Graphically Structured Diffusion Models

Christian Weilbach **William Harvey** **Frank Wood**
Department of Computer Science
University of British Columbia
Vancouver, Canada
{weilbach,wsgh,fwood}@cs.ubc.ca

Abstract

We introduce a framework for automatically defining and learning deep generative models with problem-specific structure. We tackle problem domains that are more traditionally solved by algorithms such as sorting, constraint satisfaction for Sudoku, and matrix factorization. Concretely, we train diffusion models with an architecture tailored to the problem specification. This problem specification should contain a graphical model describing relationships between variables, and often benefits from explicit representation of subcomputations. Permutation invariances can also be exploited. Across a diverse set of experiments we improve the scaling relationship between problem dimension and our model's performance, in terms of both training time and final accuracy.

1 Introduction

There has been a recent wave of progress in deep generative modelling, especially with the emergence of diffusion models (DMs) [Sohl-Dickstein et al., 2015, Ho et al., 2020]. Compelling conditional variants can take as input text [Rombach et al., 2022], previous video frames [Ho et al., 2022], or subsets of their output dimensions [Tashiro et al., 2021]. These successes come from the considerable effort that has been spent developing models to process data streams matching the human sensorium. In short, within a certain problem domain, DMs provide a flexible mapping from inputs to outputs. It remains unclear how similar models can be applied to the highly-structured problem domains more typically considered the domain of algorithms. Take the example of matrix factorization, where the input is a matrix and the desired output is two matrices which multiply to give the input. A typical approach to tackle matrix factorization is then to acquire a real or synthetic dataset of input-output pairs to train a mapping. Unlike traditional algorithm design, this approach must learn all necessary problem structure from data. For highly structured problem domains, the required training compute will therefore be large and scale poorly with problem dimensionality.

A natural framework to reason about structure in the probabilistic setting of deep generative models is that of graphical models [Koller and Friedman, 2009]. We propose Graphically Structured Diffusion Models, or GSDMs, which have an architecture based on a graphical model representation of a dataset. We find that incorporating structure from the graphical model can lead to architectures that scale well with problem dimension. This includes the ability to deal with problems of varying dimension by scaling the graphical model correspondingly. We design the framework to be accessible and widely-applicable and therefore do not prescribe a specific form of graphical model. We find that the quality of the GSDM model is consistent across reasonable choices of graphical model.

Structured computation graphs often contain "intermediate variables," variables (stochastic or deterministic) whose values are computed as part of a model or algorithm, but that do not directly form the input or output. When these are present in a graphical model, GDSM is able to leverage both their values and the relevant graphical structure to, roughly speaking, break down the learning of a complex problem into the learning of multiple simpler problems. This helps GSDM to exhibit a more graceful scaling with problem dimension than pure DM baselines.

To summarize our contributions, we **(1)** describe how to automatically derive a structured, sparse attention mechanism from a graphical model representation, and release an efficient implementation (Appendix C)[1]. **(2)** We showcase the integration of a generative model's intermediate vari-

Preprint.

[1] https://github.com/plai-group/gsdm.

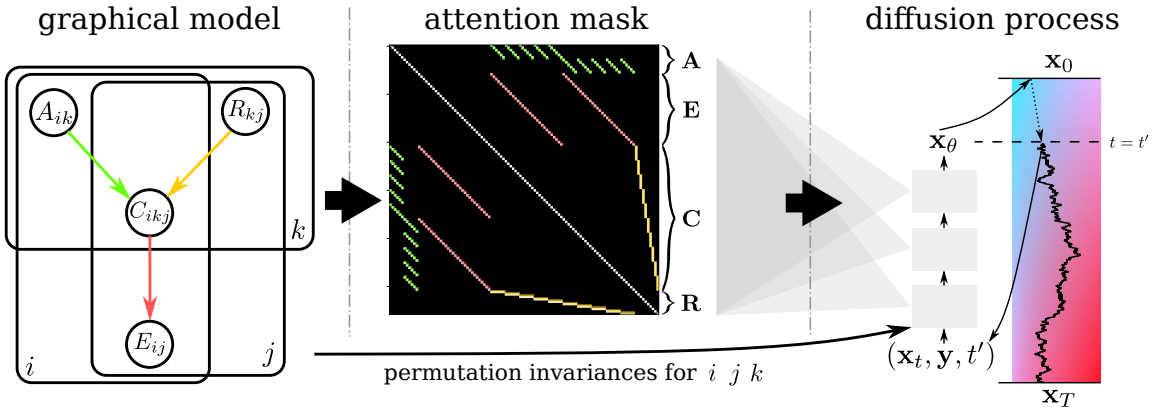


Figure 1: An application of our framework to binary-continuous matrix factorization. In the first panel the computational graph of the multiplication of the continuous matrix $\boldsymbol{A} \in \mathbb{R}^{6\times 2}$ and the binary matrix $\boldsymbol{R} \in \mathbb{R}^{2\times 5}$ is expanded as a probabilistic graphical model in which intermediate products $\boldsymbol{C}$ are summed to give $\boldsymbol{E} = \boldsymbol{AR}$. This graph is used to create a structured attention mask $\boldsymbol{M}$, in which we highlight 1's with the color of the corresponding graphical model edge and self-edges in white. In the third panel the projection into the sparsely-structured neural network guiding the diffusion process is illustrated. In the bottom the translation of permutation invariances of the probability distribution into the embeddings of $\mathbf{x}_t$ and $\mathbf{y}$ is shown (Section 3.6).

ables into GSDM and empirically demonstrate their benefits. **(3)** We present a method for translating known permutation invariances of a model into permutation invariances of GSDM. **(4)** We demonstrate that GSDM architectures can solve hard combinatorial problems. **(5)** As part of the experiments we also provide a new approximate algorithm for binary continuous matrix factorization.

2 Background

2.1 Conditional diffusion models

Defining $\mathbf{x}_0$ to be data sampled from a data distribution $q(\mathbf{x}_0)$, a diffusion process constructs a chain $\mathbf{x}_{0:T}$ with noise added at each stage by the transition distribution

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

leading to the joint distribution

$$q(\mathbf{x}_{0:T}) = q(\mathbf{x}_0)\prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}). \tag{2}$$

The sequence $\beta_{1:T}$ controls the amount of noise added at each step and, along with T itself, is chosen to be large enough that the marginal $q(\mathbf{x}_T|\mathbf{x}_0)$ resulting from Eq. (1) is approximately a unit Gaussian for any $\mathbf{x}_0$.

This diffusion process inspires a diffusion model [Sohl-Dickstein et al., 2015, Ho et al., 2020, Song et al., 2021b], or DM, which approximately "inverts" it using a neural network that outputs $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \approx q(\mathbf{x}_{t-1}|\mathbf{x}_t)$. We can sample from a diffusion model by first sampling $\mathbf{x}_T \sim p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$ and then sampling $\mathbf{x}_{t-1} \sim p_\theta(\cdot|\mathbf{x}_t)$ for each $t = T, \ldots, 1$, eventually sampling $\mathbf{x}_0$. In the conditional DM variant [Tashiro et al., 2021] the neural network is additionally conditioned on information $\mathbf{y}$ so that the modelled distribution is

$$p_\theta(\mathbf{x}_{0:T}|\mathbf{y}) = p(\mathbf{x}_T)\prod_{i=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{y}). \tag{3}$$

The transitions $p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{y})$ are typically approximated by a Gaussian with non-learned diagonal covariance, and so the learning problem is simply to fit the Gaussian's mean. Ho et al. [2020] parameterize this mean as an affine function of $\mathbb{E}[\mathbf{x}_0|\mathbf{x}_t, \mathbf{y}]$ and, by doing so, reduce the problem to fitting an estimator of $\mathbf{x}_0$ from $\mathbf{x}_t$ and $\mathbf{y}$ with the loss

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \mathbb{E}_{q(\mathbf{x}_0, \mathbf{x}_t, \mathbf{y})}\left[\lambda(t) \cdot \|\hat{\mathbf{x}}_\theta(\mathbf{x}_t, \mathbf{y}, t) - \mathbf{x}_0\|_2^2\right]. \tag{4}$$

Ho et al. [2020], Song et al. [2021a] show that there exists a weighting function $\lambda(t)$ such that this loss is (the negative of) a lower-bound on the marginal log-likelihood $\log p_\theta(\mathbf{x}_0|\mathbf{y})$. We instead use uniform weights $\lambda(t) = 1$ which has been shown to give better results in practice [Ho et al., 2020].

2.2 Transformer architecture

A detailed circuit diagram of our neural architecture for $\hat{\mathbf{x}}_\theta$ is shown in Figure 2, where learnable neural network mappings are denoted by m's. The following is happening at a fixed diffusion time step t. We view the neural network's input $(\mathbf{x}_t, \mathbf{y})$ as a list $\{x_1, \ldots, x_n\}$, for which each element x_j corresponds to one node in the graphical model

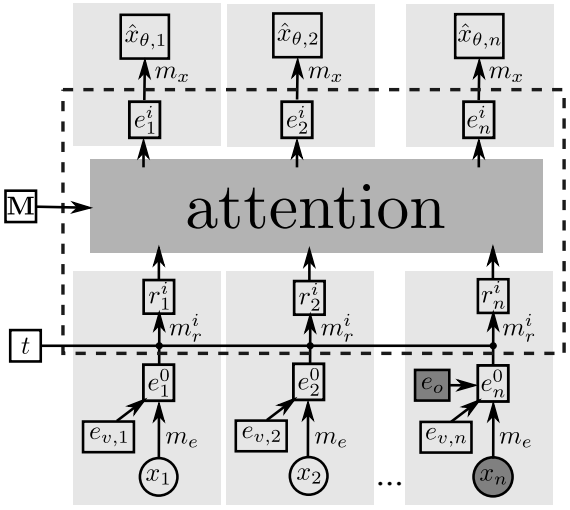


Figure 2: Circuit diagram for $\hat{\mathbf{x}}_\theta$ evaluated at time t as described in Section 2.2. The dashed box represents one repeatable attention layer i that acts on an embedding (its residual connection is not shown). The light grey boxes show parts of the circuit that operate on each dimension independently while the dark grey box shows the masked pairwise attention mechanism.

of interest and comes with side-information denoting which node it corresponds to and whether or not it is observed (i.e. whether it belongs to $\mathbf{x}_t$ or $\mathbf{y}$). Interactions between these elements are governed by self-attention based transformer layers [Vaswani et al., 2017]. Before the first attention layer we embed x_j as $e_j^0 = m_e(x_j)$, add an embedding $e_{v,j}$ describing which graphical model node it belongs to and, for observed variables (like x_n in Figure 2), also add a single globally-shared embedding e_o. Each transformer layer $i \in \{1, ..., L\}$ first applies a residual neural network [He et al., 2016] m_r^i on each of the embedded dimensions e_j^{i-1} independently, $r_j^i = m_r^i(e_j^{i-1}, t)$. Inside the attention each r_j^i is then projected into a query $q_j^i = m_q^i(r_j^i)$, key $k_j^i = m_k^i(r_j^i)$ and value $v_j^i = m_v^i(r_j^i)$, all in $\mathbb{R}^d$. Stacking these n values yields the matrices $\boldsymbol{Q}^i, \boldsymbol{K}^i, \boldsymbol{V}^i \in \mathbb{R}^{n \times d}$. The self-attention, shown in the dark grey box in Figure 2, is then calculated as

$$\boldsymbol{e}^i = \boldsymbol{e}^{i-1} + \boldsymbol{W}^i \boldsymbol{V}^i = \boldsymbol{e}^{i-1} + \mathrm{softmax}\left(\boldsymbol{Q}^i \boldsymbol{K}^{iT}\right) \boldsymbol{V}^i \tag{5}$$

where the addition of $\boldsymbol{e}^{i-1}$ corresponds to a residual connection. We note that $\boldsymbol{Q}^i \boldsymbol{K}^{iT}$ yields a pairwise interaction matrix which lets us impose an additional attention mask $\boldsymbol{M}$ before calculating the output $\boldsymbol{e}^i = \boldsymbol{e}^{i-1} + \mathrm{softmax}\left(\boldsymbol{M} \odot \boldsymbol{Q}^i \boldsymbol{K}^{iT}\right) \boldsymbol{V}^i$. This masking interface is central to structure the flow of information between graphical model nodes in Section 3.1 .

2.3 Permutation invariance

Large probabilistic models often contain permutation invariance, in the sense that the joint probability density $q(\mathbf{x}_0)$ is invariant to permutations of certain indices [Bloem-Reddy and Teh, 2020]. For example the matrix multiplication in Fig. 1 is invariant with respect to permutations of any of the plate indices.[2] If the joint probability density is invariant to a particular permutation, this can be enforced in a distribution modelled by a DM by making the neural network architecture equivariant to the same permutation [Hoogeboom et al., 2022]. We show how to encode such equivariances in GSDM in Section 3.6.

3 Method

A GSDM is a DM which operates over the nodes of a graphical model and whose architecture reflects side information, chiefly in the form of a graphical model structure. This section describes both how to provide useful side information and the process for constructing GSDM from given side information.

3.1 Structure from graphical models

Integrating a graphical model for the data distribution speeds up learning and increases computational efficiency, and these benefits scale with problem dimension. If only a data set without a generative model is available, additional reasoning is required to define a graphical structure. We emphasize that there is considerable flexibility in the specification of this structure. It can be written as a directed graphical model, undirected graphical model, factor graph (where group constraints are represented by connecting all nodes within the group) or a combination of these. We find in Section 4.3 that GSDM is not sensitive to small changes in the specification of the graphical model and that there can be multiple modeling perspectives yielding similar GSDM performance. In general, we use the most intuitive graphical model that we can come up with for each problem, whether directed or undirected. One way to derive a directed model is to follow the definition of a causal forward model for an inverse problem where the terminal node will be the input to our approximate algorithm, as for BCMF in Fig. 1. An undirected graphical model can be seen as implicitly grouping nodes by constraints, e.g. for Sudoku with row, column and block constraints (Appendix D). Note that if the data is simulated and source code is available then we can automatically extract the simulator's compute graph as a graphical model (Appendix B).

Given a graphical model, we train a DM to model the joint distribution of its nodes, and use the edges to construct attention masks $\boldsymbol{M}$ for the DM's transformer layers. Precisely,

[2] In general, plate notation implies permutation invariance as long as no link functions depend on the plate indices themselves.

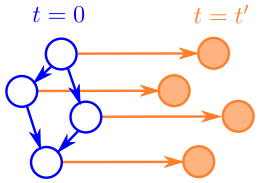


Figure 3: Example graphical model of $q(\mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)$. Nodes are latent/blue if in $\mathbf{x}_0$, observed/orange if in $\mathbf{x}_t$.

we allow variable i to attend to variable j iff there is an edge between node i and node j, and irrespective of the direction of the edge or whether it has a direction.

3.2 Faithful structured attention

According to the DM loss in Eq. (4), the neural network $\hat{\mathbf{x}}_\theta$ is tasked at each timestep $t = t'$ with predicting $\mathbf{x}_0$ from $\mathbf{x}_{t'}$. Figure 3 shows an example graphical model factorization for $q(\mathbf{x}_0|\mathbf{x}_{t'})$ corresponding to this task. Since $\mathbf{x}_{t'}$ can be generated by adding independent noise to each dimension of $\mathbf{x}_0$, this graphical model is derived from the graphical model of the data distribution $q(\mathbf{x}_0)$ by simply adding an edge from each variable in $\mathbf{x}_0$ to the corresponding variable in $\mathbf{x}_{t'}$.

Theorem 1 (Dependence in diffusion models)**.** *Given that the data distribution $q(\mathbf{x}_0)$ is represented by a connected graphical model $\mathcal{G} = (\mathcal{X}, \mathcal{A})$, with nodes $\mathcal{X}$ and edges $\mathcal{A}$, we can represent the temporally combined graphical model at times $t = 0$ and $t = t'$ as $\mathcal{G}_{DM} = (\{\mathbf{x}^i_{t'}\}_i \cup \{\mathbf{x}^i_0\}_i, \{(\mathbf{x}^i_0, \mathbf{x}^i_{t'})\}_i \cup \mathcal{A})$. Then there are no pairs i, j such that $\mathbf{x}^i_0$ can be assumed independent of $\mathbf{x}^j_{t'}$ after conditioning on all other dimensions of $\mathbf{x}_{t'}$. In other words, for any i, j pair, we have to assume $\mathbf{x}^i_0 \not\perp\!\!\!\perp \mathbf{x}^j_{t'} \mid \mathbf{x}^{-j}_{t'}$, where $\mathbf{x}^{-j}_{t'}$ stands for all nodes in $\mathbf{x}_{t'}$ except node j.*

Proof. For any j, $\mathbf{x}^j_{t'}$ is directly connected to $\mathbf{x}^j_0$. Since we assumed that the graphical model for $q(\mathbf{x}_0)$ is connected, there will further be a path from $\mathbf{x}^j_0$ to $\mathbf{x}^i_0$ which does not pass through any conditioned on nodes for any i. Therefore $\mathbf{x}^j_{t'}$ cannot be d-separated from $\mathbf{x}^i_0$ [Koller and Friedman, 2009]. □

The consequence of Theorem 1 is that every node in the neural network output $\hat{\mathbf{x}}_\theta(\mathbf{x}_t, \mathbf{y}, t)$ should depend on every node in its input $\mathbf{x}_t$. Neural network architectures without this property might not be able to faithfully predict $\mathbf{x}_0$ given $\mathbf{x}_t$ and so might not faithfully model $q(\mathbf{x}_0)$. This consideration motivated our previously-described design choice that variable i can attend to j if there is any edge between them, irrespective of the direction of the edge. Otherwise, if the graphical model is directed and acyclic there will not be a path between every pair of nodes. This would cause GSDM to make false independence assumptions, which we show impacts performance in Appendix H. We also note that if

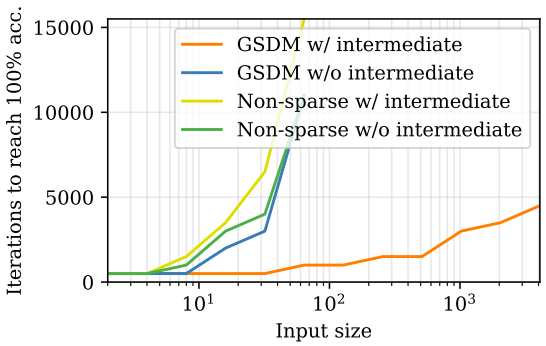


Figure 4: Number of iterations to reach 100% accuracy, validating on 16 examples every 500 iterations.

node i is not directly connected to node j in our attention mask, information about node i may have to be passed to node j via other nodes. Since messages are only passed along one edge per transformer layer, the number of transformer layers should be chosen to be at least as great as the maximum path length in the symmetrized graphical model.

To reduce our memory usage and computation compared to a dense matrix multiplication of the masked matrix we provide an efficient sparse attention implementation as described in Appendix C. The computational cost of our approximate algorithms is $\mathcal{O}(nmL)$ for L attention layers, where n is the number of dimensions and m is the number of entries in the densest row of $\boldsymbol{M}$.

3.3 Intermediate variables

When translating a generative model into a graphical model, the modeler must choose how fine or coarse the graphical representation should be by selecting which, and how many, "intermediate" variables are included and therefore modelled by GSDM. The optimal choice of granularity is model-specific but we argue that fine-grained representations are often better for GSDMs due to **(1)** the better learning signal that we describe below and **(2)** the reduced computational cost of sparse attention that is related to the number of graphical model *edges* more so than the number of nodes, and so is not necessarily increased by adding intermediate variables.

GSDM can be much easier to train using a fine-grained representation because each node in a fine-grained graphical model will typically have few parents and be a relatively simple function of its parents.[3] Since the DM training objective involves predicting (and receiving supervision for) the value of every node, approximately we can imagine

[3]To take this argument to an extreme, we could represent a generative model with a compute graph where each node represents a single output of a CPU's arithmetic logic unit (ALU). Since the ALU operates on pairs of inputs, each node will have at most two parents and each link function will be simple enough to execute in one CPU cycle.

Experiment	Graphical model	Conditioned on	Struct. attn.	Interm. vars	Disc. & cont.	Emb. sharing
BCMF	directed	$\boldsymbol{E}$	✓	✓	✓	(✓)
Sudoku	factor graph	random subset	(✓)	-	-	(✓)
Sorting	mixed	$\boldsymbol{u}$	✓	✓	✓	(✓)
Boolean	directed	input	✓	✓	-	-

Table 1: Problems tackled. A tick, ✓, highlights where our contributions were necessary to learn at all or scale with problem dimension and (✓) where they improved performance. The improvements from structured attention and intermediate variables are shown in Fig. 4 and Fig. 7.

each link function as being learned in parallel during training. From this perspective, making the link functions very simple means that DM training is similar to learning many very simple functions in parallel rather than, e.g., one complex function that would require many training steps.

As an illustrative example, consider a Boolean logic circuit which takes an input of size 2^n. The input is split into pairs and each pair is mapped through a logic gate to give an output of size 2^{n-1}. After n layers and a total of $2^n - 1$ logic gates, there is a single output. Suppose that you know that each gate is randomly assigned to be either an OR gate or an AND gate, and you wish to infer which from data. If the data contains only the inputs and the single output, it contains only 1 bit of information. Inferring the function computed by each of the $\mathcal{O}(2^n)$ gates will therefore require $\mathcal{O}(2^n)$ data points. On the other hand, if the data contains intermediate variables in the form of the output of every logic gate, each data point contains $\mathcal{O}(2^n)$ bits of information so the task may be solvable with few data points. Figure 4 shows that this reasoning holds up when we train a DM on this example. Without intermediate variables, the number of training iterations needed scales exponentially with n. With the combination of intermediate variables and structured attention, however, the training behaviour is fundamentally changed to scale more gracefully with n.

3.4 Handling mixed continuous/discrete variables

Our simple approach to combining discrete and continuous variables in a DM is to map the discrete variables to one-hot encodings in a continuous space before running the diffusion process. Sampled one-hot encodings can then be mapped back to the discrete space with an $\arg\max$. We project the entire (diffused) one-hot encoding for each discrete variable into a single embedding before passing it into the transformer, so that the transformer performs the same amount of computation for a discrete variable as for a continuous variable.

3.5 Flexible conditioning

Optimizing the DM loss in Eq. (4) requires a partitioning of data into latent variables (outputs) $\mathbf{x}_0$, and observed variables (inputs) $\mathbf{y}$. While traditional amortized inference requires choosing this partitioning before training [Gershman and Goodman, 2014, Ritchie et al., 2016, Le et al., 2017], our approach allows for *flexible conditioning* by training over a distribution of partitions so that certain variables can be latent in some test examples and observed in others. The neural network distinguishes between variables in $\mathbf{x}_t$ and $\mathbf{y}$ via a learned observation embedding vector $\boldsymbol{e}_o$ that is added to the embeddings of observed variables. This approach also naturally allows us to deal with missing values at inference time, something that neither standard algorithms nor traditional amortized inference artifacts can deal with.

3.6 Embeddings and varying dimensionality

GSDM's architecture contains positional embeddings which inform the neural network which inputs correspond to which graphical model nodes. The simplest variation of GSDM learns one embedding per graphical model node independently, and we call this approach **independent embeddings**, or **IE**. An issue with IE is that it cannot generally be adapted to changing input dimension. For example, a network trained with IE to operate on inputs of size length 10 couldn't be applied to inputs of length 11. As a generic way to tackle problems defined over variable-size arrays we suggest **array embeddings**, or **AE**. Creating these involves analyzing the problem to group together variables that are in the same (potentially multi-dimensional) array, treating scalars as arrays of size 1 in each dimension. We then create the embedding for each variable as the sum of a shared array embedding, learned independently for every array, and a sinusoidal positional embedding [Vaswani et al., 2017] for its position within an array.

For models exhibiting the permutation invariances discussed in Section 2.3 we can optionally use **exchangeable embeddings**, or **EE**. To do so for any index that a model is permutation invariant with respect to, we share embeddings between all variables which are replicas of each other along that index. For example, the matrix multiplication in Fig. 1 is invariant to permutations of i and k so we share embeddings across all A_{ik}. In combination with the structured attention mechanism (which prevents communication between nodes with different indices) this renders the neural network equivariant with respect to these permutations. It also allows the number of repetitions of each plate to be changed

arbitrarily, allowing generalisation for models including our matrix factorization experiment. We derive why the resulting GSDM is fully permutation equivariant in Appendix I.

4 Experiments

Binary continuous matrix factorization (BCMF) Our first experiment tackles the challenging BCMF problem, where we factorize a continuous matrix into one binary and one continuous component. Our BCMF generative model samples a binary matrix $\boldsymbol{R} \in \mathbb{R}^{k \times n}$ elementwise from Bernoulli(0.3) and a continuous matrix $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ from an elementwise Uniform$(0, 1)$ prior. The BCMF task is to infer these conditioned on $\boldsymbol{E} := \boldsymbol{AR}$. To obtain intermediate variables as discussed in Section 3.3 we break the matrix multiplication into two steps, $C_{ijk} := A_{ik} R_{kj}$ and then $E_{ij} := \sum_k C_{ijk}$. Our latent variables $\mathbf{x}_0$ are therefore the combination of all elements of $\boldsymbol{R}$, $\boldsymbol{A}$, and $\boldsymbol{C}$ and the observed variables $\mathbf{y}$ are the elements of $\boldsymbol{E}$. Fig. 6 shows that our learned GSDM can find factorisations that are consistent with the observations. Furthermore, we show in Fig. 5 that GSDM scales well, even to larger m and n. The model was trained on ranges uniformly sampled in the range 1 to 10 for m, n and k, but generalizes to 20 times larger m and n.

Sudoku A Sudoku grid is a 9×9 array of numbers such that each number is in $\{1, \dots, 9\}$ and no two numbers in the same row, column, or 3×3 block are the same. Solving a Sudoku, i.e. completing one given a partially-filled in grid, is a difficult problem for deep learning methods, and has previously been addressed with hand-designed modules for reasoning [Palm et al., 2018] or semi-definite programming [Wang et al., 2019]. We use GSDM without such custom modules for combinatorial reasoning. We model a Sudoku with a factor graph. There is one factor for each row, column, and block representing the constraint that it contains all numbers $\{1, \dots, 9\}$. The resulting GSDM attention mask lets each variable attend to all other variables in the same row, column, and block. Our data generator[4] creates complete 9×9 Sudokus. In order to train GSDM as a Sudoku solver for arbitrary Sudoku puzzles, we create each training example by randomly partitioning the grid into latent and observed portions by sampling n_o uniformly from 0 to 80 and then uniformly sampling n_o variables to observe. We show samples from GSDM in Fig. 8, which exhibit diversity when multiple solutions exist. In our evaluations, the samples were valid Sudokus 76% of the time when no cells were observed; 55% of the time when 16 of the 81 cells were observed; 99% of the time when 40 cells were observed; and always when 64 of the 81 cells were observed.

[4]We generated complete Sudokus with a port of `https://turtletoy.net/turtle/5098380d82`

Sorting We demonstrate GSDM on sorting as evidence of it's broad applicability to a wide variety of problems. Our graphical model is as follows. (1) Sample an unsorted list $\mathbf{u} \in \boldsymbol{R}^n$. with each element u_i sampled from a unit normal. (2) Sample a permutation matrix $\boldsymbol{P} \in \{0, 1\}^{n \times n}$. Similarly to the Sudoku case, factors on each row and column enforce the constraints that there is a single 1 in each. (3) Multiply $\boldsymbol{P}$ and $\mathbf{u}$. We integrate intermediate variables $C_{ij} := P_{ij} u_j$ and sum them as $s_i := \sum_j C_{ij}$ to yield $\mathbf{s}$. (4) We use factors between each pair of elements in $\mathbf{s}$ to enforce that it is sorted. This graphical model is different to, and simpler than, our true data generation procedure in which we obtain $\mathbf{s}$ and $\boldsymbol{P}$ with a pre-existing sorting algorithm. We emphasise that this approach fits into the GSDM framework nonetheless since the graphical model is a valid specification of the independences in the data distribution. We train GSDM to sort lists with sizes ranging from 2 to 40. We measure it's performance as the mismatch between the real and sampled permutation matrices $\boldsymbol{P}$, and plot progress throughout training in Fig. 7c. During evaluation, GSDM made errors on only two of 80 test examples. In each of these two cases it confused a single pair of elements that were 2×10^{-3} apart in a list of length 20 and 1×10^{-5} apart in a list of length 30.

Boolean We additionally use the Boolean circuit described in Section 3.3 (Fig. 4) to demonstrate GSDM's ability to learn structured functions over many variables.

4.1 Effect of structured attention and intermediate variables

All of our experiments rely on structured attention for their good performance, and the positive effect remains even after removing intermediate variables. We saw this for the Boolean circuit in Section 3.3 and here show experiments for Sudoku in Fig. 7b, sorting lists of length $n = 20$ in Fig. 7c, and BCMF of various dimensionalities in Fig. 7a. These are relatively simple ablations in that we do not vary the dimensionality during training but, even so, imposing structure leads to significant improvements in each case, especially in combination with intermediate variables.

We ablate intermediate variables in the same figures. In combination with structured attention, including intermediate variables is extremely helpful in all cases. For BCMF the intermediate variables reduced the error in Fig. 7a by an average factor of roughly 4.3. For the Boolean circuit, structured attention removes the exponential blow-up in the number of required training iterations. For sorting, intermediate variables sped up the time to reach 99% accuracy by a factor of 3.

Sparsely structured attention also makes the network faster to execute. Even despite the substantial number of intermediate variables needed to take advantage of sparsity in

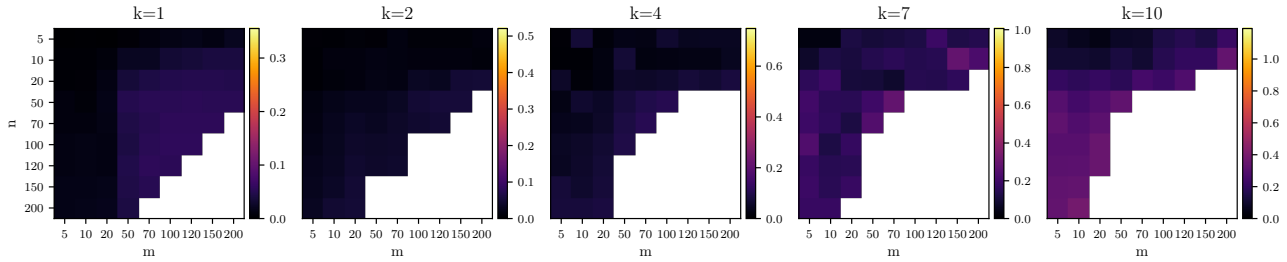


Figure 5: Error vs problem dimension for GSDM on binary continuous matrix factorization. We show the root mean square error between the observed matrix $\boldsymbol{E}$ and the product of the sampled $\boldsymbol{A}$ and $\boldsymbol{R}$. The colorbar for each value of k is scaled so that "yellow" corresponds to the error achieved by a baseline which samples $\boldsymbol{A}$ and $\boldsymbol{R}$ from the prior, ignoring $\boldsymbol{E}$. Despite never seeing a value of n, m, or k larger than 10 during training, GSDM scales well to much larger values of m and n. When they grow large enough, GSDM runs out of GPU memory. We mark entries where this occurred in white.

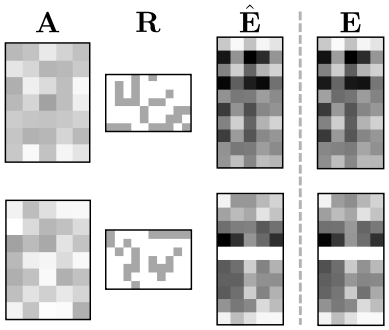


Figure 6: Two factorizations sampled by GSDM with $m = 5$, $n = 10$ and $k = 7$. Each row in the plot shows the inferred $\boldsymbol{A}$, $\boldsymbol{R}$ and the respective reconstruction $\hat{\boldsymbol{E}} = \boldsymbol{R}\boldsymbol{A}$ on the left and the $\boldsymbol{E}$ provided as input on the right. Intermediate variables $\boldsymbol{C}$ are omitted here.

e.g. BCMF, we show in Appendix G that the computational complexity of GSDM with intermediate variables and structured attention is better than that of a naive approach with neither in all experiments with parameterizable size (i.e. all but Sudoku).

4.2 Effect of embedding-sharing

Three of our problems contain permutation invariances exploitable by embedding-sharing. Our sorting model is invariant to the order of $\mathbf{u}$, Sudoku to various row/column permutations, and BCMF to the permutations of any of the plate indices in Fig. 1. We see in Fig. 7c that incorporating these invariances with EE gives much faster training than using independent embeddings. Even without specifying the invariances, AE can be used to obtain most of the benefit. The same story can be seen in Fig. 7b for Sudoku. All of our results for BCMF use EE, and we emphasise that the results showing generalization with respect to problem dimensions would not be possible without embedding sharing.

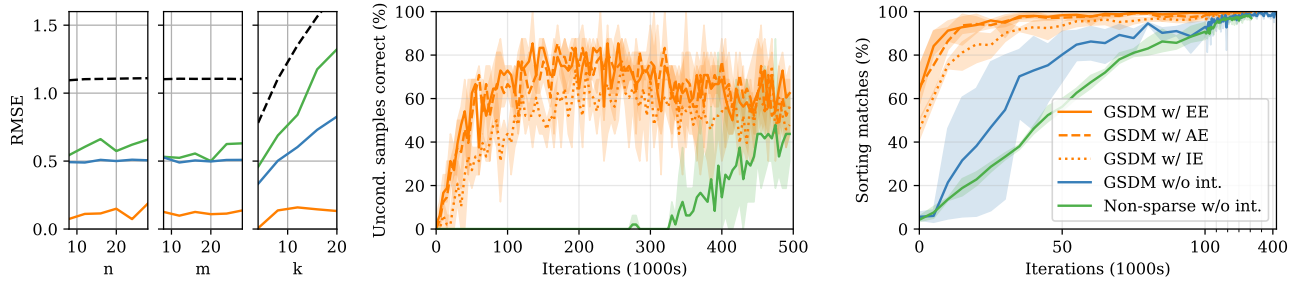
4.3 Robustness to choice of graphical model

There can be a degree of subjectivity in the choice of graphical model for a given problem. For instance, in sorting we represented the constraint that $\mathbf{s}$ is sorted with pairwise constraints between neighbouring elements. Another reasonable graphical model may have imposed a factor over all nodes of $\mathbf{s}$, making $\mathbf{s}$ fully-connected in our attention mask. We show in Fig. 9 that this choice makes negligible difference to GSDM's performance. Furthermore, some modeling choices make no difference at all to GSDM. For example we sampled $\mathbf{u}$ and $\boldsymbol{P}$ first and then computed $\mathbf{s} := \boldsymbol{P}\mathbf{u}$, but someone else may have sampled $\mathbf{s}$ first (with factors to ensure it is sorted) and then $\boldsymbol{P}$ before computing $\mathbf{u} := \boldsymbol{P}\mathbf{s}$. These two choices lead to identical GSDM networks because the only difference is the direction of edges in the graphical model (which is irrelevant when they are symmetrized to create the attention mask). Figure 9 also shows that GSDM can be robust to a misspecified model, "Unconstrained $\mathbf{s}$", where constraints are not imposed on $\mathbf{s}$. The "Random" baseline, however, in which each node is allowed to attend to 20 other nodes sampled at random, performs much worse.

5 Related work

Sparse attention mechanisms have been introduced in several forms, either to save memory footprint [Dai et al., 2019, Kitaev et al., 2020, Roy et al., 2020] or computational cost [Child et al., 2019, Beltagy et al., 2020, Zaheer et al., 2021]. A recent review is provided in Tay et al. [2022].

The framework of amortized inference [Gershman and Goodman, 2014, Ritchie et al., 2016] and probabilistic programming [Le et al., 2017, van de Meent et al., 2018] provides the framework for our approach. But instead of requiring a full probabilistic model we relax the requirement to only specify the graphical model structure. Weilbach et al. [2020] use a graphical model to structure a continuous normalizing flow for inference problems of fixed dimen-



(a) BCMF error vs dimensionality. (b) Sudoku performance throughout training. (c) Sorting performance throughout training.

Figure 7: Ablations over the use of structured attention, intermediate variables, and different types of embedding sharing. In (a) we show the test error on BCMF as each dimension is varied. While varying n, we set $m = 16$, $k = 8$. While varying m, we set $n = 16$, $k = 8$. While varying k, we set $n = m = 16$. The dashed line marks the best error with constant estimates of $\boldsymbol{A}$ and $\boldsymbol{R}$. Combining intermediate variables with structured attention greatly reduces the error, and prevents it from increasing with the rank k. In (b) we show the proportion of unconditionally-generated Sudokus that are valid over the course of training. For GSDM this begins to decrease halfway through training, but the accuracy for conditionally-generated Sudokus continues to rise. In (c) we measure how well sorted lists match the ground-truth. In all cases, attention and intermediate variables make training much faster. AE provides a significant improvement over IE, while EE provides a small further improvement over AE.

4	3	8	1	9	**6**	5	7	2
6	**9**	**2**	**7**	5	8	**3**	1	4
7	5	1	2	3	4	9	6	8
3	4	6	9	2	**1**	8	5	7
9	8	5	6	4	7	**2**	3	1
1	2	7	**5**	8	3	4	9	6
2	1	**3**	8	6	**9**	7	4	**5**
5	7	4	3	1	2	6	**8**	**9**
8	6	9	4	**7**	5	**1**	**2**	3

7	3	8	4	9	**6**	5	1	2
1	**9**	**2**	**7**	5	8	**3**	4	6
4	5	6	1	3	2	9	7	8
5	2	4	9	6	**1**	8	3	7
9	6	1	3	8	7	**2**	5	4
3	8	7	**5**	2	4	6	9	1
2	1	**3**	8	4	**9**	7	6	**5**
6	7	5	2	1	3	4	**8**	**9**
8	4	9	6	**7**	5	**1**	**2**	3

Figure 8: Two Sudoku solutions conditioned on the same 16 observed cells (in bold).

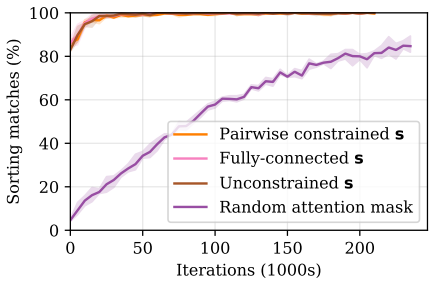


Figure 9: Ablations of GSDM with different graphical model structures for sorting. The first three lines, which represent constraints on $\mathbf{s}$ in different ways described in Section 4.3, are on top of eachother and quickly reach 100% accuracy. This speaks to the robustness of GSDM to different ways of specifying a graphical model. Randomising the attention mask works very poorly in comparison.

sion. We use the more flexible and scalable DM framework including discrete variables. We can also work directly with the forward probabilistic model to avoid computing the potentially denser stochastic inverse of Webb et al. [2017].

Close in spirit to our work in terms of combinatorial optimization are Selsam et al. [2019] for general SAT solving and Tönshoff et al. [2022] for general CSP solving, which also encode the structure between variables and constraints as message passing neural networks. But these frameworks are only applicable to deterministic discrete problem classes, while we integrate everything in the more general probabilistic inference framework.

Our approach to explicitly training conditional diffusion models is based on that of Tashiro et al. [2021], Harvey et al. [2022]. Various other methods train unconditional diffusion models before providing approximate conditioning at test-time [Song et al., 2021b, Ho et al., 2022]. Most DMs are defined over either purely continuous [Ho et al., 2020] or purely discrete spaces [Austin et al., 2021, Hoogeboom et al., 2021]. Our approach to mixed-continuous DMs is similar to that of Hoogeboom et al. [2022] but takes a variational-dequantization perspective [Ho et al., 2019] so that mapping back to the discrete space involves taking an $\arg\max$ instead of requiring sampling.

6 Discussion

GSDMs automate the reasoning required to create approximate solutions to tasks as diverse as sorting, Sudoku solving and binary-continuous matrix factorization. The two main steps needed for our approach are acquiring a data set and describing the class of problems as a graphical model. If a full generative model is available our approach additionally benefits in accuracy and scalability from the access to

computed intermediate values. Direct translation of known permutation invariances of the data distribution further improve GSDMs. Our work is a step towards the integration of the generality of statistical conditioning, the expressivity of state-of-the-art diffusion models with attention mechanisms, and the structural reasoning applied in programming language theory and algorithm design. Promising directions for future work include deeper integration of structural knowledge with the dynamics of diffusion processes, exploitation of syntactic knowledge of problem descriptions beyond the class of graphical models and integration into probabilistic programming systems.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada CIFAR AI Chairs Program, and the Intel Parallel Computing Centers program. Additional support was provided by UBC's Composites Research Network (CRN), and Data Science Institute (DSI). This research was enabled in part by technical support and computational resources provided by WestGrid (www.westgrid.ca), Compute Canada (www.computecanada.ca), and Advanced Research Computing at the University of British Columbia (arc.ubc.ca). WH acknowledges support by the University of British Columbia's Four Year Doctoral Fellowship (4YF) program.

References

Jacob Austin, Daniel D Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems*, 34:17981–17993, 2021.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer, December 2020.

Benjamin Bloem-Reddy and Yee Whye Teh. Probabilistic Symmetries and Invariant Neural Networks. *Journal of Machine Learning Research*, 21(90):1–61, 2020. ISSN 1533-7928.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating Long Sequences with Sparse Transformers, April 2019.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, June 2019.

Samuel Gershman and Noah Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the Cognitive Science Society*, volume 36, 2014.

William Harvey, Saeid Naderiparizi, Vaden Masrani, Christian Weilbach, and Frank Wood. Flexible Diffusion Modeling of Long Videos, May 2022.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning*, pages 2722–2730. PMLR, 2019.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851. Curran Associates, Inc., 2020.

Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J. Fleet. Video Diffusion Models. *arXiv:2204.03458 [cs]*, April 2022.

Emiel Hoogeboom, Alexey A Gritsenko, Jasmijn Bastings, Ben Poole, Rianne van den Berg, and Tim Salimans. Autoregressive diffusion models. *arXiv preprint arXiv:2110.02037*, 2021.

Emiel Hoogeboom, Vıctor Garcia Satorras, Clément Vignac, and Max Welling. Equivariant diffusion for molecule generation in 3d. In *International Conference on Machine Learning*, pages 8867–8887. PMLR, 2022.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The Efficient Transformer. *arXiv:2001.04451 [cs, stat]*, February 2020.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2009. ISBN 978-0-262-01319-2.

Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

Rasmus Palm, Ulrich Paquet, and Ole Winther. Recurrent Relational Networks. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Daniel Ritchie, Paul Horsfall, and Noah D. Goodman. Deep Amortized Inference for Probabilistic Programs. *arXiv:1610.05735 [cs, stat]*, October 2016.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-Resolution Image Synthesis With Latent Diffusion Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient Content-Based Sparse Attention with Routing Transformers, October 2020.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. *arXiv:1802.03685 [cs]*, March 2019.

Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2256–2265. PMLR, June 2015.

Yang Song, Conor Durkan, Iain Murray, and Stefano Ermon. Maximum Likelihood Training of Score-Based Diffusion Models. In *Advances in Neural Information Processing Systems*, volume 34, pages 1415–1428. Curran Associates, Inc., 2021a.

Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-Based Generative Modeling through Stochastic Differential Equations. *arXiv:2011.13456 [cs, stat]*, February 2021b.

Yusuke Tashiro, Jiaming Song, Yang Song, and Stefano Ermon. CSDI: Conditional score-based diffusion models for probabilistic time series imputation. In *Advances in Neural Information Processing Systems*, 2021.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey, March 2022.

Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One Model, Any CSP: Graph Neural Networks as Fast Global Search Heuristics for Constraint Satisfaction. August 2022. doi: 10.48550/arXiv.2208.10227.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. *arXiv:1809.10756 [cs, stat]*, September 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Po-Wei Wang, Priya L. Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *arXiv:1905.12149 [cs, stat]*, May 2019.

Stefan Webb, Adam Golinski, Robert Zinkov, N. Siddharth, Tom Rainforth, Yee Whye Teh, and Frank Wood. Faithful Inversion of Generative Models for Effective Amortized Inference. December 2017.

Christian Weilbach, Boyan Beronov, William Harvey, and Frank Wood. Structured Conditional Continuous Normalizing Flows for Efficient Amortized Inference in Graphical Models. In *International Conference on Artificial Intelligence and Statistics*, pages 4441–4451. PMLR, June 2020.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences, January 2021.

A Experimental details

Table 2: Experimental parameters. For sorting and BCMF we have two sets of hyperparameters: one for the main results and one for the ablations in Figs. 7, 9 and 15. Problem dimension is listed as "varying" where we vary the dimensions between each training batch (sampling n between 2 and 10 for the "Sorting" column and sampling n, m, and k independently from 1 to 10 for the "BCMF" column) and "multiple" where we show results with different problem dimensionalities.

Parameter	Sorting	Sorting abl.	Sudoku	BCMF	BCMF abl.	Boolean
Problem dimension	varying	$n = 20$	9×9	varying	multiple	multiple
Training time	1 day	8 hours	7-8 hours	1 day	1 day	40-160 min.
Training iters (1000s)	140	300-450	500	680	180-680	20
Batch size	16	16	32	32	8	16
Learning rate	2×10^{-4}	2×10^{-4}	2×10^{-5}	2×10^{-5}	2×10^{-5}	2×10^{-5}
Embedding dim.	64	64	128	64	64	64
# transformer layers	6	6	6	12	12	12
# attention heads	8	1	8	2	2	2
GPU type	A100	A5000	A5000	A100	A5000	A5000

Table 2 presents our experimental parameters. Our ablations on sorting and BCMF give all networks equal training time, and the number of iterations therefore varies depending on the time to run the network. We tuned the learning rates through small grid searches but this yielded only a small improvement to training. In keeping with common deep learning wisdom, we found that increasing the embedding dimension and number of transformer layers improved performance, as does using multiple attention heads. Conversely, the results degrade gracefully in smaller embedding dimensions, less transformer layers or varying numbers of attention heads. We set these architectural hyperparameters with the goal of obtaining networks that were both lightweight and trained quickly, and tuned them via some experimentation for sorting, Sudoku, and BCMF. We use NVIDIA A100 GPUs for sorting and BCMF, and smaller NVIDIA RTX A5000s for all ablations and other problems. We did little tuning on the batch sizes, other than ensuring that they were large enough to obtain good GPU utilization and small enough to avoid out-of-memory errors. All data is sampled synthetically on-the-fly, so data points used in one minibatch are never repeated in another minibatch. We use 1000 diffusion timesteps in all experiments and set the hyperparameters $\beta_1, \ldots, \beta_{1000}$ using a linear interpolation schedule [Ho et al., 2020] from $\beta_1 = 10^{-4}$ to $\beta_{1000} = 0.005$. Finally, we use the Adam optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ [Kingma and Ba, 2015], no weight decay and gradient clipping at 1.0. We release code to ensure full reproducibility of our results.

B Automatic compilation of BCMF

Building on the probabilistic programming language defined in van de Meent et al. [2018], we demonstrate a compiler which maps from programs into a corresponding graphical model structure. We will publish our implementation on acceptance. We demonstrate it on the program on page 16, which multiplies two random matrices $\boldsymbol{A} \in \mathbb{R}^{3\times2}, \boldsymbol{R} \in \mathbb{R}^{2\times3}$ similarly to our BCMF experiment. Samples from Dirac distributions are used to introduce the intermediate nodes of $\boldsymbol{C}$ and the terminal nodes of $\boldsymbol{E}$. Our compiler first translates it into a graphical model and then into the attention mask as shown in Figure 10. We envisage a future extension which "compiles" directly from such source code to a trained GSDM network.

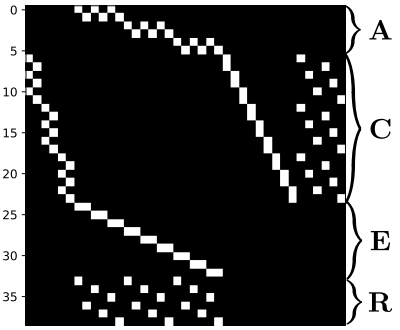
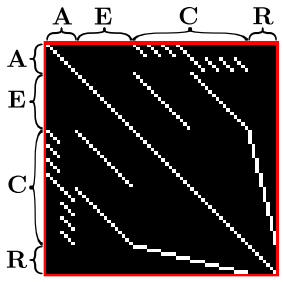


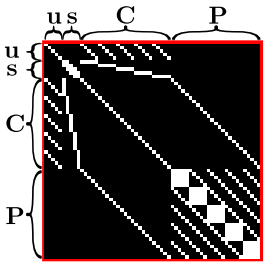
Figure 10: Connectivity mask extracted from BCMF source code. This is the same structure as in Figure 1 but with permuted indices and before the addition of the diagonal self-edges.

C Structured attention

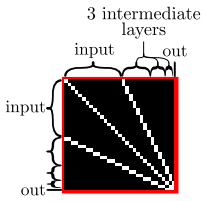
The optimal choice for our structured attention would be sparse matrix multiplication on the accelerator, unfortunately this was not yet available at the time of writing of this paper. We therefore provide a packed dense implementation of structured attention. Our structured attention mechanism lets us reduce the computational and memory cost of an n-dimensional DM from $\mathcal{O}(n^2)$ to $\Theta(nm)$, where m is the maximum number of ones in any row of our attention mask $\boldsymbol{M}$.



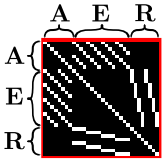
(a) BCMF ($n = 4, m = 4, k = 2$).



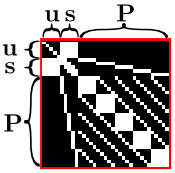
(b) Sorting ($n = 5$).



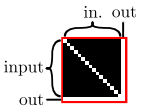
(c) Boolean circuit ($n = 4$).



(d) BCMF w/o intermediate.



(e) Sorting w/o intermediate.



(f) Boolean circuit w/o intermediate

Figure 11: Attention masks for BCMF, sorting, and the Boolean circuit. All are shown with (top row) and without (bottom row) intermediate variables. Variable i can attend to variable j iff the cell in row i and column j is white. These masks all become more sparse (as measured by proportion of entries which are non-zero) as the problem dimensionality is increased.

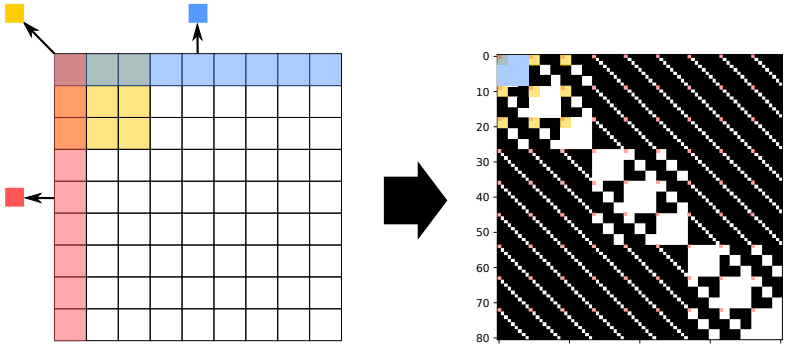


Figure 12: Correspondence between a 9×9 Sudoku grid (left) and the resulting attention mask. We draw a factor for each of the first row, column and block on the left. On the right the respective entries in the attention structured mask $\boldsymbol{M}$ are highlighted with the same color.

For our BCMF experiment, the reduction in memory footprint was necessary for us to scale to the dimensions demonstrated while using a single GPU. Recall that, after computing keys $\boldsymbol{K}$, values $\boldsymbol{V}$, and queries $\boldsymbol{Q}$ (all with shape $n \times d$, where d is the embedding dimension), and given a attention mask $\boldsymbol{M}$ with shape $n \times n$, we compute

$$\boldsymbol{e} = \text{softmax}\left(\boldsymbol{M} \odot \boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}. \tag{6}$$

If implemented naively with dense matrix multiplications, both computing $\boldsymbol{Q}\boldsymbol{K}^T$ and the outer multiplication by $\boldsymbol{V}$ involve $\mathcal{O}(n^2)$ scalar operations. We attempt to avoid this cost while still taking advantage of the dense matrix multiplications for which GPUs are designed for. To do so, we project $\boldsymbol{K}$ and $\boldsymbol{V}$ into 3-dimensional matrices $\bar{\boldsymbol{K}}$ and $\bar{\boldsymbol{V}}$ of shape $n \times m \times d$. We perform this projection such that $\bar{\boldsymbol{K}}_i$ is a sequence of the key vectors for every variable that variable i is connected to. Equivalently, letting a_{ij} be the index of the jth variable that variable i is connected to, $\bar{\boldsymbol{K}}_{i,j}$ is equal to $\boldsymbol{K}_{a_{ij}}$. We define $\bar{\boldsymbol{V}}$ similarly for value vectors. If variable i connects to less than m entries, then we pad $\bar{\boldsymbol{K}}_i$ and $\bar{\boldsymbol{V}}_i$ with zeros. We then compute an $n \times m$ array of unnormalised weights $\boldsymbol{U}$ (encompassing all interactions allowed by our attention mask) such that $\boldsymbol{U}_{i,j} = \boldsymbol{Q}_i \cdot \bar{\boldsymbol{K}}_{i,j}$. Doing so involves only $\mathcal{O}(nm)$ operations, rather than the $\mathcal{O}(n^2)$ required to compute dense attention weights. We then mask all entries in $\boldsymbol{U}$ that were padded by setting them to $-\infty$ before applying the usual softmax$(\boldsymbol{U})$ row-wise to get $\bar{\boldsymbol{W}}$. Finally, we can compute the output h by setting each $\boldsymbol{e}_i = \sum_j \bar{\boldsymbol{W}}_{i,j}\bar{\boldsymbol{V}}_j$ (again requiring only $\mathcal{O}(nm)$ operations), which is equal to the output that would be obtained through dense matrix multiplications including the mask $\boldsymbol{M}$.

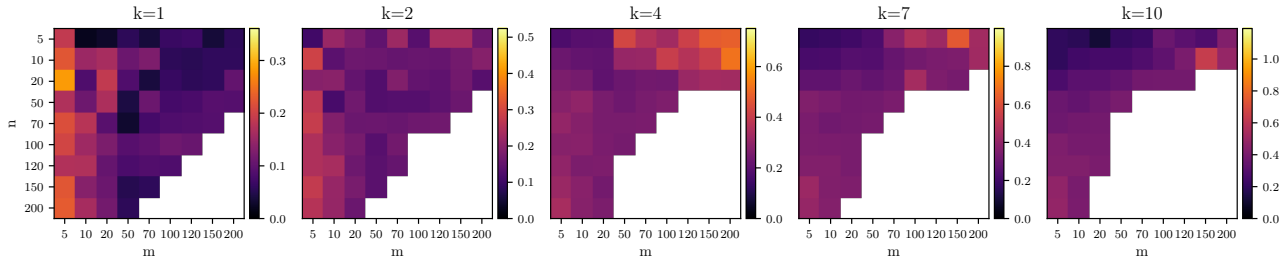


Figure 13: Similar to Figure 5 but with a GSDM model trained to sampled $\boldsymbol{A}$, $\boldsymbol{R}$, and $\boldsymbol{E}$ jointly instead of conditioning on $\boldsymbol{E}$. We plot a heatmap of the root mean squared error (RMSE) between the matrix $\boldsymbol{E}$ and the product $\boldsymbol{AR}$. The ranges for each rank are scaled so that a yellow color represents the expected RMSE if $\boldsymbol{A}$, $\boldsymbol{R}$, and $\boldsymbol{E}$ are all sampled independently from the prior. Entries colored in white exceed the GPU's memory limit.

		Cost of ResNets	Cost of attention	Overall cost	Reduction
Boolean with input size n	Naive	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	-
	GSDM	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Sorting with input size n	Naive	$\mathcal{O}(n^2)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	-
	GSDM	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n)$
BCMF with $k = m = n$	Naive	$\mathcal{O}(n^2)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	-
	GSDM	$\mathcal{O}(n^3)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^4)$	$\mathcal{O}(1)$

Table 3: Comparison of computational complexities of a GSDM layer and a naive DM layer without structured attention or intermediate variables. We do not include Sudoku since the problem has fixed dimension. GSDM yields reductions in complexity that scale with n for the Boolean and sorting experiments, while giving the same complexity as a naive approach for BCMF yet much better performance.

D Attention masks

We show attention masks used by GSDM in each of our experiments. Figure 11 compares masks with and without intermediate variables for BCMF, sorting and the Boolean circuit. Figure 12 explains the derivation of the Sudoku attention mask from a factor graph.

E Unconditional BCMF

In Figure 13 we plot a heatmap similar to Figure 5 but with a GSDM which generates unconditional joint samples of $\boldsymbol{A}$, $\boldsymbol{R}$, and $\boldsymbol{E}$ instead of samples conditioned on $\boldsymbol{E}$. We measure the mismatch between $\boldsymbol{E}$ and the product $\boldsymbol{AR}$ and, interestingly, see that it is greater for the unconditional model (for problem dimensions both inside and outside the training distribution). This suggests that it may be possible to improve unconditional generation performance by adjusting the diffusion process hyperparameters so that $\boldsymbol{E}$ is sampled early in the diffusion process and then $\boldsymbol{A}$ and $\boldsymbol{R}$ are sampled later, conditioned on $\boldsymbol{E}$, but we do not attempt to do so here.

F Ablation on BMF embedding type

Figure 14 shows the effect of embedding type on BCMF performance. As in Fig. 7a, we vary each of m, n, and k in turn and set $m = 16$, $k = 8$ while varying n; set $n = 16$, $k = 8$ while varying n; and set $m = n = 16$ while varying k. The scale for the y-axis is smaller than in Fig. 7a since the embedding type makes a smaller difference to performance than the use of sparsity and intermediate variables. This makes the noise in the results more visible, but it is noticeable that exchangeable embeddings (solid line) mostly provide the best performance. The array embeddings (dashed) are worse than independent embeddings (dotted) for higher dimensionality. This may be because, with sufficient training time, the independent embeddings can learn to approximately match the exchangeable embeddings. When we look at error earlier in training, array embeddings perform slightly better than independent embeddings.

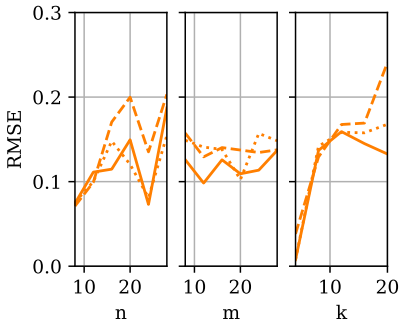


Figure 14: Final RMSE on BCMF as a function of the problem dimension. The solid line uses exchangeable embeddings; the dashed line uses array embeddings; the dotted line uses independent embeddings.

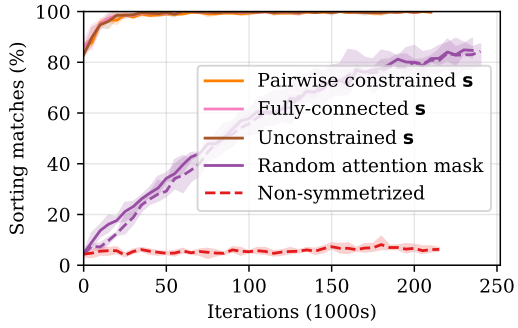


Figure 15: Performance throughout training of sorting with different attention masks corresponding to different graphical models. The solid line uses EE and the dashed line uses AE.

G Computational complexities

Table 3 compares the computational cost of GSDM with that of naively applying a neural network without intermediate variables or structured attention.

H Expanded ablation on attention mask specification

Figure 15 is an expanded version of our ablation on the sorting attention mask. In addition to the lines in Fig. 9 we show the training performance of a non-symmetrized mask (dashed red). For sorting, edges added during symmetrization are necessary to allow any information to flow from the intermediate variables $\boldsymbol{C}$ to the permutation matrix $\boldsymbol{P}$. This blocks the path between the input $\mathbf{u}$ and output of interest $\boldsymbol{P}$, and so the sampled $\boldsymbol{P}$ does no better than random chance at any point in training.

I Permutation equivariant GSDM

The stochastic process of the DM is composed of a drift term determined by $\hat{\mathbf{x}}_\theta$ and a fixed diffusion term (noise). The diffusion term employs a diagonal covariance structure in the DM and hence can be varied in dimension and does not induce dependencies between dimensions. This leaves the neural network of $\hat{\mathbf{x}}_\theta$ to reflect the permutation equivariances. First we observe in Figure 2 that all neural networks m are applied independently to each dimension (they are contained in the light grey boxes), leaving only the attention mechanism itself to relate different dimensions. Attention can only distinguish dimensions through added positional embeddings [Vaswani et al., 2017]. Consequently our exchangeable embeddings setting shares the same embedding across all realizations of an index i if the model is expected to be invariant to permutations of i. The attention mechanism can therefore not distinguish between variables with different values of i and so the messages passed are not affected by permutations of i. This means that the network as a whole is equivariant to permutations of i and so the modeled distribution is invariant to permutations of i [Hoogeboom et al., 2022].

J Matrix inversion

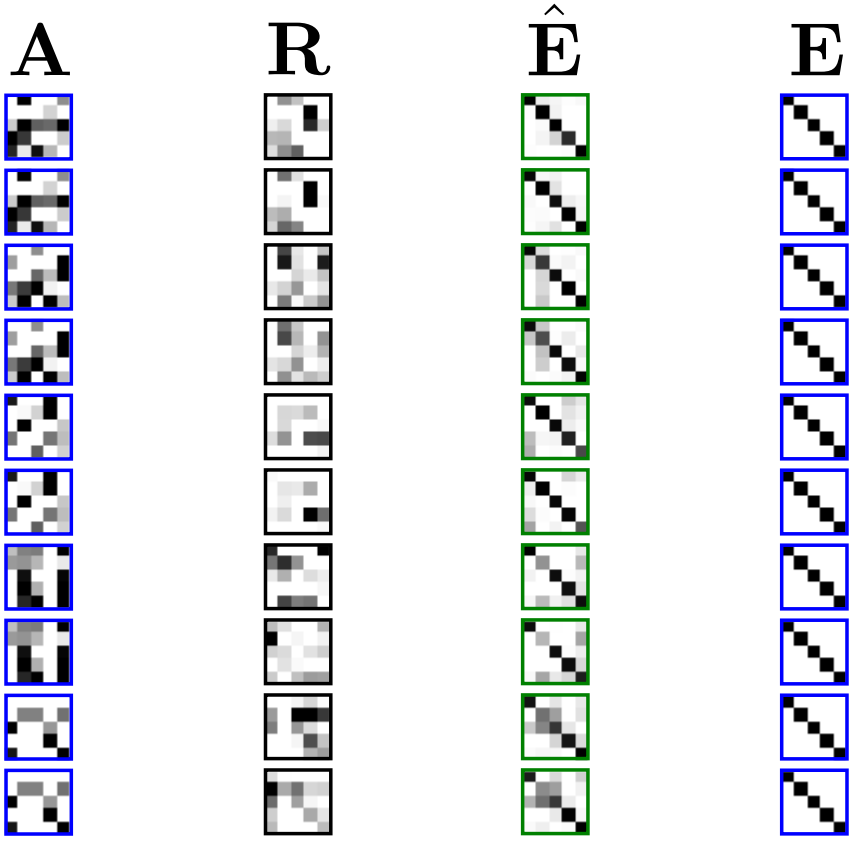


Figure 16: Rows of matrix inversion examples. Similar to Fig. 6, but here we condition both on $\boldsymbol{A}$ and $\boldsymbol{E} = \mathbf{1}$ (blue). Each of the 5 pairs of rows show a solution for the same $\boldsymbol{A}$. Reconstructions are shown as $\hat{\boldsymbol{E}} = \boldsymbol{A}\boldsymbol{R}$ (green).

In this experiment we explore a purely continuous variation of our BCMF example from Section 4. We train the model on fixed size full rank matrices of dimension 5 and condition on both $\boldsymbol{E}$ and $\boldsymbol{A}$ during training and testing. All entries of both $\boldsymbol{A} \in \mathbb{R}^{5\times5}$ and $\boldsymbol{R} \in \mathbb{R}^{5\times5}$ are now sampled from a Normal$(0, 1)$ prior and we set $\boldsymbol{E} = \boldsymbol{A}\boldsymbol{R}$ (as in BCMF). Despite training on randomly sampled $\boldsymbol{A}$ and $\boldsymbol{R}$, we demonstrate that GSDM implicitly learns matrix inversion. At test time we set $\boldsymbol{E}$ to the identity matrix and solve for $\boldsymbol{R} \approx \boldsymbol{A}^{-1}$. Example solutions can be seen in Fig. 16. Each pair of rows contains two approximate solutions for the same $\boldsymbol{A}$ to illustrate sample diversity. Most reconstructions for $\hat{\boldsymbol{E}}$ are close to the identity matrix, but GSDM is not perfect. We did not specialize our prior from BCMF; a more targeted prior could be constructed by directly providing pairs of matrices and their inverse. This experiment shows that we are able to calculate approximate inverses, even though we have not specialized our graphical model or training distribution to do so.

```
(defn rand-matrix [size name]
  (foreach (first size) [i (range (first size))]
           (foreach (second size) [j (range (second size))]
                    (sample name (normal 0 1)))))

(defn dot-helper [t state a b]
  (+ state
     (sample "C" (dirac (* (get a t)
                           (get b t))))))

(defn dot [a b]
  (loop (count a) 0 dot-helper a b))

(defn row-mul [t state m v]
  (conj state (sample "E" (dirac (dot (get m t) v)))))

(defn transpose [m]
  (foreach (count (first m)) [j (range (count (first m)))]
           (foreach (count m) [i (range (count m))]
                    (get (get m i) j))))

(defn matmatmul [m1 m2]
  (let [m2_ (transpose m2)]
    (foreach (count m1) [i (range (count m1))]
             (foreach (count m2_) [j (range (count m2_))]
                      (sample
                       "E"
                       (dirac (dot (get m1 i) (get m2_ j))))))))

(let [A (rand-matrix [3 2] "A")
      R (rand-matrix [2 3] "R")
      E (matmatmul A R)]
  E)
```

Figure 17: Source code of a full generative model for the BCMF experiment. Passing this into our compiler yields the attention mask in Fig. 10. Note that intermediate variables for C are explicitly created by sampling from a dirac distribution.