NeuroBE: Escalating NN Approximations to Bucket Elimination

Sakshi Agarwal, Kalev Kask, Alex Ihler, Rina Dechter

University of California, Irvine
{sakshia1, kkask, ihler, dechter}@ics.uci.edu

Abstract

A major limiting factor in graphical model inference is the complexity of computing the partition function. Exact message-passing algorithms such as Bucket Elimination (BE) require exponentially high levels of memory to compute the partition function, therefore approximations are necessary. In this paper, we build upon a recently introduced methodology called *Deep Bucket Elimination (DBE)* that uses classical Neural Networks (NNs) to approximate messages generated by BE when buckets have large memory requirements. The main feature of our new scheme called $NeuroBE$ is that it customizes the architecture and learning of the NNs to the message size and its distribution. We also explore a new loss function for training taking into account the estimated message cost distribution. Our experiments demonstrate that these enhancements provide significant improvements over DBE in both time and accuracy. We also study the impact of the messages local errors on the global accuracy of the estimate of the partition function.

Introduction

Two of the critical goals of probabilistic modeling are the compact representation of probability distributions and the efficient computation of their marginals and modes. Probabilistic graphical models, such as Markov networks (Pearl 1988; Darwiche 2009; Dechter 2013) provide a framework to represent distributions compactly as normalized products or factors : $P(X) = \frac{1}{Z}\prod_\alpha f_\alpha(X_\alpha)$, where X is a set of variables, each potential f_α is a function over a subset X_α of the variables (its scope) and $Z = \sum_X \prod_\alpha f_\alpha(X_\alpha)$ is the *partition function*. Computing the partition function, or performing inference, is still exponential in the induced width of the model's graph even for distributions that admit a compact representation.

The partition function Z is defined by two types of operations: sums and products. It can be evaluated efficiently if $\sum_X \prod_\alpha f_\alpha(X_\alpha)$ can be reorganized using the distributive law along a variable ordering (Dechter 2003). This organization can be described using buckets as data structures, one for each variable in the ordering. When a bucket is processed, its associated variable is removed by creating a bucket output function, also called a *message*, that is passed to a subsequent bucket. The complexity of computing this function is exponential in its number of arguments, called scope or the bucket's width. Overall, Bucket Elimination (BE) (Dechter 1999a), is time and memory exponential in the induced-width of the model's graph along the ordering. A common approach for approximating BE is to approximate each bucket message with a surrogate fnction whenever it cannot be computed exactly, that is, when its width is too high.

Such schemes that bound the time and space complexity of *BE* include the (weighted) mini-bucket scheme (Dechter and Rish 2003; Liu and Ihler 2012) and generalized belief propagation schemes (Yedidia, Freeman, and Weiss 2000; Mateescu et al. 2010). Our recent approach, *Deep Bucket Elimination (DBE)* (Razeghi et al. 2021), approximates each bucket function with a neural network (NN). While this approach is inherently time consuming requiring the independent training of many NNs to compute the partition function of a single problem, it has yielded more accurate approximations on several benchmarks. Unlike its main competition of weighted bucket-elimination it can improve with time even with bounded memory, providing a potential anytime framework for reasoning. Yet, DBE's original design can be improved significantly as we show in this paper.

Contributions. We present $NeuroBE$, a re-design of DBE, that addresses it's *one size fits all* policy by customizing the NN construction and training sample size to each bucket separately, in proportion to it's message size. We also consider a new loss function for training that takes into account the distribution of messages. We compare $NeuroBE$ with DBE and with other approximation schemes of BE. Lastly, we provide an analysis of the global error associated with the estimated partition function, relating it to the local errors associated with individual messages.

The paper is organized as follows. We first provide a background to *BE* and *DBE*; then we present $NeuroBE$ followed by error analysis; lastly, we demonstrate the efficiency of $NeuroBE$ empirically.

Related work. As noted, approximating and bounding *Bucket Elimination* has been carried out extensively over the years for all probabilistic queries. Well known is the *Mini-Bucket Elimination* scheme (Dechter and Rish 2003) and its variants, such as *Weighted Mini-Bucket Elimination*

(WMBE), augmented with message-passing cost-shifting (Liu and Ihler 2011).

Neural network approximation to *BE* was introduced in (Razeghi et al. 2021). The idea is closest in spirit to the Neuro-Dynamic Programming scheme as outlined in (Bertsekas and Tsitsiklis 1996) where the cost-to-go functions (similar to messages) generated by dynamic programming can be approximated by NNs. This is also highly related to Deep Reinforcement Learning (DRL) (Mnih et al. 2015) where, in the absence of a model, the value function is approximated by neural networks learned from temporal trajectories.

Recently, *Graph Neural Networks (GNNs)* (Scarselli et al. 2009) are used to learn *messages* following the message-passing reasoning methods in graphical models (Abboud, Ceylan, and Lukasiewicz 2020; Yoon et al. 2018; Heess, Tarlow, and Winn 2013). However, (Yoon et al. 2018; Heess, Tarlow, and Winn 2013) is restricted to small instances (i.e., $\sim$40 variables) and (Abboud, Ceylan, and Lukasiewicz 2020) tackles problems with a known polynomial-time approximation. GNN based methods derive a supervised end-to-end learning algorithm which generalize across different problem instances. In contrast, we consider a different class of algorithms, where we confine learning to within a problem instance *only*.

Background

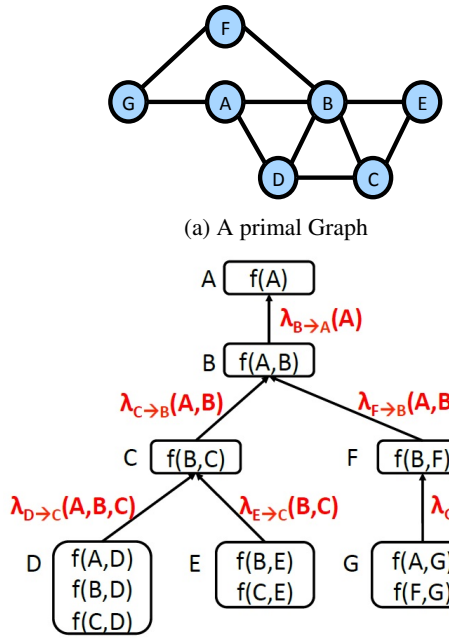
A graphical model can be defined by a 3-tuple $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, where $\mathbf{X} = \{X_i : i \in V, V = \{1, ..., n\}\}$ is a set of n variables indexed by V and $\mathbf{D} = \{D_i : i \in V\}$ is the set of finite domains for each X_i (i.e. each X_i can only assume values in D_i, and each D_i is finite). Each function $f_\alpha \in \mathbf{F}$ is defined over a subset of the variables called its scope, X_α, where $\alpha \subseteq V$ are the indices of variables in its scope and D_α denotes the Cartesian product of their domains, so that $f_\alpha : D_\alpha \rightarrow R \geq 0$.

The **primal graph** of a graphical model associates each variable with a node. An edge between node i and node j is created if and only if there is a function containing X_i and X_j in its scope. Figure 1a shows a primal graph of a graphical model with variables indexed from A to G with functions over pairs of variables that are connected by an edge. Graphical models can be used to represent a global function, often a probability distribution, defined by $Pr(X) \propto \prod_\alpha f_\alpha(X_\alpha)$. An important task is to compute the normalizing constant, also known as the partition function $Z = \sum_X \prod_\alpha f_\alpha(X_\alpha)$.

Bucket Elimination

Bucket Elimination (BE) (Dechter 1999b) is a universal exact algorithm for probabilistic inference. It is a variable elimination algorithm that can answer a wide-range of queries, including the partition function ranging from constraint satisfaction, to pure combinatorial optimization (e.g., Most Probable Explanation (MPE/MAP)), and weighted counting (Partition Function, Probability of Evidence).

Given a variable ordering d, BE (presented in Algorithm 1, omitting steps 9-12) creates a *bucket tree* where each node



(a) A primal Graph

(b) Bucket Elimination example

Figure 1: (a) A primal graph of a GM with 7 variables. (b) Illustration of *BE* with an ordering A B C E D F G.

is a bucket representing a variable in the ordering d. Figure 1b shows a bucket tree for the primal graph in Figure 1a along an ordering. Each bucket in this tree contains a set of the model's functions depending on the given order of processing. For example, Bucket G in Figure 1b has functions $\{f(A, G), f(F, G)\}$, an exhaustive set of model's functions with variable G in its scope. There is an arc from a bucket, say B_c, to a parent bucket, B_p, if X_p is the latest variable in bucket B_c's message scope along the ordering (constants are placed in B_1). In the same example, there is an arc from Bucket G to Bucket F.

BE then, performs inference along the bucket tree as a 1-iteration message-passing algorithm (bottom-up). It processes each bucket from leaves to the root passing messages from child (c) to parent (p). For a child variable X_c, it considers all the functions in bucket B_c. This includes the original functions in the graphical model as well as the messages received by processing previous variables. It then marginalizes X_c out from the product of functions in B_c generating a new, so called, *bucket function* or message, denoted $\lambda_{c \rightarrow p}$, or λ_c for short:

$$\lambda_c = \sum_{X_c} \prod_{f_\alpha \in B_c} f_\alpha \tag{1}$$

The λ_c function is placed in B_p, the bucket of X_p. Once all the variables are processed, *BE* outputs all the messages and the exact value of Z by taking the product of all the constant present in the bucket of the first variable. We illustrate *BE* message flow on our example problem in Figure 1b.

Complexity. Both the time and space complexity of *BE* are exponential in the **induced width**, which is the size of

the largest number of variables in the scope of any message in a graph (Dechter 2013). Clearly, *BE* becomes impractical if the induced width is large.

Deep Bucket Elimination

Given a variable ordering d, *Deep Bucket Elimination (DBE)* (Razeghi et al. 2021), approximates each message generated in the bucket tree by training a NN when the scope of any bucket message (S_c) is high ($>$ i-bound). For example, in figure 1b, if we use an i-bound $= 2$, instead of sending an exact function from the bucket of D to the bucket of C, $\lambda_{D\to C}(A, B, C)$, DBE sends a NN approximation $\mu_{\theta,D\to C}(A, B, C)$ with parameters θ, as we describe next.

Let B_c be a bucket with width $w_c > i$-bound and output message $\lambda_c(S_c)$ with scope S_c. DBE then, constructs a fully-connected feed-forward NN having w_c nodes in the input layer. This is followed by L hidden layers with a constant h hidden nodes per layer with $ReLU$ activation function. Finally, the output layer contains one node with a real-valued output. Subsequently, DBE generates a training set $\{(s_n, \lambda_c(s_n))\}$ of size N, where s_n denotes a configuration over S_c sampled uniformly at random and $\lambda_c(s_n)$ is the message value defined in Eq. 1. The NN parameters, θ, are then trained to minimize the mean square error loss:

$$L(\theta) = \frac{1}{N}\sum_{n=1}^{N}(\lambda_c(s_n) - \mu_{\theta,c}(s_n))^2$$

where s_n is the n^{th} sample in the training set and $\mu_{\theta,c}(s_n)$ is the NN output. Once training is complete, DBE passes the trained NN $\mu_{\theta,c}(s_n)$ to its parent bucket. Typically, during its course, DBE may approximate many bucket messages in order to compute the partition function Z.

However, it often requires a large training sample size per message to achieve desired performance. This leads to a substantial increase in the algorithm's time and memory requirements. Therefore, to make DBE efficient, we redesign the message approximation procedure, elaborated in the following section.

NeuroBE

We rename DBE to $NeuroBE$ acknowledging the use of shallow neural networks (2 layers) to approximate each message. Algorithm 1 presents *NeuroBE*. Similar to DBE, $NeuroBE$ first creates a bucket tree along a given ordering in line 2. While processing each bucket along the ordering, if it's width $\leq i-$bound, then the message, $\mu^*_{c\to p}$, is computed exactly in line 7. Otherwise, the message is approximated using a NN in line 10. Note that in either case, if a bucket contains a NN function, then computing μ^* in Line 7 or in NN-train function (Algorithm 2) requires evaluating the trained NN. Finally, line 14 calculates the partition function using the functions in bucket B_1.

Note that we denote $\mu^*_{c\to p}$ as the exact message computed in a bucket while reserve the notation $\lambda_{c\to p}$ to the messages computed by exact BE. We do this to distinguish the exact local computation of a message that may be based on inexact functions in the bucket from the globally exact messages λ computed by BE. In the latter, each bucket function is computed exactly and also all the functions in a bucket are exact. Hence, we refer to μ^*_c as the *local exact message*. Further, we denote $\mu_{\theta_c,c\to p}$ as the NN approximations of the *local* exact message, $\mu^*_{c\to p}$.

Algorithm 1: NeuroBE

Input: Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, Ordering $d = X_1, ..., X_n$
Parameters: i-bound i, #layers L, constants b, η
Output: the partition function constant and bucket messages

1: **for** c in n...1 **do**
2: (initialize buckets and form a bucket-tree) put all unplaced functions mentioning X_c in B_c.
3: **end for**
4: **for** c in n...1 **do**
5: Let X_p be the parent variable of X_c in the bucket-tree
6: **if** $width(B_c) \leq i$ **then**
7: $\mu^*_{c\to p} \leftarrow \sum_{X_c}\prod_{f_\alpha\in B_c} f_\alpha$
8: **else**
9: (denote by $\mu^*_{c\to p}$ the function $\sum_{X_c}\prod_{f_\alpha\in B_c} f_\alpha$)
10: $\mu_{\theta_c,c\to p} \leftarrow$ NN-train ($\mu^*_{c\to p}$, L, b, η)
11: **end if**
12: Put $\mu^*_{c\to p}$ or $\mu_{\theta_c,c\to p}$ in B_p
13: **end for**
14: $Z = \sum_{X_0}\prod_{f_\alpha\in B_1} f_\alpha$
15: **return** Z and All messages generated

The difference between $NeuroBE$ and DBE is solely in the individual message approximation scheme, $NN\text{-}train$. As noted before, DBE often uses a constant, large sized training set for each message approximation. A simple brute-force reduction of the sample size only to reduce training time, may lead to *overfitting*. Hence $NeuroBE$ customizes the NN architecture and its training set size to the message complexity (see (Vapnik 1999)).

NN Architecture selection

It is obvious that the NN size should be dependant on the dimensionality of the message function. In our case, the function's scope size is the induced-width, w. We propose to adjust the NN size by making the number of hidden units, h, a function of w while keeping the number of layers, L, constant. We select $h = b \cdot w$, where b is a constant satisfying $b \geq 1$. Figure 2 illustrates an example NN model with an input layer of size w and 2 hidden layers with dimension h, varying linearly with b. When $b = 1$ all layers are of the same size w. Through such a rule, $NeuroBE$ *fits* NN to message size. We now quantify the capacity of such NNs and apply it to determine it's train sample sizes.

NN complexity. The *pseudo-dimension* (Pollard 1984; Anthony and Bartlett 2002) is used to estimate the expressive power or complexity of NNs in regression problems. Bounds to the pseudo-dimension of such NNs with ReLU activation functions is provided by (Bartlett et al. 2019). We use the lower bound as a proxy to estimate the pseudo-

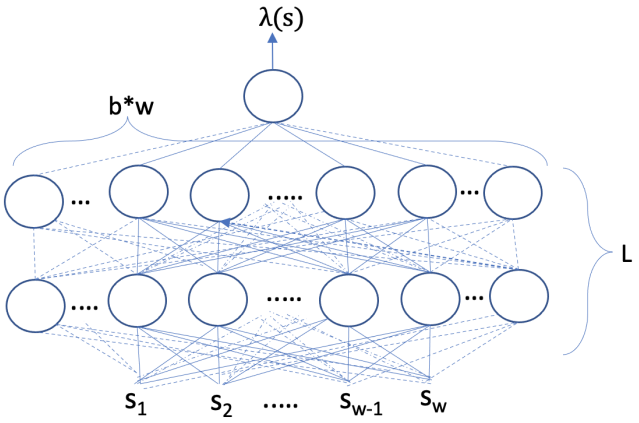


Figure 2: For a bucket of width w, we illustrate a NN architecture with L layers and $\#bw$ hidden-units with $b \geq 1$.

dimension (ρ_c) of the NN approximating a message μ_c^* having width w_c, yielding (see appendix for derivation):

$$\rho_c(w_c) \propto (L * b * w_c)^2 log[(b * w_c)] \tag{2}$$

We use the above equation as a correlation between the complexity of the candidate NN and the width of the message μ_c^* it approximates.

Sample Complexity As suggested in (Vapnik 1999). we choose a sample size (N) that is proportional to the pseudo-dimension (Eq. 2) of each NN:

$$N = \eta * (L * b * w)^2 log(b * w) \tag{3}$$

where η is a constant, and w is the number of arguments to the estimated function. We will use $N(w)$ to emphasize that N varies with w. The sample size $N(w)$ often exceeds memory limits for higher width buckets with even the simplest NN architecture ($L = 1, b = 1$). Hence, we threshold sample size per bucket to 1000k. In general, for high induced-width problems, we keep η small to favour training NNs with small sample sizes. However, for problems with small induced-width, we let η take high values.

Learning NNs

For training a NN, DBE generated samples in a bucket uniformly at random and used the mean square error as the loss function. We wondered whether generating the samples in a non-uniform distribution, in a way that is related to the message distribution, would be more effective. Alternatively, we asked if a loss function that is dependant on the actual message values should be used. To that end we define the distribution, $F_c(S)$, of an output message $\mu_c^*(S)$ by:

$$F_c(s) = \frac{\mu_c^*(s)}{\sum_{s \in D(S)} \mu_c^*(s)}, \tag{4}$$

where the denominator is summed over each configuration s from the set of all possible configurations $D(S)$ over scope S. However, since sampling from $F_c(S)$ is hard we sampled from the uniform distribution, but changed our loss to a weighted estimate of the mean square error (w.m.s.e):

$$L(\theta_c) = \frac{1}{N_c} \sum_{s \in D(S)} (\mu_c^*(s) - \mu_{\theta_c}(s))^2 \frac{\hat{F}(s)}{\hat{U}(s)} \tag{5}$$

where c is an index to the bucket, s is a uniformly sampled configuration in scope S_c and $\mu_{\theta_c}(s)$ is the approximated message value computed by the NN; $\hat{F}(s)$ is the probability of s following the message distribution (eq 4), and $\hat{U}(s)$ is the probability of s according to the uniform distribution.

Algorithm 2 describes this learning procedure. For a bucket B_c of variable X_c having local exact output message μ_c^*; Algorithm NN-train, using its input parameters L, b, η constructs a NN with L layers and $b \cdot w_c$ hidden units (line 3). It then determines it's training sample size N (line 4) using Eq. 3. A major step occurs next (line 5) where it creates the training, validation and testing sets by generating samples uniformly from the domain of the function scope S as follows. For each sample configuration s, and a variable X_c, we compute it's target value $\mu^*(s)$, as:

$$\mu^*(s) = \sum_{x \in D_{X_c}} \prod_{f \in B_c} f(s, x)$$

If the bucket B_c contains a trained NN, then this step requires evaluating that NN. Lines 9-12 then trains the NN parameters by dividing the train set into batches. Line 10 shows the function NN(data,μ_θ) as the output computed by

Algorithm 2: NN-train(μ_c^*,L,b, η, $\#epochs$)

Input: μ_c^* message on a set of variables X
Parameters: L: # layers in NN, $\#epochs$, η, b: constants
Output: $\mu_{\theta_c,c}$: NN message approximation, $\hat{\epsilon}$: an estimated bucket error bound, $\hat{\epsilon}_{avg}$: estimated average bucket error

1: $w_c \leftarrow$ scope(μ_c^*)
2: $\#h \leftarrow b * w_c$
3: Initialize NN $\mu_{\theta_c,c}$ with L layers, $\#h$ hidden-units
4: $N \leftarrow$ sample-size(w_c, η, L, b)
5: trainSet, valSet, testSet $\leftarrow$ generate-samples(μ_c^*, N)
6: p=1, error_val = $+\infty$, $\mu_\theta \leftarrow \mu_{\theta_c,c}$
7: **while** p $\leq$ #epochs and $\neg$ early_stopping(error_val) **do**
8: $\quad \mu_{\theta_c,c} \leftarrow \mu_\theta$
9: $\quad$ **for** train$_i$ in batches(trainSet) **do**
10: $\quad\quad \mu_\theta$_train$_i \leftarrow$ NN(train$_i$,μ_θ)
11: $\quad\quad$ loss $\leftarrow$ w.m.s.e(train$_i$, μ_θ_train$_i$)
12: $\quad\quad \mu_\theta \leftarrow$ optimize(Adam, loss, θ)
13: $\quad$ **end for**
14: $\quad \mu_\theta$_val $\leftarrow$ NN(valSet,μ_θ)
15: $\quad$ error_val $\leftarrow$ w.m.s.e(valSet, μ_θ_val)
16: $\quad p \leftarrow p + 1$
17: **end while**
18: error $\leftarrow$ log(testSet $\setminus$ NN(testSet,$\mu_{\theta_c,c}$))
19: $\hat{\epsilon}_c \leftarrow$max(error), $\hat{\epsilon}_c^{avg} \leftarrow$ avg(error)
20: **return** $\mu_{\theta_c,c}$, $\hat{\epsilon}_c$, $\hat{\epsilon}_c^{avg}$

the NN μ_{θ_c} on input set data. Line 11 computes the w.m.s.e between the target values in the train set and the NN output, μ_θ_train. The NN parameters (θ_c) are then updated using the Adam optimizer (Kingma and Ba 2014) (line 12). After each epoch, the current trained model is evaluated on a holdout validation set (line 14-15). We stop training when either the maximum limit $\#epochs$ is reached or the validation error meets our early stopping criteria, that is if the validation error increases for two consecutive epochs. Once training is complete, we compute the maximum and average log relative errors between the target and NN approximated messages over a test set (lines 18-19). In the next section, we use this to analyse error propagation in $NeuroBE$. The $NN\text{-}train$ procedure then returns the approximated message $\mu_{\theta_c,c}$ along with it's estimated error.

Complexity. Clearly, the time and space complexity for learning a single message in $NeuroBE$ is linear in the sample size. In contrast to DBE, here the sample size is customized to the message complexity, or bucket width.

Error Analysis

We next analyse the relationship between the local errors contributed by each approximated message and the global partition function error.

Definition 1 (local and global bucket errors)**.** Let λ_c be the (global) exact message generated in B_c by the exact BE algorithm, μ_c^* be the local exact message in B_c computed by the functions in it and $\mu_c = NN\text{-train}(\mu_c^*)$ be its NN approximation. Then, the *Local Bucket Error* is the function

$$E_c = log\mu_c^* - log\mu_c$$

The *Global Bucket Error* is

$$G_c = log\lambda_c - log\mu_c$$

The above error corresponds to a log of the relative errors. We use log relative error here because bounding the global error as a function of the local errors turned out to be easier. Note that the true partition function, $Z^* = \lambda_1$. So, the global bucket error G_1 is the error in the estimated partition function, thus measuring $NeuroBE$'s performance.

Theorem 1. *Assume a bucket-chain along an ordering d and let B_c be a bucket along the chain. Let $E_c(x) = ln\mu_c^*(x) - ln\mu_c(x)$ as defined above and let $\epsilon_c = max_{s\in D(S_c)}|E_c(s)|$, where S_c is the scope of outgoing message from B_c and $D(S_c)$ is the set of all possible configurations on S_c. Then,*

$$E_c = ln\lambda_c - ln\mu_c \leq \sum_{k=0}^{n-c} \epsilon_{c+k}$$

In particular, since $\lambda_1 = Z$, the partition function

$$E_1 = lnZ - ln\mu_1 \leq \sum_{k=0}^{n-1} \epsilon_{1+k} \tag{6}$$

For the proof see the Appendix.

Calculating e_c is hard because it involves computing the local bucket error E_c over all configurations in the scope of the bucket. Therefore, we calculate the maximum error over a sampled test set in steps 18-19 of algorithm 2 as $\hat{\epsilon}_c$. Additionally, we calculate the average local bucket error, $\hat{\epsilon}_c^{avg}$ over the same test set. According to Eq. 6, summing over all bucket error bounds, $\hat{\epsilon}_c$, bounds the global error of the estimated partition function. Clearly, this bound is very lose. We therefore also use the average local bucket error, $\hat{\epsilon}_{avg}$ to give us some additional information on the global error empirically. In the next section we evaluate $NeuroBE$ and provide some information on local vs global errors.

Experiments

Experiment Setup

We ran experiments comparing *NeuroBE* against the Weighted Mini Bucket Elimination scheme (*WMBE*) (Dechter and Rish 2003; Liu and Ihler 2012) and DBE (Razeghi et al. 2021). Following the methodology in DBE, we evaluated $NeuroBE$ on instances selected from three well-known benchmarks from the UAI repository used in (Kask et al. 2020), i.e. grids (vision domain), pedigree (genetic linkage analysis) and DBNs. We targeted diverse benchmarks (in structure and level of determinism) and aimed for different levels of hardness. Thus, in each benchmark, we distinguish between problems that can be solved exactly, which we call "easy", and those that cannot be solved, called "hard". We also distinguish benchmarks that possess *determinism*, namely have a high proportion of zero probabilities, a feature which can impact training. We randomly selected 14 instances from Grids, with easy ones (i.e., width 20-30) and hard ones (i.e., 1600 variables, width 55), 6 from pedigrees, which posses high level of determinism and 6 from DBNs, totalling 25 instances. To trigger bucket message approximations, we used i-bound=10 for easy problems and at most i-bound=20 for hard ones.

For problems with determinism, such as pedigree, the structure of the NN in $NeuroBE$ is the same as that of a MaskedNet in DBE ((Razeghi et al. 2021)), varying only the number of hidden units per layer.

NNs architecture Here, we show how the NN architecture and sample size is tuned across the different benchmarks. We keep the #layers fixed (=2) across all benchmarks. We then pick a random problem instance from each benchmark to fix the hyper-parameters regulating the NN architectures and sample sizes. For each problem instance with it's width w^*, we first keep $h = w$ and the number of samples corresponding to $\frac{w^*}{2}$ roughly around $300k$ for hard problems and $100k$ for easy problems as a heuristic and derive a value for η from equation 3. Keeping η fixed and then again, varying $h \in [w, 5w]$, we pick the value of h which gives the smallest average error to estimate the partition function for each representative problem instance. We then use this configuration for varying NN architecture and sample size for the rest of the problem instances in that benchmark. In particular, we selected $h = 3w$ and $N_{avg} \in [149k, 350k]$ for pedigrees; $h = \{3w, 5w\}$ and $N_{avg} \in [80k, 180k]$ for DBN; $h = w$ and $N_{avg} \in [12k, 121k]$ for grid-easy; and $h = w$ and $N_{avg} \in [60k, 209k]$ for grid-hard.

Problem Description					ref Z	WMB	#NB	DBE (#h=100, N=320k)				NeuroBE (#h=3w, N(w,η=10), N_{min} = 49k)						
i-bound=20						error		statistics on error over 10 runs				Statistics on resources			statistics on error over 10 runs			
Id	name	k	#v	w				avg error	min error	stdev	time	h_{max}	N_{avg}	N_{max}	avg error	min error	std	time (h)
1	pedigree13	3	888	33	-31.18	6.4696	127	5.32	2.62	3.06	11.4	96	218k	706k	1.11	0.76	0.25	7.5
2	pedigree41	5	885	32	-76.04	4.1497	92	4.27	3.25	0.734	9.9	93	190k	658k	0.47	0.153	0.21	6.2
3	pedigree51	5	871	35	-77.27	9.7624	120	23.92	9.23	12.74	13.1	102	259k	809k	3.51	1.96	0.89	10.4
4	pedigree34	5	922	33	-64.23	7.0762	106	5.91	1.57	5.98	11.1	96	211k	706k	0.65	0.23	0.29	6.96
5	pedigree7	4	867	34	-64.82	6.0012	108	11.26	5.18	7.8	10.5	99	350k	900k	1.75	1.21	0.7	10.7
6	pedigree19	5	693	28	-59.020	2.5809	43	6.054	5.41	0.92	9.35	71	149k	482k	2.61	1.91	0.6	5

(a) pedigree

Problem Description					ref Z	WMB	#NB	DBE (#h=100, N=320k)				NeuroBE (#h=w, N(w,η=4), N_{min} = 19k)						
i-bound=20						error		statistics over 10 runs				Statistics on resources			statistics on error over 10 runs			
Id	name	k	#v	w				avg error	min error	stdev	time	h_{max}	N_{avg}	N_{max}	avg error	min error	std	time (h)
1	grid4040f10	2	1600	55	5490	215.45	308	**97.1**	**11.81**	65.15	11.8	55	60k	182k	**24**	**16.51**	8	5.4
2	grid4040f5	2	1600	55	2800	84.92	308	**39.9**	**6.28**	34.96	11.7	55	60k	182k	**21.2**	**18.87**	6.4	5.3
3	grid4040f2	2	1600	55	1220	25.24	308	**7.34**	**1.2**	5.4	11.5	55	60k	182k	**5.1**	**3.46**	1.8	5.2
4	grid4040f15	2	1600	55	8200	338.2	308	**83.46**	**41.78**	34.2	13.4	55	60k	182k	**22.4**	**5.38**	13.3	5.4
5	grid4040f10w	2	1600	114	5637	297.7	376	100.5	**6.4**	82.15	21.2	114	209k	900k	**51.05**	**29.72**	18.9	13.7
6	grid4040f5w	2	1600	114	2819	136.99	376	78.2	**72.62**	5.6	**21.3**	114	209k	900k	**16.6**	**12.28**	3.92	13.9
7	grid4040f2w	2	1600	114	1231	32	376	**15.12**	**0.92**	20.52	18.2	114	209k	900k	25.78	**12.61**	17.9	11.8
8	grid4040f15w	2	1600	114	8230	657.03	376	**220.91**	**95.23**	192.2	17.7	114	209k	900k	103.9	79.59	27.8	**13.7**

(b) Grid-hard

Problem Description					ref Z	WMB	#NB	DBE (#h=100, N=320k)				NeuroBE (#h=w, N(w,η=20), N_{min} = 10k)						
i-bound=10						error		statistics on error over 10 runs				Statistics on resources			statistics on error over 10 runs			
Id	name	k	#v	w				avg error	min error	stdev	time	h_{max}	N_{avg}	N_{max}	avg error	min error	std	time (h)
1	grid1010f10w	2	100	21	333.32	32	31	**4.45**	**0.89**	3.71	1.53	21	25k	48k	**3.51**	**1.45**	1.21	0.35
2	grid1010f10	2	100	13	303.086	1.58	8	**0.7**	**0.05**	0.54	0.32	13	11.6k	15.5k	**1.1**	**0.34**	0.7	0.055
3	grid2020f2	2	400	27	291.733	11.24	114	**1.98**	**0.36**	1.28	5.34	27	68k	480k	**0.37**	**0.174**	0.21	2.24
4	grid2020f10	2	400	27	1311.98	80.86	114	**10.04**	**1.05**	9.3	5.5	27	68k	480k	**1.37**	**0.08**	1.05	2.22
5	grid2020f5	2	400	27	665.12	39.44	114	5.75	**0.24**	3.1	5.5	27	121k	965k	**0.67**	**0.018**	0.56	0.91
6	grid2020f15	2	400	27	1962.98	122.91	114	17.8	**3.08**	9.8	**2.95**	27	68k	480k	**2.35**	**0.53**	1.57	2.2

(c) Grid-easy

Problem Description					ref Z	WMB	#NB	DBE (#h=100, N=320k)				NeuroBE (#h=3w, N(w,η=30), N_{min} = 147k)						
i-bound=20						error		statistics on error over 10 runs				Statistics on resources			statistics on error over 10 runs			
Id	name	k	#v	w				avg error	min error	stdev	time	h_{max}	N_{avg}	N_{max}	avg error	min error	std	time (h)
1	rbm20	2	40	21	58.53	0.0007	20	0.22	0.034	0.17	1.05	60	147k	147k	0.23	0.002	0.21	0.45
2	rbm21	2	42	22	63.15	6.39	22	0.48	**0.27**	0.19	1.25	63	163k	164k	0.78	0.133	0.8	0.46
3	rbm22	2	40	21	66.55	8.65	24	0.47	**0.14**	0.35	1	66	180k	182k	0.32	0.035	0.26	0.57
4	rbm-ferro20	2	44	23	151.16	0.005	20	1.33	0.29	1.21	1.03	60	147k	147k	1.58	0.38	1.15	0.44
5	rbm-ferro21	2	42	22	152.62	1.98	22	3.43	**0.83**	1.89	1.17	63	163k	164k	2.79	0.125	2.57	0.48
6	rbm-ferro22	2	44	23	166.11	0.517	24	6.52	3.86	1.5	1.3	66	180k	182k	4.69	1.09	3.2	0.59
i-bound=10												NeuroBE (#h=5w, N(w,η=20), N_{min} = 20k)						
1	rbm20	2	40	21	58.53	7.85	30	0.49	0.11	0.35	1.57	100	82k	87k	1.35	0.46	0.96	0.42
2	rbm21	2	42	22	63.15	15.73	32	0.73	0.14	0.47	1.71	105	90k	109k	0.43	0.14	0.35	0.5
3	rbm22	2	40	21	66.55	27.46	34	0.65	0.19	0.41	1.83	110	99.5k	121k	0.46	0.19	0.49	0.54

(d) DBN

Figure 3: Results on performance of *NeuroBE* against *DBE* and *WMB*. k:domain size, $\#v$:variables, w:induced width, $\#NB$: number of buckets that are trained with NNs, $\#h$: number of hidden units per layer (reported maximum $\#h$ for *NeuroBE*), N: number of training samples (reported minimum, average and maximum $\#N$ for *NeuroBE*), $error$: L1 error for referenced and estimated $log(Z)$ (reported minimum, average, and standard deviation over 5 runs for *DBE* and *NeuroBE*), $time$: average time taken to get the estimated $error$:. *Note: Here, referenced Z is approximated by (Kask et al. 2020)

Problem Description					ref Z	NeuroBE		Statistics on Global Error (over 5 runs)					
i-bound=20						(h=3w)		m.s.e loss			w.m.s.e loss		
Id	name	k	#v	w		#NB	N_{avg}	avg error	min error	stdev	avg error	min error	stdev
1	pedigree13	3	888	33	-31.18	127	218k	9.04	7.86	0.8	1.11	0.76	0.25
2	pedigree41	5	885	32	-76.04	92	190k	10.2	8.4	2.16	0.47	0.153	0.21
3	pedigree51	5	871	35	-77.27	120	259k	11.725	10.37	1.26	3.51	1.96	0.89
4	pedigree34	5	922	33	-64.23	106	211k	6.14	4.56	4.14	0.65	0.23	0.29
6	pedigree19	5	693	28	-59.020	43	149k	9.14	8.7	0.6	2.61	1.91	0.6

Figure 4: Comparing m.s.e and w.m.s.e loss function with NeuroBE on pedigrees. $\#NB$: # buckets trained, N_{avg}: average samples, $avg\ error$: average global error, $stdev$: standard deviation on global error (over 5 runs).

Training NNs. Bucket output messages can either have very small values (eg. $exp(-11)$) as in the pedigree benchmarks (and possess determinism) or can be very large (eg. $exp(51)$) as for grids and DBNs. To handle large values in messages of non-deterministic benchmark domains, we use log transformations to handle overflow issues. In addition, we normalize the input and output values for NNs across benchmarks to be in $[-1, 1]$ and $[0, 1]$ respectively to accelerate training (Cun, Kanter, and Solla 1991)). As per algorithm 2, we create the training set of size $N(w_c)$ (Eq 3), validation set of size $\frac{N(w_c)}{9}$, and test set of size $50k$. We then train the network using the Adam optimizer with a learning rate of 0.001 and a batch-size of 256 across all benchmarks.

Performance measures We evaluate the performance of $NeuroBE$ using: $error = |log_e Z - log_e \hat{Z}|$ where $\hat{Z}$ is the generated estimate of the partition function, Z. When the exact Z is not available (for hard Grid benchmark), Z^* is a surrogate to Z, which is obtained using an advanced sampling scheme for a duration of $100 * 1hr$ (Kask et al. 2020).

Results

Figure 3 compares $NeuroBE$ against DBE and WMB over the 3 benchmarks. We report the results over 5 runs for each instance of both $NeuroBE$ and DBE due to stochasticity of their behavior. The first few columns show the problem statistics for instances in the respective benchmarks (pedigree, DBN and grids). We then show wmb error. For $NeuroBE$, we report the average and maximum #training samples, (N_{max}, N_{avg}) and maximum #hidden units, (h_{max}) across all buckets of the instance. We also report the average time (in hours), the average error, minimum error and standard deviation over the 5 runs.

Pedigrees We see a consistent decrease in both average error and standard deviation for the partition function estimates with $NeuroBE$ when compared to DBE, being $\geq$5 times more accurate than DBE for 5 out of the 6 instances. It achieves this better estimates with less time, since it uses far less training samples. Also $NeuroBE$ outperforms WMB on 5 instances ($\geq$5 times more accurate for 4 instances). Here DBE yields either a similar accuracy as WMB or even a worse one (instances 3,5,6).

Grids. Here too we observe that $NeuroBE$ outperforms DBE in accuracy as reflected by the average error and standard deviation, even though it uses far less time. In most cases we see a reduction in time by a factor of 2 or more (IDs 1,3,4,5 from grid-easy and IDs 1,2,3,4,6,7 from grid-hard) still producing a far better estimate. For 2 instances, however, (easy #2, hard #5) we see slightly worse performance than DBE. $NeuroBE$ and DBE outperform WMB across all problem instances.

DBN We report results for the DBN benhcmark for 2 i-bounds. For i-bound=20, ***NeuroBE*** achieves a higher accuracy than DBE with far less time (instances 3,5,6). It is superior to wmb on instances 2,3. However, WMB performs better on instance 1,4 & 6, as the induced-width is closer to the i-bound and is comparable to DBE in accuracy, yet it takes half of the time. For i-bound=10, $NeuroBE$ shows better accuracy than WMB for all three instances. It outperforms DBE on instances 2,3.

Overall, compared with DBE, $NeuroBE$ is faster on 11 grid + DBN instances by a factor of 2; 6 times more accurate on 4 easy grid instances; more accurate on 5 grid-hard instances by a factor of 2 and 5 times more accurate on 5 pedigree instances.

The impact of loss functions. Overall, on most of the instances of Grids and DBNs we did not observe a significant impact of weighted loss function on the quality of training (results are not explicitly shown). Yet, for pedigree instances we did observe a significant impact. We suspect this for pedigrees because the NN outputs are messages (i.e., they do not undergo log transformation) and the weights in the w.m.s.e reflect the message distribution. Figure 4 compares the w.m.s.e and m.s.e loss for pedigrees. We observe that with w.m.s.e loss, $NeuroBE$ shows better performance on all instances (for instance 2, training with w.m.s.e is 20 times more accurate).

How local errors impacts the global error. Figure 5 reports some statistics on local bucket errors (Def. 1) and test w.m.s.e across buckets for 4 Grid problem instances; We also show the global errors estimated by Eq. 6; and the empirical global errors averaged over 5 runs. We report the results for 2 sets of average sample size $N_{avg} = \{60k, 150k\}$. 1) We observe a consistent decrease in the local bucket and global errors when more samples are given. However, instance 4 shows an increase in the empirical global error, which we account to stochastic behaviour. 2) We also observe a direct correlation between the estimated local bucket errors and empirical average global errors across instances. For example, the average local bucket error for instance 1 is 1.67 with an empirical global error of 24.01; instance 2 shows an average local bucket error of 0.784 with 21.15 as the empirical global error. Note that both errors simultaneously decrease for instance 2. This trend continues for instance 3, where the local bucket error is 0.336 while the global error is 5.05. We observe such a correlation of the empirical average global errors on the estimated local bucket errors across all instances, which is affirmative.

Time & accuracy. Figure 6 shows the expected relationship between time and accuracy on a few problem instances. We depict the average error of our partition function estimate

	N_{avg} = 60k, h=w								N_{avg} = 150k, h=w							
	Statistics on local bucket errors				Statistics on global errors				Statistics on local bucket errors				Statistics on global errors			
Id	test w.m.s.e		local bucket errors		estimated bounds		empirical errors		test w.m.s.e		local bucket errors		estimated bounds		empirical errors	
	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max	avg	max
1	2.19E-04	0.06	1.67	3.81	251.7	1664	24.01	37.21	1.63E-05	0.006	0.29	3.71	71.75	1380	8.11	13.62
2	1.74E-04	0.053	0.784	2.44	115.6	796	21.15	31.32	1.40E-05	0.0068	0.131	1.45	34.34	680	6.4	12.3
3	1.06E-04	0.053	0.336	0.661	34.98	264.91	5.05	7.88	7.46E-06	0.004	0.045	0.46	10.28	228.59	4.3	5.95
4	1.94E-04	0.057	1.988	10.97	359.38	2549.28	22.43	27.68	1.85E-05	0.009	0.382	4.665	106.75	2166	31.03	57.46

Figure 5: Statistics of Local & bucket errors compared with global error over 5 runs for 4 grid-hard instances having w=55 with $i-$bound=20, where h=w, # buckets trained, $\#NB = 308$ for two different scales of smaples sizes. *test wmse* is the w.m.s.e of the learned NN over the test set; *local bucket error* is the average L1 error for $log\lambda$ approximations over all buckets; *estimated bounds* is the bound obtained in eq 9; *empirical error* is the average global error over 5 runs.

i-bound=20			h=3*w				h=5*w			
Id	#v	w	#N_{avg} (10^3)	standard deviation	error	t(h)	#N_{avg} (10^3)	standard deviation	error	t(h)
1	888	33	218	0.25	1.11	7.5	437	0.17	0.34	14.3
3	871	35	259	0.89	**3.51**	10.4	518	1.57	**1.42**	18.3
6	693	28	149	0.6	2.61	5	298	0.51	0.68	8.2

(a) pedigree

i-bound=20			h=w				h=w			
Id	#v	w	#N_{avg} (10^3)	standard deviation	error	t(h)	#N_{avg} (10^3)	standard deviation	error	t(h)
1	1600	55	60	7.98	24.01	5.4	150	5.17	8.12	9.1
2	1600	55	60	6.4	**21.2**	5.3	150	4.12	**6.4**	9.2
4	1600	55	60	1.8	5.1	5.2	300	1.23	2.08	14
7	1600	114	209	3.92	16.6	13.9	392	1.01	6.67	22.7

(b) Grid-hard

i-bound=10			h=w				h=3*w			
Id	#v	w	#N_{avg} (10^3)	standard deviation	error	t(h)	#N_{avg} (10^3)	standard deviation	error	t(h)
1	100	21	25	1.21	3.51	0.35	42	0.88	1.38	0.4
2	100	13	11.6	0.7	**1.1**	0.055	19	0.4	**0.42**	0.07

(c) Grid-easy

i-bound=20			h=3*w				h=5*w			
Id	#v	w	#N_{avg} (10^3)	standard deviation	error	t(h)	#N_{avg} (10^3)	standard deviation	error	t(h)
1	40	21	147	0.21	0.23	0.45	489	0.09	0.2	1.1
2	42	22	163	0.8	**0.78**	0.46	544	0.33	**0.55**	1.5
4	44	23	147	1.15	1.58	0.44	489	0.63	0.62	1.13
5	42	22	164	2.57	2.79	0.48	13	0.82	1.12	0.16
6	44	23	182	3.2	4.69	0.59	600	0.78	2.87	1.7

(d) DBN

Figure 6: Performance of NeuroBE when increasing # samples &/or NN complexity. N_{avg}: average samples, $t(h)$: average time, $Error$: global error (reported average and standard deviation over 5 runs; except instance 7 from grid-hard(#runs=2)).

(what we call global error), and standard deviation (over 5 runs) and the computation time for 2 different NN architecture and their associated sample sizes. As expected we see that increasing the NN and sample sizes increases the time and accuracy for pedigrees. For grid-hard instances, we just increased the sample sizes for the same architecture where $h = w$, and observe that the average error is reduced by about a factor of 3. Instances from grid-easy and DBN show a similar improvement in performance with a larger NN and a corresponding larger training sample size. This shows that the algorithm has an anytime characteristics, as it can improve its performance by controlling the size of the approximating NN matched by a suitable sample size for training.

Conclusion & Future Work

In this work we advance our earlier theme of using the power of Neural networks to approximate the class of bucket-elimination algorithms that is at the heart to probabilistic reasoning. The central aim of our paper is to improve the *efficiency* of such schemes by enhancing its NN training aspect. *NeuroBE*'s main new design feature is that it customizes the NN architectures and the training samples to the messages, thus achieving higher accuracy often with less time when compared to DBE. We presented *NeuroBE* and illustrated, on challenging instances over three benchmarks that it can be far more accurate and requires less time compared with its earlier version of *Deep Bucket Elimination DBE* and is also superior to *weighted mini-bucket WMB* that cannot improve its accuracy once their memory is exhausted.

Future Work. While $NeuroBE$ adjusts the NN architectures according to a bucket's scope, it keeps the #layers fixed and seeks for user inputs for such variations. We will explore how to fine-tune NN capacity dynamically during training. Since the number of buckets trained have a direct effect on $NeuroBE$'s accuracy and time performance, we will explore reducing the number of trained functions by training a single function per union of buckets, which yield a cluster in a tree-decomposition (Dechter 2013). This can significantly reduce the number of trained functions at the cost of more time for sample generation, a trade-off we plan to study. We will also explore the task of parameter sharing and thus, training of multiple bucket functions simultaneously.

References

Abboud, R.; Ceylan, I.; and Lukasiewicz, T. 2020. Learning to reason: Leveraging neural networks for approximate

DNF counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 3097–3104.

Anthony, M.; and Bartlett, P. L. 2002. *Neural Network Learning - Theoretical Foundations.* Cambridge University Press. ISBN 978-0-521-57353-5.

Bartlett, P. L.; Harvey, N.; Liaw, C.; and Mehrabian, A. 2019. Nearly-tight VC-dimension and Pseudodimension Bounds for Piecewise Linear Neural Networks. *Journal of Machine Learning Research*, 20(63): 1–17.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*, volume 3 of *Optimization and neural computation series*. Athena Scientific. ISBN 1886529108.

Cun, Y. L.; Kanter, I.; and Solla, S. A. 1991. Eigenvalues of covariance matrices: Application to neural-network learning. *Phys. Rev. Lett.*, 66: 2396–2399.

Darwiche, A. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press.

Dechter, R. 1999a. Bucket Elimination: A Unifying Framework for Reasoning. *Artif. Intell.*, 113(1-2): 41–85.

Dechter, R. 1999b. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113: 41–85.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

Dechter, R. 2013. Reasoning with probabilistic and deterministic graphical models: Exact algorithms. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 7(3): 1–191.

Dechter, R.; and Rish, I. 2003. Mini-buckets: A General Scheme for Bounded Inference. *Journal of the ACM (JACM)*, 50(2): 107–153.

Heess, N.; Tarlow, D.; and Winn, J. 2013. Learning to Pass Expectation Propagation Messages. In *NIPS*, volume 26, 3219–3227.

Kask, K.; Pezeshki, B.; Broka, F.; Ihler, A. T.; and Dechter, R. 2020. Scaling Up AND/OR Abstraction Sampling. In *Proceedings of IJCAI 2020*, 4266–4274.

Kingma, D. P.; and Ba, J. 2014. Adam: A Method for Stochastic Optimization.

Liu, Q.; and Ihler, A. 2012. Belief Propagation for Structured Decision Making. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, 523–532.

Liu, Q.; and Ihler, A. T. 2011. Bounding the Partition Function using Holder's Inequality. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, 849–856.

Mateescu, R.; Kask, K.; Gogate, V.; and Dechter, R. 2010. Join-Graph Propagation Algorithms. *J. Artif. Intell. Res. (JAIR)*, 37: 279–328.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann.

Pollard, D. 1984. *Convergence of Stochastic Processes*. Springer-Verlag.

Razeghi, Y.; Kask, K.; Lu, Y.; Baldi, P.; Agarwal, S.; and Dechter, R. 2021. Deep Bucket Elimination. In Zhou, Z.-H., ed., *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, 4235–4242. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The Graph Neural Network Model. *Trans. Neur. Netw.*, 20(1): 61–80.

Vapnik, V. N. 1999. *The Nature of Statistical Learning Theory*. Springer, second edition. ISBN 0387987800.

Yedidia, J. S.; Freeman, W. T.; and Weiss, Y. 2000. Generalized Belief Propagation. In *(NIPS) 2000*, 689–695. MIT Press.

Yoon, K.; Liao, R.; Xiong, Y.; Zhang, L.; Fetaya, E.; Urtasun, R.; Zemel, R. S.; and Pitkow, X. 2018. Inference in probabilistic graphical models by Graph Neural Networks.

Appendix

Estimating the pseudo-dimension of a NN:

In our work, we use NN architectures with ReLU activation functions. To construct a NN with L layers and a variable #hidden-units per layer to model a specific bucket message λ_c, we pick the rule $h = b * w_c$ where w_c is the width and b is a constant. By doing this, the #parameters in the NN is :

$$|\theta_c| = (L-1) * b^2 * w_c^2 + b * w_c^2 + (L+1) * b * w_c + 1 \quad (7)$$

We make use of the lower bound of pseudo-dimension for NNs with ReLU activation functions from the work in (Bartlett et al. 2019) to get:

$$\rho_c = |\theta_c| * Llog(|\theta_c|/L) \quad (8)$$

By substituting (7) in (8) and ignoring all linear in w_c terms we get that $\rho_c(w_c)$ can be dominated by:

$$\rightarrow \rho_c(w_c) \propto (L * b * w_c)^2 log[(b * w_c)]$$

Estimating error in partition function:

Theorem. 1 *Let B_c be a bucket in a bucket chain along an ordering d; let B_c contain the original functions as ϕ_c and μ_{c+1} as the message passed to it from the previous bucket; let λ_c be the (global) exact message generated in B_c, μ_c^* be the local exact message in B_c and $\mu_c = APP(\mu_c^*)$ its approximation (e.g., by a trained neural network). Let $E_c = ln\mu_c^* - ln\mu_c$ and and $\epsilon_c = max_{B_c}|E_c|$. Then,*

$$ln\lambda_c - ln\mu_c \leq \sum_{k=2}^{n-c} \epsilon_{c+k}$$

In particular, since $\lambda_1 = Z$, the partition function

$$lnZ - ln\mu_1 \leq \sum_{k=1}^{n-1} \epsilon_{1+k} \quad (9)$$

Proof. We will next derive the recursion, starting at the first processed bucket B_n and going down in order. Remember throughout that $\ln\mu^*_{n-i} = \sum_{X_{n-i}} ln(e^{ln\phi_{n-i}+ln\mu_{n-i+1}})$

For B_n $\lambda_n = \mu^*_n$, therefore

$$ln\lambda_n - ln\mu_n = ln\mu^*_n - ln\mu_n = E_n$$

For B_{n-1}, by definition

$$ln\lambda_{n-1} - ln\mu_{n-1} = ln \sum_{X_{n-1}} e^{ln\phi_{n-1}+ln\lambda_n} - ln\mu_{n-1}$$

Substituting $ln\lambda_n$ from B_n

$$= ln \sum_{X_{n-1}} e^{[(ln\phi_{n-1}+ln\mu_n)+E_n]} - ln\mu_{n-1}$$

$$= ln[\sum_{X_{n-1}} e^{(ln\phi_{n-1}+ln\mu_n)} e^{E_n}] - ln\mu_{n-1}$$

If $max_{scope(\mu^*_n)}|E_n| = \epsilon_n$, then,

$$\leq ln[e^{\epsilon_n} \sum_{X_{n-1}} e^{(ln\phi_{n-1}+ln\mu_n)}] - ln\mu_{n-1}$$

$$\leq \epsilon_n + ln \sum_{X_{n-1}} e^{(ln\phi_{n-1}+ln\mu_n)} - ln\mu_{n-1}$$

Since $ln \sum_{X_{n-1}} e^{ln\phi_{n-1}+ln\mu_n} = ln\mu^*_{n-1}$ we get

$$ln\lambda_{n-1} - ln\mu_{n-1} \leq \epsilon_n + ln\mu^*_{n-1} - ln\mu_{n-1} \quad (10)$$

or equivalently,

$$ln\lambda_{n-1} - ln\mu_{n-1} \leq \epsilon_n + E_{n-1} \quad (11)$$

Moving to B_{n-2}, by definition:

$$ln\lambda_{n-2} - ln\mu_{n-2} = ln \sum_{X_{n-2}} e^{ln\phi_{n-2}+ln\lambda_{n-1}} - ln\mu_{n-2} \quad (12)$$

Substituting $ln\lambda_{n-1}$ from Eq. (10) we get

$$ln\lambda_{n-2} - ln\mu_{n-2} \leq ln \sum_{X_{n-2}} e^{ln\phi_{n-2}+[ln\mu_{n-1}+\epsilon_n+E_{n-1}]} - ln\mu_{n-2}$$

$$\leq ln \sum_{X_{n-2}} e^{ln\phi_{n-2}+\mu_{n-1}} e^{\epsilon_n+E_{n-1}} - ln\mu_{n-2}$$

Taking $\max_{scope(\mu^*_{n-1})} E_{n-1} = \epsilon_{n-1}$,

$$\leq lne^{\epsilon_n+\epsilon_{n-1}} \sum_{X_{n-2}} e^{ln\phi_{n-2}+\mu_{n-1}} - ln\mu_{n-2}$$

$$\leq \epsilon_n + \epsilon_{n-1} + ln \sum_{X_{n-2}} e^{ln\phi_{n-2}+\mu_{n-1}} - ln\mu_{n-2}$$

$$\leq \epsilon_n + \epsilon_{n-1} + ln\mu^*_{n-2} - ln\mu_{n-2}$$

yielding,

$$ln\lambda_{n-2} - ln\mu_{n-2} \leq E_{n-2} + \epsilon_{n-1} + \epsilon_n \quad (13)$$

Moving to bucket B_{n-3}, by definition

$$ln\lambda_{n-3} - ln\mu_{n-3} = ln \sum_{X_{n-3}} e^{ln\phi_{n-3}+ln\lambda_{n-2}} - ln\mu_{n-3}$$

Substituting for λ_{n-2} from Eq. (13) we get with some algebra

$$ln\lambda_{n-3} - ln\mu_{n-3}$$
$$\leq ln \sum_{X_{n-3}} e^{ln\phi_{n-3}+[ln\mu_{n-2}+E_{n-2}+\epsilon_{n-1}+\epsilon_n]} - ln\mu_{n-3}$$

yielding

$$ln\lambda_{n-3} - ln\mu_{n-3} \leq E_{n-3} + \epsilon_{n-2} + \epsilon_{n-1} + \epsilon_n$$

and so on. Clearly the emerging expression for bucket B_c is

$$ln\lambda_c - ln\mu_c \leq E_c + \epsilon_{c+1} + \epsilon_{c+2} + ... \quad (14)$$

or,

$$ln\lambda_c - ln\mu_c \leq E_c + \sum_{k=0}^{n-c-1} \epsilon_{c+1+k} \quad (15)$$

The general transition from $n-i$ to $n-i-1$ can be easily followed to complete the inductive proof.

Assuming that we control the derivation of μ_c for each B_c to ensure that $E_c = ln\mu^*_c - ln\mu_c \leq \epsilon_c$ and substituting in the expression we get from Eq. (15) that

$$ln\lambda_c - ln\mu_c \leq \epsilon_c + \sum_{k=0}^{n-c-1} \epsilon_{c+1+k} \leq (n-c+1) * \epsilon \quad (16)$$

□