Belief Propagation Neural Networks

Abstract

We introduce belief propagation neural networks (BPNNs), a new neural architecture that operates directly on factor graphs. BPNNs draw inspiration from two lines of research: variational methods used to approximate #P-complete integration problems, and solving simpler NP-complete problems by learning from labeled examples. Without training, BPNNs perform exact loopy belief propagation. During training, they learn to improve upon the standard belief propagation updates in a data-driven manner. Our evaluation considers two integration tasks, computing the partition function of Ising models and model counting. Surprisingly, BPNNs are able to effectively learn from only 10's of training examples. On the Ising model task, BPNNs frequently provide 10-100 times more accurate estimates than standard graph neural networks and loopy belief propagation. For approximate model counting, while state-of-the-art methods often time out, BPNNs provide estimates of comparable quality within a second.

1 INTRODUCTION

#P-complete integration problems arise in many domains, from statistical physics to machine learning. There is little hope that efficient, exact solutions to these problems exist as they are at least as hard as NP-complete decision problems. Significant research has been devoted across the fields of machine learning, statistics, and statistical physics to develope variational and sampling based methods to approximate these challenging problems (Chandler, 1987; Mézard et al., 2002; Wainwright et al., 2008; Baxter, 2016; Owen, 2013).

Loopy belief propagation and the mean field approximation fall into the category of variational methods. These algorithms reduce inference to an optimization problem, but compute an estimate with weak (or without) guarantees on its accuracy. Markov chain Monte Carlo is the gold standard of sampling based methods, famously delivering a fully polynomial-time randomized approximation scheme (FPRAS) (Jerrum et al., 2004) to estimate the permanent of a matrix, a problem known to be #P-complete (Valiant, 1979). However these methods suffer from high computational complexity. The FPRAS for estimating the matrix permanent scales as $O(n^7 \log^4 n)$, with respect to matrix dimensionality, with large coefficients thus making it practically unusable. Randomized hashing methods are another, relatively recent, technique for performing approximate integration (Gomes et al., 2006; Chakraborty et al., 2016; Ermon et al., 2014). While very successful in some domains, they repeatedly solve NP-complete problems during execution making them prohibitively slow for most applications.

Existing approximate inference techniques rely on handcrafted heuristics, which may or may not perform well (in terms of runtime and accuracy) on different classes of input instances. Inspired from recent work that has successfully used neural networks to *learn* how to solve NP-complete decision problems (Selsam et al., 2018; Prates et al., 2019), we introduce belief propagation neural networks (BPNNs). These networks define a parameterized computation graph that subsumes the computation performed by traditional belief propagation on any given graphical model. In particular, BPNNs can perform exact belief propagation for a certain choice of the learnable parameters, but can improve on it through training. The recent performance gains of neural networks have been fueled by increasingly massive training datasets and parallel computation power. Our BPNN violates both of these paradigms. BPNNs outperform both modern neural networks with GPU compute power and classical algorithms when trained on 10's of examples using only

CPU computation. We perform experiments where we estimate the partition function of an Ising model and approximate the number of satisfying solutions to a boolean formula (model counting). On the Ising model task our BPNN frequently estimates the partition function with accuracy that is 10-100 times better than a maximally powerful graph neural network, loopy belief propagation, and the mean field approximation. On the model counting task our BPNN requires orders of magnitude less computation than state-of-the-art randomized hashing methods, while returning estimates with comparable quality.

2 BACKGROUND

We provide background on belief propagation and graph neural networks (GNN) to motivate and clarify belief propagation neural networks (BPNN).

2.1 BELIEF PROPAGATION

We describe a general version of belief propagation (Yedidia et al., 2005) that operates on factor graphs.

Factor Graphs. A factor graph (Kschischang et al., 2001; Yedidia et al., 2005) is a general representation of a distribution over n discrete random variables, $\{X_1, X_2, \ldots, X_n\}$. Let x_i denote a possible state of the i^{th} variable. We use the shorthand $p(\mathbf{x}) = p(X_1 = x_1, \ldots, X_n = x_1)$ for the joint probability mass function, where $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ is a specific realization of all n variables. Without loss of generality, $p(\mathbf{x})$ can be written as the product

$$p(\mathbf{x}) = \frac{1}{Z} \prod_{a=1}^{M} f_a(\mathbf{x}_a). \tag{1}$$

The functions $f_1, f_2, \ldots, f_m$ each take some subset of variables as arguments; function f_a takes $\mathbf{x_a} \subset \{x_1, x_2, \ldots, x_n\}$. We require that all functions are non-negative and finite. This makes $p(\mathbf{x})$ a well defined probability distribution after normalizing by the distribution's partition function

$$Z = \sum_{\mathbf{x}} \left(\prod_{a=1}^{M} f_a(\mathbf{x}_a) \right). \tag{2}$$

A factor graph is a bipartite graph that expresses the factorization of the distribution in equation 1. A factor graph's nodes represent the n variables and m functions present in equation 1. The nodes corresponding to functions are referred to as factor nodes. Edges exist between factor nodes and variables nodes if and only if the variable is an argument to the corresponding function.

Message Updates. Belief propagation performs iterative message pass. The message $n^{(k)}_{i\to a}(x_i)$ from variable node i to factor node a during iteration k is computed according to the rule

$$n^{(k)}_{i\to a} k(x_i) \coloneqq \prod_{c \in \mathcal{N}(i)\setminus a} m^{(k-1)}_{c\to i}(x_i). \tag{3}$$

The message $m^{(k)}_{a\to i}(x_i)$ from factor node a to variable node i during iteration k is then computed according to the rule

$$m^{(k)}_{a\to i}(x_i) \coloneqq \sum_{\mathbf{x}_a \setminus x_i} f_a(\mathbf{x}_a) \prod_{j \in \mathcal{N}(a)\setminus i} n^{(k)}_{j\to a}(x_j). \tag{4}$$

The BP algorithm estimates approximate marginal probabilities of each variable, referred to as beliefs. We denote the belief at variable node i, after message passing iteration k is complete, as $b^{(k)}_i(x_i)$ which is computed as

$$b^{(k)}_i(x_i) = \frac{1}{z_i} \prod_{a \in \mathcal{N}(i)} m^{(k)}_{a\to i}(x_i), \tag{5}$$

where z_i is the normalizing term

$$z_i = \sum_{x_i} \prod_{a \in \mathcal{N}(i)} m^{(k)}_{a\to i}(x_i). \tag{6}$$

Similarly, BP computes joint beliefs over the sets of variables $\mathbf{x}_a$ associated with each factor f_a. We denote the belief over variables $\mathbf{x}_a$, after message passing iteration k is complete, as $b^{(k)}_a(\mathbf{x}_a)$ which is computed as

$$b^{(k)}_a(\mathbf{x}_a) = \frac{f_a(\mathbf{x}_a)}{z_a} \prod_{i \in \mathcal{N}(a)} n^{(k)}_{i\to a}(x_i), \tag{7}$$

where z_a is the normalizing term

$$z_a = \sum_{\mathbf{x}_a} f_a(\mathbf{x}_a) \prod_{i \in \mathcal{N}(a)} n^{(k)}_{i\to a}(x_i). \tag{8}$$

Partition Function Approximation. The belief propagation algorithm proceeds by iteratively updating variable to factor messages (Equation 3) and factor to variable messages (Equation 4) until they converge to fixed values or a predefined maximum number of iterations is reached. At this point the beliefs are used to compute a variational approximation of the factor graph's partition function. This approximation, originally developed in statistical physics, is known as the Bethe free energy $F_{\text{Bethe}} \approx -\ln Z$ (Bethe, 1935). It is defined in terms of the Bethe average energy U_{Bethe} and the Bethe entropy H_{Bethe}.

Definition 1. *The Bethe average energy is*

$$U_{Bethe} \coloneqq -\sum_{a=1}^{M}\sum_{\mathbf{x}_a} b_a(\mathbf{x}_a)\ln f_a(\mathbf{x}_a). \tag{9}$$

Definition 2. *The Bethe entropy is*

$$\begin{aligned} H_{Bethe} \coloneqq &-\sum_{a=1}^{M}\sum_{\mathbf{x}_a} b_a(\mathbf{x}_a)\ln b_a(\mathbf{x}_a) \\ &+\sum_{i=1}^{N}(d_i-1)\sum_{x_i} b_i(x_i)\ln b_i(x_i), \end{aligned} \tag{10}$$

where d_i is the degree of variable node i.

Definition 3. *The Bethe free energy is defined as $F_{Bethe} = U_{Bethe} - H_{Bethe}$.*

Numerically Stable Belief Propagation. Standard belief propagation is generally performed in log-space for numerical stability. The message $n^{(k)}_{i\to a}(x_i)$ from variable node i to factor node a during iteration k given in Equation 3 becomes

$$n^{(k)'}_{i\to a}k(x_i) = \sum_{c\in\mathcal{N}(i)\setminus a} m^{(k-1)'}_{c\to i}(x_i). \tag{11}$$

The message $m^{(k)}_{a\to i}(x_i)$ from factor node a to variable node i during iteration k given in Equation 4 becomes

$$m^{(k)'}_{a\to i}(x_i) = \underset{\mathbf{x}_a\setminus x_i}{\text{LSE}}\left(\phi_a(\mathbf{x}_a) + \sum_{j\in\mathcal{N}(a)\setminus i} n^{(k)'}_{j\to a}(x_j)\right), \tag{12}$$

where we use the shorthand

$$\underset{i}{\text{LSE}}\, w_i = \ln\left(\sum_i \exp(w_i)\right), \tag{13}$$

and $\phi_a(\mathbf{x}_a) = \ln\left(f_a\left(\mathbf{x}_a\right)\right)$ denotes the log factors. Plugging the definition of $n^{(k)'}_{i\to a}k(x_i)$ from Equation 11 into Equation 12 gives a rule for computing factor to variable messages at iteration k in terms of factor to variable messages from iteration $k-1$.

2.2 GRAPH NEURAL NETWORKS

We switch topics in this section to provide background on graph neural networks (GNNs), a form of neural network that operates directly on graphs. They have seen recent success in representation learning on graph structured data. GNNs perform iterative message passing operations between neighboring nodes in graphs, updating the learned, hidden representation of each node after every iteration. The computational similarities between GNNs and belief propagation have been noted (Yoon et al., 2018). We will revisit these similarities and also less frequently noted differences in the next section. Xu et al. (2018) showed that graph neural networks are at most as powerful as the Weisfeiler-Lehman graph isomorphism test (Weisfeiler & Lehman, 1968), which is a strong test that generally works well for discriminating between graphs. Additionally, Xu et al. (2018) presented a GNN architecture called the Graph Isomorphism Network (GIN), which they showed has discriminative power equal to that of the Weisfeiler-Lehman test and thus strong representational power. We will use GIN as a baseline GNN for comparison in our experiments because it is provably as discriminative as any GNN.

We now describe in detail the GIN architecture that we use. Our architecture performs regression on graphs, learning a function $f_{\text{GIN}} : \mathcal{G} \to \mathbb{R}$ from graphs to a real number. Our input is a graph $G = (V, E) \in \mathcal{G}$ with node feature vectors $\mathbf{h}^{(0)}_v$ for $v \in V$ and edge feature vectors $\mathbf{e}_{u,v}$ for $(u, v) \in E$. Our output is the number $f_{\text{GIN}}(G)$, which should ideally be close to the ground truth value y_G. Let $\mathbf{h}^{(k)}_v$ denote the representation vector corresponding to node v after the k^{th} message passing operation. We use a slightly modified GIN update to account for edge features as follows:

$$\mathbf{h}^{(k)}_v = \text{MLP}^{(k)}_1\left(\mathbf{h}^{(k-1)}_v + \sum_{u\in\mathcal{N}(v)} \text{MLP}^{(k)}_2\left(\mathbf{h}^{(k-1)}_u, \mathbf{e}_{u,v}\right)\right). \tag{14}$$

A K-layer GIN network with width M is defined by K successive GIN updates as given by Equation 14, where $\mathbf{h}^{(k)}_v \in \mathbb{R}^M$ is an M-dimensional feature vector for $k \in \{1, 2, \ldots, K\}$. All MLPs within GIN updates (except $\text{MLP}^{(0)}_2$) are multilayer perceptrons with a single hidden layer whose input, hidden, and output layers all have dimensionality M. $\text{MLP}^{(0)}_2$ is different in that its input dimensionality is given by the dimensionality of the original node feature representations. The final output of our GIN network is given by

$$f_{\text{GIN}}(G) = \text{MLP}^{(K+1)}\left(\underset{k=1}{\overset{K}{\text{CONCAT}}} \sum_{v\in G} \mathbf{h}^k_v\right), \tag{15}$$

where we concatenate summed node feature vectors from all layers and $\text{MLP}^{(K+1)}$ is a multilayer perceptron with a single hidden layer. Its input and hidden layers have dimensionality $M \cdot K$ and its output layer has dimensionality 1.

3 BELIEF PROPAGATION NEURAL NETWORK (BPNN)

We now present the architecture of our belief propagation neural network (BPNN). It is analogous to general GNN architectures, but two points must be emphasized. First, *BPNN can perform computations that standard GNNs are not capable of.* Second, *BPNN subsumes belief propagation* as a strict generalization.

BPNN Iterative Layers. Our BPNN architecture adds two multilayer perceptrons to the standard belief propagation updates. BPNNs generalize Equation 12 and compute factor to variable messages as

$$m_{a\to i}^{(k)'}(x_i) = \underset{\mathbf{x}_a \setminus x_i}{\text{LSE}}\left(\phi_a(\mathbf{x}_a) + \text{LNE}_2\Big[\sum_{j\in\mathcal{N}(a)\setminus i} \text{LNE}_1\Big(n_{j\to a}^{(k)'}(x_j)\Big)\Big]\right), \tag{16}$$

where we use the shorthand,

$$\text{LNE}(\mathbf{h}) = \ln\Big(\text{MLP}_\theta\big(\exp(\mathbf{h})\big)\Big), \tag{17}$$

and MLP_θ is a multilayer perceptron parameterized by θ. We exponentiate before applying the multilayer perceptron because we empirically find that this improves training. Variable to factor messages are unchanged from Equation 11.

Damped BPNN Iterative Layers. We add a residual connection to message updates, which serves the same purpose as damping in standard belief propagation. With this modification, factor to variable messages become

$$m_{a\to i}^{(k)'}(x_i) = \alpha_0 m_{a\to i}^{(k-1)'}(x_i) + (1-\alpha_0) \underset{\mathbf{x}_a \setminus x_i}{\text{LSE}}\left(\phi_a(\mathbf{x}_a) + \text{LNE}_1\Big[\sum_{j\in\mathcal{N}(a)\setminus i} \text{LNE}_2\Big(n_{j\to a}^{(k)'}(x_j)\Big)\Big]\right). \tag{18}$$

Finally, we add a residual connection directly around both MLPs. With this modification, factor to variable messages become

$$m_{a\to i}^{(k)'}(x_i) = \alpha_0 m_{a\to i}^{(k-1)'}(x_i) + (1-\alpha_0)\underset{\mathbf{x}_a \setminus x_i}{\text{LSE}}\left(\phi_a(\mathbf{x}_a) + \text{LNE}_1^{\alpha_1}\Big[\sum_{j\in\mathcal{N}(a)\setminus i} \text{LNE}_2^{\alpha_2}\Big(n_{j\to a}^{(k)'}(x_j)\Big)\Big]\right), \tag{19}$$

where we use the shorthand

$$\text{LNE}^{\alpha}(\mathbf{h}) = \alpha\mathbf{h} + (1-\alpha)\ln\Big(\text{MLP}_\theta\big(\exp(\mathbf{h})\big)\Big). \tag{20}$$

Weight Tying in BPNN Iterative Layers. Weights can be tied between iterative layers in a BPNN, such that $\theta_1 = \theta_2 = \cdots = \theta_K$ for a K-layer BPNN. In this setting the single iterative layer can either be applied for a fixed number of iterations or until messages converge as in standard belief propagation. In the second setting BPNNs become learned, iterative fixed point solvers.

Bethe Free Energy Layer. A K-layer BPNN applies K iterative layers followed by a final Bethe free energy layer. This final layer is an MLP that takes a concatenation of Bethe average energy and entropy terms across all iterations, summed across factors within iterations but not factor states (which is analogous to the summation and concatenation of node feature vectors in a standard GNN). It outputs a scalar as follows:

$$f_{\text{BPNN}}(G_{\text{factor}}) = \text{MLP}_\theta^{(K+1)}\Bigg[\underset{k=1}{\overset{K}{\text{CONCAT}}}\Bigg(\text{CONCAT}\Bigg(\sum_{a=1}^{M} b_a^{(k)}(\mathbf{x}_a)\ln f_a(\mathbf{x}_a),\; -\sum_{a=1}^{M} b_a^{(k)}(\mathbf{x}_a)\ln b_a^{(k)}(\mathbf{x}_a),\; \sum_{i=1}^{N}(d_i-1)b_i^{(k)}(x_i)\ln b_i^{(k)}(x_i)\Bigg)\Bigg)\Bigg]. \tag{21}$$

3.1 PROPERTIES OF BPNN

Proposition 1. *Belief propagation neural networks subsume belief propagation as a strict generalization.*

Please see the appendix for a proof of proposition 1.

The following theorem formalizes a particular instance of computation that BPNNs are capable of that GNNs are not. Furthermore, GNNs aggregate messages between neighboring nodes in a permutation invariant manner. BPNNs achieve this behavior with pairwise factors. However, BPNNs that operate on factors over more than 2 variables leverage more complicated variable dependencies that are discarded by permutation invariant GNNs.

Theorem 1. *Belief propagation neural networks with weight tying between layers converge within l iterations on tree structured factor graphs with height l.*

Proof. If we consider a BPNN with weight tying, then regardless of the number of iterations or layers, the output messages are the same if the input messages are the same. Without loss of generality, let us first consider any

node r as the root node, and consider all the messages on the path from the leaf nodes through r. Let $d_{r,i}$ denote the depth of the sub-tree with root i when we consider r as the root (e.g. for a leaf node i, $d_{r,i} = 1$). We use the following induction argument:

- At iteration 1, the message from all nodes with $d_{r,i} = 1$ to their parents will be fixed for subsequent iterations since the inputs to the BPNN for these messages are the same.
- If at iteration $t-1$, the message from all nodes with $d_{r,i} \leq t-1$ to their parents are fixed for all subsequent iterations, then the inputs to the BPNN for all the messages from all nodes with $d_{r,i} = t$ to their parents will be fixed (since they depend on lower level messages that are fixed). Therefore, at iteration t, the messages from all the nodes with $d_{r,i} \leq t$ to their parents will be fixed because of weight tying between BPNN layers.
- The maximum tree depth is l, so $\max_i d_{r,i} \leq l$. From the induction argument above, after at most l iterations, all the messages along the path from leaf nodes to r will be fixed.

Since the BPNN layer performs the operation over all nodes, this above argument is valid for all nodes when we consider them as root nodes. Therefore, all messages will be fixed after at most l iterations, which completes the proof. □

4 EXPERIMENTS

4.1 ISING MODEL

We follow a common experimental setup used to evaluate approximate integration methods (Hazan & Jaakkola, 2012; Ermon et al., 2013). We randomly generated grid structured Ising models whose partition functions can be computed exactly using the junction tree algorithm (Lauritzen & Spiegelhalter, 1988). We used our belief propagation network to estimate the partition function of a set of random models and compared with loopy belief propagation (Yedidia et al., 2005; Murphy et al., 1999), the Graph Isomorphism Network GNN, and the mean field approximation (Wainwright et al., 2008).

Data Generation. An $N \times N$ Ising model is defined over binary variables $x_i \in \{-1, 1\}$ for $i = 1, 2, \ldots, N^2$, where each variable represents a spin. Each spin has a local field parameter J_i which corresponds to its local potential function $J_i(x_i) = J_i x_i$. Each spin variable has 4 neighbors, unless it occupies a grid edge. Neighboring spins interact with coupling potentials $J_{i,j}(x_i, x_j) = J_{i,j} x_i x_j$. The probability of a complete variable configuration $\mathbf{x} = \{x_1, \ldots, x_{N^2}\}$ is defined to be

$$p(\mathbf{x}) = \frac{1}{Z} \exp\left(\sum_{i \in V} J_i x_i + \sum_{(i,j) \in E} J_{i,j} x_i x_j \right), \quad (22)$$

where the normalization constant Z, or partition function, is defined to be

$$Z = \sum_{\mathbf{x}} \exp\left(\sum_{i \in V} J_i x_i + \sum_{(i,j) \in E} J_{i,j} x_i x_j \right). \quad (23)$$

We performed experiments using datasets of randomly generated Ising models. Each dataset was created by first choosing N, c_{max}, and f_{max}. We sampled $N \times N$ Ising models according to the following process

$$\begin{aligned} c &\sim \text{Unif}[0, c_{\text{max}}), \\ f &\sim \text{Unif}[0, f_{\text{max}}), \\ (J_i)_{i \in V} &\overset{\text{i.i.d.}}{\sim} \text{Unif}[-f, f), \\ (J_{i,j})_{(i,j) \in E} &\overset{\text{i.i.d.}}{\sim} \text{Unif}[0, c). \end{aligned}$$

Training Protocol. We trained a 10 layer BPNN to predict the natural logarithm of an Ising model's partition function (Z). We simplified the model by freezing the final MLP to always output the Bethe free energy computed from the final layer beliefs. We set the residual parameters to $\alpha_0 = \alpha_1 = \alpha_2 = .5$, and trained on 50 attractive Ising models generated with $N = 10$, $f_{\text{max}} = .1$, and $c_{\text{max}} = 5$. We used mean squared error as our training loss. We used the Adam optimizer (Kingma & Ba, 2015) with an initial learning rate of .0005 and trained for 100 epochs, with a decay of .5 after 50 epochs. Batching was over the entire training set (of size 50) with one optimization step per epoch.

Baselines. We trained a 10 layer GNN with width 4 on the same dataset of attractive Ising models that we used for our BPNN. We set edge features to the coupling potentials; that is, $\mathbf{e}_{u,v} = J_{u,v}$. We set the initial node representations to the local field potentials of each node, $\mathbf{h}_v^{(0)} = J_v$. We used the same training loss and optimizer as for our BPNN. We used an initial learning rate of 0.001 and trained for 5k epochs, decaying the learning rate by .5 every 2k epochs.

We consider two additional baselines: Bethe approximation from running standard loopy belief propagation and mean field approximation. We used the libDAI (Mooij, 2010) implementation for both. We ran loopy belief

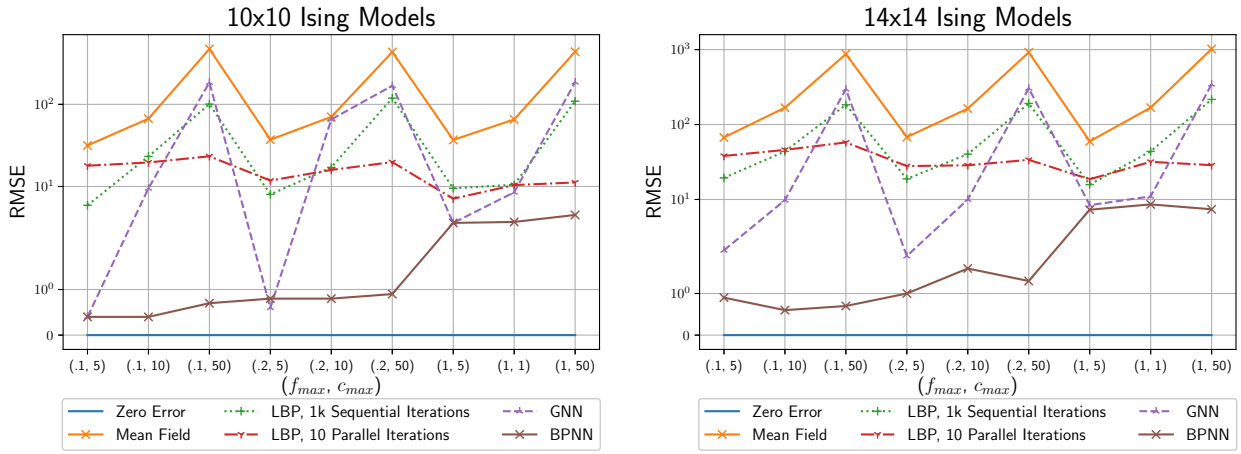

Figure 1: Each point represents the root mean squared error (RMSE, y-axis) of the specified method on a test set of 50 Ising models sampled with the parameters $f_{\max}$ and $c_{\max}$ (x-axis). The leftmost point shows results for test data drawn from the same distribution as training. BPNN significantly improves upon loopy belief propagation (LBP) for both in and out of distribution data. BPNN also significantly outperforms GNN on out of distribution data and larger models.

propagation with a variety of configurations and show results for two settings that have very different convergence properties: (1) run for a maximum of 10 iterations with parallel updates and damping set to .5, and (2) run for a maximum of 1000 iterations with sequential updates using a random sequence and no damping.

Out of Distribution Generalization. We tested all methods on data sampled from distributions that differed from the training distribution. We sampled test data from distributions with c_{max} and f_{max} increased by factors of 2 and 10 from their training values, with N set to 14 (for 196 variables instead of the 100 seen during training). Full test results are shown in figure 1. The leftmost point in the left figure shows results for test data that was drawn from the same distribution used for training the BPNN and GNN. The BPNN and GNN perform similarly for data drawn from the same distribution seen during training. However, our BPNN significantly outperforms the GNN when the test distribution differs from the training distribution and when generalizing to the larger models. Our BPNN also significantly outperforms loopy belief propagation, both for test data drawn from the training distribution and for out of distribution data.

4.2 MODEL COUNTING

In this section we use our BPNN to estimate the number of satisfy solutions to a boolean formula. This is a classic #P-complete problem, commonly known as approximate model counting. We consider the general case where input formulas over n boolean variables, $\{X_1, X_2, \ldots, X_n\}$, are in conjunctive normal form (CNF). Formulas in CNF form are a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation.

Dataset. We evaluated the performance of our BPNN using the suite of benchmarks from Soos & Meel (2019). Some of these benchmarks come with a sampling set. The sampling set redefines the model counting problem, asking how many configurations of variables in the sampling set correspond to at least one complete variable configuration that satisfies the formula. (A formula with n variables may have at most 2^n satisfying solutions, but a sampling set over i variables will restrict the number of solutions to at most 2^i). We stripped all problems of sampling sets since they are outside the scope of this work. We ran the exact model counter DSharp (Muise et al., 2012) on all benchmarks with a timeout of 5k seconds to obtain ground truth model counts for 928 of the 1,896 benchmarks. Only 50 of these problems had more than 5 variables in the largest factor, so we discarded these problems and set the BPNN architecture to run on factors over 5 variables. We categorized the remaining 878 by their arcane names into groupings. With some sleuthing we determined that categories ‘or_50’, ‘or_60’, ‘or_70’, and ‘or_100’ contain network DQMR problems with 150, 121, 111, and 138 benchmarks per category respectively. Categories ‘75’ and ‘90’ contain network grid problems with 20 and 107 benchmarks per category respectively. Category ‘blasted’ conains bit-blasted versions of SMTLIB (satisfiability modulo theories library) benchmarks (Chakraborty et al., 2016) and has 147 benchmarks. Category ‘s’ contains representations of circuits with a subset of outputs randomly xor-ed and

has 68 benchmarks. We discarded 4 categories that contained fewer than 10 benchmarks. For each category that contained more than 10 benchmarks, we split 70% into the training set and left the remaining benchmarks in the test set. We then performed two splits of the training set for training and validation; for each category we (1) trained on a random sampling of 70% of the training problems and performed validation on the remaining 30% and (2) trained on 70% of the training problems that DSharp solved fastest and performed validation on the remaining 30% that took longest for DSharp to solve. These hard validation sets are significantly more challenging for Dsharp. The median runtime in each category's hard validation set is 4 to 15 times longer than the longest runtime in each corresponding easy training set.

Baseline Approximate Model Counters. For comparison, we ran the state of the art approximate model counter ApproxMC3[1] (Chakraborty et al., 2016; Soos & Meel, 2019) on all benchmarks. ApproxMC3 is a randomized hashing algorithm that returns an estimate of the model count that is guaranteed to be within a multiplicative factor of the exact model count with high probability. Improving the guarantee, either by tightening the multiplicative factor or increasing the confidence, will increase the algorithm's runtime. We ran ApproxMC3 with the default parameters; confidence set to 0.81 and epsilon set to 16.

We also compare with the state of the art randomized hashing algorithm F2[2] from (Achlioptas & Theodoropoulos, 2017; Achlioptas et al., 2018), run with CryptoMiniSat5[3] (Soos et al., 2009; Soos & Meel, 2019). This algorithm gives up the probabilistic guarantee that the returned estimate will be within a multiplicative factor of the true model count in return for significantly increased computational efficiency. We computed only a lower bound and ran F2 with variables appearing in only 3 clauses. This significantly speeds up the reported results (Achlioptas & Theodoropoulos, 2017, p.14), at some additional cost to accuracy. For example, on the problem 'blasted_case37' Achlioptas & Theodoropoulos (2017, p.14) report an estimate of $\log_2(\#\text{models}) \approx 151.02$ and a runtime of 4149.9 seconds. Running F2 with variables appearing in only 3 clauses, we computed the lower bound on $\log_2(\#\text{models})$ of 148 in 2 seconds.

As a preprocessing step, we attempted to find a set of variables that define a *minimal independent support* (MIS) (Ivrii et al., 2016) for each benchmark using the authors' code[4] with a timeout of 1k seconds. A set of variables that define a MIS for a boolean formula fully determine the values of the remaining variables. Randomized hashing algorithms can run significantly faster when given a set of variables that define a MIS. When we could find a set of variables that define a MIS, we recorded the time that each randomized hashing algorithm required without the MIS and the sum of the time to find the MIS and perform randomized hashing with the MIS. We report the minimum of these two times.

We also attempted to train a GNN, using the architecture from Selsam et al. (2018) to perform regression instead of classification. We used the author's code, making slight modifications to perform regression. However, we were not successful in achieving non-trivial learning.

Training Protocol. We trained our BPNN to predict the natural logarithm of the number of satisfying solutions to an input formula in CNF form. The boolean formula was converted into a factor graph where each clause corresponds to a factor. Factors take the value of 1 for variable configurations that satisfy the clause and 0 for variable configurations that do not satisfy the clause. The partition function of this factor graph is equal to the number of satisfying solutions. All experiments were run with the same hyper-parameters. We trained 2, 3, 5, and 10 layer BPNNs (see the appendix for complete results). ReLUs in the Bethe free energy layer where shifted, $\text{ReLU}'(x) = \text{shift} + f(x - \text{shift})$, with the shift set to -500. We set the residual parameters to $\alpha_0 = \alpha_1 = \alpha_2 = .5$, using mean squared error between the natural logarithm of the number of satisfying solutions and the BPNN estimate as our training loss. We used the Adam optimizer (Kingma & Ba, 2015) with an initial learning rate of .0001. We trained for 1000 epochs. We decayed the learning rate, multiplying it by .5 every 100 epochs. We performed batching over the entire training set with one optimization step per epoch. We used two MLPs per layer as in equation 16 and a final Bethe free energy layer.

Learning from Limited Data. We demonstrate that we are able to learn in an extremely data limited regime in table 1. Results in each row titled 'Random Split' were obtained by training on 70% of benchmarks from a single category while holding out a random 30% of the benchmarks for validation. This left *only 9 to 73 benchmarks for training*. In contrast, prior work has performed approximate model counting on boolean formulas in disjunctive normal form (DNF) by creating a large training set of 100k examples that whose model counts can be approximated with an efficient polynomial time algorithm. Such an algorithm does not exist for model counting on CNF formulas, making this approach intractable. How-

[1] https://github.com/meelgroup/ApproxMC
[2] https://github.com/ptheod/F2
[3] https://github.com/msoos/cryptominisat
[4] https://github.com/meelgroup/mis

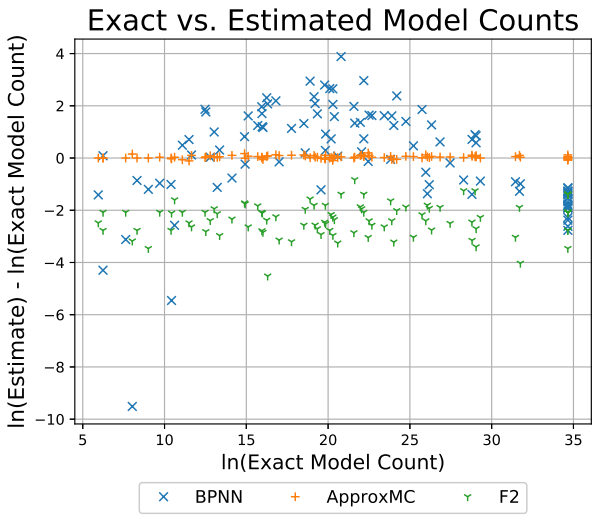

Figure 2: Error in estimate log model count is plotted against the exact model count for 'or_50' training and validation benchmarks. BPNN achieves a validation RMSE of 1.97 on this category compared with a RMSE of 2.5 for F2 across all training problems.

ever, we achieve training and validation RMSE comparable to the randomized hashing method F2 across a range of benchmark categories using only 10's of training problems (F2 generally has an RMSE between 2 and 3). Figure 2 shows estimates by our BPNN on the category 'or_50' compared with the randomized hashing algorithms ApproxMC3 and F2. While ApproxMC3 provides much better estimates than BPNN, it does so by using signficantly more computation as we will see. See Appendix for complete results for ApproxMC3 and F2.

Generalizing from Easy Data to Hard Data. BPNN trained on 'easy' problems can still generalize to 'hard' problems. The rows in table 1 labeled 'Easy / Hard' were obtained by training on 70% of the benchmarks in a single category that were easiest for DSharp to solve, and validating on the remaining 30%. While validation RMSE is worse than training RMSE here, the validation problems are significantly more difficult. DSharp solving time among validation sets is 4-15 times longer than the longest solving time in each corresponding training set. Notably, BPNN significantly outperformed F2 in terms of validation accuracy on the category '90', even when trained on only the easiest problems (F2 has a RMSE of 12.4 on this category). This category was also the second hardest for ApproxMC3 with a 16% completion rate within the time limit of 5k seconds. This demonstrates that BPNNs have the potential to be trained on available data and then generalize to related problems that are too difficult for any current methods.

BPNN RMSE by SAT Category		
Benchmark Category	Train / Val Split	Train / Val RMSE
'or_50'	Random Split	1.92 / 1.97
	Easy / Hard	1.56 / 5.50
'or_60'	Random Split	2.41 / 2.57
	Easy / Hard	1.79 / 5.85
'or_70'	Random Split	1.85 / 2.55
	Easy / Hard	1.69 / 3.59
'or_100'	Random Split	2.85 / 3.24
	Easy / Hard	2.49 / 6.20
'blasted'	Random Split	3.12 / 4.14
	Easy / Hard	2.68 / 307.77
's'	Random Split	300.02 / 15.40
	Easy / Hard	1.49 / 3755.19
'75'	Random Split	2.02 / 3.40
	Easy / Hard	1.73 / 14.78
'90'	Random Split	2.85 / 2.94
	Easy / Hard	5.00 / 6.65

Table 1: RMSE of BPNN for each training/validation set. 'Random Split' rows show that BPNNs are capable of learning a distribution from a tiny dataset of only 10s of training problems. 'Easy / Hard' rows show that BPNNs are able to generalize from simple training problems to significantly more complex validation problems.

BPNNs Provide Excellent Computational Value. Compared with randomized hashing techniques, BPNNs provide a better tradeoff between accuracy and runtime. The RMSE values reported in table 1 are large compared with the RMSE values for ApproxMC3, which range from .03 to .07 for problems that ApproxMC3 could complete within the 5k second time limit. However, even when excluding category 's' (ApproxMC3 could not solve any problems in category 's' within the time limit), ApproxMC3 could only complete 75% of the remaining training problems. Among the problems that it could solve we calculated the ratio of ApproxMC3's runtime to our BPNN's runtime. We found median, mean, and maximum ratios of 41, 6.5k, and 130k respectively, making ApproxMC incomparably slower when only considering problems that it could solve.

However, ApproxMC3 is not the fastest randomized hashing algorithm. It sacrifices speed for strict probabilistic guarantees on the quality of its estimate. F2 delivered significant speedups in our experiments, using state-of-the-art hashing matrices derived from low density parity check error-correcting codes. F2 solves 93% of the

training benchmarks within the time limit. We found median, mean, and maximum ratios of 4.6, 45, and 2800 between ApproxMC3's and F2's runtime on only those problems that could be solved by both methods.

Still, our BPNN is in a separate performance class from F2. We computed runtime ratios between F2 and our BPNN on only those the problems that F2 completed. We found median, mean, and maximum ratios of 32, 1200, and 45000 respectively. BPNN computes model counting estimates with accuracy on par with F2 while generally using orders of magnitude less computation. This gives the quality of BPNN's estimates an excellent computational value compared to randomized hashing approaches. Furthermore, note that we ran our BPNN on a cpu throughout comparisons with randomized hashing approaches. Note that we could achieve orders of magnitude additional speedups by running the BPNN on a GPU with parallel computation (Bixler & Huang, 2018).

Learning Across Domains. We also trained our BPNN on a random sampling of 70% of the problems from the 'or_50', 'or_60', 'or_70', 'or_100', 'blasted', '75', and '90' categories. These problems span the domains of network grid problems, bit-blasted versions of SMTLIB benchmarks, and network DQMR problems. We trained a 3 layer BPNN according to the same protocol followed for individual category training, except that we stopped early after 660 epochs. The BPNN achieved a final training RMSE of 4.4, validation RMSE of 5.4, and test RMSE of 6.4, demonstrating that the BPNN is capable of capturing a broad distribution that spans multiple domains from a small training set.

5 RELATED WORK

Apart from background previously mentioned, the most similar work to ours is Abboud et al. (2020). Here, the authors use a graph neural network to perform approximate weighted disjunctive normal form (DNF) counting. This is a special case of weighted model counting (weighted model counting assigns a weight to every satisfying solution of a boolean formula and asks for the sum of these weights). Weighted DNF counting is a #P-complete problem. However, in contrast to model counting on CNF formulas, there exists an $O(nm)$ polynomial time approximation algorithm for weighted DNF counting (where n is the number of variables and m is the number of clauses in the DNF formula). The authors leverage this fact to generate a large training dataset of 100k DNF formulas with approximate solutions. In comparison, our BPNN can learn and generalize from a very small training dataset of less than 50 problems. This result provides the significant future work alluded to in the conclusion of Abboud et al. (2020). "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."

Yoon et al. (2018) perform marginal inference using GNNs on relatively small smalls. Heess et al. (2013) consider improving message passing in expectation propagation for probabilistic programming, when users can specify arbitrary code to define factors and the optimal updates are intractable. Wiseman & Kim (2019) consider learning Markov random fields and address the problem of estimating marginal likelihoods (generally intractable to compute precisely). They use a transformer network that is faster than loopy BP but computes comparable estimates. This allows for faster amortized inference during training when likelihoods must be computed at every training step. In contrast, BPNNs significantly outperform LBP and generalize to out of distribution data, although they are no faster than LBP.

6 CONCLUSION

We introduced belief propagation neural networks, a GNN architecture that generalizes BP and can perform computations impossible for standard GNNs. We empirically demonstrated that BPNNs can learn from tiny data sets containing only 10s of training points, and generalize to test data drawn from a different distribution than seen during training. BPNNs significantly outperform loopy belief propagation and standard graph neural networks in terms of accuracy. BPNNs provide excellent computational efficiency, running orders of magnitudes faster than state-of-the-art randomized hashing algorithms while maintaining comparable accuracy.

References

Abboud, R., Ceylan, I. I., and Lukasiewicz, T. Learning to reason: Leveraging neural networks for approximate dnf counting. *AAAI*, 2020.

Achlioptas, D. and Theodoropoulos, P. Probabilistic model counting with short XORs. In *SAT*, 2017.

Achlioptas, D., Hammoudeh, Z., and Theodoropoulos, P. Fast and flexible probabilistic model counting. In *SAT*, pp. 148–164, 2018.

Baxter, R. J. *Exactly solved models in statistical mechanics*. Elsevier, 2016.

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

Bixler, R. and Huang, B. Sparse-matrix belief propagation. In *UAI*, 2018.

Chakraborty, S., Meel, K. S., and Vardi, M. Y. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *IJCAI*, 7 2016.

Chandler, D. Introduction to modern statistical. *Mechanics. Oxford University Press, Oxford, UK*, 1987.

Ermon, S., Gomes, C., Sabharwal, A., and Selman, B. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *ICML*, pp. 334–342, 2013.

Ermon, S., Gomes, C. P., Sabharwal, A., and Selman, B. Low-density parity constraints for hashing-based discrete integration. In *ICML*, pp. 271–279, 2014.

Gomes, C. P., Sabharwal, A., and Selman, B. Model counting: A new strategy for obtaining good bounds. In *AAAI*, pp. 54–61, 2006.

Hazan, T. and Jaakkola, T. S. On the partition function and random maximum a-posteriori perturbations. In *ICML*, pp. 991–998. ACM, 2012.

Heess, N., Tarlow, D., and Winn, J. Learning to pass expectation propagation messages. In *NIPS*, pp. 3219–3227, 2013.

Ivrii, A., Malik, S., Meel, K. S., and Vardi, M. Y. On computing minimal independent support and its applications to sampling and counting. *Constraints*, 21(1): 41–58, 2016.

Jerrum, M., Sinclair, A., and Vigoda, E. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM (JACM)*, 51(4):671–697, 2004.

Kingma, D. P. and Ba, J. L. Adam: a method for stochastic optimization. In *ICLR*, 2015.

Kschischang, F. R., Frey, B. J., and Loeliger, H.-A. Factor graphs and the sum-product algorithm. *IEEE Trans. on information theory*, 47(2):498–519, 2001.

Lauritzen, S. L. and Spiegelhalter, D. J. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 50(2):157–194, 1988.

Mézard, M., Parisi, G., and Zecchina, R. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

Mooij, J. M. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *JMLR*, 11:2169–2173, August 2010. URL `http://www.jmlr.org/papers/volume11/mooij10a/mooij10a.pdf`.

Muise, C., McIlraith, S. A., Beck, J. C., and Hsu, E. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, 2012.

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

Owen, A. B. Monte carlo theory, methods and examples, 2013.

Prates, M., Avelar, P. H., Lemos, H., Lamb, L. C., and Vardi, M. Y. Learning to solve NP-complete problems: A graph neural network for decision TSP. In *AAAI*, volume 33, pp. 4731–4738, 2019.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT solver from single-bit supervision. In *ICLR*, 2018.

Soos, M. and Meel, K. S. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *AAAI*, 1 2019.

Soos, M., Nohl, K., and Castelluccia, C. Extending SAT solvers to cryptographic problems. In *SAT*, 2009.

Valiant, L. G. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

Wainwright, M. J., Jordan, M. I., et al. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2): 1–305, 2008.

Weisfeiler, B. and Lehman, A. A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2 (9):12–16, 1968.

Wiseman, S. and Kim, Y. Amortized bethe free energy minimization for learning mrfs. In *NeurIPS*, pp. 15520–15531, 2019.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR*, 2018.

Yedidia, J. S., Freeman, W. T., and Weiss, Y. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Trans. on information theory*, 51(7):2282–2312, 2005.

Yoon, K., Liao, R., Xiong, Y., Zhang, L., Fetaya, E., Urtasun, R., Zemel, R. S., and Pitkow, X. Inference in probabilistic graphical models by graph neural networks. *ArXiv*, abs/1803.07710, 2018.

A PROOFS

Proposition 1. Consider a BPNN with weight tying between layers where the single iterative layer is applied until the messages converge. This particular architecture performs belief propagation and returns the exact Bethe free energy with the following choice of parameters:

- Bias parameters are set to 0 and weight matrices to identity matrices in the single iterative layer.
- Bias parameters are set to 0 in the final Bethe free energy layer.
- All weight matrices are set identity matrices in the final Bethe free energy layer, except for the final weight matrix. The final matrix is set all 0's except for 1's along the diagonal for corresponding to terms in the Bethe free energy.
- All ReLUs are shifted, $\text{ReLU}'(x) = \text{shift} + f(x - \text{shift})$. There exists a shift small enough such that ReLUs never modify their inputs.

Other configurations of BPNNs strictly generalize belief propagation. □

B ADDITIONAL EXPERIMENTS (SAT)

Table 2 shows the root mean squared error (RMSE) of estimates from the approximate model counters ApproxMC3 and F2 across all training benchmarks in each category. Error was computed as the difference between the natural logarithm of the number of satisfying solutions and the estimate. The fraction of benchmarks that each approximate counter was able to complete within the time limit of 5k seconds is also shown. For each benchmark category we show runtime percentiles for ApproxMC3, F2, and the exact model counter DSharp. The DSharp runtime column shows the runtime dividing our easy training sets and hard validation sets for each benchmark category. It also shows the median run time of each hard validation set (85th percentile). The median runtime in each category's hard validation set is 4 to 15 times longer than the longest runtime in each corresponding easy training set. We observe that F2 is generally tens or hundreds of times faster than ApproxMC3. On these benchmarks DSharp is generally faster than F2, however there exist problems that can be solved much faster by randomized hashing (ApproxMC3 or F2) than by DSharp (Achlioptas et al., 2018; Soos & Meel, 2019).

Runtime. Table 3 shows runtime percentiles in seconds for DSharp, ApproxMC3, F2, and a 2 layer BPNN run on all benchmarks in each category in the training dataset. We ran all methods on a cpu. Note that we could achieve orders of magnitude speedups by running the BPNN on a GPU with parallel computation (Bixler & Huang, 2018). However, we still significantly outperform all other methods, generally by orders of magnitude on difficult problems.

Baselines RMSE by SAT Category		
	RMSE (% Completed)	
Category	ApproxMC3	F2
'or_50'	0.07 (89%)	2.4 (100%)
'or_60'	0.07 (87%)	2.3 (100%)
'or_70'	0.06 (78%)	2.4 (100%)
'or_100'	0.06 (73%)	2.4 (100%)
'blasted'	0.04 (80%)	2.4 (84%)
's'	– (0%)	2.9 (81%)
'75'	0.04 (92%)	2.0 (100%)
'90'	0.03 (16%)	12.4 (68%)

Table 2: Root mean squared error (RMSE) of estimates of the natural logarithm of the number of satisfying solutions is shown. The fraction of benchmarks within each category that each approximate counter was able to complete within the time limit of 5k seconds is shown in parentheses.

	Runtimes By Percentile			
Category	DSharp (0/70/85/100)	ApproxMC3 (0/70/100)	F2 (0/70/100)	BPNN (0/70/100)
‘or_50’	0.0 / 0.8 / 12.4 / 48.1	0.1 / 336.6 / 5k	0.2 / 4.0 / 89.9	0.0 / 0.0 / 0.1
‘or_60’	0.0 / 0.3 / 2.1 / 79.1	0.1 / 276.6 / 5k	0.2 / 5.0 / 353.2	0.1 / 0.1 / 0.1
‘or_70’	0.0 / 0.7 / 3.6 / 46.6	0.1 / 748.3 / 5k	0.2 / 11.9 / 491.3	0.1 / 0.1 / 0.1
‘or_100’	0.0 / 0.3 / 4.8 / 54.2	0.1 / 1918.8 / 5k	0.2 / 33.0 / 3021.1	0.1 / 0.1 / 0.2
‘blasted’	0.0 / 1.7 / 29.3 / 1390.8	0.0 / 952.6 / 5k	0.0 / 742.3 / 5k	0.0 / 0.3 / 1.2
‘s’	0.0 / 0.7 / 2.9 / 101.8	5k / 5k / 5k	0.2 / 252.9 / 5k	0.0 / 0.2 / 5.0
‘75’	0.0 / 6.0 / 29.0 / 160.3	279.6 / 805.1 / 5k	1.1 / 2.3 / 9.0	0.1 / 0.2 / 0.3
‘90’	0.0 / 1.8 / 16.7 / 479.9	326.3 / 5k / 5k	1.1 / 5k / 5k	0.1 / 0.4 / 0.7

Table 3: Runtime percentiles (in seconds) are shown for DSharp, ApproxMC3, F2, and a 2 layer BPNN. Percentiles are computed separately for each category’s training dataset.