Deep Bucket Elimination

Yasaman Razeghi , **Kalev Kask** , **Yadong Lu** , **Pierre Baldi** , **Sakshi Agarwal** and **Rina Dechter**
University of California, Irvine
{yrazeghi, kkask, yadongl1, pfbaldi, sakshia1, dechter}@ics.uci.edu

Abstract

Bucket Elimination (BE) is a universal inference scheme that can solve most tasks over probabilistic and deterministic graphical models exactly. However, it often requires exponentially high levels of memory (in the induced-width) preventing its execution. In the spirit of exploiting Deep Learning for inference tasks, in this paper, we will use neural networks to approximate BE. The resulting *Deep Bucket Elimination (DBE)* algorithm is developed for computing the partition function. We provide a proof-of-concept empirically using instances from several different benchmarks, showing that *DBE* can be a more accurate approximation than current state-of-the-art approaches for approximating BE (e.g. the mini-bucket schemes), especially when problems are sufficiently hard.

1 Introduction

Probabilistic graphical models, including Bayesian networks and Markov random fields, provide a framework for information representation and reasoning [Pearl, 1988; Darwiche, 2009]. *Bucket Elimination (BE)* [Dechter, 1999] is a universal exact algorithm for probabilistic inference. It is a variable elimination algorithm that can answer queries, ranging from constraint satisfaction, to pure combinatorial optimization (e.g., Most Probable Explanation (MPE/MAP)), and weighted counting (Partition Function, Probability of Evidence, Solution Counting). Even the most challenging mixed inference tasks, involving both optimization and summation, such as computing the Marginal Map or the Maximum Expected Utility over an influence diagram can be addressed with *BE* [Dechter, 2013]. Bucket Elimination algorithms are time and space exponential in the *induced-width* of the underlying dependency primal graph of the model. Thus when the induced-width is too high these algorithms cannot be executed. In this work, we propose to address this fundamental problem using Neural Network (NN) approximation.

To better understand how the induced width affects *BE*, note that the central operation of *BE* is processing the bucket function of each variable, one at a time, along a reverse ordering *starting from the last variable in the ordering and going towards the first*. Processing a bucket involves combining all its functions by a combination operator (often product) and then *eliminating* the bucket's variable yielding the output bucket's function λ (also called a message). The arguments of λ (called its scope) are the set of all the bucket's variables excluding itself. The largest scope size of all the buckets corresponds to the induced width of the model given the ordering. The bucket's function λ is then placed in a *parent bucket* in the ordering which, among all of its scope variables is the closest (i.e., latest). However, if the induced-width is too high then a bucket's function is too large to fit memory and the computation becomes infeasible. Therefore, significant research went into bounding *BE*. This includes the (weighted) mini-bucket scheme [Dechter and Rish, 2003; Liu and Ihler, 2012] and generalized belief propagation schemes [Yedidia *et al.*, 2000; Mateescu *et al.*, 2010].

We focus here on BE for the sum-product task, yet the scheme that we will present is applicable to pure optimization and to mixed, max-sum queries as well. In probabilistic graphical models, the sum-product problem, which includes the *partition function* and the *probability of evidence* as special cases, has many applications in areas such as computational protein design, genetic linkage analysis, and scheduling [Fishelson *et al.*, 2005; Sontag *et al.*, 2008].

Providing good approximations to *BE* is important not only because it generates an answer to a query, but primarily because it compiles a structure and a set of messages that can be used to answer multiple queries (e.g., the probability of evidence for various evidence variables [Darwiche, 2009]). Also because, the messages can be used as building blocks for generating heuristics for search or for providing good proposal distributions for sampling, to further improve performance. We will therefore consider and evaluate Deep Bucket Elimination (*DBE*) within the class of *approximate BE* schemes.

Contributions We present a novel algorithm, *Deep Bucket Elimination (DBE)*, that addresses the memory bottleneck of bucket elimination by training NNs to approximate the bucket-functions or messages. We provide an analysis of our algorithm and compare *DBE* against one of the most powerful approximations of bucket-elimination, the weighted mini-bucket scheme (*WMB*). Our empirical results show that *DBE* is overall significantly more accurate than *WMB* especially on hard instances and even when the latter is provided the most

generous memory resources feasible.

While *DBE* is not yet competitive time-wise, as it requires training of many NNs, it can yield far more accurate approximations of *BE* compared against other bucket-elimination approximations which *cannot improve their performance even when given more time*. Therefore, at this initial exploration stage, we focus more on *DBE's* accuracy, leaving speed optimization issues for followup studies. We provided the source code to reproduce the results of this paper at https://github.com/dechterlab/DBE.

Related work As noted, approximating and bounding the *Bucket Elimination* algorithm 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 (WMB)*, augmented with message-passing cost-shifting [Liu and Ihler, 2011]. Those schemes also extend into iterative versions such as generalized belief propagation (IBP, IJGP) [Mateescu *et al.*, 2010; Liu and Ihler, 2012]. We therefore judge *DBE* within this class of algorithms.

The approach we take here is to approximate *BE*'s messages by exploiting *Deep Learning (DL)* [Goodfellow *et al.*, 2016; Baldi, 2020], leveraging the well known universal approximation properties of neural networks (NN) [Hornik *et al.*, 1989; Cybenko, 1989; Baldi, 2020]. Our idea is closest in spirit to the Neuro-Dynamic Programming scheme as outlined in [Bertsekas and Tsitsiklis, 1996] where the cost-to-go functions generated by dynamic programming (that parallel the bucket's messages) can be approximated by neural networks. 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.

Our scheme can also fit within the unifying framework of structural message passing (SMP) [Gogate and Domingos, 2013] for approximating inference algorithms exploiting functions' structures in messages (e.g., ADD/AOMDD [Mateescu *et al.*, 2014] and sparse hash tables). These are defined relative to message passing algorithms over cluster-graphs.

At first, our approach may seem closely related to Graph Neural Networks [Baldi, 2020; Z and Savelsbergh, 1999; Scarselli *et al.*, 2009; Yoon *et al.*, 2018; Heess *et al.*, 2013] which use message-passing architectures exploring the underlying graph structure of the problem, however our scheme differs, significantly. In particular, we confine our learning to within each problem instance *only*.

2 Background

A graphical model, such as a Bayesian or a Markov network [Pearl, 1988; Darwiche, 2009; Dechter, 2013] 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. 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 $Z = \sum_X \prod_\alpha f_\alpha(X_\alpha)$.

Algorithm 1: [Deep] Bucket Elimination (DBE)

Input: Graphical model $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{F})$, Ordering $d = X_1, ..., X_n$, i-bound i, error bound ϵ,
Output: the partition function constant and bucket messages

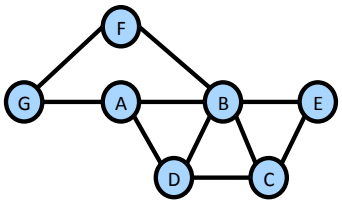
1 **foreach** *p from n to* 1 **do**
2 (Initialize buckets) put all unplaced functions mentioning X_p in B_p.
3 **foreach** *p from n to* 1 **do**
4 Let X_a be the closest ancestor variable of X_p in d that is in bucket B_p; if none exists, let $a = 0$.
5 Formulate the bucket function: $\lambda_{p\rightarrow a} \leftarrow \sum_{X_p} \prod_{f_\alpha \in B_p} f_\alpha$
6 If $width(X_P) \leq i$ then $\lambda_{\Phi,p\rightarrow a} \leftarrow \lambda_{p\rightarrow a}$,
7 else
8 $\lambda_{\Phi,p\rightarrow a} \leftarrow$ approximate-NN ($\lambda_{p\rightarrow a}$, ϵ)
9 Put $\lambda_{\Phi,p\rightarrow a}$ in B_a
10 $Z = \prod_{\lambda_\Phi \in B_0} \lambda_\Phi$
11 **return** *Z and All* λ_Φ*-messages generated*

Bucket Elimination Given a variable ordering d, *BE* (presented in Algorithm 1) processes variables one by one with respect to the reverse ordering. For the next variable X_p, it considers all the functions in bucket B_p. This includes the original functions in the graphical model as well as the messages created by processing previous variables (we do not distinguish between the different functions in step 5). It then marginalizes X_p out from the product of functions in B_p generating a new, so called, *bucket function* or message, denoted $\lambda_{p\rightarrow a}$, or λ_p for short.

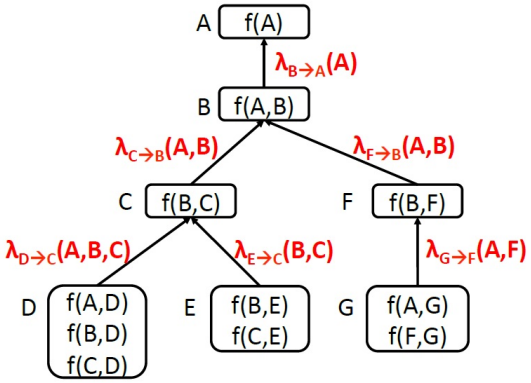
$$\lambda_{p\rightarrow a} = \sum_{X_p} \prod_{f_\alpha \in B_i} f_\alpha \tag{1}$$

where X_a is the latest variable in λ's scope along the ordering (constants are placed in B_1), called its parent bucket. The λ function is placed in the bucket of X_a, B_a. 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. **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. In this particular example $F = \{f(A), f(A,B), f(A,D), f(A,G), f(B,C), f(B,D), f(B,E), f(B,F), f(C,D), f(C,E), f(F,G)\}$.

Bucket-Elimination can be viewed as a 1-iteration message-passing algorithm along its *bucket-tree* (bottom-up). The nodes of the tree are the different buckets. Each bucket of



(a) A primal graph.



(b) Bucket elimination example

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

a variable contains a set of the model's functions depending on the given order of processing (see Algorithm 1). There is an arc from bucket B_p to a parent bucket B_a, if the function created at bucket B_p is placed in bucket B_a. 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 can be computed as a graph parameter based on the ordered primal graph [Dechter, 2013]. The induced width is the size of the largest number of variables, in the scope of any message. Clearly, *BE* becomes impractical if the induced-width is large. We denote by $scope(f)$ the set of arguments of function f and for a bucket B, scope(B) is the number of variables in B when processed (which is its induced-width + 1).

3 Deep Bucket Elimination

Algorithm 1 presents *DBE*. Note that *DBE* is identical to *BE* except that, when the scope size of a bucket's message to be generated is beyond a given i-bound, it approximates the bucket's function by training a neural network as described in lines 6-8. As before, the central operation of *BE* is processing the bucket of each variable one at the time. However, when the bucket's functions are too large, *BE* cannot be executed. To overcome this limitation, we approximate the bucket's function by training a neural network architecture having a manageable size, aiming towards achieving an error bounded by a given ϵ. For example, in Figure 1, if we use an i-bound $i = 2$ with *DBE*, then instead of sending an exact function from the bucket of D to the bucket of C, $\lambda_{D\to C}(A,B,C)$, we will send a compact NN approximation $\mu_{\Phi,D\to C}(A,B,C)$, as we will describe next.

Approximating a bucket message in *DBE* is carried out by training a neural network using the *approximate-NN* in Algorithm 2. Given a target function $\lambda(S)$ (where $S = scope(\lambda)$) and an error bound ϵ, the training scheme first generates a given number of samples from the function that serve as examples to train the neural network until either the ϵ error bound is obtained, the validation set error increases for two consecutive epochs (early stopping criteria), or until a cap on the number of training iterations (#epochs) is reached. Each sample is a pair $(s, \lambda(s))$, where s is a configuration of S. Ideally, the number of samples needed can be tailored to the size of the neural network's architecture (i.e., number of parameters), which in turn should be tuned to the complexity of the function λ, and in particular to the size of its scope. In principle, neural networks can approximate any reasonable function [Hornik *et al.*, 1989; Cybenko, 1989; Baldi, 2020]. So, the main question is how to fit a NN's architecture to approximate a function, while maintaining a desirable error bound, and how these local function errors translate into a global error. Should we use a single or multiple architectures per problem instance or across instances of the same benchmark? To reduce the space of design choices we commit to a single NN architecture for all buckets. In summary, the main design questions are: 1. How to select an effective NN architecture for a given benchmark that accommodates effective learning? 2. Given an architecture, how should we generate samples? should we aim for a particular distribution? 3. How many samples do we need? Following we address the aforementioned questions.

Algorithm 2: approximate-NN(λ, ϵ)

Input: λ function on a set of variables X, ϵ bounds the bucket's error, set the $\#epochs$ a bound on the number of epochs, NN, the neural network structure, ns: number of samples

Output: $\mu_\Phi(x)$, the trained neural network

1 train_samples, val_samples ← generate-samples(λ, ns)
2 p=1
3 **while** *val_error* $\geq \epsilon$ & $\neg early\_stopping(val\_error)$ & $p \leq$ *#epochs* **do**
4 $\quad \mu_\Phi \leftarrow train(NN, p, train\_samples)$
5 $\quad$ *val_error* $\leftarrow test(NN, p, val\_samples)$
6 $\quad p \leftarrow p+1$
7 **return** μ_Φ *and a bound on the error*

Selecting the Architecture For benchmark instances having no determinism (zero as the function value), we first started with training a feedforward neural network with fully connected layers. We experimented with several activation functions, such as ReLU, Sigmoid, and Tanh activation functions for each layer. We found that ReLU activation yields stronger performance. The number of layers and the number of units in each layer were tuned according to the complexity of the instance (details in the Empirical Evaluation sec-

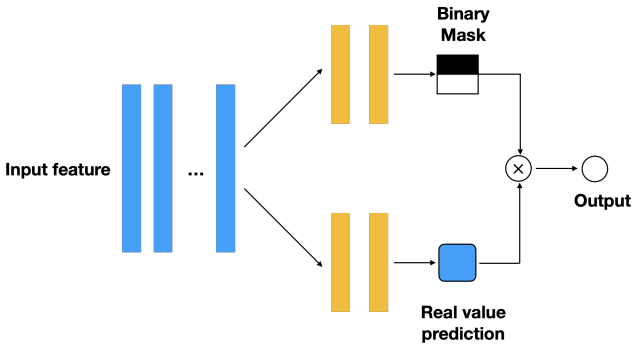


Figure 2: The structure of the MaskedNet.

tion). While the fully connected network works well when the function has no determinism, it does not work well with determinism. We found that the fully connected network with ReLU were unable to correctly predict deterministic outputs. Therefore for such benchmarks, we used a MaskedNet. The MaskedNet first predicts whether the output is inconsistent (zero), and if not, it predicts the value. The structure of the MaskedNet is described in Figure 2. The input is sent to several fully connected layers to obtain a feature vector which summarizes the high level representation of the input. Then this feature vector is sent to two sister networks: 1) a network that outputs a binary mask which is responsible for determining whether the final output is zero, and 2) a network responsible for predicting the target value of the Bucket's function. The activation functions of the final layer for the first and second sub-networks are the logistic function and the softplus function, respectively. The outputs from the two sister networks are multiplied together to get the final output of the MaskedNet. In our experiments, we found that the MaskedNet effectively decreases the error of the Bucket's function approximation compared to a plain feedforward network in the case of problems with determinism.

Sample Generation Given a generic bucket B of a variable X and a target bucket-function $\lambda(S)$, where S is the scope of the bucket's output function, we generate a required number, ns, of training examples $(s, \lambda(s))$ by sampling configurations $\{S = s\}$ uniformly at random from the domain of scope S. Each generated configuration s should be paired with its function value $\lambda(s)$. We consult the definition of the function bucket in Eq. 1 (see also step 5 of Algorithm 1). Specifically, given a configuration s over a bucket B, $S = scope(B) \setminus \{X\}$, for each $x \in D_X$ we compute the product $\psi(s, x) = \prod_{f \in B} f(s, x)$ (i.e., a product of constants) and then sum over the different values of x:

$$\lambda(s) = \sum_{x} \psi(s, x).$$

Therefore, if we have r functions in a bucket, for each we need to perform up to k function evaluations, where k is the maximum variable's domain size, yielding a total of $O(rk)$ function evaluations per sample s. For a tabular function the evaluation is a simple table lookup. However, if the bucket contains a trained NN, it will take longer to evaluate it.

Proposition 1. *The time to generate m samples in a bucket having r functions, when k bound the domain size and t_{NN} bounds the NN evaluation time is $O(m \cdot r \cdot k \cdot t_{NN})$.*

Training. Once the samples are available, we split them into training (80%), validation (10%), and test sets (10%). Given a NN architecture, it is then trained to minimize the average mean square error on the training samples:

$$\frac{1}{ns_{train}} \sum_{n=1}^{ns_{train}} (\lambda_\Phi(s_n) - \lambda(s_n))^2$$

where ns_{train} is the number of training samples and s_n is the n^{th} sample in the training set. We train the network using the Adam optimizer [Kingma and Ba, 2014] with a learning rate of 0.001 and a batchsize of 256. In our experiments, Adam optimizer tends to converge faster and is more stable than the SGD optimizer. The search of hyper-parameters such as learning rate and batchsize is carried out using the Sherpa software. [Hertela *et al.*, 2020]. The NN is trained up to a #epochs bound of 100. The model is evaluated on a holdout validation set after each epoch. We stop training either when the validation error is below ϵ, or when the bound #epochs is reached. We perform early stopping [Prechelt, 1996] if the validation error starts to increase for two consecutive epochs. Subsequently, we evaluate the trained model on the test set to ensure the performance of the model.

4 Complexity Analysis

We can immediately observe the following properties of *DBE(i)*:

1. The scope S of the NN functions of trained buckets satisfies $|S| \geq i + 1$.
2. The number of buckets trained, $\#NB(i)$, can be computed apriori from the graph along the ordering and it corresponds to the number of variables whose induced-width is larger or equal to i.
3. Exactly computed buckets include NN functions only when their scopes is subsumed in the scope of a bucket processed earlier (namely it is later in the ordering).

We use $T_{NN}(m)$ to denote the time bounds for training a NN approximating a discrete function regardless of its number of arguments when using m samples.

Theorem 1 (complexity of *DBE(i)*)**.** *Given a problem having n variables, r functions, domain size k, then* DBE(i)*, when using a neural network NN whose size is bounded by $|NN|$, and whose evaluation time is bounded by t_{NN}, has time complexity*

$$O(n \cdot T_{NN}(m) + n \cdot t_{NN} \cdot r \cdot k^{i+1}))$$

and space complexity

$$O(nk^i + n \cdot |NN|).$$

When $T_{NN} >> k^i$ and when $\#NB(i)$ is the number of trained buckets, $DBE(i)$ time is also bounded by

$$O(\#NB(i) \cdot T_{NN}(m) + n \cdot r \cdot t_{NN} \cdot k^{i+1}))$$

and its memory is

$$O(\#NB(i) \cdot |NN|).$$

Proof. Exact buckets (that produce a tabular representation) having at most i variables can be processed in time $O(r \cdot k^{i+1})$. But, if they include a NN, then computing an exact bucket is $O(r \cdot t_{NN} \cdot k^{i+1})$ time-wise.

When approximating a bucket, we need to generate m samples from the bucket's function which is $O(m \cdot r \cdot k \cdot t_{NN})$ time (Proposition 1). The training time is $T_{NN}(m)$, yielding a total of $O(T_{NN}(m) + m \cdot r \cdot k \cdot t_{NN})$. Given $\#NB(i)$ trained buckets the total processing time is

$$O(\#NB(i) \cdot (T_{NN}(m) + m \cdot r \cdot k \cdot t_{NN}))+$$

$$(n - \#NB(i))(r \cdot t_{NN} \cdot k^{i+1}))$$

which, when assuming that $T_{NN} >> k^i$ and $m \leq k^i$ simplifies to

$$O(\#NB(i) \cdot T_{NN}(m) + n \cdot r \cdot t_{NN} \cdot k^{i+1}))$$

yielding the time claims.

Regarding memory, a bucket can include tabular functions (with size $O(k^i)$), and NN functions (with size $O(|NN|)$). Therefore the memory requirement is $O((n - \#NB(i))k^i + \#NB(i) \cdot |NN|)$, yielding the memory claims.

5 Empirical Evaluation

Algorithms We ran experiments comparing *DBE* against the weighted mini bucket *WMB* scheme [Dechter and Rish, 2003; Liu and Ihler, 2012]. We use *WMB* as our baseline because this is one of the strongest *BE* schemes that approximates buckets functions. Algorithms in this class, such as generalized BP [Mateescu *et al.*, 2010; J. S. Yedidia and Weiss, 2005] will be included in our future work. Other anytime solvers for the partition function, e.g. [Gogate and Dechter, 2011; Broka *et al.*, 2018; Kask *et al.*, 2020], are not included in the current comparison because they are not *approximations of BE* but rather schemes that build upon such approximations and thus can benefit from *DBE* as well.

Both *WMB* and *DBE* use the i-bound parameter. For *WMB* it is well known that higher i-bounds generally lead to more accurate bounds but at the cost of more time and memory. For *DBE* we observed immediately that higher i-bounds tend to improve *both* accuracy and time because of the reduced number of trained buckets $\#NB(i)$ as a function of i. Consequently, we focused primarily on reporting the highest feasible i-bound for both schemes. As noted before, *WMB(i)*'s accuracy is bounded by the highest feasible i-bound (around 20) while the estimate by *DBE* is more flexible memory-wise and can yield more accuracy in an anytime fashion.

Benchmarks We carried our experiments on instances selected from three well-known benchmarks from the UAI repository used in [Kask *et al.*, 2020] such as grids (vision domain), Pedigrees (from 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 12 instances from Grids, with easy ones (i.e., width 20-30) and hard ones (i.e., 1600 variables, width 55), 7 from pedigrees, which posses high level of determinism and 6 from DBNs, totalling 25 instances.

Performance measures and methodology We evaluate the performance using: $error = |log_{10}Z^* - log_{10}\hat{Z}|$ where $\hat{Z}$ is the generated estimate of the partition function, Z, and Z^* is reference value. When the exact Z is not available (for hard Grid benchmark), Z^* is a surrogate to Z, which is obtained from [Kask *et al.*, 2020]. Their estimate is obtained using an advanced sampling scheme for a duration of $100 * 1hr$. We also converted solvable problem instances into hard ones, by selecting a bad, high induced-width, variable ordering. We used this methodology for the Pedigree benchmark, since all Pedigree instances can be solved exactly.Hence, we were able to show the performance on instances having high induced width while having access to the exact Z value for reference.

Our experiment used a fixed number of training samples throughout. Clearly tuning #samples to the function's complexity is essential if we want an effective scheme, but we leave this aspect to future explorations. We used fully connected feedforward NN for benchmarks with no determinism and MaskedNet for benchmarks with determinism to account better for the sparsity in the target. To specify the parameters of the NN, we first conducted NN architecture optimization for a single bucket function from a single selected problem instance of each benchmark. This preprocessing step was used to tune the NN structure (number of hidden units and layers) and other hyperparameters (e.g., the learning rate). During *DBE* execution, we used the optimized structure and hyperparameters for all trained buckets for all instances. In all the experiments, we used 5×10^5 samples for training the NNs with an error bound of $\epsilon = 10^{-6}$. The $\#epochs$ was bounded at 100. As explained, we are concerned primarily with accuracy (and less with time) aiming to show a proof of concept.

5.1 Results

Our results are shown in the four tables of Figure 3, one for each benchmark. The first few columns describe the problems statistics, followed by reporting the performance of *DBE*. In particular we display the number of trained buckets (#NB) and the average of mean squared error on the validation set of all of the trained buckets for each instance over 10 runs; we use this value as a representation of the local error for each trained bucket. In our experiments, the validation and test set were very close to each other. We also report the average and the smallest error over 10 runs to capture the randomness in training. Finally, we show the error obtained by *WMB* and the baseline reference Z value. For legend and description of the tables see its caption.

Grids The results for the Grid benchmarks are shown in Figures 3a (easy) and 3b (hard). For the easy problems we used a lower i-bound of 10 to facilitate the training of a relatively large number of buckets. As expected, when an instance has a low induced-width only a small number of buckets are trained (e.g. Id 2) and both schemes obtain high accuracy. As the induced-width increases, more buckets are trained yet *DBE* still obtains far higher accuracy compared

i-bound=10					DBE						WMB	ref Z
Id	name	k	#v	w	Arch	#NB	avg val mse	statistics on error over 10 runs			error	
								stdev	avg error	smallest error		
1	grid1010f10w	2	100	21	ff-2layers, 100 hidden units each	31	7.62E-06	3.71	**4.45**	**0.89**	32	331.321
2	grid1010f10	2	100	13		8	6.04E-06	0.54	**0.703**	**0.05**	1.58	303.086
3	grid2020f2	2	400	27		114	5.90E-06	1.28	**1.98**	**0.36**	11.24	291.733
4	grid2020f10	2	400	27		114	6.41E-06	9.3	**10.04**	**1.05**	80.86	1311.98
5	grid2020f5	2	400	27		114	6.60E-06	3.1	**5.748**	**0.24**	39.44	665.12
6	grid2020f15	2	400	27		114	7.01E-06	9.81	**17.8**	**3.08**	122.91	1962.98

(a) **Grid, easy, Without Determinism**

i-bound=20					DBE						WMB	ref Z
Id	name	k	#v	w	Arch	#NB	avg val mse	statistics on error over 10 runs			error	
								stdev	avg error	smallest error		
1	grid4040f10	2	1600	55	ff-2layers, 100 hidden units each	308	9.29E-06	65.15	**97.14**	**11.81**	215.45	5490
2	grid4040f5	2	1600	55		308	9.17E-06	34.96	**39.9**	**6.28**	84.92	2800
3	grid4040f2	2	1600	55		308	7.50E-06	5.4	**7.34**	**1.2**	25.24	1220
4	grid4040f2w	2	1600	55		376	1.07E-05	20.52	**15.12**	**0.92**	32	1231
5	grid4040f15	2	1600	55		308	9.38E-06	34.2	**83.46**	**41.78**	338.2	8200
6	grid4040f15w	2	1600	55		376	1.37E-05	192.2	**220.91**	**95.23**	657.03	8230

(b) ***Grid, Hard, Without Determinism**

i-bound=20					DBE						WMB	ref Z
Id	name	k	#v	w	Arch	#NB	avg val mse	statistics on error over 10 runs			error	
								stdev	avg error	smallest error		
1	pedigree13	3	888	33	masked-net ff-3layers, 100 hidden units each	127	2.18E-02	3.374	**3.873**	**0.8927**	6.4696	-31.18
2	pedigree41	5	885	32		92	4.63E-03	0.701	**2.894**	**1.933**	4.1497	-76.04
3	pedigree51	5	871	35		120	4.85E-03	3.193	**8.662**	**4.539**	9.7624	-77.27
4	pedigree34	5	922	33		106	9.71E-03	1.191	**5.93**	**4.14**	7.0762	-64.23
5	pedigree7	4	867	34		108	8.04E-03	0.84	**6.002**	**4.628**	**6.0012**	-64.82
6	pedigree31	5	1006	30		85	9.16E-03	2.169	**5.863**	**0.0178**	12.3603	-78.52
7	pedigree19	5	693	28		43	3.76E-03	1.364	3.663	**1.5882**	2.5809	-59.020

(c) **Pedigree, Hard, With Determinism**

i-bound=20					DBE						WMB	ref Z
Id	name	k	#v	w	Arch	#NB	avg val mse	statistics on error over 10 runs			error	
								stdev	avg error	smallest error		
1	rbm20	2	40	21	ff-2layers, 100 hidden units each	20	4.59E-06	0.173	0.221	0.034	**0.0007**	58.53
2	rbm21	2	42	22		22	8.81E-06	0.281	**0.382**	**0.05**	6.3913	63.15
3	rbm22	2	40	21		20	9.34E-06	0.35	**0.47**	**0.14**	8.6549	66.55
4	rbm-ferro20	2	44	23		24	1.97E-06	0.333	0.423	0.12	**0.005**	151.16
5	rbm-ferro21	2	42	22		22	1.78E-06	0.662	**1.061**	**0.105**	**1.984**	152.62
6	rbm-ferro22	2	44	23		24	1.97E-06	0.5	2.13	1.3	**0.517**	166.11

(d) **DBN, meduim, Without Determinism**

Figure 3: Results on performance of *DBE* against *WMB*. k:domain size, $\#v$:variable numbers, w:induced width, $Arch$:the architecture of the NN, $\#NB$: number of buckets that are trained with NNs, $avg\ val\ mse$: average mean square error for validation set over trained buckets and 10 runs, $error$: L1 error for referenced and estimated $log(Z)$ (reported smallest (minimum), average, and standard deviation over 10 runs for *DBE*). *Note: Here, referenced Z is approximated by [Kask *et al.*, 2020]

Grid					DBE i-bound 20			DBE i-bound 15		
Id	name	H	#v	w	#NB	error	T(h)	#NB	error	T(h)
1	grid2020f10	e	400	27	31	4.12	1.196	69	18.32	3.156
2	grid2020f5	e	400	27	31	2.07	1.187	69	4.715	3.086
3	grid4040f10	h	1600	55	308	70.5	11.74	421	122.14	19.264
4	grid4040f5	h	1600	55	308	33.4	11.8	421	54.61	19.299

Figure 4: The impact of the i-bound, H:hardness, e:easy, h:hard, $\#v$:number of variables, w:induced width, $\#NB$: number of NNs, $error$: L1 error for referenced/estimated $log(Z)$, T: algorithm time.

with *WMB*. See for example instance 6 whose induced-width is 27. In this case *DBE*'s average error is 17.8 while for *WMB* the error is 122.91. On the hard instances, (Figure 3b) we used the highest possible i-bound of 20 and we see that *DBE* can achieve a far lower error than *WMB*. For all of the instances the average error of the *DBE* is far better than the WMB error.

Pedigree Pedigree results are presented in Table 3c. Since Pedigrees can be solved exactly using good variable orderings, we selected an alternative variable ordering in order to artificially create hard instances while still having access to the exact Z for reference. Again, we see that *DBE* achieved significantly smaller error than *WMB* (by a factor of more than 2 in most cases) with the same i-bound of 20. As expected, the number of trained buckets vary with the induced-width.

DBN DBN instances (see Figure 3d) all have exact solutions (obtained by other schemes), yet they are almost at the edge or utterly intractable memory-wise having widths 20-23. Here too *DBE* can achieve high accuracy, sometime significantly higher than *WMB* (instances 2 and 3 that are the hardest in this set). *DBE* still achieves low error for problems which *WMB* is almost exact (instances 1 and 4).

Buckets' Local Errors We also reported for each benchmark the average validation error we obtained per bucket function to illustrate the relationship between the ϵ used for training and the estimate of the local error obtained. We observe good correspondence except for the pedigree benchmark.

The Impact of the i-bound As noted earlier, the highest i-bound possible is around 20 depending also on the domain size. For the *WMB* scheme, both theory and practice show that higher i-bounds yield more accurate approximations most of the time [Dechter and Rish, 2003]. We initially explored several i-bounds with *DBE* and we immediately observed that for higher i-bound *DBE* achieves higher accuracy with less time, primarily because more buckets are processed exactly and thus fewer need to be trained (see complexity analysis). For illustration, see results in Figure 4 on four grid instances with $i = 20$ and $i = 15$. We thus clearly see that *DBE*'s accuracy and time is significantly better when the i-bound is higher and, as expected, that the number of buckets that need to be trained is reduced.

Time performance As noted *WMB* time performance can be order of magnitude faster (taking seconds or minutes) while *DBE* may take many hours. Clearly this is due to *DBE*'s needs to train tens to hundreds of neural networks. Yet, *DBE*'s performance can improve far beyond *WMB* even in its best performance due to its memory limit especially, on hard problem instances.

6 Conclusion and Future Work

This work brings the power of Neural Networks to approximate a class of variable elimination algorithms known as *Bucket Elimination (BE)* for probabilistic and deterministic graphical models. We present *Deep Bucket Elimination (DBE)* and show, on challenging instances from three benchmarks, that it can be far more accurate compared with *weighted mini-bucket WMB*, one of the most powerful approximations of Bucket Elimination. This holds true even when *WMB* uses the largest feasible i-bound. *DBE* can be viewed as a realization of Neuro-Dynamic Programming schemes [Bertsekas and Tsitsiklis, 1996], in the context of graphical models. That being said, *DBE* requires training of numerous NN per problem instance and thus is more time consuming than other approximation schemes of *BE*. However, *WMB* and other iterative *BE* approximations [Mateescu *et al.*, 2010], cannot improve once their memory is exhausted. We believe that the *DBE* has great potential to become more time efficient and may also extend into learning across a set of instances from the same benchmark domain.

Future work. We will explore speeding up the training in *DBE*. Many design choices we made were addressed to merely provide a proof-of-concept regarding accuracy while deferring efficiency issues to future work. For example, we use the same network architecture and the same number of samples across all the instances of a benchmark and across all their buckets, ignoring difference in their function complexity. We generated samples uniformly at random while other policies should be considered, for example sampling in proportion to the target function distribution.

One relatively straightforward idea to reduce the number of trained functions is to train a single function per union of buckets, which yield a cluster in a tree-decomposition [Kask *et al.*, 2005; 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 ideas from transfer learning [Pan and Yang, 2010] techniques; retraining a network to approximate a specific bucket function, and then fine-tuning the same network to approximate the others.

Acknowledgement

We thank our reviewers for their valuable comments. We would also like to thank Bobak Pezeshki for his help and constructive feedback on this work. This work was supported in part by NSF grants IIS-2008516.

References

[Baldi, 2020] P. Baldi. *Deep Learning in Science: Theory, Algorithms, and Applications*. Cambridge University Press, Cambridge, UK, 2020. In press.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic programming*, volume 3 of *Optimization and neural computation series*. Athena Scientific, 1996.

[Broka *et al.*, 2018] Filjor Broka, Rina Dechter, Alexander T. Ihler, and Kalev Kask. Abstraction sampling in graphical models. In *UAI 2018, 2018*, pages 632–641, 2018.

[Cybenko, 1989] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

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

[Dechter and Rish, 2003] Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM (JACM)*, 50(2):107–153, 2003.

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

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

[Fishelson *et al.*, 2005] M. Fishelson, N. Dovgolevsky, and D. Geiger. Maximum likelihood haplotyping for general pedigrees. *Human Heredity*, 2005.

[Gogate and Dechter, 2011] Vibhav Gogate and Rina Dechter. Samplesearch: Importance sampling in presence of determinism. *Artif. Intell.*, 175(2):694–729, 2011.

[Gogate and Domingos, 2013] Vibhav Gogate and Pedro M. Domingos. Structured message passing. In Ann Nicholson and Padhraic Smyth, editors, *UAI 2013, 2013*, 2013.

[Goodfellow *et al.*, 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[Heess *et al.*, 2013] Nicolas Heess, Daniel Tarlow, and John Winn. Learning to pass expectation propagation messages. In *NIPS*, volume 26, pages 3219–3227, 2013.

[Hertela *et al.*, 2020] Lars Hertela, Julian Collado, Peter Sadowskic, Jordan Ott, and Pierre Baldi. Sherpa: Robust hyperparameter optimization for machine learning. *SoftwareX*, 2020.

[Hornik *et al.*, 1989] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[J. S. Yedidia and Weiss, 2005] W.T. Freeman J. S. Yedidia and Y. Weiss. Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transaction on Information Theory*, pages 2282–2312, 2005.

[Kask *et al.*, 2005] Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying tree decompositions for reasoning in graphical models. *Artif. Intell.*, 166(1-2):165–193, 2005.

[Kask *et al.*, 2020] Kalev Kask, Bobak Pezeshki, Filjor Broka, Alexander T. Ihler, and Rina Dechter. Scaling up AND/OR abstraction sampling. In *IJCAI 2020*, pages 4266–4274, 2020.

[Kingma and Ba, 2014] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[Liu and Ihler, 2011] Qiang Liu and Alexander T. Ihler. Bounding the partition function using holder's inequality. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, pages 849–856, 2011.

[Liu and Ihler, 2012] Qiang Liu and Alexander Ihler. Belief propagation for structured decision making. In *UAI*, pages 523–532, 2012.

[Mateescu *et al.*, 2010] Robert Mateescu, Kalev Kask, Vibhav Gogate, and Rina Dechter. Join-graph propagation algorithms. *J. Artif. Intell. Res. (JAIR)*, 37:279–328, 2010.

[Mateescu *et al.*, 2014] Robert Mateescu, Rina Dechter, and Radu Marinescu. AND/OR multi-valued decision diagrams (aomdds) for graphical models. *CoRR*, abs/1401.3448, 2014.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[Pan and Yang, 2010] S.J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

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

[Prechelt, 1996] Lutz Prechelt. Early stopping-but when? In Genevieve B. Orr and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 1996.

[Scarselli *et al.*, 2009] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009.

[Sontag *et al.*, 2008] David Sontag, Talya Meltzer, Amir Globerson, Tommi Jaakkola, and Yair Weiss. Tightening lp relaxations for map using message passing. In *UAI*, pages 503–510, 2008.

[Yedidia *et al.*, 2000] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *(NIPS) 2000*, pages 689–695. MIT Press, 2000.

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

[Z and Savelsbergh, 1999] G. L. Nemhauser Z, Gu and M. W. P. Savelsbergh. Lifted flow covers for mixed 0-1 integer programs. *Mathematical Programming*, pages 439–467, 1999.