GTLayout: Learning General Trees for Structured Grid Layout Generation

Pengfei Xu[1], Weiran Shi[1], Xin Hu[1], Hongbo Fu[2], and Hui Huang[1]

[1] Shenzhen University, Shenzhen, China
[2] City University of Hong Kong, Hong Kong, China

Abstract. Structured grid layouts are preferable in many scenarios of 2D visual content creation since their structures facilitate further layout editing. Multiple geometry-based methods can effectively create structured grid layouts but require user-provided constraints or rules. Existing data-driven approaches have achieved remarkable performance on layout generation, but fail to produce appropriate layout structures. We present *GTLayout*, a novel generative model for structured grid layout generation. We adopt general trees to represent structured grid layouts and exploit a recursive neural network (RvNN) for this generation task. Our model can handle grid layouts with varied structures and regular arrangements. Qualitative and quantitative experiments on public grid layout datasets show that our method outperforms several baselines in the tasks of layout reconstruction and layout generation, especially when the datasets contain a small number of samples. We also demonstrate that the structured layout space constructed by our method enables structure blending between structured layouts. We will release our code upon the acceptance of the paper.

Keywords: Grid layout · Recursive neural network · Layout structure · Layout generation · Layout interpolation.

1 Introduction

Creating grid layouts [30, 33] is a fundamental step for creating 2D visual content in various forms, including documents, magazines, webpages, GUIs, *etc*. Grid lines can be used to regularize such layouts' structures, including the spatial relations and organizations of graphical layout elements [5]. These structures can be abstracted as graphs [8, 44] or trees [14, 20] and can facilitate further editing (see Figure 1), *e.g*., adjusting elements in a structure-preserving manner [8], adapting layouts to various display configurations [14], *etc*. Given the great usability of layout structures, multiple geometry-based methods have been proposed to effectively create structured layouts interactively [44, 47] or automatically [5, 40, 43]. However, they are not always preferable since the interactive methods require heavy labor inputs and the automatic ones demand high-level constraints or rules provided by users. It seems more natural to address these problems with data-driven approaches.

Very recently, learning techniques, including GAN [9], VAE [23], Transformer [41], GNN [38], Diffusion model [11], *etc*., have benefited the generation of layouts [1, 4, 10, 12, 13, 17, 18, 21, 24, 48]. These layout generation methods have achieved remarkable

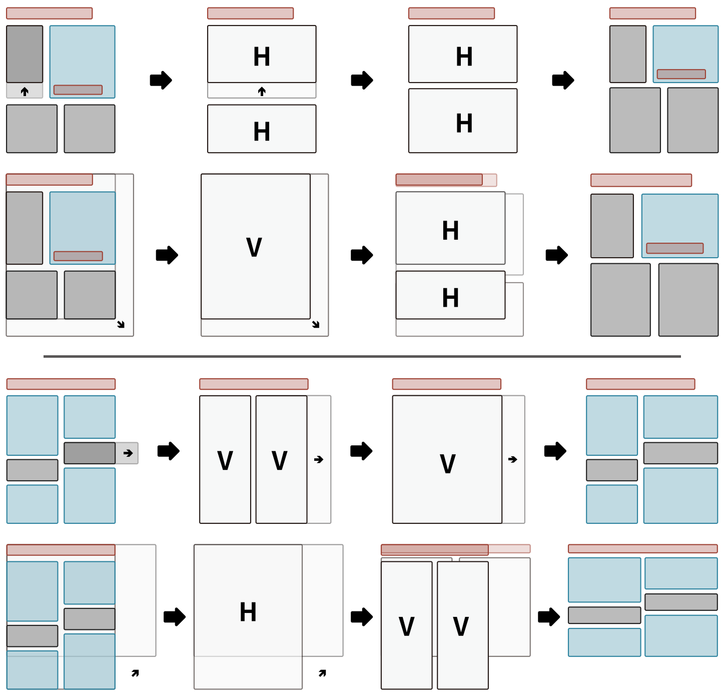

Fig. 1. Structured layouts can facilitate further editing, *e.g.*, adjusting elements in a structure-preserving manner (Row 1 and 3) and adapting layouts to various display configurations (Row 2 and 4).

results in automatic layout creation with different problem formulations. Early methods [48] consider layouts as raster images and represent graphic layout elements as color regions. The succeeding methods often adopt a parametric representation of layouts, *i.e.*, representing graphic layout elements as bounding boxes with semantic labels. This parametric representation enables convenient applications of the generated layouts since the bounding boxes can be replaced with semantic elements to synthesize realistic visual content. However, this procedure requires the semantic elements to possess the same geometries as the corresponding bounding boxes. This requirement cannot be achieved in some scenarios. For example, when synthesizing a magazine page, the image aspect ratio or the text length might not be consistent with the corresponding bounding boxes in the generated layouts. In this case, the layout structures that explicitly indicate the spatial relations and organizations of graphic elements can help automatically adjust the layout geometries. Nevertheless, none of the above layout generation methods can produce such layout structures.

Recursive neural networks (RvNNs) [39] are effective in generating structured data and have succeeded in several generation problems, *e.g.*, 3D shape generation [28, 32, 50] and indoor scene generation [29]. They are also suitable for the problem of structured grid layout generation. READ [35] is the first learning-based method that rec-

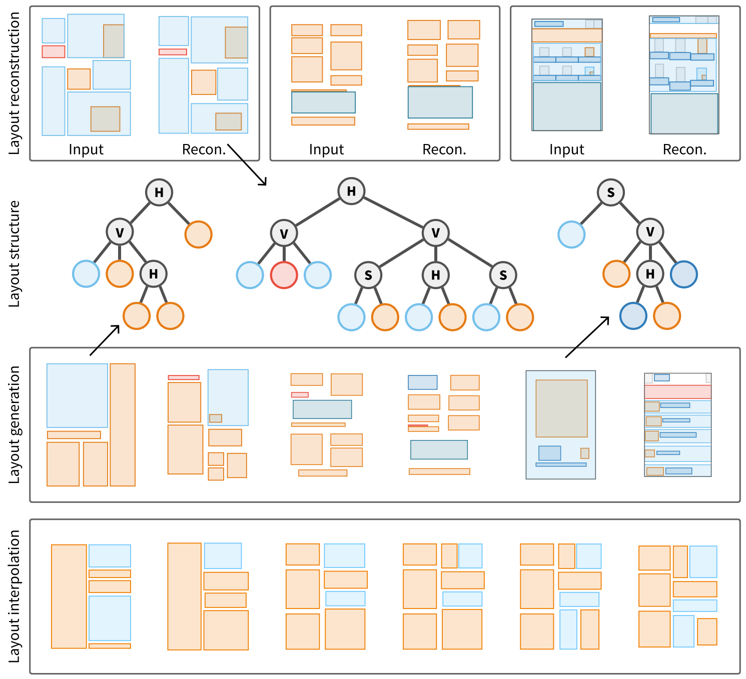

Fig. 2. Our method is effective in producing high-quality structured grid layouts. Row 1: layout reconstruction with our method. Row 2: layout structures produced by our method. Row 3: layouts generated by our method with randomly sampled latent codes. Row 4: structured layout blending achieved by our method. The leftmost and rightmost layouts are the inputs for blending.

ognizes the importance of layout structures. It adopts RvNN for structured document layout generation and can produce realistic document layouts. However, READ focuses more on exploiting the structures as tools for layout generation, instead of generating plausible layout structures. It adopts binary trees as the representation of layout structures. Although this structural representation is effective in the layout generation task, it deviates from people's perceptions of layout structures. A more appropriate representation for grid layout structures would be general trees [7, 15, 17, 20, 34, 43], in which leaf nodes represent graphic elements and internal nodes represent horizontal or vertical arrangements of their children. However, READ fails to produce layouts with such structures.

This paper presents *GTLayout*, a novel generative model for the structured grid layout generation (Figure 2). Inspired by the recent structured object generation methods[28, 29, 32, 35, 50], we also adopt RvNN for our generation task. Compared with existing RvNN-based generative models, our model needs to address the following new challenges. First, compared with 3D shapes, the grid layout structures are highly varied and involve spatial relations among different numbers of elements. Second, high-quality grid layouts are often composed of regular arrangements from local to global, while this regularity only exists in substructures of indoor scenes.

To address these challenges, we introduce the following new designs. Instead of binary trees adopted by existing works [28, 35], we adopt general trees as the structural representation of grid layouts. The hierarchical structures of grid layouts are estimated using a method similar to the one described in [15, 17, 43]. In addition, different from existing works [29, 32], we do not constrain the structures of these trees, and thus they can represent any structured grid layouts. With a dataset of structured grid layouts, we train a variational recursive autoencoder (RvNN-VAE), which embeds layouts into a structure-aware layout space in a recursive bottom-up manner. Specifically, we introduce a set of encoders and decoders, including geometry encoder/decoder, label encoder/decoder, element encoder/decoder, and arrangement encoders/decoders. These encoders are recursively applied to the substructures of layouts and encode the overall structures and geometries of layouts into fixed-length codes that roughly follow a Gaussian distribution. A new structured grid layout can be obtained hierarchically by decoding a randomly generated code with the decoders.

Our method is effective in producing high-quality structured grid layouts (see Figure 2). We compare our method with several baseline methods, including LayoutGAN++ [21], VTN [1], and READ [35]. All these three methods can construct layout spaces in which a latent code represents a layout. The comparison includes layout generation, layout reconstruction, and layout interpolation. The experiments are conducted on three public layout datasets, including Magazine [48], PubLayNet [49], and RICO [6]. To better examine how these methods are affected by the datasets' scales, we prepare a series of datasets by gradually reducing the samples in these three public layout datasets. The experiments show that our method outperforms these methods qualitatively and quantitatively in the tasks of layout generation and reconstruction. The superiority of our method is more significant for the datasets of small scales, indicating that our method is more suitable for practical usage. For the layout interpolation task, only our method achieves smooth and reasonable structure blending. To the best of our knowledge, this is accomplished for the first time by a learning-based method.

2 Related work

Interactive layout creation. Grid layouts exist in various forms of 2D visual content and usually possess regular structures. Many techniques have focused on facilitating the interactive creation of such layouts, *e.g.*, snapping tools [2, 3], arrangement commands [37, 42]. These techniques help create regular grid layouts incrementally but do not extract the layout structures. Xu *et al.* [44] proposed a framework for globally beautifying roughly aligned grid layouts. The spatial relations among the elements in a layout were inferred and could serve as the layout structures. Zeidler *et al.* [47] proposed the Auckland layout editor to help users interactively create structured GUI layouts. Although these techniques have improved the efficiency of structured layout creation, they still require heavy labor input. This problem becomes severer when producing a large number of layouts.

Geometry-based layout generation. Several geometry-based methods have been proposed for the automatic generation of structured grid layouts. For example, O'Donovan *et al.* [34] presented an optimization method for generating structured grid layouts based on the design principles extracted from existing layouts. Kikuchi *et al.* [20] introduced a method for generating webpage layouts by formulating the layout generation as a hierarchical optimization problem. The framework named Grids proposed by Dayama *et al.* [5] adopted a mixed integer linear programming solution to automatically generate structured grid layouts based on only several heuristic rules. Swearngin *et al.* [40] presented Scout, a system that helped designers explore structured grid layouts, which were generated based on user-provided high-level constraints. Xu *et al.* [43] introduced a method for creating novel structured grid layouts by blending existing ones. The core of their method was a correspondence algorithm devised according to high-level rules. Although these methods have achieved the automatic generation of structured grid layouts, the high-level constraints or rules may not be available or preferable for different types of grid layouts. Yang *et al.* [46] presented a method for generating urban layouts by recursively splitting regions. Although their method was not designed for grid layout generation, we were inspired by their idea of the recursive procedure for layout generation.

Learning-based layout generation. Learning techniques have benefited the task of layout generation in recent years. A pioneer work was LayoutGAN [27], which adopted a generative adversarial network [9] for layout generation. LayoutGAN++ [21] adopted a similar network and further improved the quality of generated layouts. LayoutVAE [18] exploited two variational autoencoders [23], *i.e.*, CountVAE and BBoxVAE, to generate layouts. LayoutTransformer [10] and VTN [1] leveraged Transformer [41] to produce layouts and achieved remarkable results. Based on layout generation, CanvasVAE [45] provided a Transformer-based method for vector graphic document generation. Jiang *et al.* [17] also exploited Transformer for layout generation. Their coarse-to-fine strategy was insightful. NDN [26] adopted a graph neural network [38] for generating layouts satisfying user constraints. Recently, several works focused on controllable layout generation. BLT [24] extended BERT [19] to the layout generation task. It learned to predict the masked attributes of layouts based on known attributes. Layoutformer++ [16] could take geometric relations among layout elements as conditions for the layout generation. Two LayoutDMs [4, 13] and LDGM [12] adopted Diffusion model [11] for the layout generation. They also enabled controllable layout generation. These recent works have greatly advanced the research on layout generation, However, none of them can produce layout structures that explicitly indicate the spatial relations and organizations of graphic elements, though some of them [4, 12, 16, 26] exploited layout structures as conditions for layout generation. READ [35] was the first learning-based layout generation method that exploited and produced layout structures. However, its adopted layout structures were represented as binary trees, which deviated from peoples' perceptions. In contrast, our method adopts general trees as a more appropriate representation of layout structures, constructs a structure-aware layout space, and achieves better performance in the tasks of layout reconstruction, layout generation, and layout blending (Section 4).

RvNN for structure generation. We adopt RvNN [39] for our structured layout generation task. This network has been proven effective in various structured data generation

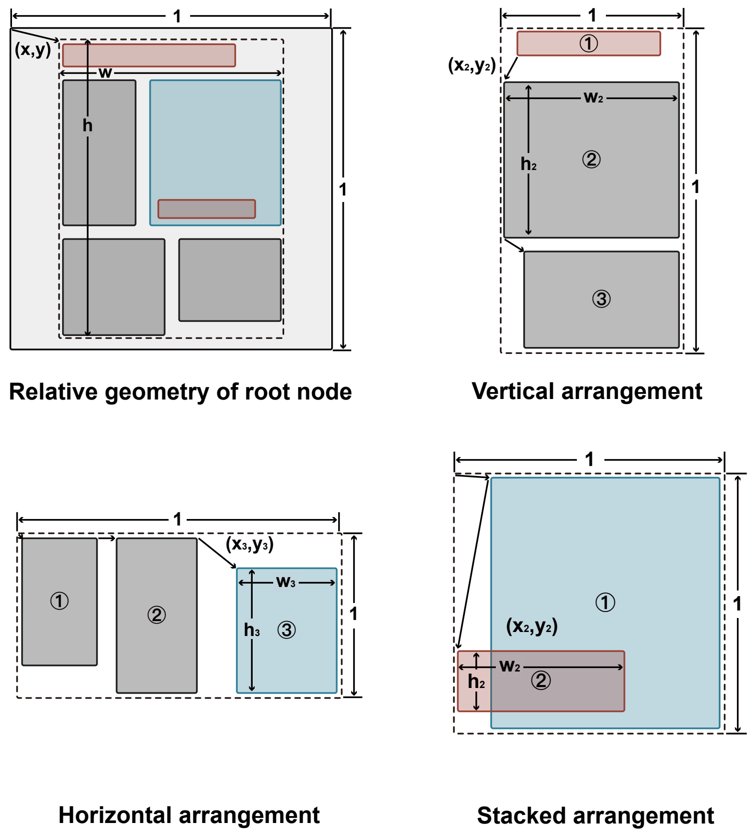

Fig. 3. To better capture arrangement patterns, we store the relative positions and sizes of node bounding boxes. The four sub-figures show how we compute the relative geometries in different arrangements. Top left: the relative geometry of the root node of a layout. Top right: vertical arrangement. Bottom left: horizontal arrangement. Bottom right: stacked arrangement.

tasks. For example, Li *et al.* [28] presented GRASS, the first work for exploiting RvNN for structured 3D shape synthesis. Zhu *et al.* [50] presented SCORES that leveraged RvNN for structured 3D shape composition. The shape structures were represented as binary trees in both works. StructureNet [32] was another RvNN-based method for structured shape generation. It adopted general trees to represent shape structures. However, it was designed to handle shapes in the same categories, i.e., shapes with similar structures. Li *et al.* [29] presented GRAINS, an RvNN-based method for synthesizing indoor scenes. This method aimed to produce 2D layouts in natural. Compared with indoor scene layouts, grid layouts possess more varied and regular structures, thus posing new challenges to layout generation.

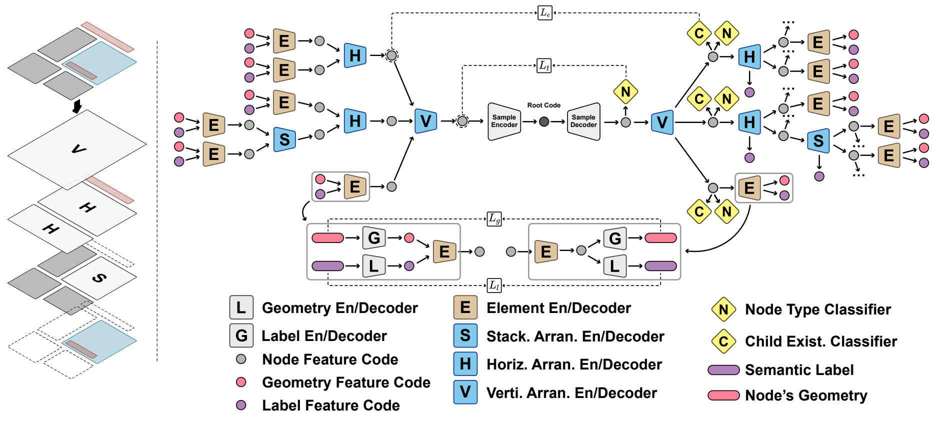

Fig. 4. Left: the procedure for extracting a layout tree from a grid layout. Right: the architecture of our generative model illustrated with an example. The encoders in our generative model map a structured grid layout to a latent feature code. The encoding is achieved in a recursive bottom-up manner. The decoders covert a latent feature code to a structured grid layout reversely.

3 Method

Our method adopts RvNN-VAE to embed structured grid layouts into a structure-aware layout space in a recursive bottom-up manner. In this space, layouts are represented as fixed-length codes that roughly follow a Gaussian distribution. Novel structured layouts can be obtained by decoding randomly generated codes. We first describe the structural representation adopted by our method and how we extract such a representation (Section 3.1). Then we introduce our generative model, including the encoders, the decoders, and the encoding/decoding procedures (Section 3.2). Finally, we explain the training objective and training details of our generative model (Section 3.3).

3.1 Structural layout representation

Most existing learning-based layout generation techniques consider a layout as a set of bounding boxes with semantic labels. READ [35] exploits layout structures for layout generation but adopts binary trees as a structural representation. As discussed in [15, 43], structured grid layouts are naturally hierarchical and can be represented by general trees. We thus adopt general trees, which we term layout trees, as the representation of structured layouts.

To extract a layout tree from a grid layout, in which elements are represented as labeled bounding boxes, we use a method similar to the one described in [15, 43], but allows elements to stack. Figure 4 (left) shows an example to illustrate this procedure. Specifically, we recursively split a layout horizontally and vertically with grid lines that do not traverse elements. If a sub-layout contains multiple elements but cannot be split, we consider that it contains stacked elements. We then decompose this sub-layout into two parts with the first part being the largest element in this sub-layout and the

Algorithm 1 Layout Tree Extraction

```
Input: A set of layout elements $\mathcal{T} = \{e_i\}$, each $e_i$ having attributes $\{label, x, y, w, h\}$.
Output: Hierarchical structured layout tree root node $N$.
1:
2: Class Node:
3:     Attributes: $children$, $parent$, $elements$
4:     Method: add_child(node)
5:
6: Function ExtractTree($\mathcal{L}$, $N$):
7: if $\mathcal{L}$ can be divided horizontally then
8:     Split $\mathcal{L}$ horizontally into subsets $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$.
9:     for each subset $\mathcal{L}_i \subseteq \mathcal{L}$ do
10:        Create a new Node $N_i$ with $\mathcal{L}_i$ as $elements$.
11:        $N$.add_child($N_i$).
12:        Call ExtractTree($\mathcal{L}_i$, $N_i$).
13: else if $\mathcal{L}$ can be divided vertically then
14:     Split $\mathcal{L}$ vertically into subsets $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$.
15:     for each subset $\mathcal{L}_i \subseteq \mathcal{L}$ do
16:        Create a new Node $N_i$ with $\mathcal{L}_i$ as $elements$.
17:        $N$.add_child($N_i$).
18:        Call ExtractTree($\mathcal{L}_i$, $N_i$).
19: else
20:     Let $e_{max}$ be the largest element in $\mathcal{L}$.
21:     Let $\mathcal{L}_{remain} = \mathcal{L} \setminus \{e_{max}\}$.
22:     Create a new Node $N_{max}$ with $e_{max}$ as $elements$.
23:     $N$.add_child($N_{max}$).
24:     if $\mathcal{L}_{remain}$ is not empty then
25:        Create a new Node $N_{remain}$ with $\mathcal{L}_{remain}$ as $elements$.
26:        $N$.add_child($N_{remain}$).
27:        Call ExtractTree($\mathcal{L}_{remain}$, $N_{remain}$).
28:
29: Initialize root node $N_{root}$ as ExtractTree($\mathcal{T}$, $N_{root}$).
```

second part being the rest elements. The elements in the second part can be further split recursively. This procedure stops when all elements are separated. Algorithm 1 shows this procedure as pseudocode.

In the extracted layout trees, the internal nodes represent arrangements of elements, including horizontal arrangements, vertical arrangements, and stacked arrangements. The leaf nodes represent graphical layout elements. The node types, *i.e.*, the arrangement or element types, and the bounding box geometries are stored in the nodes. To better capture the arrangement patterns, we store the relative positions and sizes of the bounding boxes in the nodes, instead of absolute ones. Please see Figure 3 for an illustration. Specifically, for an internal node and its children in a layout, we first normalize the children's geometries with their parent's geometry. Then the first child's position is relative to the parent, and the other children's positions are relative to their left neighbored siblings. The root node stores the relative geometry of the layout regarding a fixed-geometry canvas.

3.2 Generative model for structured grid layouts

The encoders in our generative model map a structured grid layout to a latent feature code $f \in \mathbb{R}^N$. In our experiments, we set N as 256. The encoding is achieved in a recursive bottom-up manner. The decoders convert a latent feature code to a structured grid layout reversely. Please see Figure 4 for an illustration. Below we describe the encoders/decoders in detail.

Geometry encoder/decoder. The geometry encoder is used to map the relative geometries of a node i in a layout tree to a geometry feature code $f_i^{\mathrm{g}} \in \mathbb{R}^N$:

$$f_i^{\mathrm{g}} = e^{\mathrm{g}}([x_i, y_i, w_i, h_i]), \tag{1}$$

where the input is the node i's geometry vector obtained by concatenating its relative position and size. The geometry decoder converts a geometry feature code f_i^{g} to a node i's relative geometries:

$$[x_i, y_i, w_i, h_i] = d^{\mathrm{g}}(f_i^{\mathrm{g}}). \tag{2}$$

We employ single-layer perceptrons (SLPs) for the geometry encoder/decoder since they are sufficient for the tasks.

Label encoder/decoder. The label encoder is used to map the semantic label of a leaf node i in a layout tree to a label feature code $f_i^{\mathrm{l}} \in \mathbb{R}^N$:

$$f_i^{\mathrm{l}} = e^{\mathrm{l}}(l_i), \tag{3}$$

where the input is the leaf node i's semantic label represented as a one-hot vector. The label decoder converts a label feature code f_i^{l} to a leaf node i's semantic label:

$$l_i = d^{\mathrm{l}}(f_i^{\mathrm{l}}). \tag{4}$$

The networks for the label encoder/decoder are SLPs.

Element encoder/decoder. The element encoder combines the geometry feature code f_i^{g} and the label feature code f_i^{l} of a leaf node i into a node feature code $f_i \in \mathbb{R}^N$:

$$f_i = e^{\mathrm{e}}([f_i^{\mathrm{g}}, f_i^{\mathrm{l}}]), \tag{5}$$

where the input is a vector obtained by concatenating f_i^{g} and f_i^{l}. The element decoder does the reverse task:

$$[f_i^{\mathrm{g}}, f_i^{\mathrm{l}}] = d^{\mathrm{e}}(f_i). \tag{6}$$

The networks for the element encoder/decoder are SLPs.

Arrangement encoders/decoders. We consider three types of arrangement encoders/decoders, *i.e.*, horizontal arrangement encoder/decoder, vertical arrangement encoder/decoder, and stacked arrangement encoder/decoder. The networks for these encoders/decoders are all MLPs with one hidden layer. The encoders/decoders are distinguished since the nodes' relative geometries in different arrangements are defined in different ways (see Figure 3). Given an internal node j and its child nodes $\{i_1, i_2, ..., i_K\}$, the encoders combine the child nodes' feature codes $\{f_{i_1}, f_{i_2}, ..., f_{i_K}\}$ and the internal node j's geometry feature code f_j^{g} into a new node feature code $f_j \in \mathbb{R}^N$:

$$f_j = e^*([f_{i_1}, f_{i_2}, ..., f_{i_K}, f_j^{\mathrm{g}}]), \tag{7}$$

where $* \in \{\mathrm{h}, \mathrm{v}, \mathrm{s}\}$, corresponding to the horizontal, vertical, or stacked arrangements, respectively. K is the maximum children number that is dataset-dependent. If an internal node has K' children, where $K' < K$, we append zero feature codes to obtain a vector $[f_{i_1}, ..., f_{i_{K'}}, \mathbf{0}, ..., \mathbf{0}, f_j^{\mathrm{g}}]$ whose length is still $(K+1)N$.

The decoders convert a node feature code f_j into its child nodes' feature codes and its geometry feature code. Before applying the decoders, we need to first determine which decoder should be selected. We thus design an auxiliary node type classifier c^{n} (to be described latter) that predicts the node j's type. According to the output of this classifier, an appropriate decoder is applied to f_j:

$$[f_{i_1}, f_{i_2}, ..., f_{i_K}, f_j^{\mathrm{g}}] = d^*(f_j), \tag{8}$$

where $* \in \{\mathrm{h}, \mathrm{v}, \mathrm{s}\}$. Note that f_j may also be a leaf node's feature code. In this case, this code is fed to the element decoder. After applying an arrangement decoder, the obtained long feature vector is then split into several new node feature codes and a geometry feature code. Since the number of child nodes of an internal node is not fixed, we introduce a child existence classifier c^{c} to discard the invalid children.

Auxiliary classifiers. We have two auxiliary classifiers in our model, *i.e.*, the node type classifier and the child existence classifier. The node type classifier c^{n} is an MLP with one hidden layer. It takes as input a node feature code and outputs a vector indicating the node type. The child existence classifier c^{c} is an SLP. It takes a child feature code as input and outputs a value to indicate the validity of this child.

3.3 Training

We train our model on structured layout datasets. The goal is to train the encoders and decoders so that they can perform a reversible mapping between a structured layout and a feature code. Given structured layouts, we recursively apply appropriate encoders at the nodes of the corresponding layout trees until reaching the root nodes. The feature codes of the roots are approximated to a Gaussian distribution by the VAE. We then reverse the process by feeding randomly sampled feature codes to the decoders. Finally we can obtain structured layouts in the form of layout trees.

Loss. We adopt several losses in our training procedure. The total training loss is:

$$\mathcal{L} = \lambda_{\mathrm{g}}\mathcal{L}_{\mathrm{g}} + \lambda_{\mathrm{t}}\mathcal{L}_{\mathrm{t}} + \lambda_{\mathrm{l}}\mathcal{L}_{\mathrm{l}} + \lambda_{\mathrm{e}}\mathcal{L}_{\mathrm{e}} + \lambda_{\mathrm{KL}}\mathcal{L}_{\mathrm{KL}}, \tag{9}$$

where $(\lambda_{\mathrm{g}}, \lambda_{\mathrm{t}}, \lambda_{\mathrm{l}}, \lambda_{\mathrm{e}}, \lambda_{\mathrm{KL}})$ is set as $(1, 0.3, 0.3, 0.4, 0.004)$ in our experiments. $\mathcal{L}_{\mathrm{g}}$ is the geometry loss formulated as the sum of squared difference between the nodes' relative geometries in the input and output layout trees. $\mathcal{L}_{\mathrm{t}}$ is the node type loss formulated as a cross-entropy loss between the nodes' types in the input and output layout trees. $\mathcal{L}_{\mathrm{l}}$ is the semantic label loss formulated as a cross-entropy loss between the leaf nodes' semantic labels in the input and output layout trees. $\mathcal{L}_{\mathrm{e}}$ is the child existence loss formulated as a binary cross-entropy with logits loss between the child existence values in the input and output layout trees. $\mathcal{L}_{\mathrm{KL}}$ is the KL-divergence loss for approximating the space of all root node feature codes.

Other details. We implement our *GTLayout* in PyTorch. We use Adam optimizer [22] with an initial learning rate of 10^{-3} reduced by a factor of 0.9. The batch size is 128.

4 Evaluation

Our method constructs a structure-aware layout space in which a latent code represents a structured layout. With this space, our method can generate novel structured layouts by decoding randomly sampled codes. We also demonstrate that this layout space is an appropriate representation of structured layouts by faithfully reconstructing them with their latent codes. In addition, this layout space enables interpolation between given structured layouts. In this section, we evaluate our method in the tasks of layout generation, layout reconstruction, and layout interpolation respectively. We also conducted an ablation study to verify the design of our method.

Baselines. Many existing learning-based works have already achieved novel layout generation with given layout datasets. As discussed in Section 2, these works adopt different learning techniques, *e.g.*, GAN, VAE, Transformer, GNN, Diffusion model, *etc*. Only a few of them, *e.g.*, LayoutGAN++ [21], VTN [1], and READ [35], have explicitly constructed latent spaces to represent layouts. With their constructed layout spaces, they can perform layout generation, layout reconstruction, and layout interpolation similarly to our method. In contrast, other methods that fail to construct such spaces can only accomplish parts of these three tasks using different strategies. We thus compare our method with LayoutGAN++, VTN, and READ in the tasks of layout generation, reconstruction, and interpolation to demonstrate the superiority of our method and its constructed layout space.

Datasets. The evaluation is conducted on several public grid layout datasets, including Magazine [48] which includes $3,919$ layouts with 5 labels, PubLayNet [49] which includes 330K layouts with 5 labels, and RICO [6] which includes 66K layouts with 27 labels. Due to memory limit, we remove the layouts that contain excessive elements and labels as existing works [16, 21] did. We also remove the trivial layouts that contain less than 3 elements. After this filtering, we have the following datasets: Magazine

Datasets		Layouts (#)	Labels (#)	Elements (#)
Magazine	0.5K	500	4	30
	1.0K	1,000	4	30
	1.9K	1,929	4	30
	2.5K	2,705	4	30
PubLayNet	0.5K	500	5	24
	5K	5,000	5	27
	40K	40,000	5	32
	297K	297,066	5	48
RICO	0.5K	500	14	10
	2K	2,000	17	13
	10K	10,000	21	66
	46K	46,086	23	74

Table 1. The statistics, including the number of layouts, the number of labels, and the maximum number of elements per layout, of the constructed datasets. These datasets are used in the experiments included in the paper.

($2,705$ layouts, 4 labels, up to 30 elements per layout), PubLayNet ($297,066$ layouts, 5 labels, up to 48 elements per layout), and RICO ($46,086$ layouts, 23 labels, up to 74 elements per layout). To exhaustively examine how the compared methods are affected by the scales of the datasets, we further construct a series of datasets by gradually removing the layouts in the original datasets. Table 1 shows the statistics of the constructed datasets that are used in the experiments included in the paper. The training/testing ratio for all these datasets is $9:1$.

4.1 Layout generation

Since all the compared methods can construct layout spaces in which a latent code represents a layout, the layout generation task is achieved by decoding randomly sampled codes. It is worth noting that, besides a latent code, LayoutGAN++ also needs a list of labels as input to generate a layout. We adopt the same strategy as described in [21] to get such a list, *i.e.*, randomly selecting a layout in the testing set and using this layout's label list as input.

Evaluation metrics. We evaluate the generated layouts in three aspects. The first is the arrangement quality of the generated layouts. We use the overlap score (*Overlap*) and the alignment score (*Align*) for the evaluation. Since *Overlap* and *Align* have different definitions in existing works, we reiterate our adopted definitions as below.

Overlap is defined on a layout $\mathcal{T}$ as:

$$\mathit{Overlap}(\mathcal{T}) = \frac{1}{|\mathcal{T}|(|\mathcal{T}|-1)} \sum_{i\in\mathcal{T}} \sum_{j\in\mathcal{T}, j\neq i} \frac{A(b_i \cap b_j)}{A(b_i \cup b_j)}, \tag{10}$$

where i and j are elements in the layout $\mathcal{T}$. b_i is the bounding box of the element i. $A(b_i)$ means the area of the bounding box b_i.

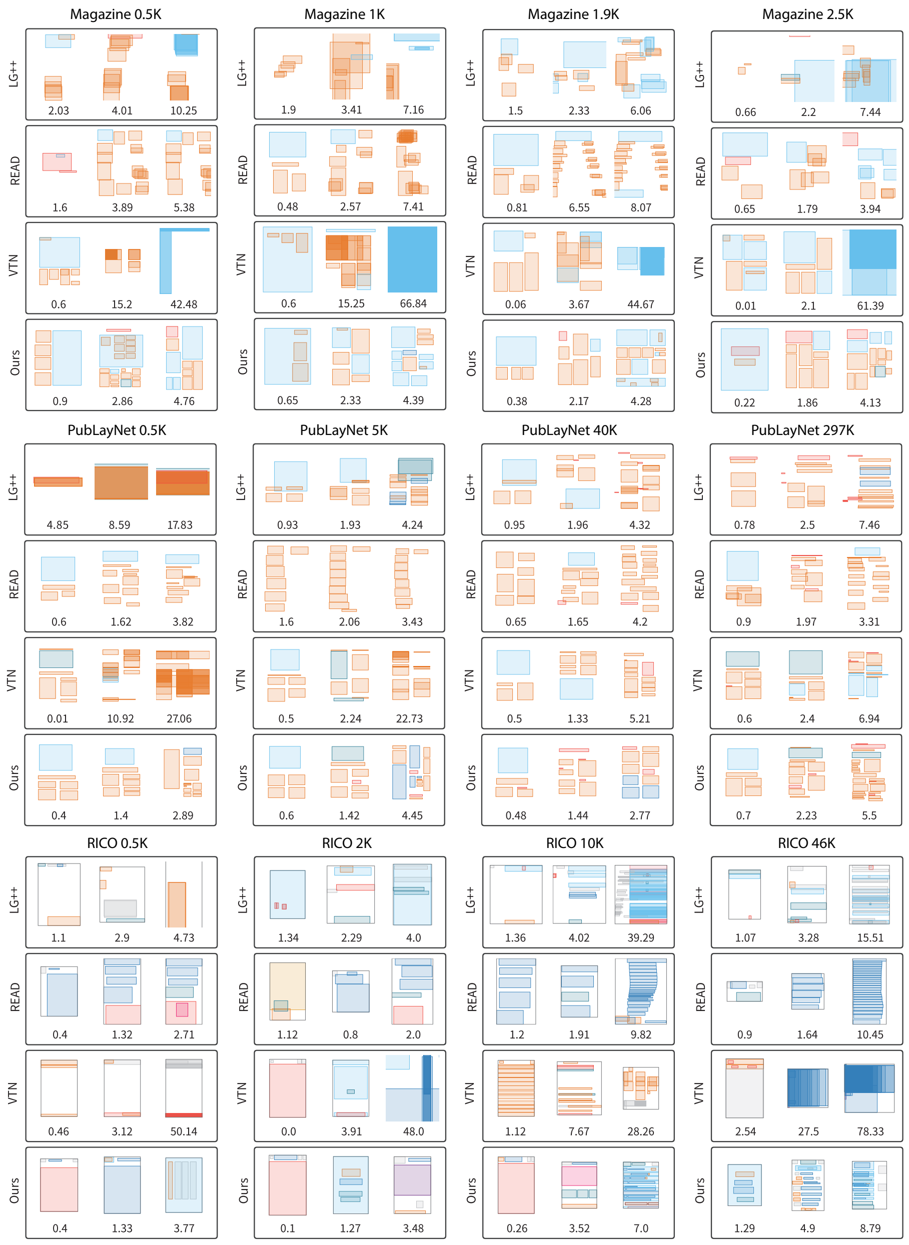

Fig. 5. Layout generation achieved by the compared methods. For each method and dataset, we select three representative layouts according to the layout similarity scores (displayed under each layout). Our method is stable in producing high-quality layouts across all datasets. In contrast, the other methods may produce low-quality layouts.

		Magazine				PubLayNet				RICO			
		0.5K	1.0K	1.9K	2.5K	0.5K	5K	40K	297K	0.5K	2K	10K	46K
Align ↓	LG++	0.057	0.066	0.080	0.075	$\underline{0.011}$	0.034	0.034	0.034	0.034	0.037	0.031	0.036
	READ	0.068	0.068	0.077	0.056	0.061	0.039	0.032	0.033	0.066	0.057	0.038	0.039
	VTN	**0.009**	**0.006**	**0.026**	**0.024**	0.013	**0.004**	**0.004**	**0.003**	**0.007**	$\underline{0.033}$	$\underline{0.027}$	**0.006**
	Ours	$\underline{0.027}$	$\underline{0.028}$	$\underline{0.029}$	$\underline{0.029}$	**0.010**	$\underline{0.011}$	$\underline{0.011}$	$\underline{0.017}$	$\underline{0.014}$	**0.018**	**0.018**	$\underline{0.016}$
Overlap ↓	LG++	0.090	0.067	0.022	$\underline{0.042}$	0.421	0.008	$\underline{0.004}$	0.009	0.019	0.019	0.011	0.013
	READ	$\underline{0.016}$	$\underline{0.018}$	$\underline{0.013}$	0.084	$\underline{0.004}$	$\underline{0.001}$	0.007	0.007	**0.007**	$\underline{0.007}$	$\underline{0.008}$	$\underline{0.007}$
	VTN	0.267	0.291	0.154	0.103	0.117	0.089	0.044	$\underline{10^{-4}}$	0.227	0.127	0.079	0.105
	Ours	**0.002**	**0.002**	**0.002**	**0.003**	$\mathbf{10^{-4}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-4}}$	$\mathbf{10^{-4}}$	$\underline{0.010}$	**0.006**	**0.005**	**0.006**
W label ↓	LG++	$\underline{0.053}$	$\underline{0.071}$	$\underline{0.047}$	$\underline{0.040}$	**0.070**	$\underline{0.034}$	**0.013**	0.096	**0.231**	**0.192**	$\underline{0.451}$	**0.418**
	READ	0.338	0.179	0.310	0.396	0.241	0.350	$\underline{0.069}$	**0.029**	0.864	0.689	1.014	1.081
	VTN	0.058	0.080	0.097	0.057	$\underline{0.099}$	0.091	0.093	$\underline{0.041}$	1.243	0.408	7.906	12.779
	Ours	**0.044**	**0.021**	**0.019**	**0.030**	0.188	**0.022**	0.097	0.115	$\underline{0.336}$	$\underline{0.275}$	**0.143**	$\underline{0.608}$
W bbox ↓	LG++	0.162	0.153	$\underline{0.087}$	0.094	0.469	0.079	0.073	0.075	0.142	0.122	0.107	$\underline{0.099}$
	READ	$\underline{0.083}$	$\underline{0.068}$	0.099	0.132	**0.040**	$\underline{0.062}$	$\underline{0.053}$	0.051	$\underline{0.098}$	$\underline{0.087}$	$\underline{0.053}$	**0.051**
	VTN	0.171	0.232	0.129	$\underline{0.083}$	0.082	0.080	0.075	**0.006**	0.194	0.155	0.343	0.442
	Ours	**0.037**	**0.027**	**0.026**	**0.021**	$\underline{0.052}$	**0.014**	**0.024**	$\underline{0.022}$	**0.071**	**0.039**	**0.023**	0.117
LaySim ↓	LG++	5.544	3.456	$\underline{2.974}$	$\underline{2.804}$	9.475	$\underline{2.175}$	**2.188**	3.569	$\underline{1.850}$	$\underline{1.862}$	6.367	**4.877**
	READ	$\underline{4.200}$	$\underline{3.235}$	6.012	9.280	$\underline{2.377}$	3.088	2.557	$\underline{2.550}$	2.154	1.970	**5.426**	$\underline{5.357}$
	VTN	34.470	38.625	17.327	9.607	25.760	8.875	$\underline{2.195}$	**2.381**	20.794	9.237	7.696	23.464
	Ours	**2.839**	**2.534**	**2.641**	**2.431**	**1.400**	**2.054**	2.429	3.526	**1.206**	**1.609**	$\underline{5.848}$	6.369
LayDiv ↑	LG++	$\underline{2.557}$	$\underline{1.838}$	$\underline{2.460}$	1.692	$\underline{1.257}$	$\underline{1.172}$	1.480	1.961	0.166	$\underline{1.198}$	$\underline{4.130}$	3.659
	READ	0.860	1.411	0.957	$\underline{1.796}$	1.176	0.392	1.814	1.928	**0.862**	0.946	2.506	2.327
	VTN	1.910	1.065	1.781	1.650	**1.473**	1.065	$\underline{1.959}$	$\underline{2.466}$	$\underline{0.752}$	0.979	1.597	$\underline{5.055}$
	Ours	**2.795**	**2.400**	**2.528**	**2.311**	0.652	**1.923**	**2.115**	**3.213**	0.684	**1.426**	**5.744**	**5.109**

Table 2. Quantitative comparisons on the layout generation task between the compared methods. In most metrics, our method outperforms the other methods. The advantage of our method is more significant for the datasets that contain a small number of layouts. The first and second best scores are highlighted in bold and underline.

Align is defined on a layout $\mathcal{T}$ as:

$$Align(\mathcal{T}) = \frac{1}{|\mathcal{T}|}\sum_{i\in\mathcal{T}}(\min_j \frac{D_{\mathrm{h}}(b_i, b_j)}{h_{\mathcal{T}}} + \min_j \frac{D_{\mathrm{v}}(b_i, b_j)}{w_{\mathcal{T}}}), \tag{11}$$

where $D_{\mathrm{h}}(b_i, b_j)$ and $D_{\mathrm{v}}(b_i, b_j)$ are defined as the minimal horizontal/vertical alignment distances between b_i and b_j. $w_{\mathcal{T}}$ and $h_{\mathcal{T}}$ are the width and height of the layout and are used to normalize the distance. In our comparison, we consider six alignments, *i.e.*, horizontal alignments: top, vertical center, bottom; vertical alignments: left, horizontal center, and right.

The second aspect of the evaluation is the similarity between the generated and existing layouts. We adopt the Wasserstein distance [1] to measure the distribution similarity between these two layout sets. Specifically, we compute the Wasserstein distances between the generated and testing layouts for the label distribution (*W label*) and the bounding box distribution (*W bbox*). Besides the distribution similarity, we want to measure the layout appearance similarity between the generated and existing layouts. Intuitively, the generated layouts should possess similar appearances to the existing layouts. This layout appearance similarity, termed *LaySim*, can be defined as:

$$\textit{LaySim}(\mathcal{S}, \mathcal{S}') = \frac{1}{|\mathcal{S}|} \sum_{\mathcal{T} \in \mathcal{S}} \min_{\mathcal{T}' \in \mathcal{S}'} M(\mathcal{T}, \mathcal{T}'), \tag{12}$$

where $\mathcal{S}$ and $\mathcal{S}'$ are the generated and testing layout sets. $M(\mathcal{T}, \mathcal{T}')$ is the similarity measure between a pair of layouts $\mathcal{T}$ and $\mathcal{T}'$. Several works [31, 35, 36, 43] have investigated the similarity measures between layouts. Patil *et al.* [35] introduced a combinatorial layout similarity measure called *DocSim*. This measure is effective for finding the nearest neighbors of a given layout, but the computed scores are inconsistent across different layouts. For example, the self-similarity scores of different layouts may vary in a large range; the similarity scores of two distinct pairs of layouts can not be compared directly. Xu *et al.* [43] introduced another combinatorial layout similarity measure that produces consistent similarity scores. However, it requires layouts to possess hierarchical structures. LayoutGMN [36] and GCN-CNN [31] are two learning-based methods that can predict the similarity between two layouts. They require a heavy load of training before using them to measure specific layouts' similarities. For convenient computation, we combine the combinatorial methods introduced in [43] and [35] to define the layout similarity measure $M(\mathcal{T}, \mathcal{T}')$. Specifically, we treat layouts as sets of elements [35] and use the Hungarian algorithm [25] to compute the optimal matching cost as the layout similarity measure between a pair of layouts. The element matching cost follows the definition in [43] and an element can correspond to a void.

The third aspect of the evaluation is the diversity of the generated layouts. This can be reflected by the similarities among the generated layouts, *i.e.*, the average similarity score among all pairs of layouts in the generated layout set $\mathcal{S}$. We term the layout diversity *LayDiv* and define it as:

$$\textit{LayDiv}(\mathcal{S}) = \frac{1}{|\mathcal{S}|(|\mathcal{S}|-1)} \sum_{\mathcal{T} \in \mathcal{S}} \sum_{\mathcal{T}' \in \mathcal{S}, \mathcal{T}' \neq \mathcal{T}} M(\mathcal{T}, \mathcal{T}'). \tag{13}$$

Results. For each method and each dataset, we randomly generate $1,000$ layouts for comparison. As discussed earlier, the generation is achieved by decoding randomly sampled codes. Table 2 shows the quantitative comparisons of the methods on different datasets. Figure 5 shows some representative layouts selected from the generated layouts. For each dataset and method, we select three layouts according to the similarity measures between the generated and existing layouts, *i.e.*, the most, the least, and the medially similar ones:

$$\begin{aligned} \mathcal{T}_{\text{most}} &= \arg\min_{\mathcal{T} \in \mathcal{S}} (\min_{\mathcal{T}' \in \mathcal{S}'} M(\mathcal{T}, \mathcal{T}')), \\ \mathcal{T}_{\text{least}} &= \arg\max_{\mathcal{T} \in \mathcal{S}} (\min_{\mathcal{T}' \in \mathcal{S}'} M(\mathcal{T}, \mathcal{T}')), \\ \mathcal{T}_{\text{median}} &= \arg\operatorname*{median}_{\mathcal{T} \in \mathcal{S}} (\min_{\mathcal{T}' \in \mathcal{S}'} M(\mathcal{T}, \mathcal{T}')), \end{aligned} \tag{14}$$

where $\mathcal{S}$ and $\mathcal{S}'$ are the generated and testing layout sets. From these quantitive and qualitative results, we have the following findings. First, in most metrics, our method

		Magazine				PubLayNet				RICO			
		0.5K	1.0K	1.9K	2.5K	0.5K	5K	40K	297K	0.5K	2K	10K	46K
CD ↓	READ	0.057	0.076	0.082	0.194	0.047	0.072	0.022	0.023	0.131	0.052	0.093	0.082
	VTN	0.084	0.149	0.156	0.111	0.043	0.039	0.033	**0.013**	0.379	0.230	0.460	1.948
	Ours	**0.023**	**0.031**	**0.019**	**0.022**	**0.030**	**0.017**	**0.014**	0.021	**0.037**	**0.024**	**0.014**	**0.034**
IoU ↑	READ	0.237	0.251	0.195	0.250	0.281	0.222	0.422	0.416	0.207	0.410	0.266	0.267
	VTN	0.172	0.201	0.232	0.318	0.353	0.318	0.374	**0.479**	0.372	0.484	0.165	0.115
	Ours	**0.455**	**0.480**	**0.527**	**0.537**	**0.382**	**0.416**	**0.446**	0.356	**0.422**	**0.548**	**0.366**	**0.338**

Table 3. Quantitative comparisons on the layout reconstruction task between the compared methods, in terms of the Chamfer distance (*CD*) and the Intersecion-over-Union (*IoU*). Our method achieves the best performance for almost all the datasets. The best scores are highlighted in bold.

outperforms the other methods. The advantage of our method is more significant for the datasets that contain a small number of layouts. For the datasets that contain a large number of layouts (*e.g.*, PubLayNet 297K), our method outperforms or is comparable to the other methods in most metrics. For *Overlap*, our method outperforms the other methods on almost all the datasets. This should be attributed to our recursive structure generation procedure and the design of relative geometry. For *Align*, our method is slightly behind VTN. VTN adopts discrete coordinates to represent element geometry, and thus greatly improves its performance on alignment. According to *W label* and *W bbox*, our method also achieves comparable performance in these two metrics. Since these two metrics only consider the distribution of labels and box geometries, we introduce *LaySim* to measure the appearance similarity. For *LaySim*, our method also achieves the best performance in most datasets. According to *LaySim* and *LayDiv*, we can conclude that our method produces more diverse layouts with higher qualities, especially for small datasets. In terms of qualitative comparison, all the methods can produce reasonable layouts, *e.g.*, the layouts that are most similar to the testing layout sets. However, our method is more stable than the other methods in producing high-quality layouts. As illustrated in Figure 5, all the representative layouts generated by our method, including the ones that are most, medially, and least similar to the testing layout sets, are of high quality. In contrast, some of the representative layouts generated by the other methods, including the ones that are medially or least similar to the testing layout sets, are of low quality, especially for the datasets that contain a small number of layouts (*e.g.*, Magzine 0.5K). READ also adopts RvNN as the network architecture, therefore its performance is better when training with small datasets. For large datasets, its performance degrades since the layouts in these datasets contain excessive elements, leading to over-deep binary layout trees, and resulting in the collapse of the model. VTN archives comparable performance to our method, especially for large datasets, indicating the advantage of the transformer architecture. Due to the memory limit, the training of VTN with RICO 46K dataset fails, resulting in low-quality results. In the supplemental material, we provided more results, including the generated layouts' nearest neighbors selected from the testing sets. These additional results confirm that our method can produce novel layouts that are visually similar to the existing ones.

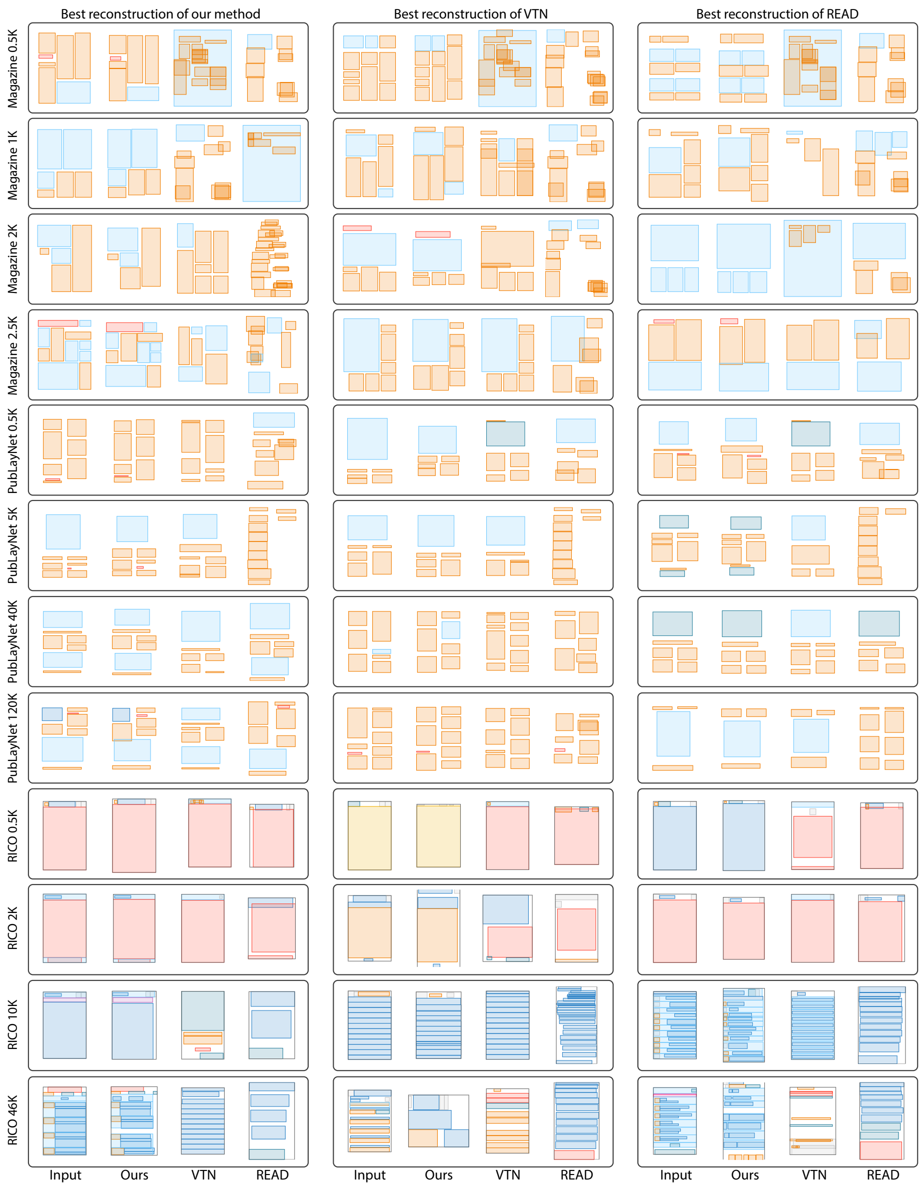

Fig. 6. The layout reconstruction achieved by the compared methods. For each dataset, we obtain three groups of layouts, according to the best reconstruction of each method. Our method outperforms the other methods in this task.

4.2 Layout reconstruction

In this task, the reconstruction is achieved by first encoding a layout to obtain a latent code and then decoding this code to reconstruct the layout. The reconstruction quality indicates whether the latent code is an appropriate representation of the layout. Since LayoutGAN++ does not have an encoder, it can not perform this reconstruction task. Therefore, we compare our method with VTN and READ.

Evaluation metrics. We adopt the Chamfer distance (*CD*) [17] and Intersection-over-Union score (*IoU*) [31, 35] between the ground-truth layouts and the reconstructed layouts as the evaluation metrics.

Results. For each testing dataset, we randomly select and reconstruct $1,000$ layouts (or all layouts for small datasets) using the compared methods. Table 3 shows the quantitative comparisons between the compared methods. According to *CD* and *IoU*, our method achieves the best performance on layout reconstruction for almost all the datasets. Figure 6 shows some representative reconstruction results obtained by the compared methods. For each dataset, we obtain three groups of layouts, each of which contains an input layout and three layouts reconstructed by the compared methods. The three input layouts are selected according to the *CD* score: for each method, we select the input layout with the best *CD* score after reconstruction. This strategy avoids the bias of manual selection. This qualitative comparison also confirmed that the reconstructed layouts obtained by our method are more visually similar to the input layouts compared with the other methods. VTN also achieves satisfactory results when training with large datasets (*e.g.*, PubLayNet 120K), but its performance degrades significantly for small datasets (*e.g.*, Magazine 0.5K). These quantitative and qualitative comparisons confirm that the latent code computed by our method is an appropriate neural representation of layouts.

4.3 Layout interpolation

Layout interpolation is achieved by interpolating the latent codes of existing layouts and then decoding the interpolated codes. We compare our method with VTN and READ in this task.

Results. Figure 7 gives a few interpolation examples. These results are obtained by training the compared methods with Magazine 2.5K and PubLayNet 297K respectively. We do not include RICO in this task since the variation of the layouts in this dataset is overly high and all the methods can not achieve satisfactory interpolation.

This qualitative comparison shows that all the methods can produce satisfactory layouts by interpolation. However, only our method achieves smooth and reasonable structure blending, since it exploits the structural information when embedding layouts into the latent space. These results confirm that the latent space constructed by our method is structure-aware. Please refer to the supplemental material for the animation of the layout interpolation.

5 Conclusion

In this paper, we have presented *GTLayout*, a novel RvNN-based generative model for structured grid layout generation. We adopt general trees as the structural repre-

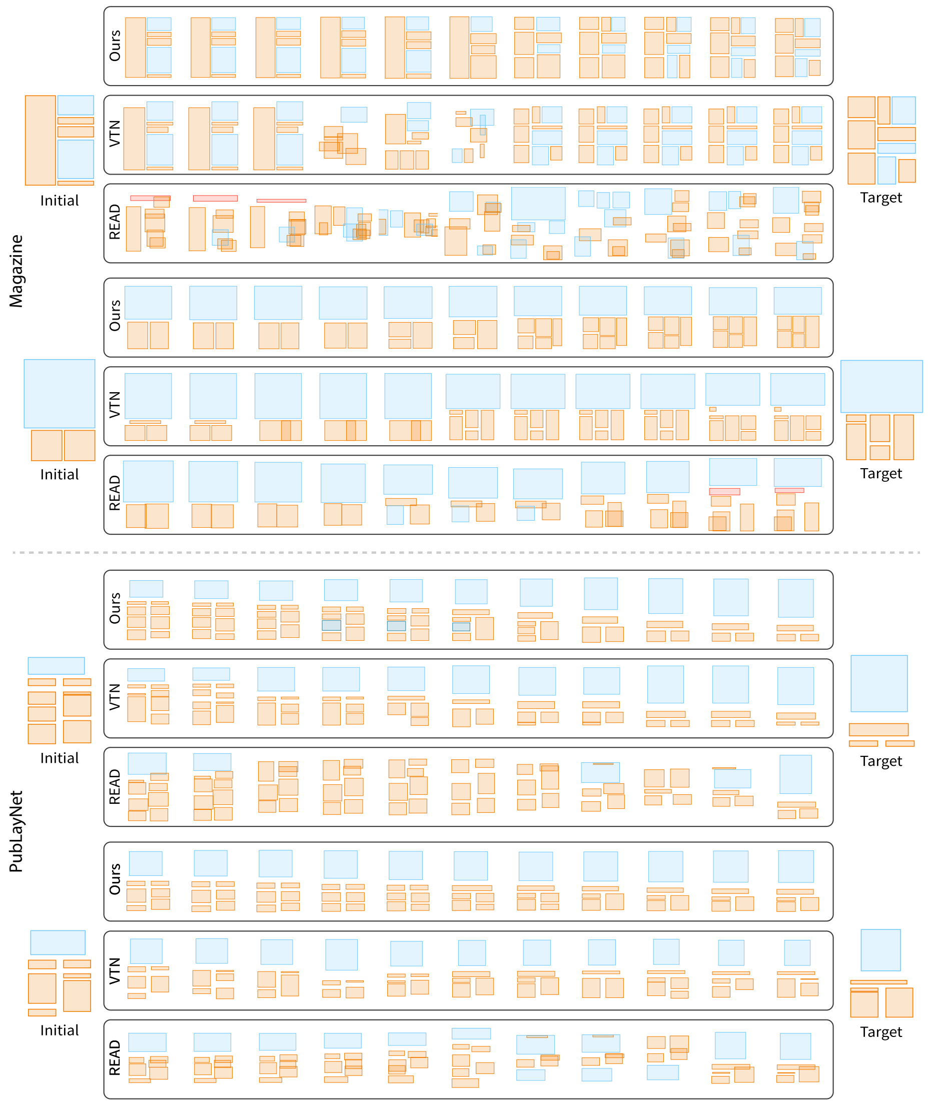

Fig. 7. The layout interpolation achieved by the compared methods. All the methods can produce satisfactory layouts by interpolation. However, only our method achieves smooth and reasonable structure blending. It confirms that the latent space constructed by our method is structure-aware.

sentation of structured grid layouts and use relative geometry to depict the spatial relations between elements. Our model is trained in a recursive bottom-up manner. We have designed several encoders and decoders according to the arrangements existing in structured gird layouts. These encoders can successfully map structured grid layouts to a structure-aware latent space. Extensive evaluations show that our method outperforms several baselines quantitatively and qualitatively in the tasks of layout reconstruction and generation, especially for small datasets. We have also demonstrated the advantage of the structure-aware latent space constructed by our method via the task of structured layout blending. To the best of our knowledge, *GTLayout* is the first learning-based method that achieves structured layout blending. We believe that the constructed structure-aware layout space has more potential applications, *e.g.*, exploiting user-provided constraints for layout generation.

Our method has some limitations. First, since our method aims to structured layout generation, it is less reliable to handle unstructured layouts. Second, our method does not consider more advanced structures in layouts, for example, symmetry or semantic groupings, since these structural information may break the tree structures. Lastly, similar to other RvNN-based methods, the training efficiency of our method is not high. It takes around 20 hours to train a model for 200 epochs on a dataset composed of $10,000$ layouts.

In the future, we plan to further explore the problem of structured layout generation. It would be interesting to resolve the limitations of our method by introducing other representations of structured layouts. We have adopted RvNN for the structured layout generation. It would be promising to exploit other advanced learning techniques, *e.g.*, Transformers [41] or Graph Neural Network [38], to solve the structured layout generation problem, or even extend them to other structured data generation problems.

Acknowledgments. This work was supported in parts by NSFC (62072316, U21B2023), NSF of Guangdong Province (2023A1515011297), DEGP Innovation Team (2022KCXTD025), Shenzhen Science and Technology Program (KQTD20210811090044003), and Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Arroyo, D.M., Postels, J., Tombari, F.: Variational transformer networks for layout generation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 13642–13652 (2021)
2. Baudisch, P., Cutrell, E., Hinckley, K., Eversole, A.: Snap-and-go: helping users align objects without the modality of traditional snapping. In: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 301–310 (2005)
3. Bier, E.A., Stone, M.C.: Snap-dragging. ACM SIGGRAPH Computer Graphics **20**(4), 233–240 (1986)
4. Chai, S., Zhuang, L., Yan, F.: Layoutdm: Transformer-based diffusion model for layout generation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 18349–18358 (2023)
5. Dayama, N.R., Todi, K., Saarelainen, T., Oulasvirta, A.: Grids: Interactive layout design with integer programming. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–13 (2020)
6. Deka, B., Huang, Z., Franzen, C., Hibschman, J., Afergan, D., Li, Y., Nichols, J., Kumar, R.: Rico: A mobile app dataset for building data-driven design applications. In: Proceedings of

the 30th Annual ACM Symposium on User Interface Software and Technology. pp. 845–854 (2017)

7. Dixon, M., Leventhal, D., Fogarty, J.: Content and hierarchy in pixel-based methods for reverse engineering interface structure. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 969–978 (2011)
8. Frisch, M., Kleinau, S., Langner, R., Dachselt, R.: Grids & guides: multi-touch layout and alignment tools. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. pp. 1615–1618 (2011)
9. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 27. Curran Associates, Inc. (2014), https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf
10. Gupta, K., Lazarow, J., Achille, A., Davis, L.S., Mahadevan, V., Shrivastava, A.: Layouttransformer: Layout generation and completion with self-attention. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 1004–1014 (2021)
11. Ho, J., Jain, A., Abbeel, P.: Denoising diffusion probabilistic models. Advances in Neural Information Processing Systems **33**, 6840–6851 (2020)
12. Hui, M., Zhang, Z., Zhang, X., Xie, W., Wang, Y., Lu, Y.: Unifying layout generation with a decoupled diffusion model. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 1942–1951 (2023)
13. Inoue, N., Kikuchi, K., Simo-Serra, E., Otani, M., Yamaguchi, K.: LayoutDM: Discrete Diffusion Model for Controllable Layout Generation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 10167–10176 (2023)
14. Jiang, Y., Du, R., Lutteroth, C., Stuerzlinger, W.: Orc layout: Adaptive gui layout with or-constraints. In: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems. pp. 1–12 (2019)
15. Jiang, Y., Stuerzlinger, W., Lutteroth, C.: Reverseorc: Reverse engineering of resizable user interface layouts with or-constraints. In: Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. pp. 1–18 (2021)
16. Jiang, Z., Guo, J., Sun, S., Deng, H., Wu, Z., Mijovic, V., Yang, Z.J., Lou, J.G., Zhang, D.: Layoutformer++: Conditional graphic layout generation via constraint serialization and decoding space restriction. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 18403–18412 (2023)
17. Jiang, Z., Sun, S., Zhu, J., Lou, J.G., Zhang, D.: Coarse-to-fine generative modeling for graphic layouts. In: AAAI'22 (February 2022)
18. Jyothi, A.A., Durand, T., He, J., Sigal, L., Mori, G.: Layoutvae: Stochastic scene layout generation from a label set. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 9895–9904 (2019)
19. Kenton, J.D.M.W.C., Toutanova, L.K.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT. pp. 4171–4186 (2019)
20. Kikuchi, K., Otani, M., Yamaguchi, K., Simo-Serra, E.: Modeling visual containment for web page layout optimization. In: Computer Graphics Forum. vol. 40, pp. 33–44. Wiley Online Library (2021)
21. Kikuchi, K., Simo-Serra, E., Otani, M., Yamaguchi, K.: Constrained graphic layout generation via latent optimization. In: Proceedings of the 29th ACM International Conference on Multimedia. pp. 88–96 (2021)
22. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR

2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980
23. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. In: International Conference on Learning Representations (2013)
24. Kong, X., Jiang, L., Chang, H., Zhang, H., Hao, Y., Gong, H., Essa, I.: Blt: bidirectional layout transformer for controllable layout generation. In: Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XVII. pp. 474–490. Springer (2022)
25. Kuhn, H.W.: The hungarian method for the assignment problem. Naval research logistics quarterly **2**(1-2), 83–97 (1955)
26. Lee, H.Y., Jiang, L., Essa, I., Le, P.B., Gong, H., Yang, M.H., Yang, W.: Neural design network: Graphic layout generation with constraints. In: European Conference on Computer Vision. pp. 491–506. Springer (2020)
27. Li, J., Yang, J., Hertzmann, A., Zhang, J., Xu, T.: Layoutgan: Generating graphic layouts with wireframe discriminators. arXiv preprint arXiv:1901.06767 (2019)
28. Li, J., Xu, K., Chaudhuri, S., Yumer, E., Zhang, H., Guibas, L.: Grass: Generative recursive autoencoders for shape structures. ACM Transactions on Graphics (TOG) **36**(4), 1–14 (2017)
29. Li, M., Patil, A.G., Xu, K., Chaudhuri, S., Khan, O., Shamir, A., Tu, C., Chen, B., Cohen-Or, D., Zhang, H.: Grains: Generative recursive autoencoders for indoor scenes. ACM Transactions on Graphics (TOG) **38**(2), 1–16 (2019)
30. Lupton, E.: Thinking with type: A critical guide for designers, writers, editors, & students. Chronicle Books (2014)
31. Manandhar, D., Ruta, D., Collomosse, J.: Learning structural similarity of user interface layouts using graph networks. In: Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXII 16. pp. 730–746. Springer (2020)
32. Mo, K., Guerrero, P., Yi, L., Su, H., Wonka, P., Mitra, N., Guibas, L.J.: Structurenet: Hierarchical graph networks for 3d shape generation. arXiv preprint arXiv:1908.00575 (2019)
33. Müller-Brockmann, J.: Grid systems in graphic design: A visual communication manual for graphic designers, typographers and three dimensional designers. Arthur Niggli (1996)
34. O'Donovan, P., Agarwala, A., Hertzmann, A.: Learning layouts for single-page graphic designs. IEEE transactions on visualization and computer graphics **20**(8), 1200–1213 (2014)
35. Patil, A.G., Ben-Eliezer, O., Perel, O., Averbuch-Elor, H.: Read: Recursive autoencoders for document layout generation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. pp. 544–545 (2020)
36. Patil, A.G., Li, M., Fisher, M., Savva, M., Zhang, H.: Layoutgmn: Neural graph matching for structural layout similarity. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 11048–11057 (2021)
37. Raisamo, R., Räihä, K.J.: A new direct manipulation technique for aligning objects in drawing programs. In: Proceedings of the 9th annual ACM symposium on User interface software and technology. pp. 157–164 (1996)
38. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE Transactions on Neural Networks **20**(1), 61–80 (2008)
39. Socher, R., Lin, C.C., Manning, C., Ng, A.Y.: Parsing natural scenes and natural language with recursive neural networks. In: Proceedings of the 28th international conference on machine learning (ICML-11). pp. 129–136 (2011)
40. Swearngin, A., Wang, C., Oleson, A., Fogarty, J., Ko, A.J.: Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–13 (2020)

41. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L.u., Polosukhin, I.: Attention is all you need. In: Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc. (2017), https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
42. Xu, P., Fu, H., Tai, C.L., Igarashi, T.: Gaca: Group-aware command-based arrangement of graphic elements. In: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems. pp. 2787–2795 (2015)
43. Xu, P., Li, Y., Yang, Z., Shi, W., Fu, H., Huang, H.: Hierarchical layout blending with recursive optimal correspondence. ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA) **41**(6), 249:1–249:15 (2022)
44. Xu, P., Yan, G., Fu, H., Igarashi, T., Tai, C.L., Huang, H.: Global beautification of 2d and 3d layouts with interactive ambiguity resolution. IEEE transactions on visualization and computer graphics **27**(4), 2355–2368 (2019)
45. Yamaguchi, K.: Canvasvae: Learning to generate vector graphic documents. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 5481–5489 (2021)
46. Yang, Y.L., Wang, J., Vouga, E., Wonka, P.: Urban pattern: Layout design by hierarchical domain splitting. ACM Transactions on Graphics (TOG) **32**(6), 1–12 (2013)
47. Zeidler, C., Lutteroth, C., Sturzlinger, W., Weber, G.: The auckland layout editor: An improved gui layout specification process. In: Proceedings of the 26th annual ACM symposium on User interface software and technology. pp. 343–352 (2013)
48. Zheng, X., Qiao, X., Cao, Y., Lau, R.W.: Content-aware generative modeling of graphic design layouts. ACM Transactions on Graphics (TOG) **38**(4), 1–15 (2019)
49. Zhong, X., Tang, J., Yepes, A.J.: Publaynet: largest dataset ever for document layout analysis. In: 2019 International Conference on Document Analysis and Recognition (ICDAR). pp. 1015–1022. IEEE (2019)
50. Zhu, C., Xu, K., Chaudhuri, S., Yi, R., Zhang, H.: Scores: Shape composition with recursive substructure priors. ACM Transactions on Graphics (TOG) **37**(6), 1–14 (2018)