatly benefit from advances in WMC.

Two important special cases of WMC are *weighted #CNF* and *weighted #DNF*, which require the input formula to be in conjunctive normal form (CNF) and disjunctive normal form (DNF), respectively. Inference in probabilistic graphical models typically reduces to solving weighted #CNF instances, while query evaluation in probabilistic databases reduces to solving weighted #DNF instances (Suciu et al. 2011). However, both weighted #CNF and weighted #DNF

are known to be #P-hard (Valiant 1979), and this computational complexity is a major bottleneck for solving large-scale WMC instances.

To overcome this problem, two main paradigms have been developed. The first paradigm is *knowledge compilation* (Cadoli and Donini 1997; Selman and Kautz 1996), which solves computationally difficult problems by compiling them into a new representation (i.e., a target language), where they can be subsequently solved efficiently. Following compilation, exact inference in WMC can be done in linear time (Darwiche and Marquis 2002). However, the compilation process can produce exponentially-sized problem representations (i.e., arithmetic circuits). Furthermore, knowledge compilation is not robust to changes: for every change in the underlying model, the computationally demanding knowledge compilation process needs to be repeated. As a result, approaches based on knowledge compilation struggle to scale to large and varying problem instances.

The second paradigm is approximate solving (Ermon et al. 2013; Chakraborty, Meel, and Vardi 2016; Meel, Shrotri, and Vardi 2017), which provides approximations of the model count as opposed to an exact solution. Loosening the requirement for exactitude renders WMC more tractable, especially for #DNF counting, where approximate solving admits a fully polynomial randomized approximation scheme (FPRAS) due to Karp, Luby, and Madras (1989), which we denote KLM. KLM allows for faster estimation of #DNF model counts, while also providing probabilistic guarantees on its approximations, and it is the state of the art for weighted #DNF. Nonetheless, KLM runs in $O(nm)$, where n denotes the number of variables and m the number of clauses of the input DNF formula. Hence, KLM struggles to scale to real-world DNF formulas.

In this work, we propose Neural#DNF, an approach that combines deep learning and approximate model counting and enables fast weighted #DNF approximation. We first generate instances of weighted #DNF and solve them using KLM to produce training data. We then use a graph neural network (GNN) to capture the symbolic structure of DNF formulas and train our system. By construction, Neural#DNF produces approximations in $O(m\bar{w})$, where $\bar{w}$ denotes the average clause width. This reduces to just

$O(n+m)$ for bounded width. Our approach does not provide guarantees as with KLM, but instead enables a relative speed-up of multiple orders of magnitude in the average case. This is especially true, since, in practice, $\bar{w} << n$. Our experiments show that the GNN learns to accurately estimate weighted model counts and generalizes to novel formulas. Indeed, our model computes solutions to unseen weighted #DNF instances with 99% accuracy relative to an additive error threshold of 0.1 with respect to tight KLM approximations. It also generalizes to larger problem instances involving up to 15K variables remarkably well, despite only seeing formulas with at most 5K variables during training.

In summary, Neural#DNF makes the following contributions:

- It produces *efficient* and *highly accurate* weighted #DNF approximations in $O(m\bar{w})$, and in *linear time* with bounded width.

- It *reliably scales* to #DNF instances with up to 15K variables, which, to our knowledge, is a first for neural-symbolic methods.

- It is *robust* in that it can produce approximations for any problem instance over a given domain following training.

Our findings suggest that GNNs can effectively and efficiently perform large-scale #DNF through training with dense and reliable data. Further experiments and details are deferred to the appendix of this paper.[1]

2   Preliminaries

We briefly introduce weighted model counting, the KLM algorithm, and graph neural networks.

Weighted Model Counting

Given a (finite) set S of propositional variables, a *literal* is of the form v, or $\neg v$, where $v \in S$. A *conjunctive clause* is a conjunction of literals, and a *disjunctive clause* is a disjunction of literals. A clause has *width* k if it has exactly k literals. A formula ϕ is in *conjunctive normal form (CNF)* if it is a conjunction of disjunctive clauses, and it is in *disjunctive normal form (DNF)* if it is a disjunction of conjunctive clauses. We say that a DNF (resp., CNF) has width k if it contains clauses of width at most k. An assignment $\nu : S \mapsto \{0, 1\}$ maps every variable to either 0 (false), or 1 (true). An assignment ν *satisfies* a propositional formula ϕ, denoted $\nu \vDash \phi$, in the usual sense, where $\vDash$ is the propositional entailment relation.

Given a propositional formula ϕ, its *model count* $\#\phi$ is the number of assignments ν satisfying ϕ. The *weighted model count* of ϕ is given by $\sum_{\nu \vDash \phi} w(\nu)$, where $w : \mathfrak{A} \mapsto \mathbb{R}$ is a *weight function*, and $\mathfrak{A}$ is the set of all possible assignments. In this work, we set $w : \mathfrak{A} \mapsto [0,1] \cap \mathbb{Q}$ such that every assignment is mapped to a rational probability and $\sum_{\nu \in \mathfrak{A}} w(\nu) = 1$. As common in the literature, we view every propositional variable as an independent Bernoulli random variable and assign probabilities to literals.

The KLM Algorithm

Exactly solving weighted #DNF instances is #P-hard and thus intractable. The KLM algorithm (Karp, Luby, and Madras 1989) is a fully polynomial randomized approximation scheme (FPRAS), and provides probabilistic guarantees for weighted #DNF. More formally, given an error $\epsilon > 0$ and a confidence value $0 < \delta < 1$, KLM computes $\hat{\mu}$, an approximation of the true weighted model count μ, in polynomial time such that $\Pr\left(\mu(1-\epsilon) \le \hat{\mu} \le \mu(1+\epsilon)\right) \ge 1 - \delta$.

Specifically, for a DNF ϕ with n variables and m clauses, KLM computes a number of sampling trials $\tau = 8(1+\epsilon)m \log(\frac{2}{\delta})\frac{1}{\epsilon^2}$, and initializes a trial counter N to 0. Then, at every trial, KLM performs the following steps:

1. If no current sample assignment exists, randomly select a clause C_i with probability $\frac{p(C_i)}{\sum_{j=1}^{m} p(C_j)}$, then randomly generate a satisfying assignment for C_i using the variable probability distribution.

2. Check whether the current assignment satisfies a randomly selected clause C_k. If so, increment N and generate a new sample assignment. Otherwise, do nothing.

KLM returns $\frac{\tau \sum_{j=1}^{m} p(C_j)}{mN}$ as an estimate for the weighted DNF count. Since assignment checking runs in $O(n)$, the complexity of KLM amounts to $O\left(nm\epsilon^{-2} \log(\frac{1}{\delta})\right)$. KLM is the state of the art for approximate weighted #DNF, so we use it to label DNF formulas used to train our model.

Graph Neural Networks

Graph neural networks (GNNs) (Gori, Monfardini, and Scarselli 2005; Scarselli et al. 2009) are neural networks specifically designed to process structured graph data. In a GNN, every graph node x is given a vector representation v_x, which is updated iteratively. A node x receives information from its *neighborhood* $N(x)$, which is the set of nodes connected by an edge to x. Let $v_{x,t}$ denote the value of v_x at iteration t. We write a node update as:

$$v_{x,t+1} = combine\Big(v_{x,t}, aggregate\big(N(x)\big)\Big),$$

where *combine* and *aggregate* are functions, and *aggregate* is permutation-invariant. We use layer-norm LSTMs (Ba, Kiros, and Hinton 2016) as our *combine* function, and sum as our *aggregate* function. This is similar to gated graph neural networks (Li et al. 2016), except that we replace the gated recurrent unit (GRU) (Chung et al. 2014) with a layer-norm LSTM, given the remarkable empirical success of the latter (Selsam et al. 2019; Prates et al. 2019). Upon termination of all iterations, the final node representations are used to compute the target output.

GNNs are highly expressive computational models: GNNs can be as discerning between graphs as the Weisfeiler-Lehman (WL) graph isomorphism heuristic (Xu et al. 2019; Morris et al. 2019). Unlike feature engineering (Kashima, Tsuda, and Inokuchi 2003) and static embedding methods (Wang et al. 2014), GNNs can autonomously learn relationships between nodes, identify important features, and build models that can generalize to unseen graphs.
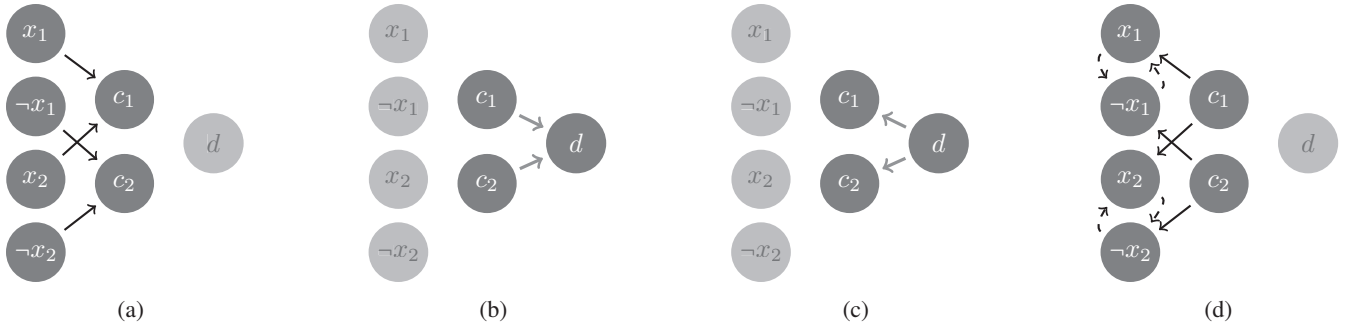
Figure 1: Message passing protocol on the DNF formula $\psi = (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$.
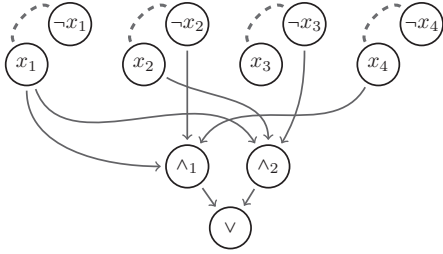

Figure 2: Graph encoding of the DNF formula $\phi = (x_1 \wedge \neg x_2 \wedge x_4) \vee (x_1 \wedge x_2 \wedge \neg x_3)$.

3   Graph Neural Network Model

We propose Neural#DNF, a new method for solving weighted #DNF problems based on GNNs. We model DNF formulas as graphs, and use a GNN to iterate over these graphs to compute an approximate weighted model count.

Model Setup

We encode a DNF formula as a graph with 3 layers as shown in Figure 2: a *literal* layer, a *conjunction* layer, and a *disjunction* layer. In the *literal* layer, every DNF variable is represented by 2 nodes corresponding to its positive and negative literals, which are connected by a (dashed) edge to highlight that they are complementary. In the *conjunction* layer, every node represents a conjunction and is connected to *literal* nodes whose literals appear in the conjunction. Finally, the *disjunction* layer contains a single disjunction node, which is connected to all nodes in the *conjunction* layer.

To approximate the model count of a DNF formula, we use a message-passing GNN model that iterates over the corresponding DNF graph and returns a Gaussian distribution. Initially, the network computes vector representations for all literal nodes, given their probabilities, using a multi-layer perceptron (MLP) f_{enc}. More formally, a k−dimensional representation $v_{x_i,0}$ of a literal x_i with probability p_i is computed as $v_{x_i,0} = f_{enc}(p_i)$. Nodes in the *conjunction* and *disjunction* layers are initialized to two representation vectors v_c and v_d, respectively, and the values for these vectors are learned over the course of training. After initialization, node representations are updated across T message passing iterations.

Message Passing Protocol

A message passing iteration consists of the following 4 steps:

(a) *Literal* layer nodes compute messages using an MLP M_l and pass them to their neighboring *conjunction* layer nodes. These *conjunction* nodes then aggregate these messages using the sum function and update their representation using a layer-norm LSTM L_{c_1}. The updated *conjunction* node representations, denoted $\hat{v}_{x_c,t+1}$, are given formally as:

$$\hat{v}_{x_c,t+1} = L_{c_1}\left(v_{x_c,t}, \sum_{x_l \in N(x_l)} M_l(v_{x_l,t})\right).$$

(b) *Conjunction* layer nodes compute and send messages to the disjunction node via an MLP M_c. The disjunction node aggregates these and updates using a layer-norm LSTM L_d, i.e.,

$$v_{x_d,t+1} = L_d\left(v_{x_d,t}, \sum_{x_c \in N(x_d)} M_c(\hat{v}_{x_c,t+1})\right).$$

(c) The disjunction node computes a message using an MLP M_d and sends it to the *conjunction* nodes, which update their representation using a different LSTM cell L_{c_2}:

$$v_{x_c,t+1} = L_{c_2}\left(\hat{v}_{x_c,t+1}, M_d(v_{x_d,t+1})\right).$$

(d) Using their latest representations, *conjunction* nodes send messages to neighboring nodes in the *literal* layer. *Literal* layer nodes aggregate these messages and concatenate them (represented with ‖) with messages from their corresponding negated literal. Then, they use this message to update their representations using a layer-norm LSTM L_l:

$$v_{x_l,t+1} = L_l\left(v_{x_l,t}, \left(\sum_{x_c \in N(x_l)} M_c(v_{x_c,t+1}) \| M_l(v_{\neg x_l,t})\right)\right).$$

A visual representation of the 4 message passing steps for a simple formula is provided in Figure 1. In this protocol, we use 2 distinct LSTM cells L_{c_1} and L_{c_2} to update the representations of *conjunction* nodes at steps (a) and (c), so that the network learns separate update procedures for literal-based and disjunction-based updates. At the end of message passing, the final disjunction node representation $v_{x_d,T}$ is passed through an MLP f_{out}. The final layer of this MLP consists of two neurons n_μ and n_σ, which return the mean and standard deviation, respectively, of a predicted Gaussian distribution.

Table 1: Distribution of formula sizes in the training set.

Size (n)	50	100	250	500
Count	30000	20000	16000	12000

Size (n)	750	1000	2500	5000
Count	10000	8000	6000	3000

Loss Function

Given ϵ and δ, KLM returns an estimate $\hat{\mu}$ of the true model count μ within a multiplicative bound with respect to ϵ, and this bound holds with probability $1-\delta$. By identifying different configurations of ϵ and δ that lead to an identical KLM running time, one can deduce that the probability mass is concentrated around $\hat{\mu}$ and decays away from it, and this holds for all DNFs. Note that the multiplicity of the bound interval on $\hat{\mu}$ w.r.t. ϵ makes it hard to fit standard distributions on it. Hence, we apply a natural logarithm to this bound to get the additive bound on $\log \mu$:

$$\log \hat{\mu} - \log(1+\epsilon) \le \log \mu \le \log \hat{\mu} + \log(1+\epsilon).$$

We can then fit a Gaussian $\mathcal{N}(\mu', \sigma)$ to this bound by setting $\mu' = \hat{\mu}$ and $\sigma = \log(1+\epsilon)/F^{-1}(1-\frac{\delta}{2})$, where F^{-1} denotes the inverse cumulative distribution function of the standard Gaussian distribution. The GNN is thus trained to predict $\log \mu$, a negative number. We adapt the exponential linear unit (ELU) (Clevert, Unterthiner, and Hochreiter 2016) activation function and apply it to n_μ and n_σ. More specifically, we use

$$ELU + 1(x) = \begin{cases} e^{-x} & \text{if } x \le 0 \\ x + 1 & \text{otherwise,} \end{cases}$$

such that n_μ uses $-ELU + 1(x)$, and n_σ uses $ELU + 1(x)$, thereby restricting their outputs to be negative and positive, respectively.

To compare the predicted Gaussian and the KLM result, we use Kullback-Leibler (KL) divergence, which for two Gaussians $\mathcal{N}_1(\mu_1, \sigma_1)$ and $\mathcal{N}_2(\mu_2, \sigma_2)$ is given by:

$$KL(\mathcal{N}_1, \mathcal{N}_2) = \log \frac{\sigma_2}{\sigma_1} - \frac{1}{2} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2}.$$

We set $\mathcal{N}_1$ to be the prediction returned by the network and $\mathcal{N}_2$ to be the KLM approximation. This choice is critical in order to avoid the system minimizing the training loss by learning to produce arbitrarily large values of σ_2.

4 Experiments

We train our model on a large set of DNF formulas and measure its generalization relative to new DNF formulas. These formulas are distinct in terms of *structure* (i.e., the underlying clauses and variables in every clause) and *size* (i.e., the number of clauses and variables is larger), so our experiments target generalization in both aspects. To evaluate *structure generalization*, we run our GNN on unseen formulas of comparable size to training formulas and measure its performance. To evaluate *size generalization*, we run tests
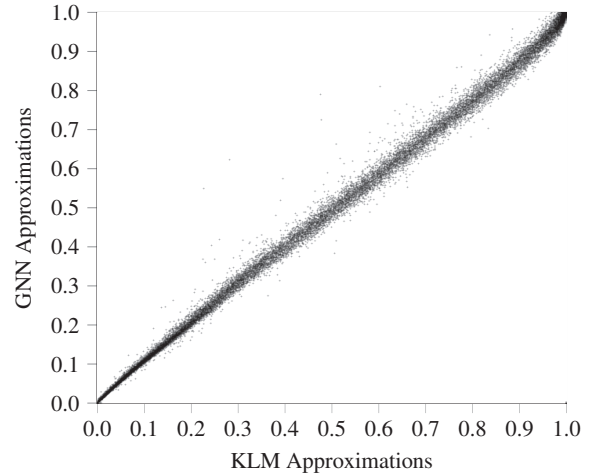

Figure 3: A gray-scale heat map representing the distribution of GNN predictions compared to KLM approximations.

on novel, larger formulas and assess how well the GNN performs. To further validate our model and data generation procedure, we also run these experiments using differently generated synthetic datasets (Meel, Shrotri, and Vardi 2018).

Experimental Setup

In our experiments, we compare Neural#DNF predictions $\hat{\mu}$ with those of KLM and check whether their absolute difference falls within pre-defined additive thresholds. We opt for additive error, as opposed to multiplicative error, as the former produces an absolute distance metric, whereas the latter is relative to the model count.

Owing to the lack of standardized benchmarks, we generate synthetic formulas using a novel randomized procedure designed to produce more variable formulas. We generate 100K distinct training formulas, where formula counts per n are shown in Table 1. For every n, formulas are generated with fixed clause width $w \in \{3, 5, 8, 13, 21, 34\}$ and number of clauses m from $\{0.25, 0.375, 0.5, 0.625, 0.75\} \cdot n$, such that every valid setting (i.e., all configurations except $w = 3$ and $m = 0.25 \cdot n$) is represented equally, and each formula has 4 variable probability distributions. More details about our data generation can be found in the appendix. The *structure* evaluation test set is generated analogously, and contains 13080 distinct formulas. The *size* evaluation set contains 348 formulas with $n = 10K$ and 116 formulas with $n = 15K$, with one probability distribution each.

For all experiments, we use $k = 128$-dimensional vector representations. We define f_{enc} as a 3-layer MLP with layer sizes 8, 32, and 128, message-generating MLPs (M_l, M_c, and M_d) as 4-layer MLPs with 128-sized layers, and f_{out} as a 3-layer MLP with layers of size 32, 8, and 2. We use the rectified linear unit (ReLU) as the activation function at MLP hidden layers, and linear activation at the output layer for all MLPs except f_{out}. For f_{out}, output activation is as defined in Section 3. Generated formulas were labelled using KLM with $\epsilon = 0.1$ and $\delta = 0.05$ to achieve a reasonable

Table 2: GNN accuracy (%) w.r.t. to additive thresholds.

Evaluation Data	Thresholds			
	0.02	0.05	0.10	0.15
Training Set	87.14	98.80	99.97	99.99
Test Set	87.37	98.76	99.95	99.98

trade-off between label accuracy and generation tractability.

We train the system for 4 epochs on a P100 GPU using KL divergence loss, the Adam optimizer (Kingma and Ba 2015), a learning rate of $\lambda = 10^{-5}$, a gradient clipping ratio of 0.5, and $T = 8$ message passing iterations.

Results

On the *structure* generalization test, Neural#DNF predictions align very closely with those of KLM, as shown in Figure 3. The model is within 0.02 of the KLM WMC estimate over 87.37% of the test set, and this rises to 99.95% for a threshold of 0.1. The model also performs consistently across different n, with accuracy varying by at most 4.5% between any two different n values for all four test thresholds. Overall test results are given in Table 2.

The proximity between training and testing accuracies at all thresholds shows that the network has not fit or memorized its training formulas, but has instead learned a general WMC procedure. The results parametrized by n are provided in Table 3. These results show that the network maintains a high accuracy (e.g., 95.5% for threshold 0.05) across all n values, and so does not rely on a particular n to achieve its high overall performance. Notably, the model is also robust against variation in w. As shown in Table 4, the model scores above 96% and 99% across all widths for thresholds 0.05 and 0.1, respectively. Interestingly, it has near-perfect performance for larger widths 13, 21, and 34, where weighted model counts are near-zero, and has relatively higher accuracy at threshold 0.02 when $w = 3$, where counts are almost one. Simultaneously performing well in both extreme cases, coupled with high accuracy on intermediate widths, further highlights the robustness of our model.

On the *size* generalization task, our model maintains accuracies of 97.13% and 94.83% with a threshold of 0.1 on 10K and 15K-variable formulas, respectively, despite having as many as triple the variables as in training. The full results for *size* generalization are given in Table 5. The same results parametrized by width w are also given in Table 6. These show that the network performs consistently across widths 3, 5, and 8, but performs less well at $w = 13$. This is due to formulas with $w = 13$ exhibiting a "phase transition" at this n and m. Indeed, in this setting, model counts fluctuate dramatically, since m is in the same order of magnitude as p^{-1}, the inverse expected clause satisfaction probability. In the training set, this phenomenon occurs at smaller widths, but never for $w = 13$, so this is an entirely new situation for the model, at a much larger scale. Nonetheless, it achieves an encouraging accuracy of 82.5% for the threshold 0.1.

These results show that reliable approximate model counting on large-scale formulas can be achieved, even with

Table 3: GNN accuracy (%) over test set by threshold versus number of formula variables (n).

n	Thresholds			
	0.02	0.05	0.10	0.15
50	85.58	98.58	99.98	100.0
100	87.87	98.87	100.0	100.0
250	87.93	99.24	100.0	100.0
500	87.67	99.40	99.99	100.0
750	87.56	99.15	100.0	100.0
1000	86.79	99.01	99.98	100.0
2500	90.06	98.17	99.85	99.94
5000	88.15	95.86	99.48	99.74

Table 4: GNN accuracy (%) over test set by threshold versus formula clause widths (w).

w	Thresholds			
	0.02	0.05	0.10	0.15
3	80.42	98.66	99.87	99.93
5	68.04	96.56	99.90	99.98
8	79.10	97.77	99.96	99.98
13	99.70	99.98	100.0	100.0
21	100.0	100.0	100.0	100.0
34	100.0	100.0	100.0	100.0

training restricted to smaller formulas. From a practical perspective, this gives further evidence that large-scale solvers can be trained using smaller formulas that are tractably labelled with existing solvers. In additional experiments run on differently generated datasets (cf. appendix), our system also maintains very high performance, and in fact performs better on fully random formulas (Meel, Shrotri, and Vardi 2018) than on formulas generated with our protocol. This further highlights the robustness of Neural#DNF, and validates the quality of our data generation procedure.

For all these results, message passing iterations are essential. Indeed, when run with just 2 message passing iterations for our ablation study, the GNN performs significantly worse across all experiments. However, this does not imply that performance always improves with more message passing. In fact, running too many message passing iterations makes the system prone to overfitting: When run with 32 iterations, the system achieves a similar performance in structure generalization, but its performance drops significantly in size generalization. This shows that a trade-off value of message passing iterations, in our case 8, must be selected, to enable sufficient communication, while not encouraging overfitting. Further details on our ablation study and experiments with 32 iterations can be found in the appendix.

All in all, our model achieves remarkable performance both in terms of structure and size generalization. These results highlight the power and scalability of neural message passing (NMP) methods to perform advanced reasoning tasks, and therefore justify further consideration of NMP.

Table 5: Accuracy (%) by threshold with respect to additive thresholds on *size* generalization test formulas.

n	Thresholds			
	0.02	0.05	0.10	0.15
10K	79.89	89.94	97.13	99.71
15K	72.41	81.90	94.83	97.41

Table 6: Accuracy (%) by threshold over *size* generalization test formulas versus w.

w	Thresholds			
	0.02	0.05	0.10	0.15
3	78.13	90.63	98.44	100.0
5	73.75	90.0	100.0	100.0
8	76.25	91.25	98.75	100.0
13	40.0	56.25	82.5	95.0
21,34	100.0	100.0	100.0	100.0

Running Time Analysis

In the average case, our GNN runs in $O(m\bar{w})$, where $\bar{w}$ denotes the average formula clause width. By contrast, KLM runs in $O(nm)$, so is much slower for standard cases in practice, where $\bar{w} << n$. In the worst case, our GNN runs in $O(nm)$, which is asymptotically identical to KLM. However, in a best-case scenario where $\bar{w}$ is upper-bounded, the GNN complexity drops to just $O(n + m)$, enabling linear-time approximations to be made, whereas KLM remains $O(nm)$, since its complexity does not depend on $\bar{w}$. Hence, our system enables much faster approximations than KLM in practice, where $\bar{w} << n$, and these approximations are in linear time with bounded clause width $\bar{w}$. A detailed complexity analysis for Neural#DNF can be found in the appendix.

Furthermore, our GNN runs on graphics processing units (GPUs) and thus benefits from accelerated computation. We show running times for the GNN vs. KLM ($\epsilon = 0.1, \delta = 0.05$) for formulas with ($w = 3, 34, m = 0.75n$) at every n in Table 7, and running times for all widths are provided in the appendix. However, we note that KLM and the GNN ran over different hardware (Haswell E5-2640v3 CPU vs. P100 GPU, resp.), since they are best suited to their respective devices (CPUs more efficiently handle multiple operations, like sampling, slicing, and comparison, whereas GPUs are efficient for repetitive floating point operations). Hence, these running times are only provided to highlight the scalability of the GNN with increasing formula size, which supports the formal running time expectations of the respective algorithms. Indeed, KLM requires 7.62 seconds for $w = 34, n = 1K$, and this rises rapidly to 305.61s for $w = 34, n = 15K$, whereas the GNN only needs 0.02 and 0.223 seconds, respectively. This is because the GNN takes advantage of limited width to deliver linear scalability, while KLM scales quadratically with n and m.

Finally, the GNN does not perform slower at smaller

Table 7: Runtimes (s) for KLM and our GNN by number of variables (n), with $w = 3, 34$ and $m = 0.75n$.

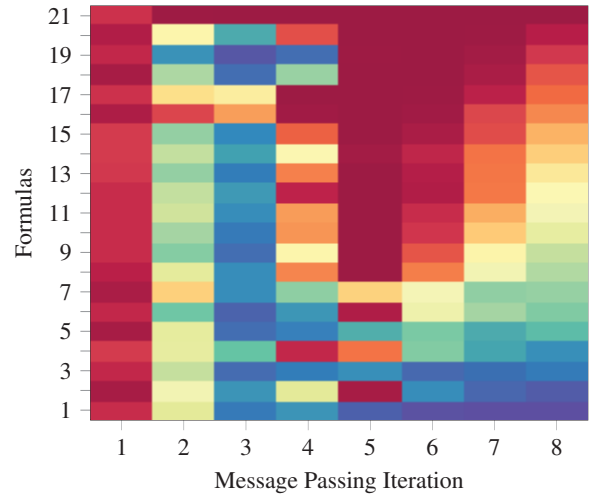| w	Algorithm	n			
		1K	5K	10K	15K
3	KLM	22.59	270.77	1151.86	2375.56
	GNN	0.017	0.040	0.073	0.104
34	KLM	7.62	43.57	164.46	305.61
	GNN	0.020	0.074	0.145	0.223

Figure 4: GNN estimates over message passing iterations. Red denotes small probability and blue high probability.

widths as with KLM, as it does not rely on sampling. With KLM, random assignments are replaced when they satisfy a clause, which means that with smaller clause widths, more replacements are made, as clause satisfaction is more likely, and this causes a heavy computational overhead. For example, KLM needs 2375 seconds (about 40 minutes) to run on a formula with $n = 15K$ and $w = 3$, using $\epsilon = 0.1$ and $\delta = 0.05$, whereas it only requires 306s when $w = 34$. By contrast, the GNN requires only 0.104 and 0.223 seconds, respectively.

Discussions: Analyzing the Model

To examine how our model makes predictions, we selected 21 formulas $f_i : i \in [1, 21]$ from the *structure* test set with weighted KLM model counts of roughly $\frac{21-i}{20}$. We then ran our GNN model on these formulas and computed the predicted probability at the end of every message passing iteration. Results are visualized in Figure 4. Initially, the network starts with a low estimate. Then, in the first 3 iterations, it accumulates probabilities and hits a "spike", which can be mapped to messages from literal nodes reaching the disjunction node. Following this, the network lowers its estimates, before adjusting and refining them in the final iterations.

Unlike (Selsam et al. 2019), where the estimate of satisfiability increases mostly monotonically, our network estimates fluctuate aggressively. A large initial estimate is made, and

then reduced and refined. In doing so, the network seems to be initially estimating the naive sum of conjunction probabilities, and subsequently revisiting its estimates as it better captures intersections between conjunctions. This falls in line with our observations, as any understanding of intersections can only occur starting from the third iteration, when the disjunction and conjunction nodes will have passed each other more global information. This also explains the limited performance observed in our ablation study: With just 2 iterations, the system cannot capture conjunction intersections, so can only make naive estimates.

5 Related Work

Weighted #DNF belongs to the wider family of WMC problems, which have been extensively studied due to their connection with probabilistic inference. Weighted #DNF is #P-hard (Valiant 1979), so is highly intractable. In fact, Toda proved that the class $P^{\#P}$ contains the entire polynomial hierarchy (Toda 1989). Surprisingly, even weighted #DNF counting on positive, partitioned DNF formulas with clause width at most 2 (Provan and Ball 1983) remains #P-hard.

As a result, many methods have been developed to exactly solve or approximate WMC solutions. One such method is *knowledge compilation (KC)*, where WMC problems are compiled into a new representation in which they are solved efficiently and exactly. KC pushes computational overhead to a preprocessing phase, but compensates for this by subsequently enabling efficient, linear-time probabilistic inference (Darwiche and Marquis 2002). However, compiled representations can be of exponential size in the worst-case. Hence, KC has limited scalability and robustness to model change, which has motivated research in approximate KC (Lowd and Domingos 2010; Friedman and Van den Broeck 2018). Our model training emulates KC preprocessing, but ultimately our model provides approximations and is more scalable. Moreover, it is robust to input changes, as it conceptually handles any formula over a fixed domain of variables and any probability distribution.

Another important paradigm is to produce approximate solutions to circumvent the intractability of WMC (Stockmeyer 1983). For the unweighted case (MC), *hashing-based methods* (Ermon et al. 2013; Chakraborty, Meel, and Vardi 2013) (see also (Chakraborty, Meel, and Vardi 2016)) produce an approximation with probabilistic guarantees. Importantly, (Chakraborty, Meel, and Vardi 2013) also yields an FPRAS when restricted to unweighted DNF; see, e.g., (Meel, Shrotri, and Vardi 2017). For hashing methods, approximation-preserving reductions from WMC to MC are known for CNF, but this remains open for the case of DNF (Chakraborty et al. 2015). Hence, *none* of these hashing methods apply to weighted #DNF. Beyond hashing techniques, *loopy belief propagation (LBP)* (Pearl 1982; Murphy, Weiss, and Jordan 1999) has been applied to approximate WMC. LBP does not provide any guarantees (Weiss 2000). Conceptually, our work also uses message passing, but instead learns messages and states so as to best capture the necessary information to relay. It also restricts all outgoing messages from a node to be identical.

Our work builds on recent applications of GNNs (Scarselli et al. 2009) to a variety of reasoning tasks, such as solving SAT (Selsam et al. 2019) and the traveling salesman problem (TSP) (Prates et al. 2019). There has also been work towards outright learning of inference in probabilistic graphical models (Yoon et al. 2018). These works achieve encouraging results, but only on very small instances (i.e., ~40 variables) of their respective problems. Indeed, they struggle to generalize to larger (but still small) instances. This is expected, since SAT and TSP are NP-complete and are hard to approximate with strong guarantees. Similarly, probabilistic inference in graphical models is #P-hard and remains NP-hard to approximate (as is weighted #CNF). Thus, significant work must be done in this direction to reach results of practical use. In contrast, our work tackles a problem with a known polynomial-time approximation, and learns from a dense dataset of approximate solutions with a very high accuracy at a large scale, and can generalize even further with tolerable loss in performance. To our knowledge, our model is the first proposal that combines reasoning and deep learning, while also scaling to realistic problem instance sizes.

6 Summary and Outlook

We presented Neural#DNF, a neural-symbolic approach that leverages the traditional KLM approximate weighted #DNF counter and GNNs to produce weighted #DNF approximations. This work shows that neural networks can be effectively and efficiently applied to large-scale weighted #DNF, given sufficiently dense and reliable training data. Therefore, it is particularly useful for query evaluation on large online probabilistic databases, where queries have computational limitations (Ceylan, Darwiche, and Van den Broeck 2016).

Looking forward, we will analyze the viability of GNNs for other reasoning problems, particularly in light of their expressive power, which could be limiting for problems with less structured graph representations. We hope that this work inspires further research leading to less data-dependent neural-symbolic methods, and a greater understanding of neural method performance over challenging problems.

7 Acknowledgements

References

Ba, J. L.; Kiros, J. R.; and Hinton, G. E. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Borgwardt, S.; Ceylan, İ. İ.; and Lukasiewicz, T. 2017. Ontology-mediated queries for probabilistic databases. In *Proc. of AAAI*.

Cadoli, M., and Donini, F. 1997. A survey on knowledge compilation. *AI Communications* 10(3-4).

Ceylan, İ. İ.; Darwiche, A.; and Van den Broeck, G. 2016. Open-world probabilistic databases. In *Proc. of KR*.

Chakraborty, S.; Fried, D.; Meel, K. S.; and Vardi, M. Y. 2015. From weighted to unweighted model counting. In *Proc. of IJCAI*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A scalable approximate model counter. In *Proc. of CP*.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2016. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Clevert, D.; Unterthiner, T.; and Hochreiter, S. 2016. Fast and accurate deep network learning by exponential linear units (ELUs). In *Proc. of ICLR*.

Darwiche, A., and Marquis, P. 2002. A Knowledge Compilation Map. *JAIR* 17(1).

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. ProbLog: A probabilistic prolog and its application in link discovery. In *Proc. of IJCAI*.

Domshlak, C., and Hoffmann, J. 2007. Probabilistic planning via heuristic forward search and weighted model counting. *JAIR* 30(1).

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2013. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*.

Friedman, T., and Van den Broeck, G. 2018. Approximate knowledge compilation by online collapsed importance sampling. In *Proc. of NeurIPS*.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2009. Model counting. In *Handbook of Satisfiability*. IOS Press.

Gori, M.; Monfardini, G.; and Scarselli, F. 2005. A new model for learning in graph domains. In *Proc. of IJCNN*.

Karp, R. M.; Luby, M.; and Madras, N. 1989. Monte-Carlo approximation algorithms for enumeration problems. *J. Algorithms* 10(3).

Kashima, H.; Tsuda, K.; and Inokuchi, A. 2003. Marginalized kernels between labeled graphs. In *Proc. of ICML*.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *Proc. of ICLR*.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2016. Gated graph sequence neural networks. In *Proc. of ICLR*.

Lowd, D., and Domingos, P. M. 2010. Approximate inference by compilation to arithmetic circuits. In *Proc. of NIPS*.

Meel, K. S.; Shrotri, A. A.; and Vardi, M. Y. 2017. On hashing-based approaches to approximate DNF-counting. In *Proc. of FSTTCS*.

Meel, K.; Shrotri, A.; and Vardi, M. 2018. Not all FPRASs are equal: Demystifying FPRASs for DNF-counting. *Constraints*.

Morris, C.; Ritzert, M.; Fey, M.; Hamilton, W. L.; Lenssen, J. E.; Rattan, G.; and Grohe, M. 2019. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *Proc. of AAAI*.

Murphy, K. P.; Weiss, Y.; and Jordan, M. I. 1999. Loopy belief propagation for approximate inference: An empirical study. In *Proc. of UAI*.

Pearl, J. 1982. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proc. of AAAI*.

Prates, M. O. R.; Avelar, P. H. C.; Lemos, H.; Lamb, L.; and Vardi, M. 2019. Learning to solve NP-complete problems - A graph neural network for the decision TSP. In *Proc. of AAAI*.

Provan, J. S., and Ball, M. O. 1983. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM* 12(4).

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20(1).

Selman, B., and Kautz, H. 1996. Knowledge compilation and theory approximation. *JACM* 43(2).

Selsam, D.; Lamm, M.; Bünz, B.; Liang, P.; de Moura, L.; and Dill, D. L. 2019. Learning a SAT solver from single-bit supervision. In *Proc. of ICLR*.

Stockmeyer, L. 1983. The complexity of approximate counting. In *Proc. of STOC*. ACM.

Suciu, D.; Olteanu, D.; Ré, C.; and Koch, C. 2011. *Probabilistic Databases*, volume 3. Morgan & Claypool.

Toda, S. 1989. On the computational power of PP and +P. In *Proc. of FOCS*.

Valiant, L. G. 1979. The complexity of computing the permanent. *TCS* 8(2).

Wang, Z.; Zhang, J.; Feng, J.; and Chen, Z. 2014. Knowledge graph embedding by translating on hyperplanes. In *Proc. of AAAI*.

Weiss, Y. 2000. Correctness of local probability propagation in graphical models with loops. *Neural Computation* 12(1).

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How powerful are graph neural networks? In *Proc. of ICLR*.

Yoon, K.; Liao, R.; Xiong, Y.; Zhang, L.; Fetaya, E.; Urtasun, R.; Zemel, R. S.; and Pitkow, X. 2018. Inference in probabilistic graphical models by graph neural networks. In *Workshop Proc. of ICLR*.