# Graphically Structured Diffusion Models

**Christian Weilbach** [1] **William Harvey** [1] **Frank Wood** [1]

## 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. Our code can be found at `https://github.com/plai-group/gsdm`.

## 1. Introduction

A future in which algorithm development is fully transformed from a challenging and labour intensive task (Cormen, 2009; Marsland, 2009; Russell & Norvig, 2010; Williamson & Shmoys, 2011) into a fully automatable process is seemingly close at hand. With prompt engineering large language models like GPT (Brown et al., 2020) and now ChatGPT have been shown to be capable of code completion and even full algorithm development from natural language task descriptions (Chen et al., 2021; Ouyang et al., 2022).

At the same time, significant advances in generative deep learning (Ho et al., 2020; Song et al., 2021c; Ho et al., 2022), AutoML (Hutter et al., 2018), and few-shot learning (Brown et al., 2020) have made it possible to learn, from data, flexible input-output mappings that generalize from ever smaller amounts of data. This approach has spawned modern aphorisms from Karpathy (2017) like " Gradient descent can write better software than you. Sorry!", appropriate attempts, in our opinion, to re-brand deep learning as differentiable programming (Baydin et al., 2017), and arguably even a new industry called "Software 2.0" in which one "programs by example" (Karpathy, 2017).

There, however, remains a chasm between these two approaches, roughly delineated along the symbolic vs. connectionist divide. Symbolically expressed algorithms can and often do generalize perfectly across all inputs and exhibit runtimes that are typically input "size" dependent. Software 2.0 algorithms struggle to generalize outside of their training data thus are most often deployed in settings where copious training data is available, the so-called "big-data" regime. Most such "neural-network algorithms" have runtimes that are not size dependent and resultingly cannot generalize in the same fashion as symbolically expressed algorithms.

Efforts to bring these two approaches closer together (Chaudhuri et al., 2021) often get lumped together under the moniker "neuro-symbolic" methods. The general shape of these methods, so to speak, is to impose some aspect of symbolic reasoning on either the structure or computation performed by a connectionist architecture. Our work can be seen as a significantly novel methodological contribution to this body of work.

We contribute a generic specification of methodology for advantageously imposing task specific symbolic structure into diffusion models and use it to demonstrate algorithm learning from data in several small-scale but foundational tasks across the algorithmic complexity spectrum. Specifically, our approach consumes a graphical model "sketch" that putatively could describe the joint data generative process. This sketch consists only of nodes for variables, edges between them, and optionally permutation invariances. We combine this information with an otherwise generic diffusion process (Ho et al., 2020), using the edges to advantageously constrain the transformer attention mechanisms (Vaswani et al., 2017) and permutation invariances to determine when parameters within our architecture can be shared. Compared to our neural baselines we improve the scaling of computational cost with problem dimension in most cases, and the scaling of problem performance with dimension in all cases.

---

[1] Department of Computer Science, University of British Columbia, Vancouver, Canada. Correspondence to: Christian Weilbach <weilbach@cs.ubc.ca>.

*Proceedings of the $40^{th}$ International Conference on Machine Learning*, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

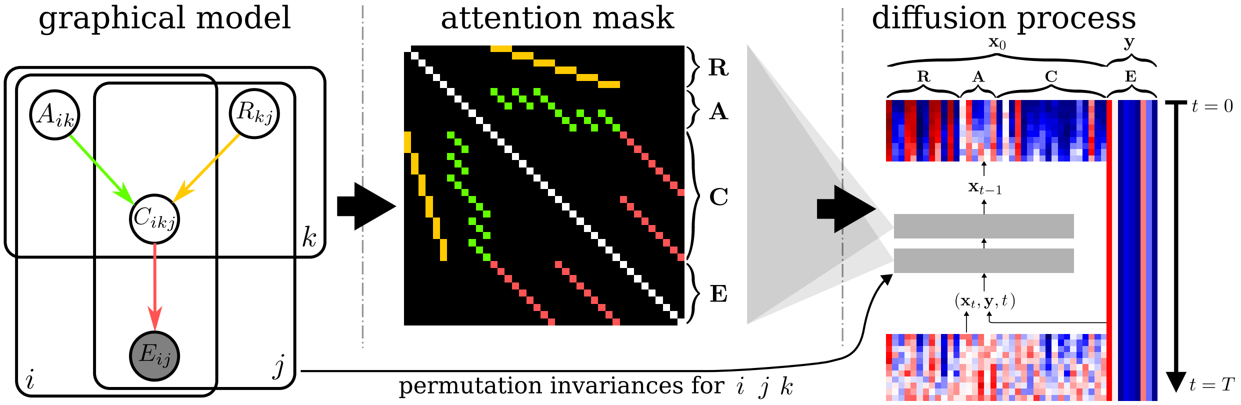


Figure 1: An application of our framework to binary-continuous matrix factorization. In the first panel the computational graph of the multiplication of the continuous matrix $\boldsymbol{A} \in \mathbb{R}^{3\times2}$ and the binary matrix $\boldsymbol{R} \in \mathbb{R}^{2\times3}$ is expanded as a probabilistic 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 (or white for self-edges). In the third panel the projection into the sparsely-structured neural network guiding the diffusion process is illustrated. The bottom shows the translation of permutation invariances of the probability distribution into shared embeddings, as detailed in Section 2.4.

As a running example to keep in mind throughout the paper, consider being given the task of developing a novel matrix factorization algorithm, one which takes a real non-negative valued matrix as input and outputs a distribution over two matrix factors, one constrained to be binary valued, the other constrained to be non-negative. The most traditional approach is to painstakingly hand-develop through intellectual willpower some new algorithm like Gram-Schmidt which may not exist and might take an entire career to develop. A more modern approach, to which we compare, is to symbolically specify a model describing a joint data generating process and employ a generic inference algorithm like MCMC (Wingate et al., 2011). Such a model is usually much easier to specify but the resulting "inversion algorithm," running a generic inference algorithm at test time, trades sure generalization with worst-case infinite runtime. Alternatively one could generate a large training dataset from such a generative description, then hand-architect and train a deep neural network to learn the desired inversion algorithm, software 2.0 style (Le et al., 2017b). This is slow to develop and train, usually requiring architectural innovation, but constant-time fast at test time, albeit with likely poor algorithm-style generalization. Our approach, most like that of (Weilbach et al., 2020), strikes a middle ground. We adopt the software 2.0 approach but provide a generic recipe for specializing a generic and powerful diffusion-based network architecture that trains quickly, generalizes reliably, and whose runtime scales with problem size.

## 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 Equation (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., 2021c), 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. (2021b) 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

Figure 2 outlines our neural architecture for $\hat{\mathbf{x}}_\theta$, which is run once for every diffusion time step $t$. Its inputs are $\mathbf{x}_t$, $\mathbf{y}$, and the diffusion timestep $t$. The (noisy) value of each latent graphical model node is represented in $\mathbf{x}_t$ and, similarly, $\mathbf{y}$ contains the value of each observed graphical model node. We use linear projections to embed all of these values, concatenating the embeddings of all latent and observed nodes to obtain, for an $n$-node graphical model and $d$-dimensional embedding space, an $n \times d$ array of embedding "tokens". We add learned "node embeddings" to these tokens to identify which graphical model node each corresponds to, and also add learned observation embedding vectors for tokens corresponding to observed nodes. The resulting $n \times d$ array is passed through a stack of self-attention (Vaswani et al., 2017) and ResNet (He et al., 2016) blocks, as summarized in Figure 2, with the ResNet blocks taking an embedding of the diffusion timestep $t$ as an additional input. All of the timestep embedder, ResNet blocks, and self-attention modules are identical to those of Song et al. (2021a), except that we replace convolutions with per-token linear projections. The tokens corresponding to non-observed graphical model nodes are then fed through a learned linear projection to produce an output for each.

The self-attention layers are solely responsible for controlling interactions between embeddings, and therefore correlations between variables in the modelled distribution $p_\theta(\mathbf{x}_0|\mathbf{y})$. Inside the self-attention, each embedding is projected into a query vector, a key vector, and a value vector, all in $\mathbb{R}^d$. Stacking these values for all embeddings yields the matrices $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{n \times d}$ (given a $d$-node graphical model). The output of the self-attention is calculated as

$$\boldsymbol{e}^{\text{out}} = \boldsymbol{e}^{\text{in}} + \boldsymbol{W}\boldsymbol{V} = \boldsymbol{e}^{\text{in}} + \text{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V} \tag{5}$$

where the addition of the self-attention layer's input $\boldsymbol{e}^{\text{in}} \in \mathbb{R}^{n \times d}$ corresponds to a residual connection. We note that $\boldsymbol{Q}\boldsymbol{K}^T$ yields a pairwise interaction matrix which lets us impose an additional attention mask $\boldsymbol{M}$ before calculating the output $\boldsymbol{e}^{\text{out}} = \boldsymbol{e}^{\text{in}} + \text{softmax}\left(\boldsymbol{M} \odot \boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}$. This masking interface is central to structure the flow of information between graphical model nodes in Section 3.1 .

### 2.3. Graphical Models

GSDM leverages problem structure described in the form of a graphical model. There is considerable flexibility in the specification of this graphical structure and we allow for both directed and undirected graphical models. A directed graphical model describes a joint distribution over $\mathbf{x} = [x_1, \ldots, x_n]$ with the density $p(\mathbf{x}) = \prod_{i=1}^{n} p_i(x_i|parents(x_i))$. This may be natural to use if the problem of interest can be described by a causal model. This is the case in the BCMF example in Figure 1, where the forward model is a matrix multiplication and we can use the matrix multiplication's compute graph as a graphical model. 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 L). Alternatively, an undirected graphical model uses the density $p(\mathbf{x}) \propto \prod_{j=1}^{m} f_j(vertices(j))$ where $vertices(j)$ are the vertices connected to factor $j$ and $f_j$ maps their values to a scalar. This is a natural formulation if the problem is defined by constraints on groups of nodes, e.g. for Sudoku with row, column and block constraints (Appendix C). Finally, the graphical model can combine directed and undirected components, using a density $p(\mathbf{x}) \propto \prod_{i=1}^{n} p(x_i|parents(x_i)) \prod_{j=1}^{m} f_j(vertices(j))$. We use this in our graphical model for sorting (Appendix C), which combines a causal forward model with constraints.

We emphasise that GSDM does not need the link functions (i.e. the form of each $p_i$ and $f_j$) to be specified as long as data is available, which is desirable as they are often intractable or Dirac in practice. Also, while the selection of a graphical model for data can be subjective, 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 it is directed, undirected, or a combination.

### 2.4. 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 & Teh, 2020). For example the matrix multiplication in Figure 1 is invariant with respect to permutations of any of the plate indices.[1] If the joint probability density is invariant to a particular permutation, this can be enforced in a distribution

---

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

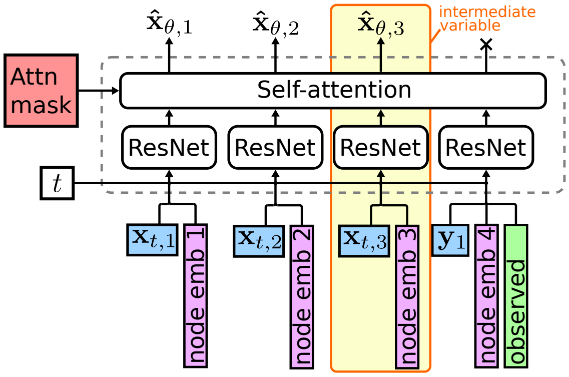


Figure 2: An example GSDM architecture for a graphical model with one observed and three latent variables. The components within the dashed lines are repeated multiple times. Arrows represent information flow. For clarity, we leave out simple operations including linear transformations; see the appendix for full detail.

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 3.2.

## 3. Method

The first stage in using GSDM is to define a graphical model as discussed previously. This section focuses on how to map from a graphical model to the corresponding GSDM architecture, an example of which is shown in Figure 2. The backbone of the architecture is a stack of transformer modules operating on a set of embeddings, with each embedding corresponding to one graphical model node. The colors in Figure 2 outline this section. Section 3.1 describes how we derive a sparse attention mechanism from a graphical model. Section 3.2 explains the node embeddings. Section 3.3 motivates our decision to model "intermediate" variables jointly with the variables of interest. Section 3.4 describes how we handle observed variables, and Section 3.5 describes how we handle discrete variables.

### 3.1. Faithful Structured Attention

Our architecture in Figure 2 runs a self-attention mechanism over a set of embeddings, each of which corresponds to a graphical model node. To add structural information, we use the graphical model's edges to construct attention masks $\boldsymbol{M}$ for the self-attention layers. Precisely, we allow variable $i$ to attend to variable $j$ if there is an edge between node $i$ and node $j$, and irrespective of the direction of the edge. If the graphical model contains factors, we additionally allow attention between any node pairs $(i, j)$ which connect to the same factor. These heuristics typically keep the graph sparse while, importantly, ensuring that given enough attention layers all output nodes can depend on all input nodes. We show in Appendix E that this is necessary to faithfully capture the dependencies in $q(\mathbf{x}|\mathbf{y})$.

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 and the released code. Its computational cost is $\mathcal{O}(nm)$, where $n$ is the number of dimensions and $m$ is the number of entries in the densest row of $\boldsymbol{M}$. We show in Appendix H that, even after accounting for cost of the modeling of additional intermediate variables as described later, the sparse attention mechanism gives GSDM a reduction in computational complexity of $\mathcal{O}(n)$ relative to a naive baseline in three of our four experiments.

### 3.2. Node Embeddings

GSDM's architecture contains positional embeddings to let the neural network distinguish 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 problem dimension. A generic solution to this involves noting that graphical model nodes can often be grouped together into "arrays". For instance, the BCMF example in Figure 1 contains 39 nodes but these belong to just 4 multi-dimensional arrays: $\boldsymbol{A}$, $\boldsymbol{R}$, $\boldsymbol{C}$, and $\boldsymbol{E}$. We suggest **array embeddings**, or **AE**, which can be automatically constructed for such problems with (potentially variable-size) ordered datatypes. With AE, we compute the embedding for each node 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. Scalars can be treated as arrays of size 1. AEs work well in our experiments and are a sensible default.

For graphical models exhibiting permutation invariances we can optionally enforce these invariances exactly using **exchangeable embeddings**, or **EE**. We do so according to the following result.

**Theorem 3.1** (Permutation invariance in GSDM)**.** *Let $\mathcal{A}$ represent the indices of a subset of the dimensions of data $\mathbf{x}$ and $\Pi_{\mathcal{A}}$ be the class of permutations that permute only dimensions indexed by $\mathcal{A}$. Assume we have a GSDM parameterised with neural network $\hat{\mathbf{x}}_\theta(\cdot; \boldsymbol{M})$, where $\boldsymbol{M}$ is the structured attention mask. If the node embeddings used by $\hat{\mathbf{x}}_\theta$ are shared across all nodes indexed by $\mathcal{A}$, then the distribution modelled by GSDM will be invariant to all permutations $\pi$ satisfying*

$$\boldsymbol{M} = \pi \boldsymbol{M} \quad \textit{and} \quad \pi \in \Pi_{\mathcal{A}} \tag{6}$$

*where $\pi\boldsymbol{M}$ is a permutation of both the rows and columns of $\boldsymbol{M}$ by $\pi$.*

*Proof.* See Appendix F. ☐

One implication of the $\boldsymbol{M} = \pi\boldsymbol{M}$ condition is that it holds trivially for a DM without sparse attention, in which $\boldsymbol{M}$ is a matrix of all ones. The modeled distribution would therefore be invariant to any permutation of $\mathcal{A}$ (Hoogeboom et al., 2022). This may be a useful permutation invariance to encode for some problems but, for the structured problems considered in this paper, it is too simple and not valid. In none of our experiments are there two variables whose values can be swapped without changing the density under the data distribution. For example in BCMF, the data density is invariant to reordering any of the plate indices, but not to swapping any single pair of nodes in them.

When $\boldsymbol{M}$ is a structured matrix as we propose for GSDM, Theorem 3.1 suggests a way to incorporate invariances that are closely tied to the problem structure. On the BCMF problem shown in Figure 1 we use only four embeddings, sharing a single embedding between all nodes in $\boldsymbol{A}$; another between all nodes in $\boldsymbol{R}$; and so on for $\boldsymbol{C}$ and $\boldsymbol{E}$. Without imposing an attention mask, this would make the network invariant to any permutation of the variables within each of $\boldsymbol{A}$, $\boldsymbol{R}$, $\boldsymbol{C}$ and $\boldsymbol{E}$. With GSDM's attention mask, it is only invariant to permutations which lead to the same mask. This means that the learned distribution is invariant only to the ordering of the indices $i$, $j$, and $k$. As represented by the plate notation in Figure 1, this is a desired invariance that matches the data distribution.

In general, Theorem 3.1 suggests a simple heuristic for checking when problem invariances can be enforced. If permuting the ordering of an index does not affect the sparsity mask for a given problem then sharing embeddings across instances of this index will enforce a permutation invariance with respect to this index in GSDM. We utilise this property in three of our four experiments. Along with our compiler which generates an attention mask for a programatically-defined graphical model, checking when Theorem 3.1 holds for a given class of permutations can reasonably be automated. We do, however, still require human knowledge to propose invariances suitable for the graphical model.

### 3.3. Intermediate Variables

When translating a generative model into a graphical model, any observed variables or latent variables of particular interest to the modeler should clearly be included as nodes. There will be other latent variables, which we call "intermediate variables", which are not directly of interest but may be included to make the graphical model more interpretable, more sparse, or otherwise preferable. Whether or not these are included has implications for GSDM as it will either be trained to model them jointly with the variables of interest if they are included, or trained without this signal if they are not. There is no model-agnostic "right" answer to whether or not intermediate variables will be helpful but we point out that including them is often beneficial for GSDMs because of **(1)** GSDM's empirical success at utilising the learning signal from these intermediate variables as described below and **(2)** the reduced computational cost of GSDM's 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.

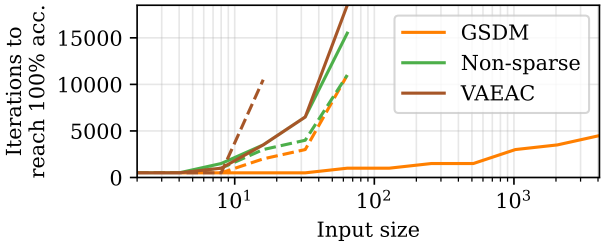


Figure 3: Time to fit the Boolean circuit with (solid lines) and without (dashed) intermediate variables. Accuracy is computed on 16 validation examples every 500 iterations, to a maximum of 20 000.

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. Identifying the function computed by each of the $\mathcal{O}(2^n)$ gates will therefore require at least $\mathcal{O}(2^n)$ 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 only a few data points. Figure 3 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. Flexible Conditioning

Optimizing the DM loss in Equation (4) requires a partitioning of data into latent variables (outputs) $\mathbf{x}_0$, and observed variables (inputs) $\mathbf{y}$. Our neural network distinguishes between variables in $\mathbf{x}_t$ and $\mathbf{y}$ via a learned observation

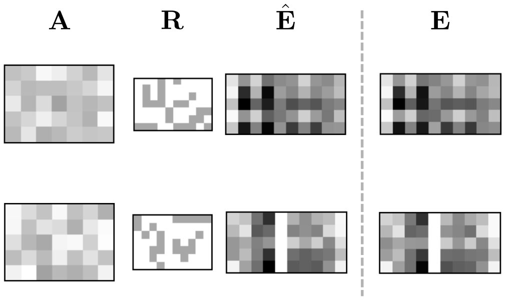


Figure 4: Two BCMF samples by GSDM with $m = 5$, $n = 10$ and $k = 7$. Each row in the plot shows the inferred $\boldsymbol{A}$, $\boldsymbol{R}$ and the respective reconstruction $\hat{\boldsymbol{E}} = \boldsymbol{A}\boldsymbol{R}$ on the left and the $\boldsymbol{E}$ provided as input on the right. Intermediate variables $\boldsymbol{C}$ are omitted here.

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, or additional observed values, at inference time. This is unlike standard algorithms or more traditional autoregressive amortized inference artifacts (Le et al., 2017a).

### 3.5. 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 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.

## 4. Experiments

Our experiments compare GSDM against ablations including a non-sparse version (i.e. a vanilla DM), as well as the variational auto-encoder for arbitrary conditioning (VAEAC) (Ivanov et al., 2019) and, where appropriate, the best performing MCMC method we tried: Lightweight Metropolis Hastings (LMH) (Wingate et al., 2011). Unlike GSDM, MCMC requires a fully specified probabilistic model and cannot operate with data and independencies only. We additionally compare against two deterministic baselines trained to predict $\mathbf{x}$ given $\mathbf{y}$ with a mean-squared error loss. One of these, "Regressor + GS" has a similar architecture to GSDM and the other, "Regressor" has a similar architecture to our vanilla DM baseline. Both perform poorly because most of our problems have multiple solutions. We test the methods on the following four problems, which were chosen to cover a wide range of problems in the space of algorithm design and demonstrate that GSDM is suitable to learn such approximate algorithms.

**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 $\boldsymbol{E} := \boldsymbol{A}\boldsymbol{R}$. 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 $\boldsymbol{E}$. Figure 4 shows that our learned GSDM can find factorisations that are consistent with the observations. We plot performance for GSDM trained on different problem sizes in Figure 5. We highlight another property here, namely that GSDM trained on BCMF can generalise well to larger $m$ and $n$ than it is trained on; a GSDM trained on $m$, $n$ and $k$ uniformly sampled in the range 1 to 10 can generalize to problems with $m$ and $n$ as large as 200 as we show in Appendix G.

In keeping with our motivation of GSDM as a tool for quickly learning approximate algorithms, we compare against several quickly-implemented heuristic algorithms for BCMF. First, K-means uses the K-means algorithm to assign each row of $\boldsymbol{E}$ to one of $k$ cluster. The factor $\boldsymbol{A}$ is then the cluster centers and $\boldsymbol{R}$ is a binary tensor describing the assigned cluster indices. Next, NMF uses an off-the-shelf continuous non-negative matrix factorization algorithm and discretizes one of the returned factors. Finally, we found that simply prompting ChatGPT to write a BCMF algorithm in Python yielded a non-trivial algorithm which we denote "ChatGPT" (see Appendix M). GSDM outperforms all of these baselines whenever $k$ is less than $n$ and $m$ (so that there are no trivial solutions). Additionally, we emphasize that GSDM targets the posterior over all solutions, while these hand-coded algorithms yield a single point estimate.

**Sudoku** A Sudoku grid is a $9 \times 9$ array of numbers such that each number is in $\{1, \ldots, 9\}$ and no two numbers in the same row, column, or $3 \times 3$ 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 $\{1, \ldots, 9\}$. The resulting GSDM attention mask lets each variable attend to all other variables

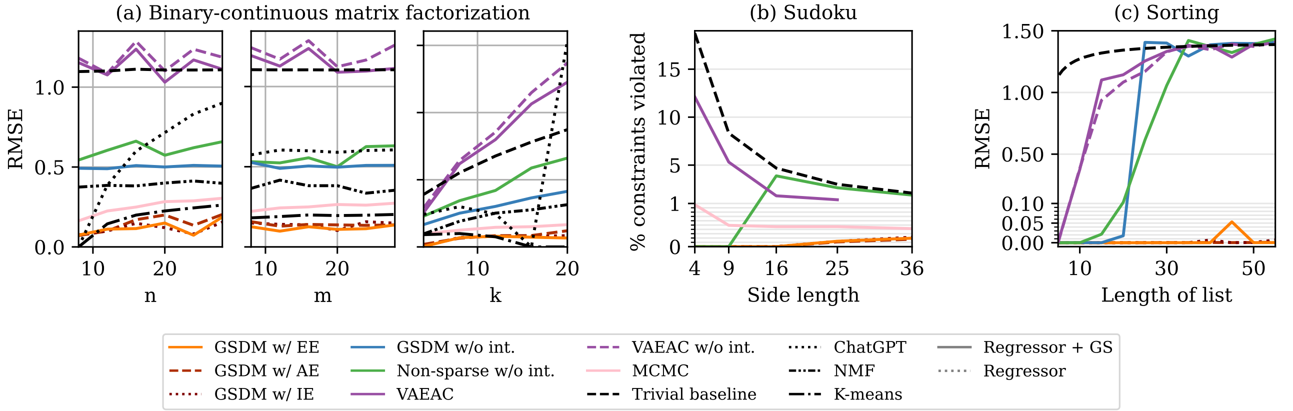


Figure 5: Performance versus problem dimension with all runs matched for wall-clock time. **(a)** BCMF reconstruction error. We start from $n = m = 16$ and $k = 8$ and vary each dimension independently. GSDM is the best-performing method for all settings, and is also the only neural method to outperform our MCMC baseline. **(b)** Number of pairwise constraint violations from samples for the Sudoku task with 20% of cells observed, normalised by the total number of constraints. Only GSDM and the MCMC baseline significantly outperform it across all tested problem dimensions. **(3)** RMSE between ground-truth sorted lists and the observed lists sorted with sampled permutation matrices. All methods work on lists of length 5 but our baselines quickly degrade in performance as the dimensionality is increased, while GSDM continues to work well. The dashed black line represents a trivial baseline given by, in (a), predicting the mean under the prior, in (b), sampling all cells from independent uniform distributions and, in (c), sampling a random permutation matrix. We share the legend with Figure 6, which contains results for "Regressor" and "Regressor + GS".

in the same row, column, and block. Our data generator[2] creates complete $9 \times 9$ 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 find that GSDM can indeed solve Sudoku puzzles: it generates valid Sudokus unconditionally with 98% accuracy; and in the case where 16 of 81 cells are observed with 96% accuracy while maintaining sample diversity (Appendix J). We also consider a generalization of Sudokus to any $n^2 \times n^2$ grid. Our results in Figure 5 show that GSDM can scale gracefully to these larger problems. This is not true of our other deep learning baselines, which have performance no better than random guessing on large problem sizes. The MCMC method consistently violates less constraints than random guessing but appears to be prone to getting stuck in local minima and so seldom finds a correct solution even with small problem sizes.

**Sorting** Our graphical model for sorting 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 that there should be 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 $\mathbf{s} = \boldsymbol{P}\mathbf{u}$. (4) We use factors between each pair of elements in $\mathbf{s}$ to enforce that it is sorted. We show a diagram of this graphical model, as well as all others used in our experiments, in Appendix B. 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. It fits into the GSDM framework nonetheless since the graphical model is a valid specification of the independences in the data distribution. We measure it's performance in Figure 5 as the RMSE between the ground-truth $\mathbf{s}$ and the observed $\mathbf{u}$ transformed by the sampled $\boldsymbol{P}$, and plot progress during training in Figure 7. Our MCMC baseline was not able to sort even 3 element lists.

**Boolean** We additionally use the Boolean circuit described in Section 3.3 (Figure 3) to demonstrate GSDM's ability to learn structured functions over many variables. Our graphical model is simply the tree-shaped compute graph of the Boolean circuit (diagram in Appendix B).

### 4.1. Structured Attention and Intermediate Variables

All of our experiments rely on structured attention for their good performance, and the positive effect remains to a lesser extent after removing intermediate variables. We saw this for the Boolean circuit in Section 3.3 and here show experi-

---

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

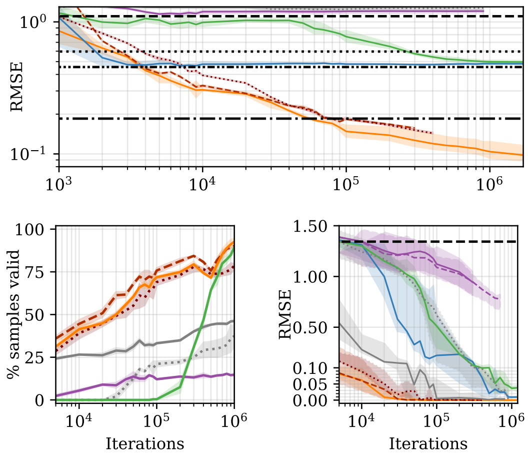


Figure 6: Training curves for BCMF (top) with $n, m, k = 16, 10, 8$; $9 \times 9$ Sudoku (left); and sorting (right) with $n = 20$. Error bars show min/max of 3 seeds. Legend in Figure 5.

ments in Figure 5 for Sudoku, sorting, and BCMF of various dimensionalities. Imposing structure leads to significant improvements in each case, especially in combination with intermediate variables. We ablate intermediate variables in the same figure, as well as for the Boolean circuit in Figure 3. The combination of structured attention and intermediate variables is extremely helpful in all cases, ensuring that the error remains low for all tested problem dimensions, even when the baselines perform no better than random chance.

### 4.2. Effect of Node Embeddings

Three of our problems contain permutation invariances exploitable by embedding-sharing. Our sorting model is invariant to permutations of $j$ (the ordering of $\mathbf{u}$). Sudoku is invariant to various permutations although EE was only able to capture a simple invariance by sharing embeddings between nodes that are within both the same row and the same block. BCMF is invariant to permutations of all plate indices in Figure 1. We see in Figure 6 that incorporating these invariances with EE always gives faster training than using IE. Even without specifying the invariances, AE can be used to obtain most of the benefit. For Sudoku AE outperforms IE, which may be due to AE's natural encoding of spatial information. Additionally, our results on generalization past the training dimensions in Appendix G are only possible with the embedding-sharing enabled by EE.

Also note that all results shown so far are for the infinite data regime, where we create data cheaply and on-the-fly during training. We show in Appendix I that our innovations are even more helpful when the number of training examples is restricted.

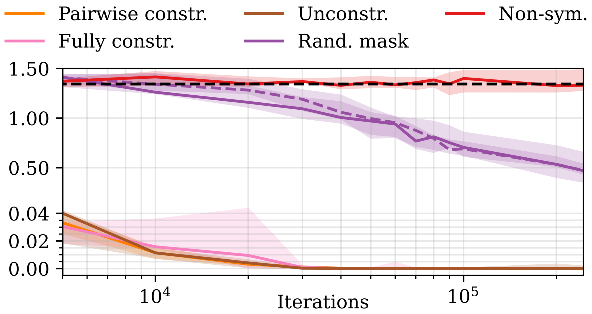


Figure 7: Ablations of GSDM with different graphical model structures for sorting with $n = 20$. The first three lines, which represent reasonable or only slightly misspecified constraints on $\mathbf{s}$ in different ways described in Section 4.3, quickly reach near-zero error. This speaks to GSDM's robustness to graphical model specification.

### 4.3. Robustness to Choice of Graphical Model

There can be a degree of freedom 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 Figure 7 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 7 also shows that GSDM can be robust to a misspecified model, "Unconstrained $\mathbf{s}$", where constraints are not imposed on $\mathbf{s}$. Two baselines perform significantly worse: a "Random" baseline, in which each node is allowed to attend to 20 other nodes sampled at random; and a baseline with a non-symmetrized version of our "standard" graphical model's attention mask. 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 demonstrates the necessity of our symmetrization, backing up the reasoning in Section 3.1.

## 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 & Goodman, 2014; Ritchie et al., 2016) and probabilistic programming (Le et al., 2017a; van de Meent et al., 2018) provides the foundation 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 dimension. 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. Freivalds & Kozlovics (2022) use a DM with a graph neual network architecture (Zhou et al., 2020) to tackle the SAT problem class. We aim to tackle a much more general class of problems.

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., 2021c; 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

We have introduced GSDM which benefits from 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. We have demonstrated that GSDMs can automate the reasoning required to create approximate solutions to tasks as diverse as sorting, Sudoku solving and binary-continuous matrix factorization. Our current implementation can be applied to graphical models with size up to roughly $50\,000$ nodes on a standard GPU and we see considerable scope for scaling further in future work. Another immediate avenue for future work to address is GSDM's relatively slow sampling speed. All samples drawn throughout this paper used 1000 network function evaluations, but recent work suggests that it may be possible to reduce this number considerably (Salimans & Ho, 2022; Karras et al., 2022). We believe that GSDM holds promise for models as complex as large scientific simulators and encourage others in the community to join us in making this a reality. In addition we are interested in integrating task-specific symbolic knowledge beyond graphical model structure into the dynamics of diffusion processes. Finally we believe that, if future work could turn our largely manual process for deriving a specific GSDM into a fully specified formal procedure, this could lead to powerful new next-generation probabilistic programming systems.

### ACKNOWLEDGMENTS

Our thanks go to Benjamin Bloem-Reddy for a helpful discussion on equivariance in probabilistic models. 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

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

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, 18: 153:1–153:43, 2017.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The Long-Document Transformer, December 2020.

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

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]*, July 2020.

Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., and Yue, Y. Neurosymbolic Programming. *Foundations and Trends® in Programming Languages*, 7 (3):158–243, December 2021. ISSN 2325-1107, 2325-1131. doi: 10.1561/2500000049.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating Large Language Models Trained on Code, July 2021.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating Long Sequences with Sparse Transformers, April 2019.

Cormen, T. H. (ed.). *Introduction to Algorithms*. MIT Press, Cambridge, Mass, 3rd ed edition, 2009. ISBN 978-0-262-03384-8 978-0-262-53305-8.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q. V., and Salakhutdinov, R. Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context, June 2019.

Freivalds, K. and Kozlovics, S. Denoising diffusion for sampling sat solutions. *arXiv preprint arXiv:2212.00121*, 2022.

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

Harvey, W., Naderiparizi, S., Masrani, V., Weilbach, C., and Wood, F. Flexible Diffusion Modeling of Long Videos, May 2022.

He, K., Zhang, X., Ren, S., and Sun, J. 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*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Hendrycks, D. and Gimpel, K. Gaussian Error Linear Units (GELUs), July 2020.

Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. Flow++: Improving flow-based generative models with variational dequantization and architecture design. In *International Conference on Machine Learning*, pp. 2722–2730. PMLR, 2019.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 6840–6851. Curran Associates, Inc., 2020.

Ho, J., Salimans, T., Gritsenko, A., Chan, W., Norouzi, M., and Fleet, D. J. Video Diffusion Models. *arXiv:2204.03458 [cs]*, April 2022.

Hoogeboom, E., Gritsenko, A. A., Bastings, J., Poole, B., Berg, R. v. d., and Salimans, T. Autoregressive diffusion models. *arXiv preprint arXiv:2110.02037*, 2021.

Hoogeboom, E., Satorras, V. G., Vignac, C., and Welling, M. Equivariant diffusion for molecule generation in 3d. In *International Conference on Machine Learning*, pp. 8867–8887. PMLR, 2022.

Hutter, F., Kotthoff, L., and Vanschoren, J. (eds.). *Automatic Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

Ivanov, O., Figurnov, M., and Vetrov, D. Variational Autoencoder with Arbitrary Conditioning. *arXiv:1806.02382 [cs, stat]*, June 2019.

Karpathy, A., Nov 2017. URL `https://karpathy.medium.com/software-2-0-a64152b37c35`.

Karras, T., Aittala, M., Aila, T., and Laine, S. Elucidating the design space of diffusion-based generative models. *arXiv preprint arXiv:2206.00364*, 2022.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Kitaev, N., Kaiser, Ł., and Levskaya, A. Reformer: The Efficient Transformer. *arXiv:2001.04451 [cs, stat]*, February 2020.

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

Kreuzer, D., Beaini, D., Hamilton, W., Létourneau, V., and Tossou, P. Rethinking graph transformers with spectral attention. *Advances in Neural Information Processing Systems*, 34:21618–21629, 2021.

Le, T. A., Baydin, A. G., and Wood, F. 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*, pp. 1338–1348, Fort Lauderdale, FL, USA, 2017a. PMLR.

Le, T. A., Baydin, A. G., Zinkov, R., and Wood, F. Using synthetic data to train neural networks is model-based reasoning. In *2017 international joint conference on neural networks (IJCNN)*, pp. 3514–3521. IEEE, 2017b.

Marsland, S. *Machine Learning - an Algorithmic Perspective*. Chapman and Hall / CRC Machine Learning and Pattern Recognition Series. CRC Press, 2009. ISBN 978-1-4200-6718-7.

OpenAI. Chatgpt. `https://chat.openai.com`, 2021. Accessed on March 17, 2023.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

Palm, R., Paquet, U., and Winther, O. Recurrent Relational Networks. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

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

Roy, A., Saffar, M., Vaswani, A., and Grangier, D. Efficient Content-Based Sparse Attention with Routing Transformers, October 2020.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 3rd edition, 2010. ISBN 0-13-604259-7 978-0-13-604259-4.

Salimans, T. and Ho, J. Progressive distillation for fast sampling of diffusion models. *arXiv preprint arXiv:2202.00512*, 2022.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT Solver from Single-Bit Supervision. *arXiv:1802.03685 [cs]*, March 2019.

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., and Ganguli, S. Deep Unsupervised Learning using Nonequilibrium Thermodynamics. In *Proceedings of the 32nd International Conference on Machine Learning*, pp. 2256–2265. PMLR, June 2015.

Song, J., Meng, C., and Ermon, S. Denoising Diffusion Implicit Models. *arXiv:2010.02502 [cs]*, November 2021a.

Song, Y., Durkan, C., Murray, I., and Ermon, S. Maximum Likelihood Training of Score-Based Diffusion Models. In *Advances in Neural Information Processing Systems*, volume 34, pp. 1415–1428. Curran Associates, Inc., 2021b.

Song, Y., Sohl-Dickstein, J., Kingma, D. P., Kumar, A., Ermon, S., and Poole, B. Score-Based Generative Modeling through Stochastic Differential Equations. *arXiv:2011.13456 [cs, stat]*, February 2021c.

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

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient Transformers: A Survey, March 2022.

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

van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. An Introduction to Probabilistic Programming. *arXiv:1809.10756 [cs, stat]*, September 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Wang, P.-W., Donti, P. L., Wilder, B., and Kolter, Z. SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. *arXiv:1905.12149 [cs, stat]*, May 2019.

Webb, S., Golinski, A., Zinkov, R., Siddharth, N., Rainforth, T., Teh, Y. W., and Wood, F. Faithful Inversion of Generative Models for Effective Amortized Inference. December 2017.

Weilbach, C., Beronov, B., Wood, F., and Harvey, W. Structured conditional continuous normalizing flows for efficient amortized inference in graphical models. In Chiappa, S. and Calandra, R. (eds.), *Proceedings of the*

*Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pp. 4441–4451. PMLR, 26–28 Aug 2020. URL `https://proceedings.mlr.press/v108/weilbach20a.html`.

Williamson, D. P. and Shmoys, D. B. *The Design of Approximation Algorithms*. Cambridge University Press, Cambridge, 2011. ISBN 978-0-511-92173-5. doi: 10.1017/CBO9780511921735.

Wingate, D., Stuhlmüller, A., and Goodman, N. D. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. pp. 9, 2011.

Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. Big Bird: Transformers for Longer Sequences, January 2021.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

# A. Experimental Details

| 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 Figure 3 and Figure 5.

Table 2: Experimental parameters. Listed training times refer to those used in Figures 3 and 5. The numbers of training iterations refer to those listed in the same plot with the listed problem dimension. They vary with problem dimensions as we trained all dimensions for a fixed training time on each problem, and the time per iteration depends on problem dimension. The training curves in Figure 6 were obtained by training for longer in some cases. The dimensions listed are those used in Figure 6; dimensions are varied and clearly stated in other results.

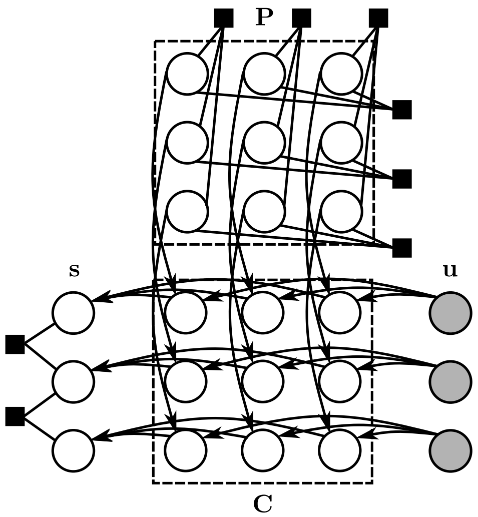
| Parameter | Sorting | Sudoku | BCMF | Boolean |
|---|---|---|---|---|
| Problem dimension | $n = 20$ | $9 \times 9$ | $n, m, k = 16, 10, 8$ | - |
| Training time | 1 day | 1 day | 8 hours | 40-160 min. |
| Training iters (1000s) | 120 | $-$ | 320 | 20 |
| Batch size | 16 | 32 | 8 | 16 |
| Learning rate | $2 \times 10^{-4}$ | $2 \times 10^{-5}$ | $2 \times 10^{-5}$ | $2 \times 10^{-5}$ |
| Embedding dim. | 64 | 128 | 64 | 64 |
| # transformer layers | 6 | 6 | 12 | 12 |
| # attention heads | 8 | 8 | 2 | 2 |
| GPU type | A100 | A5000 | A100 | A5000 |
| VAEAC learning rate | $3 \times 10^{-5}$ | $3 \times 10^{-4}$ | $3 \times 10^{-5}$ | $3 \times 10^{-5}$ |
| LMH warmup samples | - | 5000 | 5000 | - |

We re-summarize our experiments and the results obtained in Table 1. 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 not tune 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 & Ba, 2015), no weight decay and gradient clipping at 1.0. For LMH we relax the discrete delta distributions representing "hard constraints" for Sudoku and sorting by observing a Bernoulli variable with a probability of $0.9999$ to provide individual guidance to the sampling process for each constraint. For BCMF, we relax the Dirac distribution on the output matrix $\boldsymbol{E}$ to a normal distribution with standard deviation $0.01$. We release code to ensure full reproducibility of our results.

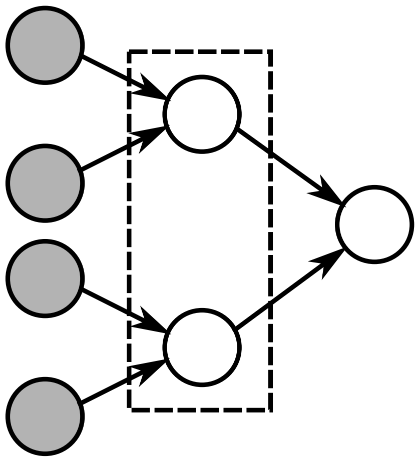
For the training plots in Figures 6, 7 and 22 we compute validation metrics regularly throughout training, in particular every 1000 iterations for BCMF and HBCMF and every 5000 iterations for sorting and Sudoku. We smooth the lines by averaging the metrics for each evaluation within each segment of the logarithmic grid (so e.g. the RMSE shown for sorting at $3 \times 10^4$

iterations is the average of the RMSE computed at $2.5 \times 10^4$ and that computed at $3 \times 10^4$). Where not otherwise specified, results for Sudoku are computed on examples with $50\%$ of cells observed.

## B. Graphical models



(a) Graphical model for sorting. The unsorted list $\mathbf{u}$ is observed and multiplied with the permutation matrix $\mathbf{P}$ (factors ensure one element is active per row and column) into intermediate variables $\boldsymbol{C}$. Summing over $\boldsymbol{C}$ yields the sorted nodes $\mathbf{s}$. The sorted nodes have pairwise constraint factors to ensure their ordering.



(b) Boolean circuit graphical model with two layers. The nodes in the dashed box are intermediate variables. The input nodes are observed and the output node is to be inferred.

The graphical model for BCMF is shown in Figure 1; for Sudoku in Figure 10; for sorting in Figure 8a; and for the Boolean circuit in Figure 8b. The Boolean circuit has a relatively simple tree structure. The others have more complex structures; note that even though BCMF may look like a tree in Figure 1, the use of plates mean that it is not as there are multiple paths between e.g. elements in $\boldsymbol{A}$ and elements in $\boldsymbol{E}$. The following section shows the resulting attention masks.

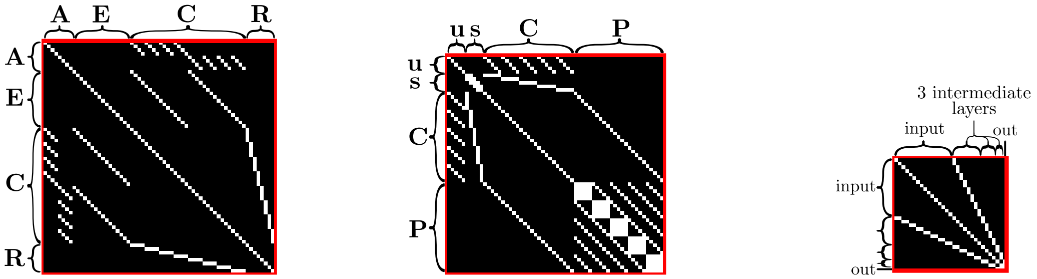
## C. Structured Attention

Figures 9 and 10 show examples of the attention masks used in all experiments, with and without intermediate variables. In Figure 11 we compare the imposed Sudoku sparsity mask to the attention weight matrices learned by our Non-sparse baseline. This confirms that, on Sudoku, the inductive bias we impose is appropriate.

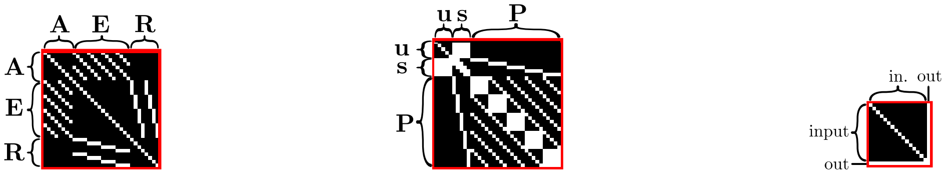
The optimal choice for our structured attention would be sparse matrix multiplication on the accelerator, unfortunately this was not yet available at the time of writing of this paper. We therefore provide a packed dense implementation of structured attention. Our structured attention mechanism lets 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 $\boldsymbol{M}$. 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

$$\boldsymbol{e} = \operatorname{softmax}\left(\boldsymbol{M} \odot \boldsymbol{Q}\boldsymbol{K}^T\right)\boldsymbol{V}. \tag{7}$$

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 $j$th 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



(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 9: 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.

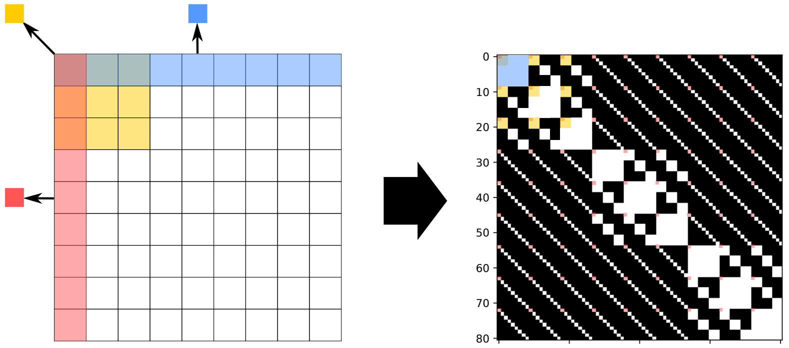


Figure 10: Correspondence between a $9 \times 9$ Sudoku grid (left) and the resulting attention mask. We draw a factor for each of the first row, column and block on the left. On the right the respective entries in the attention structured mask $\boldsymbol{M}$ are highlighted with the same color.

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 $\boldsymbol{M}$. Future work may integrate the sparse attention implementation of Kreuzer et al. (2021) that scales as $\mathcal{O}(e)$.

## D. Neural network architecture

In the following we denote Swish activations as SiLU (Hendrycks & Gimpel, 2020), group norm as norm, 2D convolution as conv2d, a linear layer as linear and neuron dropout as dropout. Following the implementation of Song et al. (2021a) our timestep embedding is implemented as the following sequence: linear; SiLU; linear. Our ResNet also follows Song et al. (2021a) by applying the sequence: norm; SiLU; conv2d; projection of time and variable embedding; norm; SiLU; dropout; conv2d. Compared to Song et al. (2021a) our convolution here is 1D and not 2D since we do not operate on images.

We repurpose the same architecture for our "Regressor + GS" and "Regressor" baselines, using the GSDM architecture for "Regressor + GS" and the non-sparse variation for "Regressor". To do so, we simply replace the inputs $\mathbf{x}_t$ and $t$ with

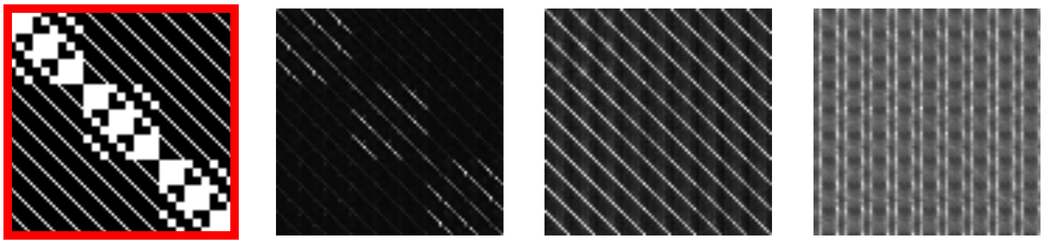


Figure 11: **Left/red border:** The attention mask imposed by GSDM on Sudoku. **Remainder:** Three attention masks obtained from different attention layers/heads of an non-sparse diffusion model on Sudoku. Soft attention to row, column and block constraints are visible, indicating that we impose an appropriate inductive bias.

appropriately-shaped arrays of zeros for every training iteration as well as during inference. The modified architectures therefore take a sole input $\mathbf{y}$ and return a sole output, their prediction of $\mathbf{x}$.

## E. Faithfulness of Attention

Our design choices regarding the construction of attention masks, and the depth of our architecture, are motivated by the following. According to the DM loss in Equation (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 12 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 E.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 & Friedman, 2009). ☐

The consequence of Theorem E.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 Figure 7. Note that if 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.

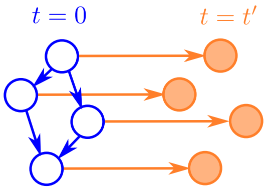


Figure 12: Example graphical model of $q(\mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)$. Nodes are latent/blue if in $\mathbf{x}_0$, observed/orange if in $\mathbf{x}_t$.

## F. Permutation invariance through shared embeddings

In this section we first provide intuition about permutation invariance in diffusion models before restating and proving Theorem 3.1. Our architecture provides an opportunity to enforce permutation invariance in the learned distribution. Consider a “vanilla” DM which has a similar architecture to ours in Figure 2 but without the attention mask. The only information this network receives about the ordering of its input comes from the node embeddings and, if the node embeddings were removed, the architecture would be entirely permutation-equivariant. Similarly, if the node embeddings were shared between a set of nodes, the architecture would be equivariant to permutations of this set of nodes. The modeled distribution would therefore be invariant to permutations of this set of nodes (Hoogeboom et al., 2022). This may be a useful permutation invariance to encode for some problems but, for the structured problems considered in this paper, it is too simple and not valid. We introduce the following theorem to get describe other, more practically applicable, types of permutation invariance.

**Theorem F.1** (Permutation invariance in GSDM)**.** *Let $\mathcal{A}$ represent the indices of a subset of the dimensions of data $\mathbf{x}$ and $\Pi_{\mathcal{A}}$ be the class of permutations that permute only dimensions indexed by $\mathcal{A}$. Assume we have a GSDM parameterised with neural network $\hat{\mathbf{x}}_\theta(\cdot;\boldsymbol{M})$, where $\boldsymbol{M}$ is the structured attention mask. If the node embeddings used by $\hat{\mathbf{x}}_\theta$ are shared across all nodes indexed by $\mathcal{A}$, then the distribution modelled by GSDM will be invariant to all permutations $\pi$ satisfying*

$$\boldsymbol{M} = \pi\boldsymbol{M} \quad \textit{and} \quad \pi \in \Pi_{\mathcal{A}} \tag{8}$$

*where $\pi\boldsymbol{M}$ is a permutation of both the rows and columns of $\boldsymbol{M}$ by $\pi$.*

*Proof.* Our architecture without sparse attention, i.e. with $\boldsymbol{M} = \mathbf{1}$, is equivariant under $\Pi_{\mathcal{A}}$ in that (writing the attention mask as an additional input)

$$\hat{\mathbf{x}}_\theta(\pi\mathbf{x}_t;\mathbf{1}) = \pi\hat{\mathbf{x}}_\theta(\mathbf{x}_t;\mathbf{1}) \qquad \forall\pi \in \Pi_{\mathcal{A}} \tag{9}$$

for any $\mathbf{x}_t$. The analysis is different when the network uses an attention mask because the attention mask provides additional information about the ordering of the inputs. Replacing $\boldsymbol{M}$ by $\pi\boldsymbol{M}$, the permutation of both the rows and columns of $\boldsymbol{M}$ by $\pi$, prevents this:

$$\hat{\mathbf{x}}_\theta(\pi\mathbf{x}_t,\pi\boldsymbol{M}) = \pi\hat{\mathbf{x}}_\theta(\mathbf{x}_t,\boldsymbol{M}) \qquad \forall\pi \in \Pi_{\mathcal{A}} \tag{10}$$

For equivariance to permutations of $\mathbf{x}$ alone, however, we require

$$\hat{\mathbf{x}}_\theta(\pi\mathbf{x}_t,\boldsymbol{M}) = \pi\hat{\mathbf{x}}_\theta(\mathbf{x}_t,\boldsymbol{M}) \qquad \forall\pi \in \Pi_{\mathcal{A}}. \tag{11}$$

In general, the equality in Equation (10) will only imply that in Equation (11) if $\boldsymbol{M} = \pi\boldsymbol{M}$. Therefore, when used in combination with a structured attention mask $\boldsymbol{M}$, sharing embeddings among all nodes in $\mathcal{A}$ will lead to equivariance only to the set of permutations $\{\pi \in \Pi_{\mathcal{A}} | \boldsymbol{M} = \pi\boldsymbol{M}\}$. ☐

## G. Generalization with BCMF problem dimension

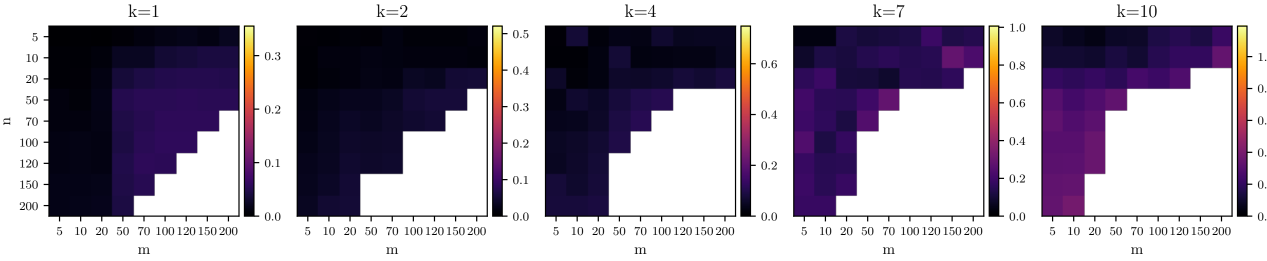


Figure 13: Error vs problem dimension for GSDM on binary continuous matrix factorization. We show the root mean square error between the observed matrix $\boldsymbol{E}$ and the product of the sampled $\boldsymbol{A}$ and $\boldsymbol{R}$. The colorbar for each value of $k$ is scaled so that "yellow" 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.

In Figure 14 we plot a heatmap similar to Figure 13 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.

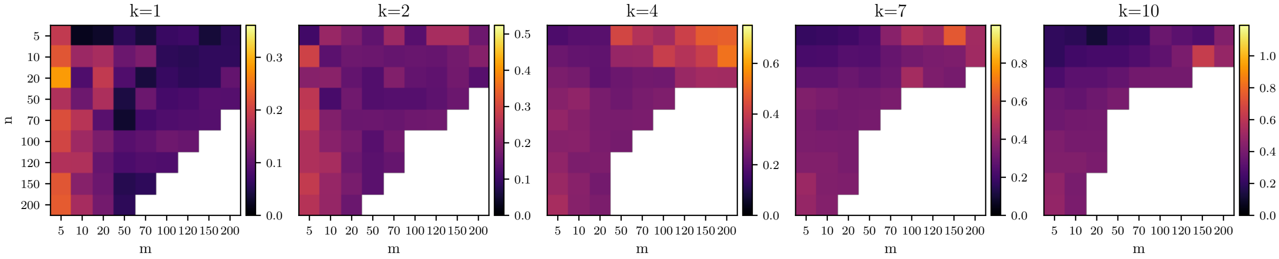


Figure 14: Similar to Figure 13 but with a GSDM model trained to sampled $\boldsymbol{A}$, $\boldsymbol{R}$, and $\boldsymbol{E}$ jointly instead of conditioning on $\boldsymbol{E}$. We plot a heatmap of the root mean squared error (RMSE) between the matrix $\boldsymbol{E}$ and the product $\boldsymbol{AR}$. 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 |
|---|---|---|---|---|---|
| BCMF with $k = m = n$ | Naive<br>GSDM | $\mathcal{O}(n^2)$<br>$\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^4)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^4)$ | -<br>$\mathcal{O}(1)$ |
| Sorting with input size $n$ | Naive<br>GSDM | $\mathcal{O}(n^2)$<br>$\mathcal{O}(n^2)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^3)$ | -<br>$\mathcal{O}(n)$ |
| Sudoku with side length $n$ | Naive<br>GSDM | $\mathcal{O}(n^2)$<br>$\mathcal{O}(n^2)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^3)$ | $\mathcal{O}(n^4)$<br>$\mathcal{O}(n^3)$ | -<br>$\mathcal{O}(n)$ |
| Boolean with input size $n$ | Naive<br>GSDM | $\mathcal{O}(n)$<br>$\mathcal{O}(n)$ | $\mathcal{O}(n^2)$<br>$\mathcal{O}(n)$ | $\mathcal{O}(n^2)$<br>$\mathcal{O}(n)$ | -<br>$\mathcal{O}(n)$ |

Table 3: Comparison of computational complexities of a GSDM layer and a naive DM layer without structured attention or intermediate variables. 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.

## H. Computational Complexities

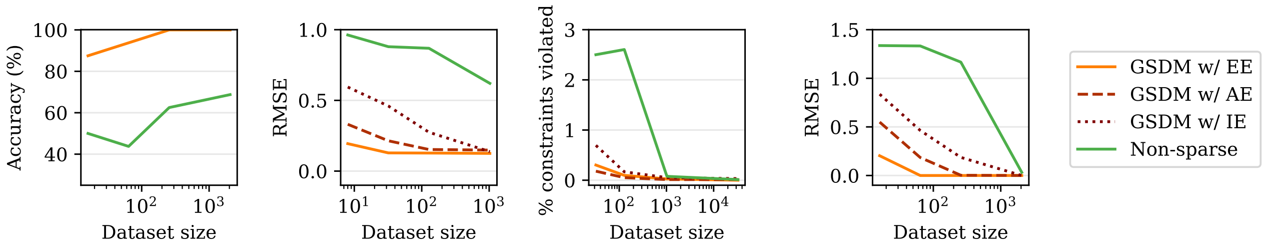
Table 3 compares the computational cost of GSDM with that of naively applying a DM without intermediate variables or structured attention.

## I. Data efficiency

Throughout this paper, we have focused on the case where infinite training data is available from a fast-to-sample generative model of $\mathbf{x}$ and $\mathbf{y}$. We now show that GSDM provides further advantages if limited data is available through the inductive biases that we impose on its architecture. In Figure 15 we demonstrate this by showing results when we train on a finite number of examples. On the left-hand side of each plot, the number of unique data points is equal to the batch size so that all data points are seen in every batch. Our non-sparse baseline does consistently worse than GSDM in this setting, and the GSDM variations with non-exchangeable embeddings have performance in between. GSDM reaches its best performance in each case with less than a thousand data points. The non-sparse baseline needs considerably more to be competitive. Training times and hyperparameters for each variation were as mentioned in Table 2.

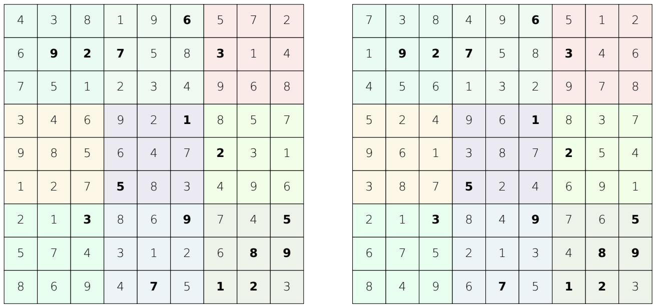
## J. Sudoku sample diversity

Sample diversity for Sudoku solving is shown in Figure 16. GSDM's objective is naturally mass-covering so enforces sample diversity.



(a) Boolean circuit (b) BCMF (c) Sudoku (d) Sorting

Figure 15: RMSE for GSDM and baselines trained on varying numbers of unique data points. GSDM needs far fewer data points to than the baselines to fit well.



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

## K. Matrix inversion

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 $\boldsymbol{E} = \boldsymbol{AR}$ (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 Figure 17. 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.

## L. 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 demonstrate it on the program on page 21, 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 18. We envisage a future extension which "compiles" directly from such source code to a trained GSDM network.

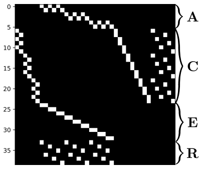


Figure 18: Connectivity mask extracted from BCMF source code. This is the same structure as in Figure 1 but with permuted indices and before the addition of the diagonal self-edges.

## M. BCMF baselines

Here we provide full detail on the "hand-coded algorithm" baselines which we compare against on BCMF. They are as follows

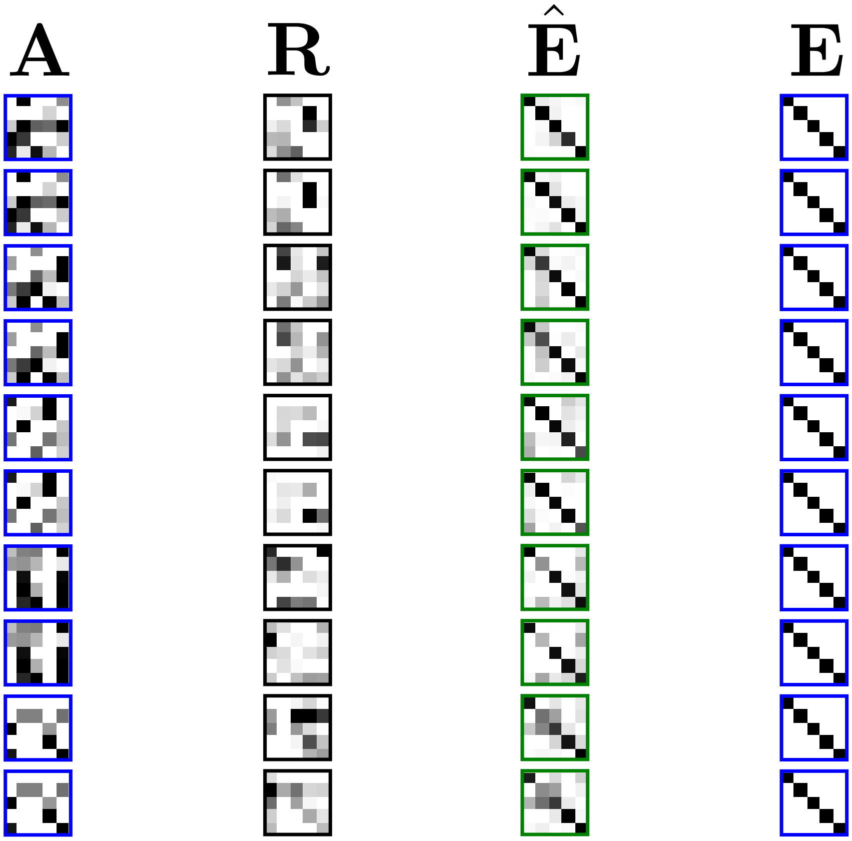


Figure 17: Rows of matrix inversion examples. Similar to Figure 4, but here we condition both on $\boldsymbol{A}$ and $\boldsymbol{E} = \mathbf{1}$ (blue). Each of the 5 pairs of rows show a solution for the same $\boldsymbol{A}$. Reconstructions are shown as $\hat{\boldsymbol{E}} = \boldsymbol{AR}$ (green).

**K-means** We treat the $m \times n$ matrix $\boldsymbol{E}$ as a dataset containing $n$ data points, each made up of $m$-dimensions. We then use an off-the-shelf implementation of K-means clustering with number of clusters $K$ set to to the rank $k$. This returns an $m \times k$ matrix of cluster centers, which we use as $\boldsymbol{A}$. It also returns the cluster indices in $\{1, \dots, k\}$ associated with each of the $n$ data points. We convert the indices to one-hot vectors and stack them to give a $k \times n$ binary matrix, which we return as $\boldsymbol{R}$. Doing a factorization of $\boldsymbol{E}$ into $\boldsymbol{A}$ and $\boldsymbol{R}$ in this manner and then reconstructing $\hat{E} := \boldsymbol{AR}$ can be understood as snapping each row of $\boldsymbol{E}$ to one of the $k$ "mean" rows in $\boldsymbol{R}$. Therefore, when $k$ is equal to $n$, this approach yields zero error, as is visible in Figure 5. The error quickly increases as $n$ grows larger than $k$.

**NMF** We perform non-negative matrix factorization of $\boldsymbol{E} \in \mathbb{R}^{m \times n}$ into $\tilde{\boldsymbol{A}} \in \mathbb{R}^{m \times k}$ and $\tilde{\boldsymbol{R}} \in \mathbb{R}^{k \times n}$ with coordinate descent, as implemented in Scikit-learn Pedregosa et al. (2011). We then simply round each element in $\boldsymbol{R}$ to be in $\{0, 1\}$.

**ChatGPT** We show the prompt and produced source code in Figure 20. The algorithm it produces finds the $k$ largest rows in $\boldsymbol{E}$ and copies them to $\boldsymbol{A}$, adding an index in $\boldsymbol{R}$ so that they are reconstructed if we compute $\hat{E} := \boldsymbol{AR}$. The reconstruction is thus perfect for the largest $k$ rows and zero for all other rows. This means that the baseline achieves zero error when $k = n$ and relatively large error otherwise. We also see in Figure 5 that the error spikes for $k$ larger than $n$ and $m$ (e.g. for $n = m = 16$ and $k = 20$) as this edge case is unhandled.

```
(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 19: Source code of a full generative model for the BCMF experiment. Passing this into our compiler yields the attention mask in Figure 18. Note that intermediate variables for $C$ are explicitly created by sampling from a dirac distribution.

## N. Hierarchical BCMF

In this section we present an extension of the binary-continuous matrix factorization model in which the generative model samples factors from a hierarchical prior. As shown in Figure 21, our generative model samples a rank variable and then sample the factors and conditioned on having exactly this many non-zero rows. We report the generative model, the associated sparsity mask, our results, and example factorizations. The differing sizes of and of the factorizations reflect the different ranks that are sampled for each. We show results in Figures 22 and 23. GSDM consistently selects an appropriate rank. We find that, as in our non-hierarchical matrix factorization experiment, GSDM is the best-performing method and indeed the only neural method to outperform our hand-crafted approximate solutions.

**Prompt:** Write a Python function which takes as input E, an m x n non-negative matrix, and factorises it into a mxk binary matrix and a kxn continuous component with values in [0, 1]

**Returned code:**

```
import numpy as np

def factorize(E, k):
    """
    Factorize an m x n non-negative matrix E into a mxk binary matrix and a kxn continuous component with values in [0, 1].

    Args:
        E (numpy.ndarray): The non-negative matrix to be factorized.
        k (int): The number of columns in the binary matrix and rows in the continuous component.

    Returns:
        A tuple containing the binary matrix and continuous component, respectively.
    """

    # Initialize the binary and continuous matrices
    B = np.zeros((E.shape[0], k), dtype=np.int)
    C = np.zeros((k, E.shape[1]), dtype=np.float)

    # Perform the factorization
    for i in range(k):
        # Find the row in E with the largest sum that hasn't been assigned to a column in B yet
        max_sum = -1
        max_row = -1
        for j in range(E.shape[0]):
            if np.sum(B[j,:]) == 0 and np.sum(E[j,:]) > max_sum:
                max_sum = np.sum(E[j,:])
                max_row = j

        # Assign the row to the ith column of B
        B[max_row, i] = 1

        # Compute the ith row of C as a weighted average of the rows in E selected by B
        selected_rows = B[:,i] == 1
        C[i,:] = np.sum(E[selected_rows,:], axis=0) / np.sum(selected_rows)

    return B, C
```

Figure 20: Binary-continuous matrix factorization code produced by ChatGPT (OpenAI, 2021) We used the prompt at the top and show the code produced, ignoring additional description and examples that it gave.

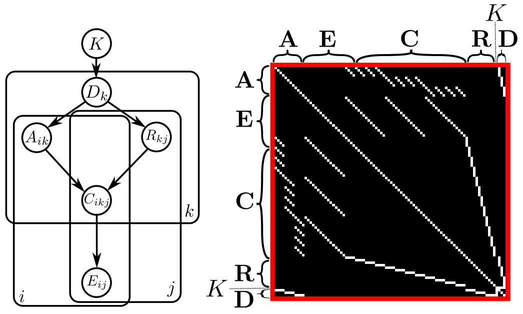


Figure 21: **Left:** A graphical model for hierarchical binary-continuous matrix factorization (HBCMF). The rank $K$ is first sampled from a uniform distribution. Given this, $\mathbf{D}_k$ is set to 1 for $k \leq K$ and 0 otherwise. Each $k$th row of $\mathbf{R}$ is then sampled conditioned on being zero iff $\mathbf{D}_k$. The same is done for each $k$th column of $\mathbf{A}$. **Right:** The sparsity mask for GSDM derived from the HBCMF graphical model. It is similar to that for BCMF but with added rows and columns for $K$ and $\mathbf{D}$. The single dimension of $K$ is indicated with a dashed line. In this model, the upper limit on $K$ is three so $\mathbf{D}$ has three dimensions. $\mathbf{D}$ interacts with both $\mathbf{A}$ and $\mathbf{R}$ following the graphical model.

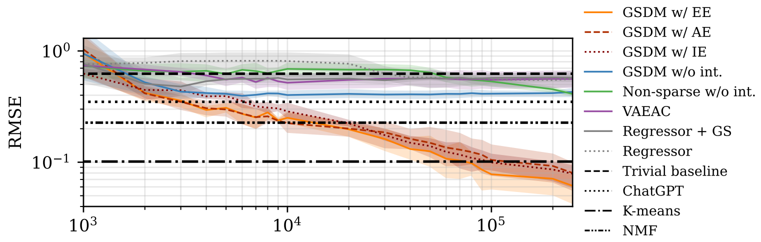


Figure 22: Experimental results on HBCMF. As in our non-hierarchical matrix factorization experiment, GSDM is the best-performing method and indeed the only neural method to outperform our hand-crafted approximate solutions.

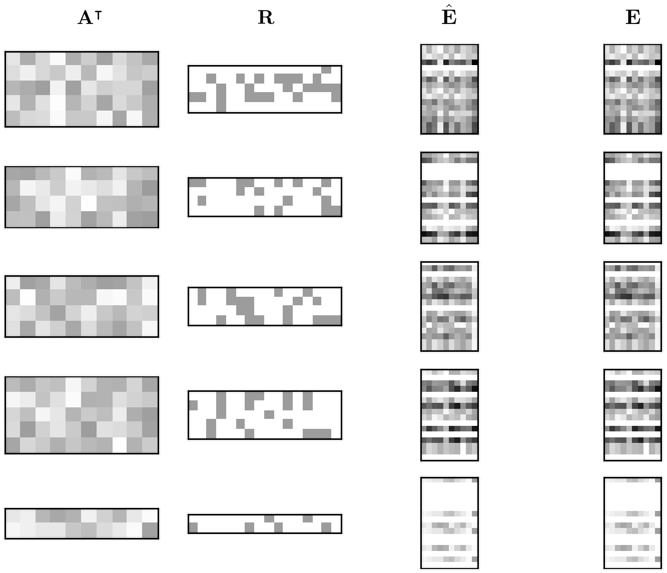


Figure 23: Example HBCMF factorizations of five matrices by GSDM, one per row. The differing sizes of $\mathbf{A}$ and $\mathbf{R}$ in this figure reflect the different ranks that are sampled for each. GSDM consistently selects an appropriate rank.