Journal of Machine Learning Research 25 (2024) 1-25 Submitted 8/23; Revised 10/24; Published 12/24

PGMax: Factor Graphs for Discrete Probabilistic Graphical Models and Loopy Belief Propagation in JAX

Guangyao Zhou[1] STANNIS@GOOGLE.COM
Antoine Dedieu[1] ADEDIEU@GOOGLE.COM
Nishanth Kumar[2] NJK@MIT.EDU
Wolfgang Lehrach[1] WPL@GOOGLE.COM
Shrinu Kushagra[1] SHRINUKUSHAGRA@GOOGLE.COM
Dileep George[1] DILEEPGEORGE@GOOGLE.COM
Miguel Lázaro-Gredilla[1] LAZAROGREDILLA@GOOGLE.COM
[1] *Google DeepMind*
[2] *Massachusetts Institute of Technology*

Editor: Alexandre Gramfort

Abstract

PGMax is an open-source Python/ JAX package for (a) easily specifying discrete Probabilistic Graphical Models (PGMs) as factor graphs; and (b) automatically running efficient and scalable differentiable Loopy Belief Propagation (LBP). PGMax supports general factor graphs with tractable factors, and leverages modern accelerators like GPUs for inference. Compared with alternative libraries, PGMax obtains higher-quality inference results with up to three orders-of-magnitude inference time speedups. PGMax interacts seamlessly with the growing JAX ecosystem, opening up new research possibilities. Our source code, examples and documentation are available at `https://github.com/google-deepmind/PGMax`.

Keywords: Probabilistic Graphical Models, Bayesian Inference, Belief Propagation, JAX

1. Introduction

Probabilistic Graphical Models (PGMs) compactly encode the full joint probability distribution of a set of random variables. PGMs are commonly specified using factor graphs (Kschischang et al., 2001), and have been successfully used in in computer vision (Wang et al., 2013), error correcting codes (McEliece et al., 1998), biology (Durbin et al., 1998), etc.

In this paper, we focus on discrete PGMs. A standard approach for performing approximate posterior inference on discrete PGMs with tractable factors[1] involves message-passing algorithms like Loopy Belief Propagation (LBP) (Pearl, 1988; Murphy et al., 1999). LBP propagates "messages" between the variables and the factors of the factor graph. However, despite several past attempts (see Section 2), there is no well-established open-source Python package that implements efficient and scalable LBP for general factor graphs. A key challenge lies in the design and manipulation of the Python data structures containing the LBP messages for supporting a large class of factor graphs with arbitrary topology and

1. By tractable factors we mean factors for which the LBP updates in Equations 6 and 9 in Appendix B can be computed in polynomial time.

different factor types (e.g., the non standard pairwise factors discussed in Section 5.2) while also guaranteeing fast and scalable inference.

In this paper, we describe PGMax, a new open-source Python package that (a) provides an easy-to-use interface for specifying a large family of factor graphs with discrete variables and tractable factors, and (b) implements efficient and scalable LBP in JAX (Bradbury et al., 2018). Compared with alternative libraries, PGMax (a) supports a larger class of tractable factors; (b) demonstrates superior inference performance in terms of quality, speed and scalability; and (c) opens up new research possibilities from its seamless interaction with the rapidly growing JAX ecosystem (Babuschkin et al., 2020).

2. Related Work

Several Python packages have been proposed for running LBP in discrete PGMs. Some examples include `pyfac`, `py-factorgraph`, `factorflow`, `fglib`, and `sumproduct`. Two of the most recent and efficient packages are `pomegranate` (Schreiber, 2018) and `pgmpy` (Ankan and Panda, 2015). As we discuss in Section 4, their LBP implementations rely on *for* loops, which are slow in Python. In contrast, PGMax introduces a novel array-based LBP implementation, which makes it faster and more scalable (see Section 5.1).

Due to the challenges in deriving an efficient LBP implementation in Python, past works resort to other programming languages. `OpenGM` (Andres et al., 2012; Kappes et al., 2013), written in C++, supports general factor graphs but is no longer maintained, while `ForneyLab` (Cox et al., 2019) and `ReactiveMP` (Bagaev and de Vries, 2022), both written in Julia, focus on conjugate state-space models and/or variational inference. These packages cannot easily interact with the vast Python scientific computing ecosystem.

Another related line of work is probabilistic programming languages (PPLs). While PPLs (van de Meent et al., 2018; Carpenter et al., 2017; Bingham et al., 2019; Salvatier et al., 2016) have appealing properties, they have limited support for undirected PGMs and discrete variables (Zhou, 2020). In contrast, PGMax is a specialized PPL that specifies discrete PGMs with tractable factors, and automatically derives GPU-accelerated LBP inference.

3. Main PGMax Features

Specification of general factor graphs: PGMax [2] handles general factor graphs with (a) arbitrary topology, (b) different factor types, and (c) variable-specific number of states (which can be heterogeneous within the same factor graph). See Section 5.2 for examples.

Efficient, scalable LBP implementation: PGMax adopts an LBP implementation using parallel message updates and damping (Pretti, 2005)— see Appendix B for details. This setup has been extensively tested in recent works (Dedieu et al., 2023; George et al., 2017; Lázaro-Gredilla et al., 2021; Lazaro-Gredilla et al., 2021; Zhou et al., 2021) on a wide range of discrete PGMs. PGMax develops a novel fully flat array-based LBP implementation (see Section 4) in JAX, and effectively leverages just-in-time compilation to run on modern accelerators like GPUs. As we show in Section 5.1, PGMax inference is up to three orders of magnitude faster than existing libraries.

2. We refer the readers to our example notebook `[link]` for a tutorial on basic PGMax usage.

Specialized inference routines for common discrete factors: To scale to large factor graphs, PGMax provides specialized, efficient, and computationally stable message updates for common discrete factors. In particular, PGMax supports three types of factors representing logical relations between discrete variables (Ravanbakhsh et al., 2016; Dedieu et al., 2023; Lazaro-Gredilla et al., 2021) for which it implements message updates with linear complexity. See Appendix B, Section B.5 for details.

Seamless interaction with JAX: PGMax implements LBP as pure functions with no side effects. This allows PGMax to seamlessly interact with the rapidly growing JAX ecosystem (Babuschkin et al., 2020). For example, we can apply JAX transformations like `jit/vmap/grad`, etc., to these functions. This allows us to run LBP in parallel over large batches, or as part of a larger end-to-end differentiable system, as discussed in Section 5.3.

Software engineering best practices: PGMax follows the best software development workflows, with enforced format and static type checking, automated continuous integration and documentation generation, and comprehensive unit tests that fully cover the codebase.

4. A Flat Array-Based LBP Implementation

LBP passes “messages” along the edges of a factor graph. It is challenging to design efficient Python data structures for storing these messages in order to support efficient and scalable inference on a large class of factor graphs. Existing Python packages (e.g. `pgmpy` and `pomegranate`) store messages in separate arrays. Their LBP implementation relies on *for* loops, which are slow in Python. In contrast, PGMax concatenates all the variables-to-factors and factors-to-variables messages of a factor graph into two separate 1D arrays, and develops a novel fully flat array-based LBP implementation. By keeping track of the global indices of variable states and the corresponding valid factor configurations, PGMax leverages the `gather/scatter` JAX primitives, which gets rid of the need for *for* loops. This guarantees efficient and scalable LBP implementation, and supports GPU acceleration.

5. Experiments

5.1 Timing Comparison with Alternative Libraries

We compare PGMax v0.6.1 with Python alternatives for maximum-a-posteriori (MAP) inference on restricted Boltzmann Machines (RBMs). We discarded `py-factorgraph` and `fglib`, which we could not get to work. `pgmpy` v0.1.25’s belief propagation uses junction tree algorithm for exact inference and is prohibitively slow. `pgmpy` additionally implements the max-product linear programming (MPLP) algorithm (Globerson and Jaakkola, 2007), which we use as a baseline, along with `pomegranate` v1.0.4’s LBP. `pgmpy` only runs on CPUs: we use it with default settings. We run `pomegranate` and PGMax for 200 LBP steps on both CPUs and GPUs—with a 0.5 damping for PGMax. We use an instance with 8 Intel(R) Xeon(R) 2.20GHz CPUs and a single NVIDIA V100 GPU. Our experiments can be reproduced using our code on PGMax GitHub `[link]`.

First, we randomly generate 50 RBMs with 24 units (12 hidden and 12 visible variables) and derive their ground truth (GT) MAP estimates via brute force. We measure the inference quality of a method by computing the energy of its MAP estimate. `pgmpy` consistently gives poor MAP estimates. PGMax (resp. `pomegranate`) recovers the GT MAP estimates for 21/50 RBMs (resp. 13/50). PGMax also reaches the lowest (best) energy for 46/50 RBMs, and a *strictly* lower energy than `pgmpy` and `pomegranate` for 26/50 RBMs.

Second, we compare the packages scalability by increasing the RBM sizes to $40, 60, \ldots, 200$ units—with an equal number of hidden and visible variables—and randomly generating 20 RBMs for each size. Again, `pgmpy`'s MPLP consistently gives poor MAP estimates, while `pomegranate` is competitive. On average, PGMax obtains the lowest (best) energy for $17.00(\pm 0.61)$ out of the 20 RBMs, and a *strictly* lower energy than `pgmpy` and `pomegranate` for $15.67(\pm 0.67)$ RBMs. Figure 3, Appendix A, displays a detailed analysis, highlighting that PGMax benefits from using a large number of 200 iterations. Figure 1 compares the average timings of each method, when using a batch size of 100. PGMax benefits from GPU acceleration and achieves significant inference speedups. This advantage is more pronounced for large models: for a RBM with 200 units, inference takes 0.59s($\pm$0.01) with PGMax on a GPU, and 408.73s($\pm$77.45) with `pomegranate` on a CPU. Finally, Figure 2, Appendix A, investigates the impact of the batch size: while PGMax is always faster on GPUs, `pomegranate` is faster on CPUs for a small batch size of 1 and as fast on GPUs as on CPUs for a large batch size of 1k. For 200 units, PGMax on GPU is three orders of magnitude faster than `pomegranate` for a small batch size of 1, and two orders of magnitude faster for a large batch size of 1k. Overall, these large speedup gains demonstrate the benefits of PGMax flat array-based LBP implementation, both in terms of inference speed and quality.

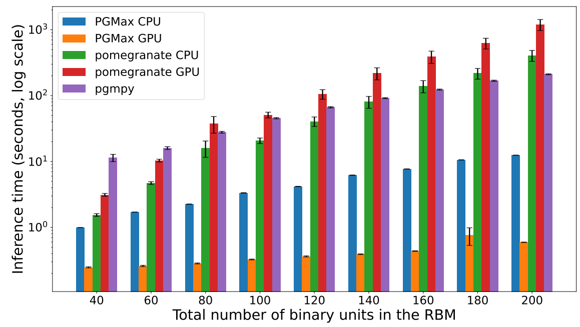

Figure 1: PGMax inference speedups on RBMs

5.2 Specifying a Large Class of Factor Graphs

One of our example notebooks `[link]` presents a PGMax implementation of max-product LBP inference for Recursive Cortical Network (RCN) (George et al., 2017). PGMax naturally supports RCN's non-standard pairwise factors: the categorical variables have a large number of states, but only a small number of explicitly enumerated joint configurations are valid. Another example notebook `[link]` uses PGMax to build a large PGM with logical factors. It then leverages the specialized message updates for logical factors to efficiently solve the 2D blind deconvolution problem from Lazaro-Gredilla et al. (2021).

5.3 Leveraging Advanced JAX Features

Two of our example notebooks illustrate how PGMax can interact with JAX advanced functionalities. The first one `[link]` draws samples from a pretrained RBM in parallel. The second one `[link]` learns the parameters of a PGM by using automatic differentiation to compute the gradient of the likelihood: the marginals are computed using LBP inference.

Acknowledgement

We thank Kevin Murphy for useful discussions about PGMax and for proofreading the paper. We also thank the anonymous reviewers for their relevant suggestions on how to improve PGMax example notebooks, and how to best compare PGMax against alternative libraries.

Contribution Statement

Guangyao Zhou came up with the flat array-based LBP implementation.

Guangyao Zhou and Nishanth Kumar implemented the initial version of PGMax, open sourced PGMax under the Vicarious repository, and wrote the first version of the paper.

Antoine Dedieu implemented logical factors with specialized inference, sped up the factor graph creation and the LBP inference runtime, led the migration of PGMax from the Vicarious repository to the DeepMind repository on GitHub, led the editing of the paper for the JMLR submission and revisions, including the writing of the technical appendices.

Guangyao Zhou and Nishanth Kumar implemented and ran the benchmarking experiments of PGMax against pomegranate and pgmpy.

Antoine Dedieu and Guangyao Zhou maintain PGMax on GitHub.

Wolfgang Lehrach improved PGMax infrastructure and provided guidance on LBP acceleration.

Shrinu Kushagra implemented the RCN example notebook.

Dileep George advised and provided funding for the project.

Miguel Lázaro-Gredilla provided guidance on LBP implementation and advised the project.

Appendix A. Additional comparisons with pomegranate

A.1 Varying the batch size

Figure 2 studies the effect of increasing the batch size from 1 to 100 and 1000 when running inference with PGMax and with `pomegranate`, both on CPU and on GPU, on a sequence of RBMs of increasing sizes. Each batch entry copies the same input: the batch size does not affect the solution returned by inference. While Figure 3 in the main paper compares timings for 200 LBP steps, here, we also include the timings for only running 20 LBP steps. Note that the time vertical axis is fixed across all the six subplots.

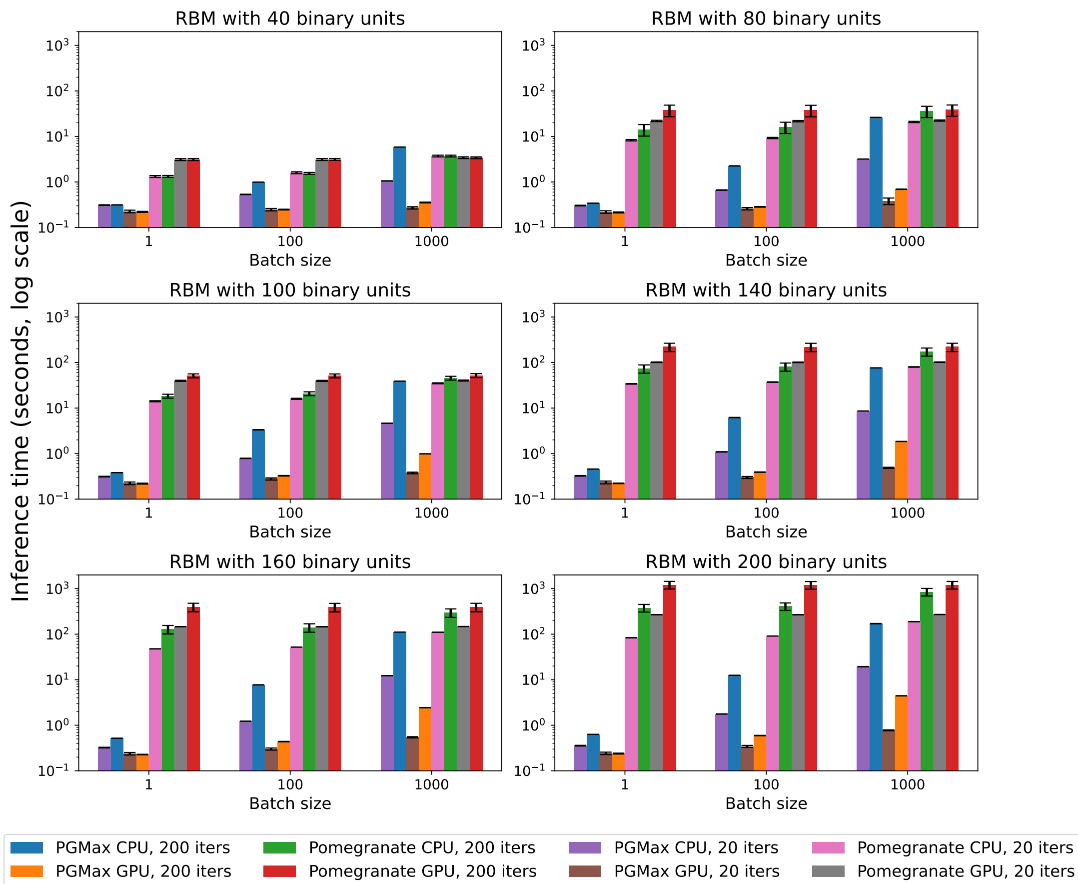

Figure 2: Inference timing comparisons when running PGMax and `pomegranate` (a) on CPU and on GPU, (b) for 20 and 200 inference steps.

We derive the following findings from Figure 2:

- PGMax runs constantly faster on GPU. While this timing difference is lower for small batch sizes, it is important for larger batches. For instance, running 200 LBP steps in parallel on 1000 RBMs with 200 unit each takes 4.43s($\pm$0.01) on a GPU and 170.51s($\pm$1.26) on a CPU.

- For RBMs with 40 units, PGMax GPU timings do not vary much as we increase the batch size. It takes 0.22s($\pm$0.01) to run 200 LBP steps on a single RBM, and 0.35s($\pm$0.01) to run 200 LBP steps in parallel on 1000 RBMs. However, for larger models with 200 units, PGMax gets slower as we increase the batch size. It takes 0.24s($\pm$0.01) to run 200 LBP steps on a single RBM and 4.43s($\pm$0.01) on 1000 RBMs.

- For a small batch size of 1, `pomegranate` is around three times faster on a CPU than on a GPU. It takes $1.32s$($\pm$0.06) to run 200 LBP steps on an RBM with 40 binary units on a CPU, and 3.11s($\pm$0.15) on a GPU. Similarly, on an RBM with 200 binary units, `pomegranate` takes 377.31s($\pm$71.67) on a CPU, and 1210.31s($\pm$230.47) on a GPU. However, as we increase the batch size, `pomegranate` CPU timings increase while its GPU timings do not vary much. Running 200 LBP steps on 1000 RBMs with 40 units takes 3.71s($\pm$0.17) on a CPU, and 3.41s($\pm$0.15) on a GPU. Similarly, running 200 iterations on 1000 RBMs with 200 units takes 849.84s($\pm$161.00) on a CPU, and 1206.90s($\pm$229.53) on a GPU. For larger batch sizes, `pomegranate` will eventually benefit from GPU acceleration.

- The numbers above also highlight that `pomegranate` timings significantly increase when we increase the number of units. For a fixed batch size and background, `pomegranate` is two orders of magnitude faster on a small RBM with 40 units than on a large RBM with 200 units.

- Consequently, the gap between `pomegranate` and PGMax also increases for larger models. For small models with 40 units, and 200 LBP steps, PGMax is only one order of magnitude faster than `pomegranate` for a batch size of 1—0.22s($\pm$0.01) vs. $1.32s$($\pm$0.06)—as well as for a batch size of 1k—0.35s($\pm$0.01) vs. 3.11s($\pm$0.15). However, for large RBM with 200 units, PGMax is up to three orders of magnitudes faster for a small batch size of 1—0.24s($\pm$0.01) vs. 377.31s($\pm$71.67)—and up to two orders of magnitude faster for a large batch size of 1k—4.43s($\pm$0.01) vs. 849.84s($\pm$161.00).

- Naturally, all the methods are slower when we increase the number of LBP iterations from 20 to 200.

A.2 Comparing the energies

For a given RBM size, Figure 3 reports the ratio of RBMs—out of the 20 randomly generated—for which each method reaches the lowest energy.[3] As above, we also compare the solutions returned by PGMax and `pomegranate` for 20 and 200 LBP steps.

Figure 3 highlights that for each RBM size, PGMax reaches the lowest energy more frequently than `pomegranate`. This gap is bigger for larger models: for an RBM with

3. Note that several methods can sometimes return the lowest energy.

200 units, PGMax with 200 iterations reaches the lowest energy for 20/20 RBMs, while `pomegranate` never does so. In addition, for larger models, PGMax inference quality benefits from using more LBP steps.

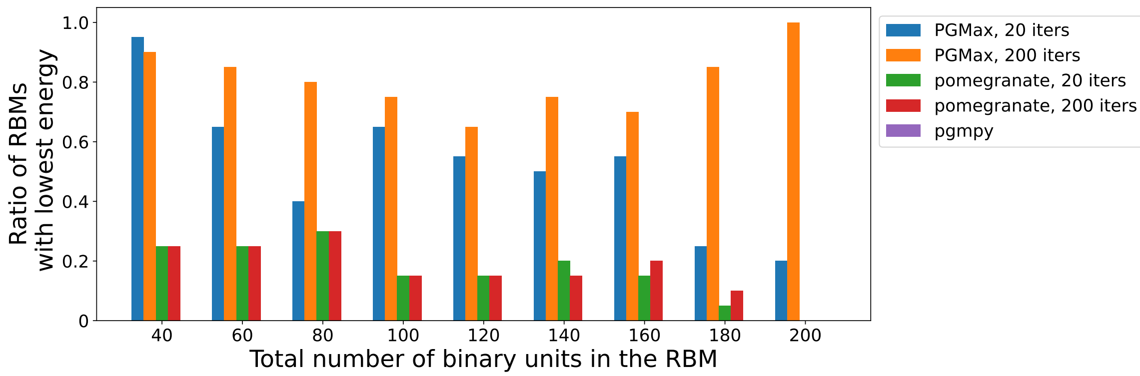

Figure 3: Number of RBMs for which each method achieves the lowest energy.

Finally, we define the set $\mathcal{M}$ of all methods compared in Figure 3. For each RBM and method $m \in \mathcal{M}$, we define $\mathcal{E}(m)$ the energy of the MAP estimate returned by the method m, as well as $\mathcal{E}^* = \min_{m \in \mathcal{M}}$ the lowest energy returned by a method. Figure 3 averages the statistic $\mathbf{1}(\mathcal{E}(m) = \mathcal{E}^*)$ which indicates whether a method returns a solution with the lowest energy.

In Figure 4, we average over the 20 RBMs of the same size the (non-negative) differences $\mathcal{E}(m) - \mathcal{E}^* \geq 0$ between the energy returned by a method and the lowest energy among all the methods. Figure 4 also shows that (a) the inference quality gap between PGMax and `pomegranate` widens as the RBMs get larger and (b) for larger models, PGMax benefits from using more LBP steps.

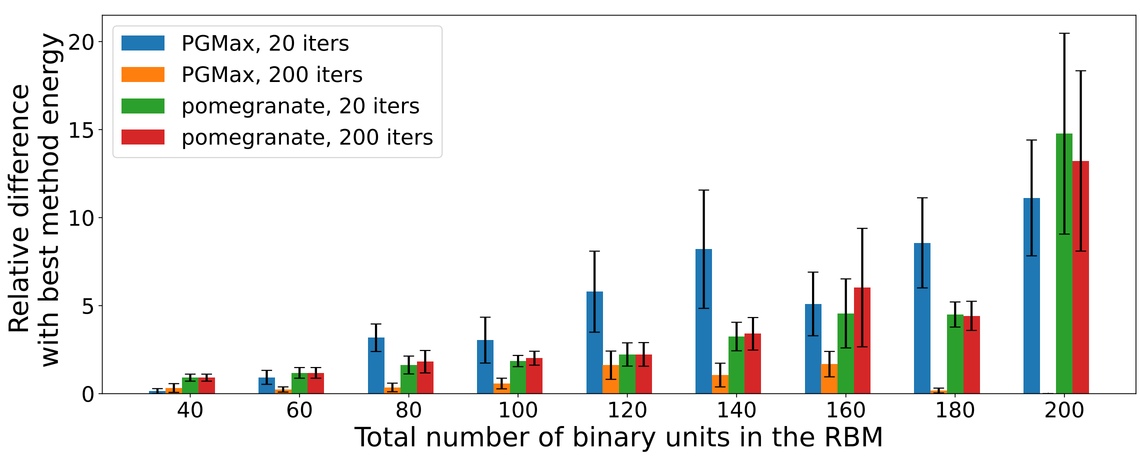

Figure 4: Difference between the energy returned by each method and the lowest energy returned across methods, averaged over the 20 RBMs (lower is better).

Appendix B. Parallel Loopy Belief propagation in PGMax

We derive herein the parallel belief propagation (BP) algorithm implemented in PGMax for the supported families of tractable factors. First, we discuss how PGMax implements BP at any temperature in the general case of "enumeration factors". Second, we discuss how PGMax implements message updates for "logical factors" that are both (a) specialized with a complexity linear in the number of connected variables (b) computationally stable at low temperatures.

B.1 General framework

We consider a discrete probabilistic graphical model (PGM) with categorical variables z described by a set of F factors $\{\theta^{f^\top} \eta^f(z^f)\}_{f=1}^F$ and I unary terms $\{\theta_i^\top \eta_i(z_i)\}_{i=1}^I$. For the fth factor, z^f is the vector of variables connected to the factor, θ^f is a vector of factor parameters and $\eta^f(z^f)$ is a vector of factor sufficient statistics. Similarly, for the variable z_i, θ_i is a vector of unary parameters and $\eta_i(z_i)$ is a vector of variable sufficient statistics. The energy of the model can be expressed as

$$E(z) = -\sum_{i=1}^{I} \theta_i^\top \eta_i(z_i) - \sum_{f=1}^{F} \theta^{f^\top} \eta^f(z^f). \tag{1}$$

The probability of a configuration z satisfies $p(z) \propto \exp(-E(z))$.

Notations: We use the following notations. First, for any $f \leq F$, let $N^f = \{i \leq I : \; z_i \in z^f\}$ be the indices of the variables connected to the fth factor. Similarly, for any $i \leq I$, let $N_i = \{f \leq F : \; z_i \in z^f\}$ be the indices of the factors connected to the ith variable.

Second, we note $V(z_i)$ (resp. $V(z^f)$) the set of valid assignments for the categorical variable z_i (resp. for the variables connected to the fth factor). Each element $w \in V(z^f)$ corresponds to an assignment for each variable in z^f: we denote $w = (w_i)_{i \in N^f}$. Naturally, we have $\eta_i(z_i) = (\mathbf{1}(z_i = k))_{k \in V(z_i)}$ and $\eta^f(z^f) = \big(\mathbf{1}(z^f = w)\big)_{w \in V(z^f)}$.

Finally, let us denote $\theta_i(k)$ the unary entry of the vector θ_i associated with $k \in V(z_i)$; and $\theta^f(w)$ the potential entry of the vector θ^f associated with $w \in V(z^f)$. It holds $\theta_i = (\theta_i(k))_{k \in V(z_i)}$ and $\theta^f = \big(\theta^f(w)\big)_{w \in V(z^f)}$.

B.2 Inference in PGMs

PGMax is specialized to solve the two following types of inference problems in the discrete PGM above.

Problem 1: mode estimation. The first problem we are interested in is estimating the mode of the probability distribution p. This problem, also referred to as maximum a posteriori **(MAP)** problem, is equivalent to finding the variables assignment with the lowest energy, that is in solving

$$z^{\text{MAP}} \in \underset{z}{\operatorname{argmin}}\, E(z) = \underset{z}{\operatorname{argmax}} \; \sum_{i=1}^{I} \theta_i^\top \eta_i(z_i) + \sum_{f=1}^{F} \theta^{f^\top} \eta^f(z^f). \tag{2}$$

The MAP Problem in Equation 2 is an integer programming problem and can be rewritten

$$\max_z \sum_{i=1}^{I} \sum_{k \in V(z_i)} \theta_i(k) \mathbf{1}(z_i = k) + \sum_{f=1}^{F} \sum_{w \in V(z^f)} \theta^f(w) \mathbf{1}(z^f = w). \tag{3}$$

Problem 2: marginals estimation. The second problem we are interested in consists in estimating the marginal distribution of p:

$$p(z_i = k) = \sum_{z:\ z_i = k} p(z), \quad \forall i,\ \forall k \in V(z_i). \tag{4}$$

Belief propagation: Exact MAP or marginal inference in complex PGMs is often intractable. To mitigate this problem, several techniques have been proposed for approximate inference, among which a popular one is belief propagation (BP) (Pearl, 1988). The BP algorithm takes as input a temperature $T \in [0, 1]$. When $T = 0$, it is often referred to as the max-product algorithm, and it returns an estimate of the MAP solution to Equation 2. When $T = 1$, it is often referred to as the sum-product algorithm, and it returns an estimate of the marginal probabilities in Equation 4. When $T \in (0, 1)$, BP estimates the soft max-marginals $\left(\sum_{z:\ z_i=k} p(z)^{1/T}\right)^T$. While BP is guaranteed to converge in trees (Yedidia et al., 2000), convergence is not guaranteed for loopy models.

In the rest of this section, we discuss the BP algorithm that we have implemented in PGMax at any temperature $T \in [0, 1]$. Section B.4 discusses message updates for a general class of "enumeration factors". Section B.5 introduces "logical factors" and Sections B.6 and B.7 places a special emphasis on presenting how PGMax derives message updates for logical factors that are both (a) specialized with a complexity linear in the number of variables connected to a factor (b) computationally stable at low temperatures.

B.3 Stable computation of two useful functions

We introduce two operators that PGMax relies on and discuss their computational stability.

Log-sum-exp: for a vector $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, we define the log-sum-exp operator at the temperature T via

$$\mathtt{LSE}(x, T) = T \log \left(\sum_i \exp\left(\frac{x_i}{T}\right) \right).$$

To guarantee computational stability, we introduce $x^* = \max_i x_i$ and implement this operator as

$$\mathtt{LSE}(x, T) = x^* + T \log \left(\sum_i \exp\left(\frac{x_i - x^*}{T}\right) \right).$$

Log-minus-exp: for two scalar $x_1, x_2 \in \mathbb{R}$ such that $x_1 \geq x_2$, we define the log-minus-exp operator at the temperature T via:

$$\begin{aligned}\texttt{LME}(x_1, x_2, T) &= T \log\left(\exp\left(\frac{x_1}{T}\right) - \exp\left(\frac{x_2}{T}\right)\right) \\ &= x_1 + T \log\left(1 - \exp\left(\frac{x_2 - x_1}{T}\right)\right) \\ &= x_1 + T \; \texttt{log1mexp}\left(\frac{x_1 - x_2}{T}\right).\end{aligned}$$

The last equation guarantees the computational stability of the implementation. We implement $\texttt{log1mexp}(x) = \log(1 - e^{-x})$ accurately using Mächler (2012).

B.4 Belief propagation in PGMax for an enumeration factor

PMax refers to a general factor ${\theta^f}^\top \eta^f(z^f)$ in Equation 1 as an "enumeration factor". An enumeration factor is then defined by a set of valid configurations represented by $\eta^f(z^f)$, paired with their respective vector of log-potentials θ^f. PGMax considers two cases when implementing the belief propagation algorithm: the max-product algorithm at $T = 0$ is treated separately from the more general case $T \in (0, 1]$.

B.4.1 At the temperature T=0

Message updates: Given a factor f and a variable $i \in N^f$ connected to this factor, the max-product algorithm estimates the solution to the **(MAP)** Problem in Equation 2 by defining positive messages $m_{z_i \to f}(k)$ from the state k of the variable z_i to the fth factor, and positive messages $m_{f \to z_i}(k)$ from the fth factor to the state k of the variable z_i. It then iterates the fixed-point updates for N_{BP} iterations:

$$\begin{aligned}m_{z_i \to f}(k) &= \exp\left(\theta_i(k)\right) \prod_{g \in N_i:\ g \neq f} m_{g \to z_i}(k) \\ m_{f \to z_i}(k) &= \max_{w \in V(z^f):\ w_i = k} \left\{ \exp\left(\theta^f(w)\right) \prod_{j \in N^f:\ j \neq i} m_{z_j \to f}(w_j) \right\}.\end{aligned} \tag{5}$$

Equations 5 are derived by setting the gradients of the Lagrangian of the Bethe free energy to 0: see Wainwright et al. (2008) for details.

Log-space mapping: For computational stability, PGMax maps the messages to log-space by defining $n_{z_i \to f}(k) = \log m_{z_i \to f}(k)$ and $n_{f \to z_i}(k) = \log m_{f \to z_i}(k)$. It then implements Equations 5 as

$$\begin{aligned}n_{z_i \to f}(k) &= \theta_i(k) + \sum_{g \in N_i:\ g \neq f} n_{g \to z_i}(k) \\ n_{f \to z_i}(k) &= \max_{w \in V(z^f):\ w_i = k} \left\{ \theta^f(w) + \sum_{j \in N^f:\ j \neq i} n_{z_j \to f}(w_j) \right\}.\end{aligned} \tag{6}$$

Parallelization: One of the appealing properties of the max-product algorithm is that it is highly parallelized. In PGMax, all the variables-to-factors message updates (top row in Equation 6) are computed in parallel. Similarly all the factors-to-variables message updates (bottom row in Equation 6) are also computed in parallel.

Optional damping: In loopy models where BP is not guaranteed to converge, a damping factor $\alpha \in (0,1)$ in the message updates from factors to variables can be used to improve convergence. After computing $n_{f \to z_i}(k)$ as in Equation 6, the messages from factors to variables are updated via

$$n^{\text{new}}_{f \to z_i}(k) = \alpha n_{f \to z_i}(k) + (1-\alpha) n^{\text{old}}_{f \to z_i}(k).$$

PGMax supports damping: the user has to specify α. $\alpha = 0.5$ offers a good trade-off between accuracy and speed in most cases.

Beliefs and MAP estimate: After N_{BP} iterations of Equation 6 with optional damping, PGMax defines the beliefs of the variables states as

$$b_i(k) = \theta_i(k) + \sum_{f \in N_i} n_{f \to z_i}(k), \ \forall i, \ \forall k \in V(z_i). \tag{7}$$

The max-product MAP estimate is then

$$z_i^* = \underset{k \in V(z_i)}{\text{argmax}} \ b_i(k), \ \forall i.$$

In addition, PGMax estimates the normalized max-marginals via the relation

$$\max_{z:\ z_i = k} p(z) \propto \frac{\exp(b_i(k))}{\sum_{\ell \in V(z_i)} \exp(b_i(\ell))}. \tag{8}$$

B.4.2 At a temperature T>0

Message updates: The belief propagation algorithm can be extended to any temperature $T \in (0,1]$ by redefining the message updates from factors to variables in Equation 5 as

$$m_{f \to z_i}(k)^{1/T} = \sum_{w \in V(z^f):\ w_i = k} \exp(\theta^f(w))^{1/T} \prod_{j \in N^f:\ j \neq i} m_{z_j \to f}(w_j)^{1/T},$$

which can be equivalently written in log-space as

$$n_{f \to z_i}(k) = T \log \left(\sum_{w \in V(z^f):\ w_i = k} \exp\left(\frac{\theta^f(w) + \sum_{j \in N^f:\ j \neq i} n_{z_j \to f}(w_j)}{T} \right) \right).$$

That is, for $T > 0$, PGMax leaves the message updates from variables to factors $n_{z_i \to f}(k)$ in Equation 6 unchanged and implements the message updates from factors to variables as

$$n_{f \to z_i}(k) = \texttt{LSE}\left(\left(\theta^f(w) + \sum_{j \in N^f:\ j \neq i} n_{z_j \to f}(w_j) \right)_{w \in V(z^f):\ w_i = k}, T \right). \tag{9}$$

Beliefs and MAP estimate: After N_{BP} iterations of Equation 9 with optional damping, PGMax estimates the beliefs as in Equation 7. Similar to Equation 8, it then estimates the normalized soft max-marginal probabilities, defined as

$$\left(\sum_{z:\ z_i = k} p(z)^{1/T} \right)^T \propto \frac{\exp(b_i(k))}{\sum_{\ell \in V(z_i)} \exp(b_i(\ell))}.$$

In the case $T = 1$, we recover the sum-product algorithm (Pearl, 1988) and PGMax estimates the marginal probabilities via

$$p(z_i = k) = \sum_{z:\ z_i = k} p(z) \approx \frac{\exp(b_i(k))}{\sum_{\ell \in V(z_i)} \exp(b_i(\ell))}.$$

B.5 Belief propagation for logical factors

In addition to the enumeration factors discussed in Section B.4, PGMax supports three classes of "logical factors", which express logical relations between the variables they connect. The supported `OR` factors, `AND` factors and `Pool` factors are displayed in Figure 5. These factors connect binary variables and are defined as follows:

OR factors : An `OR` factor connects n binary parents variables $p_1, \dots, p_n$ with one binary child variable c. $c = 1$ iff at least one p_i is equal to 1.

AND factor : An `AND` factor connects n binary parents variables $p_1, \dots, p_n$ with one binary child variable c. $c = 1$ iff if at least all the p_is are equal to 1.

Pool factor : A `Pool` factor connects one binary parent variable p with n binary child variables $c_1, \dots, c_n$. $p = 1$ iff exactly one c_i is equal to 1; and $p = 0$ iff all the c_is are equal to 0.

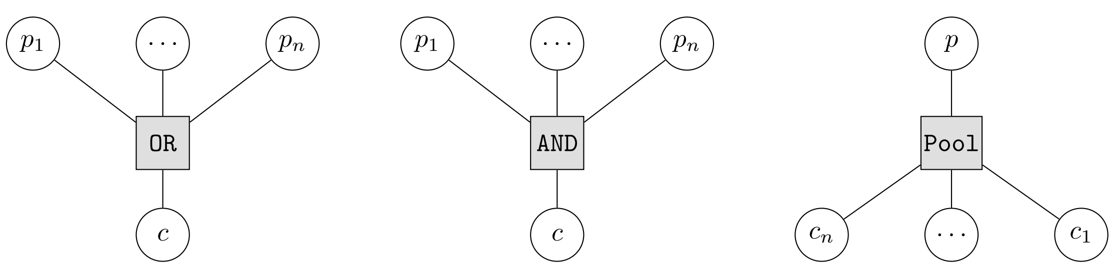

Figure 5: [Left] An `OR` factor. [Middle] An `AND` factor. [Right] A `Pool` factor.

Specialized message updates for logical factors: The `OR` and `AND` factors in Figure 5 have 2^n valid configurations. Consequently, the implementation of the BP message updates from factors to variables in Equations 6 and 9 has an exponential complexity in the number of variables connected to these factors. Similarly, the message updates for a `Pool` factor have a quadratic complexity in the number of variables. We discuss herein how PGMax leverages the structure of these logical factors to reduce, for any temperature $T \in [0, 1]$, the complexity of the message updates from factors to variables down to a linear cost in the number of connected variables. In addition, we discuss computational stability of these specialized message updates at low temperatures $T \sim 0$.

Related work: For $T = 0$ (i.e. for the max-product algorithm), some of the following equations are presented in Lázaro-Gredilla et al. (2016); Lazaro-Gredilla et al. (2021). However, we discuss herein how to efficiently implement parallel message updates by reusing computations. For $T > 0$, we are not aware of a previous work that derives the BP updates for the logical factors aforementioned.

B.6 Message updates for an `OR` factor

We reuse the previous notations and study how to efficiently implement BP updates for an `OR` factor. Note that the same updates can be used for an `AND` factor by exploiting the relation

$$c = \texttt{AND}(p_1, \dots, p_n) \iff \bar{c} = \texttt{OR}(\bar{p}_1, \dots, \bar{p}_n)$$

The message updates from variables to factors $n_{z_i \to f}(k)$ in Equation 6 are left unchanged for logical factors. Similarly, the damping step and the beliefs computation are defined as before. We therefore only study the message updates from factors to variables $n_{f \to z_i}(k)$.

Notations: We note $i_1^* \in \operatorname{argmax}_i \{n_{p_i \to f}(1) - n_{p_i \to f}(0)\}$ and $i_2^* \in \operatorname{argmax}_{i \neq i_1^*} \{n_{p_i \to f}(1) - n_{p_i \to f}(0)\}$.

Lemma 1 below justifies the need to separately consider the variable with the largest incoming log-message difference. We defer the proof to Section C.

Lemma 1 *Given the incoming variables to factor messages, the valid configuration $(p^*, 1)$ of an `OR` factor satisfying $c = 1$ and with the highest sum of incoming log-messages is given by $p^*_{i_1^*} = 1$ and $p^*_j = \operatorname{argmax}\{n_{p_j \to f}(0),\ n_{p_j \to f}(1)\})$ otherwise.*

B.6.1 At the temperature T=0

Outgoing messages to the child variable: The messages going from the fth `OR` factor to the child variable c can be derived from Lemma 1:

$$\begin{cases} n_{f \to c}(1) = n_{p_{i_1^*} \to f}(1) + \sum\limits_{j \neq i_1^*} \max\{n_{p_j \to f}(1),\ n_{p_j \to f}(0)\} \\ n_{f \to c}(0) = \sum\limits_{j} n_{p_j \to f}(0), \end{cases}$$

Outgoing messages to the parents variables: We now consider the messages going from the `OR` factor to the parent variables. For $i \neq i_1^*$, it holds:

$$\begin{cases} n_{f \to p_i}(1) = n_{c \to f}(1) + \sum\limits_{j \neq i} \max\{n_{p_j \to f}(1),\ n_{p_j \to f}(0)\} \\ n_{f \to p_i}(0) = \max \left\{ \begin{array}{l} n_{c \to f}(0) + \sum\limits_{j \neq i} n_{p_j \to f}(0), \\ n_{c \to f}(1) + n_{p_{i_1^*} \to f}(1) + \sum\limits_{j \neq i_1^*, i} \max\{n_{p_j \to f}(1),\ n_{p_j \to f}(0)\} \end{array} \right\}, \end{cases}$$

where we have applied a similar reasoning to Lemma 1 to compute the second term in $n_{f \to p_i}(0)$.

For $i = i_1^*$, $n_{f\to p_{i_1^*}}(1)$ is derived as above. For $n_{f\to p_{i_1^*}}(0)$ it holds

$$n_{f\to p_i}(0) = \max \left\{ \begin{array}{l} n_{c\to f}(0) + \sum_{j\neq i} n_{p_j\to f}(0), \\ n_{c\to f}(1) + n_{p_{i_2^*}\to f}(1) + \sum_{j\neq i_2^*, i} \max\{n_{p_j\to f}(1),\ n_{p_j\to f}(0)\} \end{array} \right\}$$

Efficient parallel message updates: To compute the updates in parallel efficiently, we first introduce the quantities

$$\begin{cases} i_1^* & \in \ \operatorname{argmax}_i \{n_{p_i\to f}(1) - n_{p_i\to f}(0)\} \\ \Delta n_1^* & = n_{p_{i_1^*}\to f}(1) - n_{p_{i_1^*}\to f}(0) \\ \Delta n_2^* & = \max_{i\neq i_1^*} \{n_{p_i\to f}(1) - n_{p_i\to f}(0)\} \\ S & = \ \sum_i n_{p_i\to f}(0) \\ SM & = \ \sum_i \max\{n_{p_i\to f}(1),\ n_{p_i\to f}(0)\} \\ SM_i & = \ SM - \max\{n_{p_i\to f}(1),\ n_{p_i\to f}(0)\} + n_{c\to f}(1), \quad \forall i. \end{cases}$$

PGMax derives all the message updates to the variables in parallel, with a linear complexity, via

$$\begin{cases} n_{f\to c}(0) & = \ S \\ n_{f\to c}(1) & = \ SM + \min\{0,\ \Delta n_1^*\} \\ n_{f\to p_i}(1) & = \ SM_i & \forall i \\ n_{f\to p_i}(0) & = \ \max\{S - n_{p_i\to f}(0) + n_{c\to f}(0),\ \ SM_i + \min\{0,\ \Delta n_1^*\}\}, & \forall i \neq i_1^* \\ n_{f\to p_{i_1^*}}(0) & = \ \max\left\{S - n_{p_{i_1^*}\to f}(0) + n_{c\to f}(0),\ \ SM_{i_1^*} + \min\{0,\ \Delta n_2^*\}\right\}. \end{cases}$$

Only computing message difference: The BP algorithm does not need to access the message updates from an `OR` factor to both states 0 and 1 of a variable. Only computing the message difference—i.e. $n_{f\to c}(1) - n_{f\to c}(0)$ for a child variable and $n_{f\to p_i}(1) - n_{f\to p_i}(0)$ for a parent variable—can make inference faster and more stable. However, alternative inference methods to BP may require to access the message updates to both variable states. Therefore, PGMax computes both message updates. When using BP, PGMax only returns the message differences and lets the JAX compiler simplify intermediate computations.

B.6.2 At a temperature T>0

Outgoing messages to the child variable: At a temperature $T > 0$, for the outgoing messages to the state 0 of the child variable c, we have

$$m_{f\to c}(0)^{1/T} = \prod_i m_{p_i\to f}(0)^{1/T}$$

from which we derive

$$n_{f\to c}(0) = S = \sum_i n_{p_i\to f}(0), \tag{10}$$

where we have reused the above quantity S. Similarly for the state 1

$$\begin{aligned} m_{f\to c}(1)^{1/T} &= \sum_{(e_1,\ldots,e_n)\neq(0,\ldots,0)} \prod_i m_{p_i\to f}(e_i)^{1/T} \\ &= \prod_i \left(m_{p_i\to f}(0)^{1/T} + m_{p_i\to f}(1)^{1/T} \right) - \prod_i n_{p_i\to f}(0)^{1/T}. \end{aligned}$$

Let us define

$$P = T \sum_i \log \left(\exp\left(\frac{n_{p_i\to f}(0)}{T}\right) + \exp\left(\frac{n_{p_i\to f}(1)}{T}\right) \right) = \sum_i \texttt{LSE}\left((n_{p_i\to f}(0),\ n_{p_i\to f}(1)), T \right).$$

Then it holds

$$n_{f\to c}(1) = T \log(e^{P/T} - e^{S/T}) = \texttt{LME}(P, S, T). \tag{11}$$

Computational stability: When the incoming messages satisfy $n_{p_i\to f}(0) \geq n_{p_j\to f}(1), \forall i$ and we are at a low temperature $T \sim 0$, P and S may be really close and the LME operator in Equation 11 may become computationally unstable. In this case, PGMax replaces $n_{f\to c}(1)$ by a lower bound of Equation 11 derived as follows.

Let us first observe that as $n_{p_i\to f}(0) \geq n_{p_j\to f}(1),\ \forall i$, the two valid factor configuration $p^{(1)}$ and $p^{(2)}$ satisfying $c = 1$ with highest sum of incoming messages are respectively (a) $p^{(1)}_{i_1^*} = 1$ and $p^{(1)}_i = 0$ o.w. and (b) $p^{(2)}_{i_2^*} = 1$ and $p^{(2)}_i = 0$ o.w.. A lower bound of $m_{f\to c}(1)^{1/T}$ is then:

$$m_{f\to c}(1)^{1/T} \geq m_{p_{i_1^*}\to f}(1)^{1/T} \prod_{i\neq i_1^*} m_{p_i\to f}(0)^{1/T} + m_{p_{i_2^*}\to f}(1)^{1/T} \prod_{i\neq i_2^*} m_{p_i\to f}(0)^{1/T}.$$

$n_{f\to c}(1)$ can then be lower bounded as

$$n_{f\to c}(1) \geq T \log \left(\exp\left(\frac{S + \Delta n_1^*}{T}\right) + \exp\left(\frac{S + \Delta n_2^*}{T}\right) \right) = \texttt{LSE}\left((S + \Delta n_1^*,\ S + \Delta n_2^*), T \right). \tag{12}$$

In practice, PGMax introduces a threshold $\epsilon = 10^{-4}$ and replaces $n_{f\to c}(1)$ with the lower bound when $P - S \leq \epsilon$. This lower bound is fast to compute and minimally affects the message updates speed. Let us note that it could be refined further if needed by adding more configurations, at the cost of more compute, to better approximate $n_{f\to c}(1)$.

Outgoing messages to the parent variables: For the outgoing messages to the state 0 of the parent variable p_i it holds:

$$m_{f\to p_i}(1)^{1/T} = m_{c\to f}(1)^{1/T} \prod_{j\neq i} \left(m_{p_j\to f}(0)^{1/T} + m_{p_j\to f}(1)^{1/T} \right).$$

By introducing $P_i = T \sum_{j\neq i} \log \left(\exp\left(\frac{n_{p_j\to f}(0)}{T}\right) + \exp\left(\frac{n_{p_j\to f}(1)}{T}\right) \right)$, we simply have:

$$n_{f\to p_i}(1) = n_{c\to f}(1) + P_i. \tag{13}$$

Similarly, for the outgoing messages to the state 1:

$$\begin{aligned} m_{f\to p_i}(0)^{1/T} = \; & m_{c\to f}(0)^{1/T} \prod_{j\neq i} m_{p_j\to f}(0)^{1/T} \\ & + m_{c\to f}(1)^{1/T} \left\{ \prod_{j\neq i} \left(m_{p_j\to f}(0)^{1/T} + m_{p_j\to f}(1)^{1/T} \right) - \prod_{j\neq i} m_{p_j\to f}(0)^{1/T} \right\}. \end{aligned}$$

We introduce $S_i = \sum_{j\neq i} n_{p_i\to f}(0)$ and observe that:

$$m_{f\to p_i}(0)^{1/T} = m_{c\to f}(0)^{1/T} \exp\left(\frac{S_i}{T}\right) + m_{c\to f}(1)^{1/T} \left\{ \exp\left(\frac{P_i}{T}\right) - \exp\left(\frac{S_i}{T}\right) \right\},$$

from which we derive

$$\begin{aligned} n_{f\to p_i}(0) &= T \log\left(\exp\left(\frac{S_i + n_{c\to f}(0)}{T}\right) + \exp\left(\frac{P_i + n_{c\to f}(1)}{T}\right) - \exp\left(\frac{S_i + n_{c\to f}(1)}{T}\right)\right) \\ &= \texttt{LME}\left(\texttt{LSE}\left((S_i + n_{c\to f}(0),\; P_i + n_{c\to f}(1)),\; T\right),\; S_i + n_{c\to f}(1),\; T\right) \end{aligned} \tag{14}$$

Computational stability: Again, if $n_{c\to f}(1) \geq n_{c\to f}(0)$ and $n_{p_j\to f}(0) \geq n_{p_j\to f}(1)$, $\forall j \neq i$ and we are at a low temperature $T \sim 0$, then P_i and S_i may be close and the `LME` operator in Equation 14 may become computationally unstable. As before, we propose to replace $n_{f\to p_i}(0)$ by a lower bound by using the valid configurations $p^{(1)}$ and $p^{(2)}$.

If $i \neq i_1^*$, we use $p^{(1)}$ to derive a lower bound of $m_{f\to p_i}(0)^{1/T}$:

$$m_{f\to p_i}(0)^{1/T} \geq m_{c\to f}(0)^{1/T} \exp\left(\frac{S_i}{T}\right) + m_{c\to f}(1)^{1/T} m_{p_{i_1^*}\to f}(1)^{1/T} \prod_{j\neq i_1^*, i} m_{p_j\to f}(0)^{1/T}$$

from which we derive a lower bound of $n_{f\to p_i}(0)$:

$$\begin{aligned} n_{f\to p_i}(0) &= T \log\left(\exp\left(\frac{S_i + n_{c\to f}(0)}{T}\right) + \exp\left(\frac{S_i + \Delta n_1^* + n_{c\to f}(1)}{T}\right)\right) \\ &= \texttt{LSE}\left((S_i + n_{c\to f}(0),\; S_i + \Delta n_1^* + n_{c\to f}(1),\; T\right). \end{aligned} \tag{15}$$

A similar lower bound can be derived for i_1^*, using $p^{(2)}$.

Stable and efficient parallel message updates: To summarize Equations 10, 11, 12, 13, 14, 15, let us first introduce the quantities

$$\begin{cases} i_1^* & \in \ \operatorname{argmax}_i \left\{ n_{p_i \to f}(1) - n_{p_i \to f}(0) \right\} \\ \Delta n_1^* & = n_{p_{i_1^*} \to f}(1) - n_{p_{i_1^*} \to f}(0) \\ \Delta n_2^* & = \max_{i \neq i_1^*} \left\{ n_{p_i \to f}(1) - n_{p_i \to f}(0) \right\} \\ S & = \ \sum_j n_{p_j \to f}(0) \\ S_i & = \ S - n_{p_i \to f}(0), \qquad \forall i \\ n_i & = \ \mathtt{LSE}\left((n_{p_i \to f}(0),\ n_{p_i \to f}(1)), T \right) \\ P & = \sum_j n_j \\ P_i & = P - n_i, \qquad \forall i \\ I_i & = (P_i - S_i \geq \epsilon) \vee (n_{c \to f}(0) - n_{c \to f}(1) \geq \epsilon)\,, \quad \forall i \end{cases}$$

where I_i is a boolean variable. PGMax derives the `OR` factor to variables message updates (a) with a linear complexity (b) stable at low temperature and (c) computed in parallel, via

$$\begin{cases} n_{f \to c}(0) & = \ S \\ n_{f \to c}(1) & = \begin{cases} \mathtt{LME}(P, S, T) & \text{if } P - S \geq \epsilon \\ \mathtt{LSE}\left((S + \Delta n_1^*,\ S + \Delta n_2^*), T \right) & \text{o.w.} \end{cases} \\ n_{f \to p_i}(1) & = \ P_i + n_{c \to f}(1), \quad \forall i \\ n_{f \to p_i}(0) & = \begin{cases} \mathtt{LME} \begin{pmatrix} \mathtt{LSE}\left((S_i + n_{c \to f}(0),\ P_i + n_{c \to f}(1)),\ T \right), \\ S_i + n_{c \to f}(1), \\ T \end{pmatrix} & \text{if } I_i \\ \mathtt{LSE}\left((S_i + n_{c \to f}(0),\ S_i + \Delta n_1^* + n_{c \to f}(1),\ T \right) & \text{if not } I_i \text{ and } i \neq i_1^* \\ \mathtt{LSE}\left((S_{i_1^*} + n_{c \to f}(0),\ S_{i_1^*} + \Delta n_2^* + n_{c \to f}(1),\ T \right) & \text{if not } I_i \text{ and } i = i_1^* \end{cases} \end{cases}$$

B.7 Message updates for a `Pool` factor

Finally, we derive efficient and computationally stable parallel message updates for a `Pool` factor. A `Pool` factor only has $n+1$ valid configurations: the variables $(p, c_1, \ldots, c_n)$ can only be assigned to the values $(0, 0, \ldots, 0)$, $(1, 1, 0, \ldots, 0)$, $\ldots$, $(1, 0, \ldots, 0, 1)$. We reuse the notations $i_1^* \in \operatorname{argmax}_i \{n_{c_i \to f}(1) - n_{c_i \to f}(0)\}$ and $i_2^* \in \operatorname{argmax}_{i \neq i_1^*} \{n_{c_i \to f}(1) - n_{c_i \to f}(0)\}$.

B.7.1 At the temperature T=0

Outgoing messages to the parent variable: The messages going from the fth `Pool` factor to the parent variable p can be derived by using an observation similar to Lemma 1:

$$\begin{cases} n_{f \to p}(1) = n_{c_{i_1^*} \to f}(1) + \sum_{j \neq i_1^*} n_{c_j \to f}(0) \\ n_{f \to p}(0) = \sum_j n_{c_j \to f}(0). \end{cases}$$

In particular, the message difference can be simplified:

$$n_{f \to p}(1) - n_{f \to p}(0) = n_{c_{i_1^*} \to f}(1) - n_{c_{i_1^*} \to f}(0).$$

Outgoing messages to the children variable: We now look at the messages going from the `Pool` factor to the child variable c_i. If $i \neq i_1^*$ we have:

$$\begin{cases} n_{f \to c_i}(1) = n_{p \to f}(1) + \sum_{j \neq i} n_{c_j \to f}(0) \\ n_{f \to c_i}(0) = \max \left\{ n_{p \to f}(0) + \sum_{j \neq i} n_{c_j \to f}(0), n_{p \to f}(1) + n_{c_{i_1^*} \to f}(1) + \sum_{j \neq i_1^*, i} n_{c_j \to f}(0) \right\}. \end{cases}$$

Again, the message difference can be simplified as

$$n_{f \to c_i}(1) - n_{f \to c_i}(0) = \min \left\{ n_{p \to f}(1) - n_{p \to f}(0), \quad n_{c_{i_1^*} \to f}(0) - n_{c_{i_1^*} \to f}(1) \right\}$$

For $i = i_1^*$, $n_{f \to c_{i_1^*}}(1)$ is derived as above while the message difference becomes:

$$n_{f \to c_{i_1^*}}(1) - n_{f \to c_{i_1^*}}(0) = \min \left\{ n_{p \to f}(1) - n_{p \to f}(0), \quad n_{c_{i_2^*} \to f}(0) - n_{c_{i_2^*} \to f}(1) \right\}$$

Efficient parallel message updates: By first introducing the quantities

$$\begin{cases} i_1^* & \in \ \operatorname{argmax}_i \left\{ n_{c_i \to f}(1) - n_{c_i \to f}(0) \right\} \\ \Delta n_1^* & = n_{c_{i_1^*} \to f}(1) - n_{c_{i_1^*} \to f}(0) \\ \Delta n_2^* & = \ \max_{i \neq i_1^*} \left\{ n_{c_i \to f}(1) - n_{c_i \to f}(0) \right\} \\ S & = \ \sum_j n_{c_j \to f}(0) \\ S_i & = \ S - n_{c_i \to f}(0) + n_{p \to f}(1), & \forall i. \end{cases}$$

PGMax derives all the message updates from a `Pool` factor to the variables connected, with a linear complexity, via

$$\begin{cases} n_{f \to p}(0) & = \ S \\ n_{f \to p}(1) & = \ S + \Delta n_1^* \\ n_{f \to c_i}(1) & = \ S_i & \forall i \\ n_{f \to c_i}(0) & = \ S_i + \min \left\{ n_{p \to f}(1) - n_{p \to f}(0), \quad - \Delta n_1^* \right\},] & \forall i \neq i_1^* \\ n_{f \to c_{i_1^*}}(0) & = \ S_i + \min \left\{ n_{p \to f}(1) - n_{p \to f}(0), \quad - \Delta n_2^* \right\}. \end{cases}$$

B.7.2 At a temperature T>0

Outgoing messages to the parent variable: At a temperature $T > 0$, the outgoing messages to the state 0 of the parent variable p can be expressed as

$$m_{f \to p}(0)^{1/T} = \prod_i m_{c_i \to f}(0)^{1/T},$$

from which we derive

$$n_{f \to p}(0) = \sum_i n_{c_i \to f}(0).$$

Similarly, for the state 1 we have

$$m_{f\to p}(1)^{1/T} = m_{c_1\to f}(1)^{1/T}\prod_{i\neq 1} m_{c_i\to f}(0)^{1/T} + \ldots + m_{c_n\to f}(1)^{1/T}\prod_{i\neq n} m_{c_i\to f}(0)^{1/T}.$$

Hence the message difference can be expressed as

$$\begin{aligned} n_{f\to p}(1) - n_{f\to p}(0) &= T\log\left(\frac{m_{f\to p}(1)^{1/T}}{m_{f\to p}(0)^{1/T}}\right) \\ &= T\log\left(\sum_i \frac{m_{c_i\to f}(1)^{1/T}}{m_{c_i\to f}(0)^{1/T}}\right) \\ &= T\log\left(\sum_i \exp\left(\frac{n_{c_i\to f}(1) - n_{c_i\to f}(0)}{T}\right)\right) \\ &= \texttt{LSE}\left(\left(n_{c_i\to f}(1) - n_{c_i\to f}(0)\right)_i,\ \ T\right). \end{aligned} \tag{16}$$

Outgoing messages to the children variable: Similarly, the outgoing messages to the state 0 of the child variable c_i is

$$m_{f\to c_i}(1)^{1/T} = m_{p\to f}(1)^{1/T}\prod_{j\neq i} m_{c_j\to f}(0)^{1/T},$$

from which we derive

$$n_{f\to c_i}(1) = n_{p\to f}(1) + \sum_{j\neq i} n_{c_j\to f}(0).$$

And the outgoing message for the state 1 is

$$m_{f\to c_i}(0)^{1/T} = m_{p\to f}(0)^{1/T}\prod_{j\neq i} m_{c_j\to f}(0)^{1/T} + \sum_{j\neq i} m_{p\to f}(1)^{1/T} m_{c_j\to f}(1)^{1/T}\prod_{k\neq i,j} m_{c_k\to f}(0)^{1/T}.$$

Hence, by reusing Equation 16, the message difference can be expressed as

$$\begin{aligned} n_{f\to c_i}(1) - n_{f\to c_i}(0) &= T\log\left(\frac{m_{f\to c_i}(1)^{1/T}}{m_{f\to c_i}(0)^{1/T}}\right) \\ &= -T\log\left(\frac{m_{p\to f}(0)^{1/T}}{m_{p\to f}(1)^{1/T}} + \sum_{j\neq i}\frac{m_{c_j\to f}(1)^{1/T}}{m_{c_j\to f}(0)^{1/T}}\right) \\ &= -T\log\left(\exp\left(\frac{n_{p\to f}(0) - n_{p\to f}(1)}{T}\right) + \sum_{j\neq i}\exp\left(\frac{n_{c_j\to f}(1) - n_{c_j\to f}(0)}{T}\right)\right) \\ &= \texttt{LME}\left(Q,\ n_{c_i\to f}(1) - n_{c_i\to f}(0),\ T\right) \end{aligned}$$

where we have defined $Q = \texttt{LSE}\left(\left(n_{f\to p}(1) - n_{f\to p}(0),\ \ n_{p\to f}(0) - n_{p\to f}(1)\right),\ \ T\right)$ to reuse Equation 16. While this reuses more computations across, the `LME` operator may not be stable at low temperatures for $i = i_1^*$. In this case, we set:

$$n_{f\to c_{i_1^*}}(1) - n_{f\to c_{i_1^*}}(0) = -\texttt{LSE}\left(\left((n_{c_i\to f}(1) - n_{c_i\to f}(0))_{i\neq i_1^*},\ n_{p\to f}(0) - n_{p\to f}(1)\right),\ \ T\right).$$

Efficient and stable parallel message updates: In summary, PGMax first introduces the quantities

$$\begin{cases} i_1^* & \in \ \operatorname{argmax}_i \{n_{c_i \to f}(1) - n_{c_i \to f}(0)\} \\ \Delta n_1^* & = n_{c_{i_1^*} \to f}(1) - n_{c_{i_1^*} \to f}(0) \\ S & = \ \sum_j n_{p_j \to f}(0) \\ S_i & = \ S - n_{c_i \to f}(0) + n_{p \to f}(1) & \forall i \\ P & = \ \texttt{LSE}\left((n_{c_i \to f}(1) - n_{c_i \to f}(0))_i , \quad T\right) \\ Q & = \ \texttt{LSE}\left((P, \quad n_{p \to f}(0) - n_{p \to f}(1)), \quad T\right), \end{cases}$$

PGMax derives stable message updates, with linear complexity, from a `Pool` factor to its connected variables via

$$\begin{cases} n_{f \to p}(0) & = \ S \\ n_{f \to p}(1) & = \ S + P \\ n_{f \to c_i}(1) & = \ S_i & \forall i \\ n_{f \to c_i}(0) & = \ S_i - \texttt{LME}\left(Q, \ n_{c_i \to f}(1) - n_{c_i \to f}(0), \ T\right) & \forall i \neq i_1^* \\ n_{f \to c_{i_1^*}}(0) & = \ S_i + \texttt{LSE}\left(\left((n_{c_i \to f}(1) - n_{c_i \to f}(0))_{i \neq i_1^*}, \ n_{p \to f}(0) - n_{p \to f}(1)\right), \quad T\right) \end{cases}$$

B.8 Energy computation

Let us finally mention that given a PGM with both enumeration and logical factors, PGMax efficiently computes the energy in Equation 1 for any configuration z.

Appendix C. Proof of Lemma 1

We present herein the proof of the Lemma 1.

Proof Let $(p^*, 1)$ be the valid configuration of an `OR` factor satisfying $c = 1$, where we have defined $p^*_{i_1^*} = 1$ and $p^*_j = \operatorname{argmax}\{n_{p_j \to f}(0), \ n_{p_j \to f}(1)\})$ otherwise; and $i_1^* \in \operatorname{argmax}_i\{n_{p_i \to f}(1) - n_{p_i \to f}(0)\}$.

The sum of the associated incoming message is

$$S_1 = n_{p_{i_1^*} \to f}(1) + \sum_{j \neq i_1^*} \max n_{p_j \to f}(p_j^*).$$

Let us assume that this sum is not the highest among all the configurations satisfying $c = 1$. Then we can find $k \neq i_1^*$ such that the configuration defined by $p_{i_1^*} = 0$, $p_k = 1$ and $p_i = \operatorname{argmax}\{n_{p_j \to f}(1), \ n_{p_j \to f}(0)\})$ otherwise (a) is valid for the `OR` factor and (b) scores higher than p^*. The sum of the incoming messages for this configuration is

$$S_2 = n_{p_{i_1^*} \to f}(0) + n_{p_k \to f}(1) + \sum_{j \neq i_1^*} \max n_{p_j \to f}(p_j^*).$$

It then holds:

$$S_2 - S_1 = n_{p_k \to f}(1) - n_{p_k \to f}(p_k^*) + n_{p_{i_1^*} \to f}(0) - n_{p_{i_1^*} \to f}(1) \geq 0.$$

If $p_k^* = 1$, then $n_{p_k \to f}(1) \geq n_{p_k \to f}(0)$. In addition, $S_2 - S_1 = n_{p_{i_1^*} \to f}(0) - n_{p_{i_1^*} \to f}(1) \geq 0$ which implies that $n_{p_{i_1^*} \to f}(0) \geq n_{p_{i_1^*} \to f}(1)$.
However, by definition, $n_{p_{i_1^*} \to f}(1) - n_{p_{i_1^*} \to f}(0) \geq n_{p_k \to f}(1) - n_{p_k \to f}(0) \geq 0$: contradiction.

Hence $p_k^* = 0$. Then, $S_2 - S_1 \geq 0$ implies $n_{p_k \to f}(1) - n_{p_k \to f}(0) \geq n_{p_{i_1^*} \to f}(1) - n_{p_{i_1^*} \to f}(0)$ which, again contradicts the definition of i_1^*.

Consequently, Lemma 1 is true. ■

References

Bjoern Andres, Thorsten Beier, and Jörg H Kappes. Opengm: A c++ library for discrete graphical models. *arXiv preprint arXiv:1206.0111*, 2012.

Ankur Ankan and Abinash Panda. pgmpy: Probabilistic graphical models using python. In *SciPy*, pages 6–11. Citeseer, 2015.

Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL `http://github.com/google-deepmind`.

Dmitry Bagaev and Bert de Vries. Reactivemp. jl: A julia package for reactive message passing-based bayesian inference. *JuliaCon Proceedings*, 1(1), 2022.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76, 2017.

Marco Cox, Thijs van de Laar, and Bert de Vries. A factor graph approach to automated design of bayesian signal processing algorithms. *International Journal of Approximate Reasoning*, 104:185–204, 2019.

Antoine Dedieu, Guangyao Zhou, Dileep George, and Miguel Lazaro-Gredilla. Learning noisy-or bayesian networks with max-product belief propagation. *arXiv preprint arXiv:2302.00099*, 2023.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

Dileep George, Wolfgang Lehrach, Ken Kansky, Miguel Lázaro-Gredilla, Christopher Laan, Bhaskara Marthi, Xinghua Lou, Zhaoshi Meng, Yi Liu, Huayan Wang, et al. A generative vision model that trains with high data efficiency and breaks text-based CAPTCHAs. *Science*, 358(6368), 2017.

Amir Globerson and Tommi Jaakkola. Fixing max-product: convergent message passing algorithms for map lp-relaxations. *Advances in Neural Information Processing Systems*, 20, 2007.

Joerg Kappes, Bjoern Andres, Fred Hamprecht, Christoph Schnorr, Sebastian Nowozin, Dhruv Batra, Sungwoong Kim, Bernhard Kausler, Jan Lellmann, Nikos Komodakis, et al. A comparative study of modern inference techniques for discrete energy minimization problems. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1328–1335, 2013.

Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, 2001.

Miguel Lázaro-Gredilla, Yi Liu, D Scott Phoenix, and Dileep George. Hierarchical compositional feature learning. *arXiv preprint arXiv:1611.02252*, 2016.

Miguel Lazaro-Gredilla, Antoine Dedieu, and Dileep George. Perturb-and-max-product: Sampling and learning in discrete energy-based models. *Advances in Neural Information Processing Systems*, 34, 2021.

Miguel Lázaro-Gredilla, Wolfgang Lehrach, Nishad Gothoskar, Guangyao Zhou, Antoine Dedieu, and Dileep George. Query training: Learning a worse model to infer better marginals in undirected graphical models with hidden variables. In *AAAI Conference on Artificial Intelligence*, volume 35, pages 8252–8260, 2021.

Martin Mächler. Accurately computing log (1- exp (- a)) assessed by the rmpfr package. Technical report, Technical report, 2012.

R. J. McEliece, D. J. C. MacKay, and J. F. Cheng. Turbo decoding as an instance of Pearl's 'belief propagation' algorithm. *IEEE J. on Selected Areas in Comm.*, 16(2):140–152, 1998.

Kevin P Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: an empirical study. In *Fifteenth conference on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.

Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan kaufmann, 1988.

Marco Pretti. A message-passing algorithm with damping. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(11):P11008, 2005.

Siamak Ravanbakhsh, Barnabás Póczos, and Russell Greiner. Boolean matrix factorization and noisy completion via message passing. In *International Conference on Machine Learning*, pages 945–954. PMLR, 2016.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

Jacob Schreiber. Pomegranate: fast and flexible probabilistic modeling in python. *Journal of Machine Learning Research*, 18(164):1–6, 2018.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

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

Chaohui Wang, Nikos Komodakis, and Nikos Paragios. Markov random field modeling, inference & learning in computer vision & image understanding: A survey. *Computer Vision and Image Understanding*, 117(11):1610–1627, 2013.

Jonathan S Yedidia, William Freeman, and Yair Weiss. Generalized belief propagation. *Advances in Neural Information Processing Systems*, 13, 2000.

Guangyao Zhou. Mixed hamiltonian monte carlo for mixed discrete and continuous variables. *Advances in Neural Information Processing Systems*, 33:17094–17104, 2020.

Guangyao Zhou, Wolfgang Lehrach, Antoine Dedieu, Miguel Lázaro-Gredilla, and Dileep George. Graphical models with attention for context-specific independence and an application to perceptual grouping. *arXiv preprint arXiv:2112.03371*, 2021.