NSNet: A General Neural Probabilistic Framework for Satisfiability Problems

Zhaoyu Li[1,2], **Xujie Si**[1,2,3]
[1]McGill University, [2]Mila – Quebec AI Institute, [3]CIFAR AI Research Chair
{zli199, xsi}@cs.mcgill.ca

Abstract

We present the Neural Satisfiability Network (NSNet), a general neural framework that models satisfiability problems as probabilistic inference and meanwhile exhibits proper explainability. Inspired by the Belief Propagation (BP), NSNet uses a novel graph neural network (GNN) to parameterize BP in the latent space, where its hidden representations maintain the same probabilistic interpretation as BP. NSNet can be flexibly configured to solve both SAT and #SAT problems by applying different learning objectives. For SAT, instead of directly predicting a satisfying assignment, NSNet performs marginal inference among all satisfying solutions, which we empirically find is more feasible for neural networks to learn. With the estimated marginals, a satisfying assignment can be efficiently generated by executing a stochastic local search. For #SAT, NSNet performs approximate model counting by learning the Bethe approximation of the partition function. Our evaluations show that NSNet achieves competitive results in terms of inference accuracy and time efficiency on multiple SAT and #SAT benchmarks.

1 Introduction

The Boolean Satisfiability Problem (SAT) and the Sharp Satisfiability Problem (#SAT, or model counting) are fundamental challenges for computer science, with numerous applications including software verification [10, 17], hardware design [26, 34], and planning [11, 12]. Although modern SAT and #SAT solvers have achieved practical success in these domains, the performance of satisfiability solvers heavily relies on custom search heuristics [25]. However, designing good heuristics is highly non-trivial and time-consuming. With the superior learning ability of neural networks, recent research on satisfiability problems has migrated from hand-engineered methods toward data-driven ones.

Various neural methods have been proposed for satisfiability problems [14]. A line of research focuses on building standalone neural solvers, predicting the satisfiability [33, 20, 8] or a satisfying assignment [2, 3, 29] of a given instance. Other approaches integrate the neural modules into classic solvers, improving the branching heuristics in SAT [32] or #SAT [39] solvers. Yet, despite recent progress, the interpretation of these neural networks still remains a mystery: *why and how can neural networks learn to tackle satisfiability problems?* Answering this question is crucial for utilizing neural networks on satisfiability problems, especially for designing better neural architectures.

This paper addresses the above question by developing an explainable neural framework that can be interpreted using graphical models. Specifically, we propose **N**eural **S**atisfiability **Net**work (NSNet), a novel graph neural network (GNN) framework taking the formulation of solving satisfiability problems as probabilistic inference. Inspired by the inference algorithm Belief Propagation (BP), NSNet adopts a new graph encoding for propositional formulas and employs a novel message passing mechanism that subsumes the BP's updating rules in the latent space. In such a manner, NSNet serves as a neural extension of BP and allows the inference to learn from data. Like BP, NSNet can estimate

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

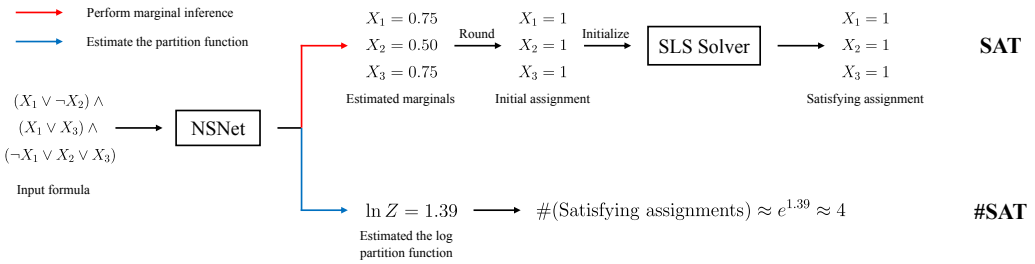

Figure 1: Overview of our pipeline for solving SAT and #SAT problems. For SAT, we apply NSNet to perform marginal inference and use the estimated marginals to guide the initialization of a SLS solver. For #SAT, NSNet itself can serve as an approximate solver by estimating the partition function.

both the marginals and the partition function of graphical models. This key observation makes NSNet a unified solution to both SAT and #SAT problems.

Existing neural SAT solvers [2, 3] aim to predict a *single* satisfying assignment for a satisfiable formula. However, there can be multiple satisfying solutions, making it unclear which particular solution should be generated. Instead of directly predicting a solution, NSNet performs *marginal inference* in the solution space of a SAT problem, estimating the assignment distribution of each variable among *all* satisfying assignments. Furthermore, without requiring computationally expensive labels, our framework can be effectively trained in an *unsupervised* manner using maximum likelihood estimation (MLE). Although NSNet is not directly trained to solve a SAT problem, its estimated marginals can be used to quickly generate a satisfying assignment. One simple way is to first round the estimated marginals to an initial assignment and then perform a stochastic local search (SLS) on it. Our experimental evaluations on the SATLIB benchmark show that (1) our initial assignments can solve 27.9% SAT instances, whereas the neural baseline only solves 7.8% instances, and (2) with a local search, NSNet can solve 71.1%, while the state-of-the-art SLS solver achieves 63.6%.

To solve #SAT, we formulate it as the *partition function estimation* of graphical models. NSNet learns the Bethe approximation of the partition function and thus acts as an approximate #SAT solver. Experiments on the BIRD and SATLIB benchmarks illustrate that our method notably surpasses both BP and the recent neural baseline. Compared with the state-of-the-art approximate #SAT solvers, NSNet can estimate the solutions with more than three orders of magnitude speedup, while still achieving competitive precision.

2 Preliminaries

Satisfiability Problems. In propositional logic, a Boolean formula consists of Boolean variables and logical operators such as negations ($\neg$), conjunctions ($\wedge$), and disjunctions ($\vee$). It is typical to represent Boolean formulas in conjunctive normal form (CNF), expressed as a conjunction of clauses, each of which is a disjunction of literals (a variable or its negation). Given a CNF formula, the SAT problem asks whether there exists an assignment to its variables that can satisfy the formula, whereas the goal for #SAT is to count the number of all satisfying solutions.

Factor Graphs. A factor graph is a bipartite graph with a set of variable nodes connected to a set of factor nodes, where each factor node expresses the dependencies among the variables it is connected to. Let p denote a distribution defined over n discrete random variables and x_i be a possible assignment of the i-th variable X_i. Then we may define the joint probability $p(\boldsymbol{x})$ of an assignment $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$ using m factors $\{f_1, f_2, \ldots, f_m\}$ as:

$$p(\boldsymbol{x}) = \frac{1}{Z} \prod_{a=1}^{m} f_a(\boldsymbol{x}_a), \quad Z = \sum_{\boldsymbol{x}} \left(\prod_{a=1}^{m} f_a(\boldsymbol{x}_a) \right), \tag{1}$$

c where each factor f_a takes an assignment of its associated variables $\boldsymbol{x}_a \subseteq \boldsymbol{x}$ as input, Z is the normalization constant (or partition function).

In the case of satisfiability problems, each Boolean variable and clause in a CNF formula correspond to a variable and a factor node in a factor graph respectively. Given a factor f_a with its associated

clause a, the input $\boldsymbol{x}_a$ represents a possible assignment for the variables appears in clause a, and $f_a(\boldsymbol{x}_a)$ takes value 1 if $\boldsymbol{x}_a$ satisfies the clause and 0 otherwise. By taking all factors into account, we derive that $p(\boldsymbol{x}) = 1/Z$ if and only if the assignment $\boldsymbol{x}$ satisfies the entire CNF formula, where the partition function Z indicates the number of all satisfying assignments. In other words, we can consider $p(\boldsymbol{x})$ as a probability measure on the space of all assignments that have a uniform distribution for all satisfying assignments and zero probability for unsatisfying ones.

Belief Propagation. BP is a variational inference algorithm to estimate the marginal distributions or the partition function Z of a factor graph. It performs iterative message passing between neighboring variable and factor nodes. Specifically, at the k-th iteration, the variable to factor messages $m^{(k)}_{i\to a}(x_i)$ and the factor to variable messages $m^{(k)}_{a\to i}(x_i)$ are computed as following:

$$m^{(k)}_{i\to a}(x_i) = \prod_{c\in\mathcal{N}(i)\setminus a} m^{(k-1)}_{c\to i}(x_i), \quad m^{(k)}_{a\to i}(x_i) = \sum_{\boldsymbol{x}_a\setminus x_i} f_a(\boldsymbol{x}_a) \prod_{j\in\mathcal{N}(a)\setminus i} m^{(k)}_{j\to a}(x_j), \tag{2}$$

where $\mathcal{N}(i)$ and $\mathcal{N}(a)$ denotes the neighbor nodes of node i and node a respectively. BP iteratively updates these messages until convergence or reaching a maximum number of iterations T. After this process, the variable beliefs $b_i(x_i)$ and the factor beliefs $b_a(\boldsymbol{x_a})$ can be computed to estimate the marginal distributions over the variables and the factors respectively:

$$b_i(x_i) \propto \prod_{a\in\mathcal{N}(i)} m^{(T)}_{a\to i}(x_i), \quad b_a(\boldsymbol{x}_a) \propto f_a(\boldsymbol{x}_a) \prod_{j\in\mathcal{N}(a)} m^{(T)}_{j\to a}(x_j). \tag{3}$$

Given the variable beliefs and the factor beliefs, BP can calculate a variational approximation of the partition function Z. Such an approximation is derived as the Bethe free energy $\mathcal{F} = -\ln Z$ in statistical physics [6], which is defined as:

$$\mathcal{F} = \sum_{a=1}^{m}\sum_{\boldsymbol{x}_a} b_a(\boldsymbol{x}_a)\ln\frac{b_a(\boldsymbol{x}_a)}{f_a(\boldsymbol{x}_a)} - \sum_{i=1}^{n}(|\mathcal{N}(i)|-1)\sum_{x_i} b_i(x_i)\ln b_i(x_i). \tag{4}$$

3 Methodology

In this section, we first rewrite standard BP to suit satisfiability problems with a probabilistic interpretation. Based on such a formulation, we introduce the graph encoding and message passing scheme of NSNet, which stands as a neural version of BP. By formulating SAT and #SAT as probabilistic inference tasks, we show how to utilize NSNet to solve these problems.

3.1 Rewriting Belief Propagation for Satisfiability Problems

For numerical stability, BP can be performed in log space, and its messages can be normalized at every iteration. Note each factor f_a can only take value 1 or 0 for satisfiability problems, thus we can remove the factor term in Equation 2 and rewrite it as:

$$m^{(k)}_{i\to a}(x_i) = -z^{(k)}_{i\to a} + \sum_{c\in\mathcal{N}(i)\setminus a} m^{(k-1)}_{c\to i}(x_i), \quad m^{(k)}_{a\to i}(x_i) = \underset{\boldsymbol{x}^*_a\setminus x_i}{\mathrm{LSE}}\left(\sum_{j\in\mathcal{N}(a)\setminus i} m^{(k)}_{j\to a}(x_j)\right). \tag{5}$$

LSE is the shorthand for the log-sum-exp function, and $z^{(k)}_{i\to a}$ is the normalization constant for the variable to factor messages at each iteration. The subscript $\boldsymbol{x}^*_a \setminus x_i$ means to fix the value x_i for the variable X_i and enumerate other *satisfying* variable assignments in clause a. Note the normalization term $z^{(k)}_{i\to a}$ ensures that $\sum_{x_i\in\{0,1\}}\exp(m^{(k)}_{i\to a}(x_i)) = 1$, while there is no such a normalization for $m^{(k)}_{a\to i}(x_i)$. From the perspective of graphical models, there is a probabilistic interpretation for the messages in Equation 5 [7]: the variable to clause message $m_{i\to a}(x_i)$ can be interpreted as the log probability that X_i takes value x_i when clause a is *not* considered and the clause to variable message $m_{a\to i}(x_i)$ represents as the log probability that clause a is satisfied when X_i takes the value of x_i. With this interpretation, BP performs message passing to approximate these probabilities iteratively.

However, BP often makes inaccurate estimates or fails to converge for satisfiability problems due to complex logical structures. To improve the inference of BP on satisfiability problems, we propose Neural Satisfiability Network (NSNet), a novel GNN framework that extends BP in the latent space, allowing the inference algorithm to learn from data.

3.2 NSNet Framework

Graph Representation. Inspired by the procedure of BP, we propose a new graph representation for CNF formulas. As shown in Figure 2, we represent a formula as an undirected bipartite graph with one type of node for each variable assignment and another type of node for each clause. An assignment node and a clause node are connected if the associated variable appears in the clause. The edges between them have two different types based on whether the clause can be satisfied using this variable assignment. Note that our graph representation can exactly express each message in BP: the direct edges from an assignment node to a clause node represent the messages $m_{i\to a}(x_i)$ and the edges from the other direction represent the messages $m_{a\to i}(x_i)$ in Equation 5.

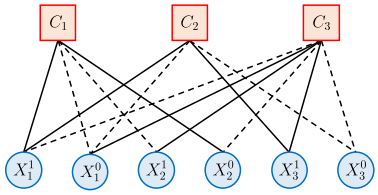

Figure 2: Our graph representation of the CNF formula $(X_1 \vee \neg X_2) \wedge (X_1 \vee X_3) \wedge (\neg X_1 \vee X_2 \vee X_3)$. X_i^1 and X_i^0 denotes variable X_i takes values 1 and 0 respectively. The solid/dashed line indicates that the variable assignment satisfies/dissatisfies the associated clause.

Message Passing Scheme. On top of our graph representation, we aim to design a learnable message passing scheme that parameterizes BP with neural modules. Formally, we define an embedding on every *directed* edge in the vector space $\mathbb{R}^d$: the variable assignment to clause embeddings $\boldsymbol{m}_{i\to a}(x_i)$ and the clause to variable assignment embeddings $\boldsymbol{m}_{a\to i}(x_i)$. Initially, we set all the embeddings of $\boldsymbol{m}_{i\to a}(x_i)$ and $\boldsymbol{m}_{a\to i}(x_i)$ to two learnable vectors $\boldsymbol{h}_{c2a}$ and $\boldsymbol{h}_{a2c}$ respectively. At the k-th iteration, these hidden representations are updated as:

$$\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(x_i) = \mathcal{A}_1\left(\sum_{c\in\mathcal{N}(i)\setminus a} \boldsymbol{m}_{c\to i}^{(k-1)}(x_i)\right),\ \boldsymbol{m}_{i\to a}^{(k)}(x_i) = \mathcal{A}_2\left(\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(x_i), \tilde{\boldsymbol{m}}_{i\to a}^{(k)}(1-x_i)\right), \quad (6)$$

$$\boldsymbol{m}_{a\to i}^{(k)}(x_i) = \mathcal{A}_3\left(\underset{\boldsymbol{x}_a^*\setminus x_i}{\mathrm{LSE}}\left(\sum_{j\in\mathcal{N}(a)\setminus i} \boldsymbol{m}_{j\to a}^{(k)}(x_j)\right)\right), \quad (7)$$

where $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ are all neural networks, which are parameterized by MLPs in our implementation. At each iteration, we adopt the aggregation operators for updating $\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(x_i)$ and $\boldsymbol{m}_{a\to i}^{(k)}(x_i)$ the same as BP in Equation 5, using summation for $\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(x_i)$ and LSE operation for $\boldsymbol{m}_{a\to i}^{(k)}(x_i)$. The subscript $\boldsymbol{x}_a^* \setminus x_i$ enumerates a set of assignment to clause embeddings connected to clause a except for the assignment x_i and ensures that at least one embedding on a 'satisfying' edge is included during the enumeration. The edge embedding $\boldsymbol{m}_{i\to a}^{(k)}(x_i)$ is updated by incorporating both $\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(x_i)$ and its flip value's $\tilde{\boldsymbol{m}}_{i\to a}^{(k)}(1-x_i)$. With the same aggregation strategy as BP, our framework can be viewed as a generalized BP algorithm that maintains the same probabilistic interpretation of BP and encodes these probabilities in the latent space.

Proposition 1. *Our message passing scheme subsumes the BP algorithm, where BP is a special case of our message passing.*

If we set the feature dimension d to 1, and let the neural networks $\mathcal{A}_1$ and $\mathcal{A}_3$ be the identity function, $\mathcal{A}_2$ be the normalization function $\mathcal{A}_2(a,b) = a - \log(\exp(a) + \exp(b))$, then the updating rules in Equation 6- 7 is equivalent to Equation 5. Thus we can consider BP as a non-parameterized GNN and NSNet as a neural version of BP that generalizes it in the vector space with learnable parameters.

In contrast to existing GNN architectures for satisfiability problems [33, 20, 8, 2, 29], NSNet defines latent representations on *edges* and performs message passing between neighboring *edges* rather than *nodes*. Such design also enables NSNet to enforce the permutation invariance and the negation equivariance of CNF formulas [33]:

Proposition 2. *For any parameterization of $\boldsymbol{h}_{c2a}$, $\boldsymbol{h}_{a2c}$, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, the embeddings of NSNet is either invariant or equivariant to the following operations: (1) permute any variables. (2) permute any clauses. (3) permute any variables within a clause. (4) negating every literal of a given variable.*

Due to the space limit, we leave the explanation and proof of Proposition 2 in Appendix A.1.

3.3 SAT Solving

We first show how NSNet can be used to solve the SAT problem. Although any satisfying assignment is a valid solution to the SAT problem, we need to consider the following question carefully:

Question 1. *If a neural network can predict a satisfying assignment, how and why does the neural network produce one particular solution instead of other feasible ones?*

Existing neural methods [20, 8, 2, 3] employ neural networks to produce a value $v \in [0, 1]$ for each variable, where this value is regarded as a soft assignment and rounded to 1 or 0 as the actual assignment. Unfortunately, these approaches fail to answer Question 1: they are trained to generate a specific satisfying solution without considering other possible ones. As a result, neural networks are biased towards specific solutions during training and uncertain about which one to predict during testing, making their generalization ability poor.

To address this question, we leverage NSNet to perform *marginal inference*, i.e., computing the marginal distribution of each variable among all satisfying assignments. In other words, instead of solving a SAT problem directly, we aim to estimate the fraction of each variable that takes 1 or 0 in the entire solution space. Note the marginal for each variable takes all feasible solutions into account and is unique, which is more stable and interpretable to be reasoned by the neural networks. Similar to Equation 3 used by BP to compute variable beliefs, NSNet estimates each marginal value $b_i(x_i)$ by aggregating the clause to variable assignment messages through a MLP and a softmax function:

$$\tilde{b_i}(x_i) = \text{MLP}\left(\sum_{a \in \mathcal{N}(i)} m^{(T)}_{a \to i}(x_i)\right), \quad [b_i(1), b_i(0)] = \text{softmax}\left(\left[\tilde{b_i}(1), \tilde{b_i}(0)\right]\right). \tag{8}$$

One may expect to train our framework in a supervised fashion by computing the ground-truth marginal for each variable. However, obtaining such labels requires us to *enumerate* all satisfying assignments, which is computationally infeasible even for medium-size instances. To alleviate this issue, we propose an *unsupervised* training strategy from a simple Bayesian perspective [16].

Let $\theta_i = b_i(1)$ denote the variable bias representing the probability that the variable X_i takes value 1 to satisfy a given formula. Then we can view the satisfiability of the formula as observation S and NSNet's outputs $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$ as parameters in the Bayesian framework. Since we perform marginal inference only on satisfiable formulas, thus the observation S will always be a scalar 1. The interpretation for such modeling is that we have known a given formula can be satisfied, but not how exactly the variables went about choosing a satisfying assignment. So we want to find the variable biases $\boldsymbol{\theta}$ that will best support our observation. This would produce the desired $P(\boldsymbol{\theta}|S = 1)$, which exactly represents the marginal value for each variable among all satisfying assignments. We can then learn the model parameters $\boldsymbol{\theta}$ by maximizing the log-likelihood of the observed variable $S = 1$, that is, $\max_{\boldsymbol{\theta}} \log P_{\boldsymbol{\theta}}(S = 1)$. Note the $P_{\boldsymbol{\theta}}(S = 1)$ stands for the probability we obtain a satisfying assignment by sampling an assignment with weight $\boldsymbol{\theta}$, we can decompose it for each clause a:

$$P_{\boldsymbol{\theta}}(S = 1) = \prod_a P_{\boldsymbol{\theta}}(S_a = 1) = \prod_a (1 - P_{\boldsymbol{\theta}}(S_a = 0)) = \prod_a \left(1 - \prod_{i \in a^+} (1 - \theta_i) \prod_{i \in a^-} \theta_i\right), \tag{9}$$

where a^+ and a^- denote the set of variables that occur positively and negatively in the clause a respectively. Such a formulation let us minimize the loss $\mathcal{L} = -\log P_{\boldsymbol{\theta}}(S = 1)$ to train NSNet *without* any supervision. We note that the loss bears a resemblance to the one used in QuerySAT [29], there are important differences: in QuerySAT, as a query mechanism, its loss is applied during both training and testing phases to produce soft assignments. However, we derive it from the Bayesian perspective and only employ it as an unsupervised loss to train NSNet to perform marginal inference.

Now we consider how to generate a satisfying assignment after obtaining the estimated marginals $\boldsymbol{\theta}$. Note that the marginal value θ_i for the variable X_i represents the probability X_i takes 1 among all satisfying solutions, X_i favors taking the value of 1 to satisfy a formula if $\theta_i > 0.5$ and vice versa. So one simple way is to round the marginal θ_i to a value $x_i = \lfloor \theta_i + 0.5 \rfloor$ for each variable to produce an assignment. However, when each variable takes its favoring value independently, such an assignment may *not* satisfy the formula even if the estimated marginals are perfect. Therefore, we also perform the local search on this assignment until finding a satisfying one. In practice, we integrate NSNet with a SLS solver by modifying the solver's initialization under the guidance of our

estimated marginals. Such a combination is easy to implement and has a modest overhead. It should be stressed that marginals are important in SAT solving as they represent the underlying distribution of individual variables. Besides using local search, one can adopt other methods such as decimation to generate a satisfying assignment. We leave the discussion of other approaches in Appendix A.2.

3.4 Model Counting

We also apply NSNet to the #SAT problem. Exact model counting is a well-known #P-Complete problem [40] and is almost infeasible without a search procedure. Due to its inherent complexity, a line of research focuses on studying approximate model counting [37, 41, 9, 35, 1, 22], where the goal is to count the number of satisfying assignments *with certain tolerance* at a lower computational cost. Similar to BP, we leverage NSNet to perform approximate model counting by learning the Bethe approximation of the partition function Z. Specifically, besides learning the variable beliefs $b_i(x_i)$ using Equation 8, we learn the factor beliefs $b_a(\boldsymbol{x_a})$ defined in Equation 3 by:

$$\tilde{b}_a(\boldsymbol{x}_a) = \text{MLP}\left(\sum_{j\in\mathcal{N}(a)} m_{j\to a}^{(T)}(x_j)\right), \quad b_a(\boldsymbol{x}_a) = \tilde{b}_a(\boldsymbol{x}_a) - \underset{\boldsymbol{x}_a}{\text{LSE}}\left(\tilde{b}_a(\boldsymbol{x}_a)\right). \tag{10}$$

Then using Equation 4, the log number of all satisfying assignments can be estimated as:

$$\ln Z = -\sum_{a=1}^{m}\sum_{\boldsymbol{x}_a} b_a(\boldsymbol{x}_a)\ln b_a(\boldsymbol{x}_a) + \sum_{i=1}^{n}(|\mathcal{N}(i)|-1)\sum_{x_i} b_i(x_i)\ln b_i(x_i). \tag{11}$$

NSNet subsumes the standard Bethe approximation with learnable estimations of variable and factor beliefs, making it a standalone approximate #SAT solver. Unlike SAT which may have multiple satisfying assignments, the solution to a #SAT problem is a unique value. We collect the ground truth value using a well-engineered #SAT solver and train NSNet by minimizing the mean square error between our approximation and the ground truth.

4 Experiments

We present the evaluation results in this section. In all experiments, we set the feature dimension $d = 64$ and message passing iteration $T = 10$ for training. All MLPs have 3 hidden layers with 64 units each and use ReLU as the activation function. We ran all experiments on a server with a single NVIDIA A100 GPU and 8 CPU cores. See Appendix B.1 for more implementation details.

4.1 SAT Solving

Experiment Setup. Instead of conducting experiments on simple synthetic SAT datasets in prior neural SAT solvers [33, 3, 29], we opt for a more diverse and challenging benchmark suite – SATLIB [15], an open-source SAT dataset containing a broad range of instances collected from various distributions. To train our model effectively, we choose the distributions with at least 100 satisfiable instances, which include the following 5 categories: (1) uniform random 3-SAT on phase transition region (RND3SAT), (2) backbone-minimal random 3-SAT (BMS), (3) random 3-SAT with controlled backbone size (CBS), (4) "Flat" graph coloring (GCP), and (5) "Morphed" graph coloring (SW-GCP). The whole dataset has 46,200 SAT instances, and we split it into training/validation/testing sets following 60%/20%/20% proportions. It should be stressed that the number of variables for SATLIB instances ranges from 100 to 600, which is significantly larger than existing synthetic datasets and makes some of them even hard for complete SAT solvers.

Evaluation & Baselines. We evaluate our approach for SAT solving in two metrics. The first one is the solving accuracy of the initial assignment obtained from the estimated marginals. In this setting, we compare NSNet against BP and the neural baseline NeuroSAT [33]. NeuroSAT is the seminal work that designs a literal-clause graph representation for a CNF formula and encodes the structure using a similar gated graph neural network [24] (GGNN). Such a representation and neural network design are widely used in the following works [20, 8, 32, 44]. We train and evaluate NeuroSAT the same way as NSNet. For both NSNet and NeuroSAT, we also examine the effectiveness of our training method against the supervised training strategy [44, 8, 20], which minimizes the binary

Table 1: Solving accuracy (%) of the initial assignments with the different number of iterations T on the SATLIB benchmark. T_{best} stands for the number of iterations achieving the best accuracy.

Learning	**Method**	**Number of Iterations (T)**						
		5	10	50	100	500	1000	T_{best}
N/A	BP	1.4	6.8	17.3	18.1	19.3	18.8	19.3
Supervised	NeuroSAT	0.4	1.7	0.3	0.4	0.7	0.9	1.7
	NSNet	1.2	4.8	8.4	8.5	9.0	9.4	9.4
Unsupervised	NeuroSAT	0.6	4.6	7.5	7.6	7.8	7.8	7.8
	NSNet	**2.1**	**19.6**	**27.9**	**27.8**	**27.8**	**27.9**	**27.9**

Table 2: Solving accuracy (%) for both neural solvers and SLS solvers on the SATLIB benchmark.

Class	**Method**	**Category**					
		RND3SAT	**BMS**	**CBS**	**GCP**	**SW-GCP**	**Total**
Neural Solver	PDP	52.1	20.8	30.1	6.0	0.0	30.2
	QuerySAT	69.4	37.0	57.2	38.8	29.3	55.8
SLS Solver	Sparrow	76.5 ± 0.5	43.5 ± 2.0	64.2 ± 0.5	50.0 ± 1.9	32.0 ± 0.8	63.6 ± 0.5
	NeuroSAT-Sparrow	78.6 ± 0.9	44.5 ± 2.5	67.7 ± 0.4	50.3 ± 1.2	$\mathbf{32.9 \pm 0.8}$	66.8 ± 0.3
	BP-Sparrow	81.1 ± 0.8	47.7 ± 3.2	71.8 ± 0.5	50.2 ± 0.9	32.7 ± 1.0	70.5 ± 0.4
	NSNet-Sparrow	$\mathbf{81.3 \pm 0.8}$	$\mathbf{49.1 \pm 2.3}$	$\mathbf{72.5 \pm 0.3}$	$\mathbf{50.6 \pm 1.3}$	32.2 ± 0.4	$\mathbf{71.1 \pm 0.3}$

cross-entropy loss of the predicted soft assignment and a certain ground-truth satisfying one. Besides using 10 message passing iterations like training, we further test the convergence of all the methods by running for more iterations of message passing.

The second metric is the accuracy of a SLS solver with different initial assignments. In particular, our modified SLS solver is Sparrow [5], one of the state-of-the-art pure SLS solvers. To evaluate the effectiveness of our method, we compare our hybrid solver against both neural baselines and other SLS solvers. These include two neural SAT solvers (PDP [3] and QuerySAT [29]), the Sparrow solver with three different initialization methods (default initialization, BP, and NeuroSAT). Unlike NeuroSAT, PDP and QuerySAT produce an assignment at each iteration of message passing and thus can generate a set of possible solutions for each SAT instance. Both neural and SLS solvers are allowed to generate up to 1000 assignments for each instance to ensure a fair comparison. For SLS solvers, mean and standard deviations are calculated across 10 runs.

Main Results. Table 1 shows the initial assignment results on the SATLIB benchmark. Generally, NSNet with unsupervised training persistently gets the best results as the iteration T increases. Using our training method, both NeuroSAT and NSNet solve multiple times more instances than that trained with supervision. It is consistent with our hypothesis that fitting on a specific satisfying assignment without considering other possible ones would confuse the neural networks and thus lead to poor generalization. Meanwhile, NSNet's performance exceeds NeuroSAT's by a large gap, which shows the advantage of our graph representation and neural architecture. Similar to BP's results, the solving accuracy of NSNet increases with more than 10 iterations. While the performance of BP hits a saturation after 500 iterations, NSNet only uses 50 iterations to converge and achieves better accuracy at the same time. This shows that NSNet learns a better inference algorithm from data, achieving better performance than BP on the SAT problem.

In Table 2, we present the performance of neural solvers and SLS solvers on the SATLIB dataset. Not surprisingly, both PDP and QuerySAT can not compete with the SLS solvers. We conjecture that this is because PDP and QuerySAT are trained to find a single satisfying assignment and thus generalize poorly during testing. For the modified SLS solvers, we choose the best number of iterations for BP, NeuroSAT, and NSNet based on their initial assignments' accuracy to initialize Sparrow. We can observe that default Sparrow performs the worst, and NeuroSAT initialization improves its performance slightly. The fact that BP significantly outperforms NeuroSAT (as shown in Table 1 and Table 2) suggests that marginals are more helpful in terms of either SAT solving directly or guiding a local search. As a learnable generalization of BP, NSNet outperforms BP and achieves the best result in most categories. More experimental results on the running time and the average number of flips for SLS solvers can be found in Appendix B.2.

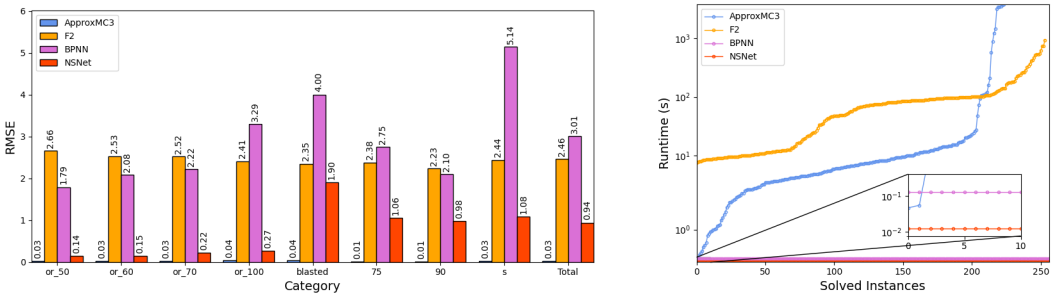

Figure 3: (Left) RMSE between estimated log countings and ground truth for each solver on the BIRD benchmark. (Right) Cactus plots of runtime for each solver on the BIRD benchmark.

4.2 Model Counting

Experiment Setup. We first follow the experiment settings in recent work BPNN [22]. Specifically, we run experiments using the same subset of BIRD benchmark [35], which contains eight categories arising from DQMR networks, grid networks, bit-blasted versions of SMTLIB benchmarks, and ISCAS89 combinatorial circuits. Each category has 20 to 150 CNF formulas, which we split into training/testing with a ratio of 70%/30%. Besides evaluating our proposed method in such a data-limited regime, we also conduct experiments on the SATLIB benchmark with more challenging instances. The splitting ratio on the SATLIB benchmark is the same as the one in SAT solving.

Evaluation & Baselines. Following BPNN [22], we use the (1) root mean square error (RMSE) between the estimated log countings and ground truth and (2) runtime as our evaluation metrics. We compare NSNet with BP, the neural baseline BPNN [22], and two state-of-the-art approximate model counting solvers, ApproxMC3 [35] and F2 [1]. Akin to SAT solving, both BP and neural models are allowed to run with more iterations until the test metric converges. However, BP consistently fails to achieve comparable results on both BIRD and SATLIB benchmarks, so we only leave the other baselines for comparison later. For ApproxMC3 and F2, we set a time limit of 5,000 seconds on each instance. See Appendix B.2 for additional setting details and results of these baselines.

Main Results. As shown in Figure 3 (Left), NSNet can estimate much tighter countings than BPNN and F2 across all categories of the BIRD benchmark. In total instances, NSNet's estimates are nearly three times preciser than F2's and BPNN's, with a RMSE of 0.94. However, NSNet can not compete with ApproxMC3, whose overall RMSE is only 0.03. We also present the running time of each solver on the BIRD benchmark in Figure 3 (Right). While ApproxMC3 can provide nearly perfect estimates, its time efficiency is undesirable: it spends 123.33 seconds on average for 224 solved ones and fails to finish on the other 33 instances within 5,000 seconds. Unlike ApproxMC3 and F2 running sequentially on CPUs, BPNN and NSNet can run all 257 testing instances parallelly in a batch on a GPU. NSNet is the most time-efficient solver than other baselines, which takes only 4.83 seconds for all problems and 0.02 seconds for each. This demonstrates that NSNet significantly outperforms BPNN and F2 in terms of both the estimate accuracy and time efficiency, and gains more than three orders of speedups over ApproxMC3 while significantly reducing the accuracy gap between them. Additionally, the failure of BP proves that NSNet can leverage the inductive bias of neural networks to synthesize a better message passing algorithm rather than approximating BP.

Table 3 summarizes the performance of each solver on the SATLIB benchmark. We train BPNN multiple times but unsuccessfully make it converge on this dataset, so we discard its performance in the table. Compared with F2, NSNet can provide more precise estimates of the countings in much less time. ApproxMC3 still performs the best among all solvers by a large margin. However, it takes 13.05 seconds on average for each instance, while the number for NSNet is less than 0.01, which is more than three orders of magnitude faster than ApproxMC3, showing the efficiency of NSNet. More experimental results on the BIRD and SATLIB datasets can be found in Appendix B.2.

Table 3: RMSE and average runtime for each solver on the SATLIB benchmark.

Method	Metric	
	RMSE	Runtime (s)
ApproxMC3	**0.05**	13.05
F2	2.36	27.79
NSNet	1.71	**< 0.01**

5 Related Work

Neural Satisfiability Solvers. Using neural networks for satisfiability problems has been explored in the last few years. One common idea is to encode the input formulas using a GNN and perform downstream tasks based on the formulas' embeddings. Recent approaches mainly follow this pipeline and can be classified into two folds [14]: standalone neural solvers and neural-guided solvers.

Standalone neural solvers solve a satisfiability task on their own. NeuroSAT [33] and the following works [20, 8] classify CNF formulas as satisfiable or unsatisfiable. Simultaneously, these methods can also construct an assignment by decoding the literal embeddings. Several alternative approaches [2, 3, 29] focus on directly generating satisfying assignments. Such frameworks leverage different GNN architectures and apply an unsupervised loss for training. However, they all attempt to predict a single satisfying solution for each instance, which fails to consider other possible ones.

Although standalone neural solvers show some promising results, neither of these solvers is competitive with the state-of-the-art satisfiability solvers in terms of accuracy or scalability. Hence, another stream of research combines neural networks with modern satisfiability solvers, hoping to improve some components of the existing solvers. These neural-guided solvers include NeuroCore [32] and #Neuro [39], which utilize neural networks to guide the branching decisions of SAT and #SAT solvers. While these works successfully reduce the number of branching steps, frequent invocation of neural networks is also required. By contrast, we call NSNet only once for both SAT and #SAT problems, which significantly diminishes the overhead of querying neural networks. For the SAT problem, NLocalSAT [44] guides the initialization of SLS solvers by the soft assignments of neural networks, which is close to our work. However, they employ the neural architecture similar to NeuroSAT and train it in a supervised fashion, while NSNet is trained to estimate marginals using MLE. In addition, there are also some works [42, 23] leveraging reinforcement learning to learn the local search or branching heuristic of SAT solvers.

Neural Inference Algorithms. While BP serves as a non-trainable inference algorithm, some recent works aim to develop learnable inference models, allowing inference to adapt from data. Some recent efforts explore the idea of enhancing BP with neural networks. In particular, [31] proposes NEBP, a hybrid inference method that augments BP by running a GNN co-jointly. BPNN [22] and FE-NBP [38] focus on improving the updates of factor to variable messages by using a learned operator or a learned damping ratio respectively.

Several other methods devise end-to-end inference models beyond the procedures of BP. [43] is the seminal work to apply a GNN to estimate the marginals on relatively small, binary graphical models. FGNN [45] is another GNN model that parameterizes the Max-Product Belief Propagation and performs MAP (maximum a posteriori) inference on factor graphs. Similar to the idea of generalizing BP, FE-GNN [38] leverages the tensor sum operator to the updates of GNNs, which combines factor potentials with message vectors in a similar fashion to BP. Nevertheless, these models are not designed for the case of satisfiability problems and fail to satisfy the negation equivariance of CNF formulas in Proposition 2 while NSNet identifies this property for satisfiability problems.

6 Discussion

Limitations. There are a few limitations to NSNet. First, NSNet takes the formulation of solving satisfiability problems as probabilistic inference. However, such a formulation is defined for satisfiable instances but is not applicable to unsatisfiable ones. Therefore, certificating unsatisfiability is beyond the scope of our work. Second, while NSNet can provide tight estimates on the #SAT problem, it requires learning from labeled data, where the labels for hard instances can be time-consuming. We consider exploring how to train NSNet without supervision for #SAT as future work.

Conclusion. In this work, we present a general neural framework for solving satisfiability problems as probabilistic inference. As a learnable inference model, our framework leverages a novel GNN architecture that subsumes BP in the latent space and performs marginal inference and partition function estimation to solve SAT and #SAT respectively. Experimental evaluations on SATLIB and BIRD benchmarks demonstrate that our approach significantly outperforms BP and other neural baselines and achieves competitive results compared with the state-of-the-art solvers.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their insightful comments. This work was supported, in part, by Individual Discovery Grants from the Natural Sciences and Engineering Research Council of Canada, and the Canada CIFAR AI Chair Program.

References

[1] Dimitris Achlioptas, Zayd Hammoudeh, and Panos Theodoropoulos. Fast and flexible probabilistic model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 148–164, 2018.

[2] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-sat: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2019.

[3] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. PDP: A general neural framework for learning constraint satisfaction solvers. *arXiv preprint arXiv:1903.01969*, 2019.

[4] Gilles Audemard and Laurent Simon. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

[5] Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In *International Conference on Theory and Applications of Satisfiability Testing*, volume 6175, pages 10–15. Springer, 2010.

[6] Hans A Bethe. Statistical theory of superlattices. *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, 150(871):552–575, 1935.

[7] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

[8] Chris Cameron, Rex Chen, Jason Hartford, and Kevin Leyton-Brown. Predicting propositional satisfiability via end-to-end learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3324–3331, 2020.

[9] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *Lecture Notes in Computer Science*, pages 200–216. Springer, 2013.

[10] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[11] Carmel Domshlak and Jörg Hoffmann. Fast probabilistic planning through weighted model counting. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 243–252, 2006.

[12] Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007.

[13] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[14] Wenxuan Guo, Junchi Yan, Hui-Ling Zhen, Xijun Li, Mingxuan Yuan, and Yaohui Jin. Machine learning methods in solving the boolean satisfiability problem. *arXiv preprint arXiv:2203.04755*, 2022.

[15] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *International Conference on Theory and Applications of Satisfiability Testing*, 2000:283–292, 2000.

[16] Eric I. Hsu and Sheila A. McIlraith. Characterizing propagation methods for boolean satisfiability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 325–338. Springer, 2006.

[17] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[18] Alexander Ivrii, Sharad Malik, Kuldeep S Meel, and Moshe Y Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.

[19] Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1):41–58, 2016.

[20] Sebastian Jaszczur, Michal Luszczyk, and Henryk Michalewski. Neural heuristics for SAT solving. *arXiv preprint arXiv:2005.13406*, 2020.

[21] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Jonathan Kuck, Shuvam Chakraborty, Hao Tang, Rachel Luo, Jiaming Song, Ashish Sabharwal, and Stefano Ermon. Belief propagation neural networks. *Advances in Neural Information Processing Systems*, 33:667–678, 2020.

[23] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Can q-learning with graph networks learn a generalizable branching heuristic for a SAT solver? In *Advances in Neural Information Processing Systems*, pages 9608–9621, 2020.

[24] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *International Conference on Learning Representations*, 2016.

[25] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer, 1999.

[26] João P Marques-Silva and Karem A Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, pages 675–680, 2000.

[27] Marc Mézard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

[28] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-dnnf compilation with sharpsat. In *Canadian Conference on Artificial Intelligence*, pages 356–361, 2012.

[29] Emils Ozolins, Karlis Freivalds, Andis Draguns, Eliza Gaile, Ronalds Zakovskis, and Sergejs Kozlovics. Goal-aware neural SAT solver. *arXiv preprint arXiv:2106.07162*, 2021.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. 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 *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[31] Victor Garcia Satorras and Max Welling. Neural enhanced belief propagation on factor graphs. In *International Conference on Artificial Intelligence and Statistics*, volume 130, pages 685–693. PMLR, 2021.

[32] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 336–353. Springer, 2019.

[33] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[34] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *International Conference on Formal Methods in Computer-Aided Design*, pages 127–144. Springer, 2000.

[35] Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1592–1599, 2019.

[36] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257. Springer, 2009.

[37] Larry J. Stockmeyer. The complexity of approximate counting. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 118–126. ACM, 1983.

[38] Fan-Yun Sun, Jonathan Kuck, Hao Tang, and Stefano Ermon. Equivariant neural network for factor graphs. *arXiv preprint arXiv:2109.14218*, 2021.

[39] Pashootan Vaezipoor, Gil Lederman, Yuhuai Wu, Chris J. Maddison, Roger B. Grosse, Sanjit A. Seshia, and Fahiem Bacchus. Learning branching heuristics for propositional model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 12427–12435, 2021.

[40] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[41] Wei Wei and Bart Selman. A new approach to model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 324–339. Springer, 2005.

[42] Emre Yolcu and Barnabás Póczos. Learning local search heuristics for boolean satisfiability. In *Advances in Neural Information Processing Systems*, pages 7990–8001, 2019.

[43] KiJung Yoon, Renjie Liao, Yuwen Xiong, Lisa Zhang, Ethan Fetaya, Raquel Urtasun, Richard S. Zemel, and Xaq Pitkow. Inference in probabilistic graphical models by graph neural networks. *arXiv preprint arXiv:1803.07710*, 2018.

[44] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. Nlocalsat: Boosting local search with solution prediction. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 1177–1183. ijcai.org, 2020.

[45] Zhen Zhang, Fan Wu, and Wee Sun Lee. Factor graph neural networks. In *Advances in Neural Information Processing Systems*, pages 8577–8587, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative social impacts of your work? [N/A] We find potential negative impacts of our work are generally indirect and hard to assess.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Section 2 and Section 6.
   (b) Did you include complete proofs of all theoretical results? [Yes] See Section 3.2 and Appendix.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] Included in the supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 4 and Appendix.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section 4.1 and Appendix.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 4.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Section 4.1 and Section 4.2.
   (b) Did you mention the license of the assets? [Yes] See Appendix.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] Included in the supplemental material.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Deferred Technical Arguments

A.1 Explanation and Proof of Proposition 2

We first define the permutation invariance and the negation equivariance of CNF formulas concretely. Given a satisfiable formula $(X_1 \vee \neg X_2) \wedge (\neg X_1 \vee X_3)$, the permutation invariance refers to the fact that all the satisfying assignments are not affected by permuting any variables (e.g. swapping X_1 and X_2 throughout the formula), by permuting any clauses (e.g. swapping the first clause with the second clause) or by permuting any literals within a clause (e.g. swapping X_1 and $\neg X_2$ in the first clause). The negation equivariance means that the assignment of a given variable should be negated if we negate every its corresponding literals (e.g. swap X_1 and $\neg X_1$ throughout the formula). Note the permutation invariance and the negation equivariance are important properties of CNF formulas, we also enforce such properties in NSNet.

Proof. Given our graph representation of a CNF formula, permuting any variables or clauses is equivalent to changing the orderings of the corresponding assignment nodes or clause nodes in the original graph representation. However, all the learnable modules are not affected by the different orderings, and our summation aggregation and LSE aggregation for updating assignment to clause embeddings and clause to assignment embeddings are also not subject to the orderings of neighboring embeddings. Thus the embeddings between a variable node and an assignment node remain the same after permutation regardless of any parameterization of $\boldsymbol{h}_{c2a}$, $\boldsymbol{h}_{a2c}$, $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$. On the other hand, permuting any literals within a clause has no effect on our graph construction, so all of the embeddings in NSNet remain unchanged. Similarly, negating every literal corresponding to a given variable X_i is equivalent to swapping two assignment nodes X_i^0 and X_i^1 in our graph representation, while all the edges in the graph remain the same. Therefore, all the edge embeddings connected to X_i^0 are the same as the embeddings connected to X_i^1 after negating and vice versa. □

A.2 Constructing A Satisfying Assignment from Marginals

Besides performing a stochastic local search, there are multiple ways to generate a satisfying assignment from the estimated marginals. One common approach is the decimation algorithm [27] (Algorithm 1), which processes the following steps iteratively: (1) estimate the variable marginals. (2) fix a variable with the highest certainty (whose marginal value is the most extreme) to the value 0 or 1. (3) simplify the given formula. If we can estimate the marginals accurately at each iteration, this process would act as an oracle search without backtracking to construct a satisfying assignment. However, if we integrate NSNet with the decimation algorithm, each iteration of the decimation process needs to query the neural networks on a new simplified formula, which is computationally demanding and impractical for large instances. In addition, one can combine NSNet with backtracking-based solvers by using the estimated marginals to guide the branching heuristic in these solvers. Similarly, frequent invocation of the neural networks is also required, making such a combination inefficient. To reduce the overhead of querying neural networks, we call NSNet only once to estimate marginals and perform a local search to find a satisfying assignment.

Algorithm 1 The decimation algorithm

Input: A satisfiable formula Φ with n variables
$\Phi_0 \leftarrow \Phi$
for $t \leftarrow 1$ to n **do**
    Estimate marginals $b_i(1), b_i(0)$ for each variable X_i of the formula Φ_{t-1}
    Find the variable X_j with the highest value $|b_j(1) - b_j(0)|$
    if $b_j(1) > b_j(0)$ **then**
        $x_j \leftarrow 1$
    else
        $x_j \leftarrow 0$
    end if
    Obtain a new formula Φ_t from Φ_{t-1} by substituting x_j for variable X_j and simplifying
end for
return The assignment $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$

B Additional Experimental Details

B.1 Implementation Details

For training, we use the Adam optimizer [21] with a learning rate of 1e-4 and a weight decay of 1e-10 and clip the gradient with a global norm of 0.65. We train all the neural networks with a batch size of 64 for 1000 epochs on the BIRD benchmark and 400 epochs on the SATLIB benchmark. For experiments on the SATLIB benchmark, we select the best checkpoint for each model based on its performance on the validation set. We run QuerySAT [1], PDP [2], and BPNN [3] using the official code, and implement NeuroSAT and NSNet using PyTorch [30] and PyTorch Geometric [13].

B.2 SAT Solving

B.2.1 Labels

We ran Glucose4[4] [4] with a time limit of 5,000 seconds to generate a satisfying assignment.

B.2.2 Baselines

Sparrow provides two simple initialization methods: zero initialization and random initialization. We try both initialization strategies and find that random initialization is slightly better. So we only report the results of Sparrow with the random initialization as default.

B.2.3 Results

Table 4 demonstrates the overall runtime for neural solvers and SLS solvers on the SATLIB benchmark. Unlike SLS solvers running the instances sequentially on a CPU, PDP and QuerySAT can process multiple instances parallelly on a GPU. However, these two neural solvers are more than ten times slower than SLS solvers. Note PDP and QuerySAT generate a possible assignment and check whether it can satisfy the formula after each iteration of message passing, such generation processes are computationally expensive. Instead, we use neural networks with fewer message passing iterations and combine a SLS solver to generate assignments. Such a combination is time efficient and causes limited overhead: the time of querying NSNet is 43.41 seconds and Sparrow takes 245.44 seconds for all 9,260 testing instances.

Table 4: Overall runtime for each solver on the SATLIB benchmark.

Method	**Runtime (s)**
PDP	6195.46
QuerySAT	4778.70
Sparrow	245.44
NeuroSAT-Sparrow	361.88
BP-Sparrow	326.18
NSNet-Sparrow	288.85

We also test the performance of the SLS solvers by reporting their average number of flips. As shown in Table 5, all the modified Sparrow solvers outperform Sparrow with default initialization. NSNet-Sparrow can not only solve more instances than other SLS solvers but also generate a satisfying assignment with the least local search steps.

Table 5: Average number of flips for SLS solvers on the SATLIB benchmark.

Method	**Category**					
	RND3SAT	**BMS**	**CBS**	**GCP**	**SW-GCP**	**Total**
Sparrow	198.2 ± 6.9	403.7 ± 26.4	400.3 ± 3.0	432.9 ± 13.9	371.0 ± 17.5	381.4 ± 2.9
NeuroSAT-Sparrow	143.8 ± 8.1	288.9 ± 26.4	284.6 ± 3.4	431.5 ± 15.7	297.0 ± 9.6	275.4 ± 2.7
BP-Sparrow	104.5 ± 6.2	226.6 ± 26.5	200.8 ± 3.2	442.2 ± 15.0	297.1 ± 22.0	199.4 ± 2.7
NSNet-Sparrow	$\mathbf{95.9 \pm 3.4}$	$\mathbf{194.1 \pm 17.9}$	$\mathbf{192.8 \pm 4.4}$	$\mathbf{424.4 \pm 10.9}$	$\mathbf{291.2 \pm 13.6}$	$\mathbf{191.0 \pm 3.9}$

[1] `https://github.com/LUMII-Syslab/QuerySAT`. MIT License.

[2] `https://github.com/microsoft/PDP-Solver`. MIT License.

[3] `https://github.com/jkuck/BPNN`.

[4] `https://www.labri.fr/perso/lsimon/glucose/`.

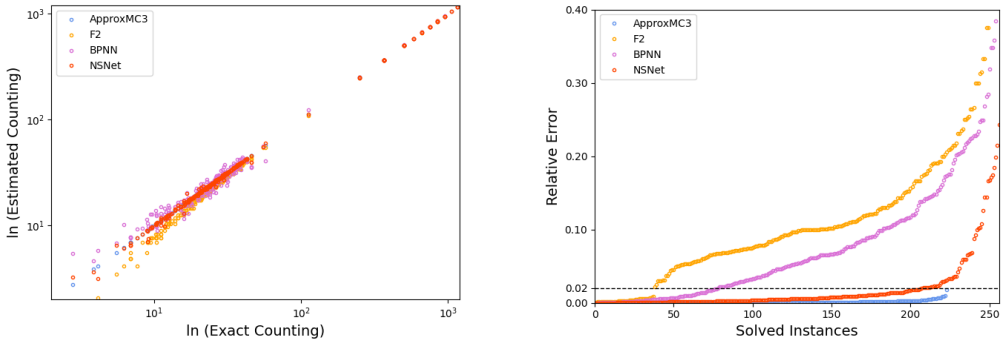

Figure 4: (Left) Scatter plot comparing the estimated log countings against the ground truth for each solver on the BIRD benchmark. (Right) Relative error between the estimated log countings and the ground truth log countings for each solver on the BIRD benchmark.

B.3 Model Counting

B.3.1 Labels

For both BIRD and SATLIB benchmarks, we ran the state-of-the-art exact #SAT solver DSharp[5] [28] with a time limit of 5,000 seconds to generate the ground truth labels. We discard those instances where DSharp fails to finish in 5,000 seconds.

B.3.2 Baselines

To ensure a fair comparison, we train BPNN using the message passing iteration of 10 rather than 5 in its original paper, which also slightly improves its performance on the BIRD benchmark. Note that ApproxMC3 provides probably approximately correct (PAC) guarantee on the estimated model count with two parameters: the tolerance ϵ and the confidence 1-δ, we first run ApproxMC3 in the default settings (ϵ=0.8, δ=0.2) and further conduct experiments with different settings. Following the evaluation of BPNN, we run F2 with the default parameters and choose the low bound mode. In addition, for these two solvers, finding *minimal independent support* (MIS) [18] is always used as a preprocessing step to boost their computations. So we also run the MIS tool [19] within 1k seconds as the preprocessing. We record two times for each instance: one is the sum of the MIS tool's runtime and the time of the #SAT solvers with the MIS support; the other is the running time of the #SAT solvers without the MIS. The minimum of these two times is reported. For BP, we try to perform message passing with $T = 5, 10, 50, 100, 500, 1000$ iterations, but only achieve the best overall RMSE on the BIRD benchmark and SATLIB benchmark of 20.42 and 17.67 respectively, which is not comparable with other baselines.

B.3.3 Results

Figure 4 (Left) shows the scatter plot comparing the estimated log countings against the ground truth for each solver on the BIRD benchmark. We can observe that both ApproxMC3 and NSNet can provide tighter estimates than both F2 and BPNN on most instances when the ground truth is less than e^{100}. While ApproxMC3 fails to finish in 5,000 seconds when the ground truth counting is more than e^{100}, NSNet can still give tight approximations when the ground truth counting is even more than $e^{1,000}$. This demonstrates the effectiveness of NSNet to solve hard and large instances. We further report the relative error between the estimated log countings and the ground truth log countings in Figure 4 (Right). On average, NSNet's relative error is less than 2%, which is significantly better than F2's and BPNN's. Note that NSNet only spends 0.02 seconds for each instance, such relative error is also acceptable in many applications.

Table 6 shows the RMSE results of each solver on the SATLIB benchmark. Compared with its performance on the BIRD benchmark, the precision of NSNet decreases by a large margin. We conjecture this is because the data of the BIRD benchmark is collected from many real-world model

[5] `https://github.com/QuMuLab/dsharp`. MIT license.

counting applications, which may share a lot of common logical structures to learn. On the other hand, the instance in the SATLIB benchmark is generated randomly, making NSNet hard to exploit common features. Nevertheless, NSNet still outperforms F2 in most categories.

Table 6: RMSE between estimated log countings and ground truth for each solver on the BIRD benchmark.

Method	**Category**					
	RND3SAT	**BMS**	**CBS**	**GCP**	**SW-GCP**	**Total**
ApproxMC3	0.04	0.05	0.05	0.06	0.05	0.05
F2	2.13	2.42	2.37	2.40	2.66	2.36
NSNet	1.57	2.45	1.68	2.14	1.37	1.71

Since ApproxMC3 can be configured to achieve different trade-offs between speed and accuracy, we also test it with different settings. Table 7 shows the performance of ApproxMC3 with different ϵ and δ. Although we significantly relax the theoretical PAC guarantee on the estimated model count to improve the speed of ApproxMC3, ApproxMC3 can still give quite tight estimates while spending orders of magnitudes time than NSNet in practice. Additionally, ApproxMC3 timeouts on more than 30 instances in 5,000 seconds while NSNet solves all the instances. We believe the overhead of ApproxMC3 is still significant with much loose bound because it needs to frequently call the CryptoMiniSat [36] to reason about subformulas of the original CNF formula. Instead, NSNet only performs message passing to provide an estimation, which is much more efficient. To trade the slight inaccuracy with significant speedup, NSNet can serve as a more feasible choice.

Table 7: RMSE between estimated log countings and ground truth for ApproxMC3 with different parameters on the BIRD benchmark.

Parameter		**Metric**		
ϵ	δ	RMSE	Avg. runtime (s)	#Failed
0.8	0.2	0.03	123.32	33
0.8	0.8	0.13	39.07	33
4	0.2	0.07	63.95	33
4	0.8	0.21	23.69	34
10	0.99	0.23	22.45	35