LayoutFormer++: Conditional Graphic Layout Generation via Constraint Serialization and Decoding Space Restriction

Zhaoyun Jiang[1*], Jiaqi Guo[2], Shizhao Sun[2], Huayu Deng[3*], Zhongkai Wu[4*],
Vuksan Mijovic[5], Zijiang James Yang[1], Jian-Guang Lou[2], Dongmei Zhang[2]
[1]Xi'an Jiaotong University, [2]Microsoft Research Asia,
[3]Shanghai Jiaotong University, [4]Beihang University, [5]Microsoft,
jzy124@stu.xjtu.edu.cn, deng_hy99@sjtu.edu.cn, 17376487@buaa.edu.cn,
zijiang@xjtu.edu.cn, {jiaqiguo, shizsu, vmijovic, jlou, dongmeiz}@microsoft.com

Abstract

Conditional graphic layout generation, which generates realistic layouts according to user constraints, is a challenging task that has not been well-studied yet. First, there is limited discussion about how to handle diverse user constraints flexibly and uniformly. Second, to make the layouts conform to user constraints, existing work often sacrifices generation quality significantly. In this work, we propose LayoutFormer++ to tackle the above problems. First, to flexibly handle diverse constraints, we propose a constraint serialization scheme, which represents different user constraints as sequences of tokens with a predefined format. Then, we formulate conditional layout generation as a sequence-to-sequence transformation, and leverage encoder-decoder framework with Transformer as the basic architecture. Furthermore, to make the layout better meet user requirements without harming quality, we propose a decoding space restriction strategy. Specifically, we prune the predicted distribution by ignoring the options that definitely violate user constraints and likely result in low-quality layouts, and make the model samples from the restricted distribution. Experiments demonstrate that LayoutFormer++ outperforms existing approaches on all the tasks in terms of both better generation quality and less constraint violation.

1. Introduction

Graphic designs greatly facilitate information communication in our daily life. During its creation, the *layout*, i.e., positions and sizes of elements, plays a critical role. To assist layout design, *conditional layout generation*, which takes user constraints as input and generates layouts as output, attracts great attention (see Figure 1). It is different from unconditional layout generation, which generates layouts freely without constraints, from at least two aspects. First, the model should be able to handle diverse user constraints, called *sufficient flexibility*. Figure 2 shows 6 typical tasks of layout generation in real-world applications including layout completion, layout refinement, layout generation conditioned on element types, element types with sizes, element relationships, or any of their combinations. Second, the model should generate layouts conforming to user requirements (i.e., constraints) as many as possible without harming quality, called *good controllability* (see Figure 1).

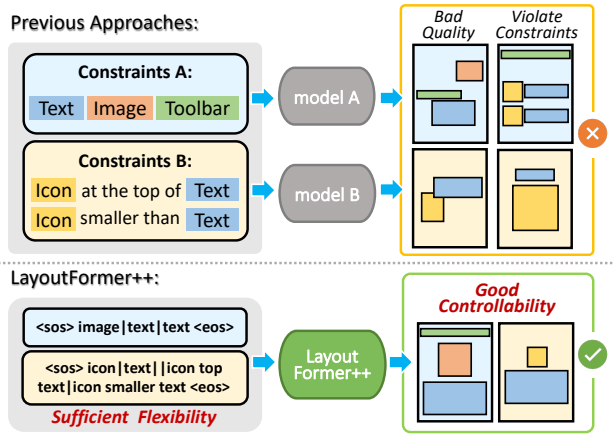


Figure 1. Comparing with previous conditional layout generation approaches, LayoutFormer++ performs better on sufficient flexibility and good controllability.

However, existing work cannot meet the above two requirements. First, there is no existing work can support all the layout generation tasks with different user constraints. Most existing approaches simply focus on tackling a single conditional layout generation task without considering whether they can be applied to other tasks. For example,

*Work done during an internship at Microsoft Research Asia.

LayoutTransformer [3] can only perform completion task and BLT [10] cannot handle element relationships. Such task-specific approaches hinder the development of solutions for the new task. Second, there are no satisfactory methods to ensure good controllability. Some approaches directly replace the values in the predicted layout with the ones specified in the user constraints [3, 10]. Another work defines a set of heuristic rules and leverages latent optimization [8]. To make the generated layouts meet user requirements, they often degrade the generation quality significantly.

In this work, we propose a unified model called *LayoutFormer++* to support the different scenarios of conditional layout generation. In *LayoutFormer++*, a set of constraints is represented as a sequence. Specifically, we use a *constraint serialization* scheme to serialize different user constraints into a sequence of tokens with a predefined format (see Figure 1). The intuition behind this design choice is as follows. First, the sequence format is widely used and effective. Its effectiveness in layout generation has also been demonstrated in recent works [3, 10]. Second, a sequence is very flexible to accommodate different constraints for layout generation. We can serialize any structured information in the user constraints as a sequence. We found although user needs are diverse, they are all about element types and five attributes including type, top coordinate, left coordinate, width and height. Thus, we can simply define a set of vocabularies to describe the attributes respectively and concatenate descriptions of different attributes and elements to construct a sequence.

Therefore, the conditional layout generation problem can be formulated as a sequence-to-sequence transformation problem. This enables us to leverage a simple yet effective encoder-decoder framework with Transformer [26] as a basic model architecture. The encoder processes the user constraints in a bidirectional way. The decoder predicts the layout sequence autoregressively, where there are multiple decoding steps and the model samples one token from the predicted distribution at each decoding step.

Furthermore, to achieve good controllability, we introduce a *decoding space restriction* strategy in the inference stage. Our key idea is to prune the infeasible options in the predicted distribution and make the model sample from the restricted distribution at each decoding step. Specifically, we leverage two kinds of information to prune the options. First, the options that definitely violate the user constraints are pruned. For example, if a user wants one image and two buttons, the option for putting one text box will not be acceptable. Second, the options with low probabilities in the predicted distribution, which will very likely result in low-quality layouts, are also pruned. As the feasible option set may be empty after pruning, we further introduce a backtracking mechanism, in which the model goes back to a certain decoding step and find a better solution. Note that the whole generation process of the proposed strategy still relies on the distribution learned from the training data. Thus, it is less likely to disturb a layout when making it better conform to user constraints.

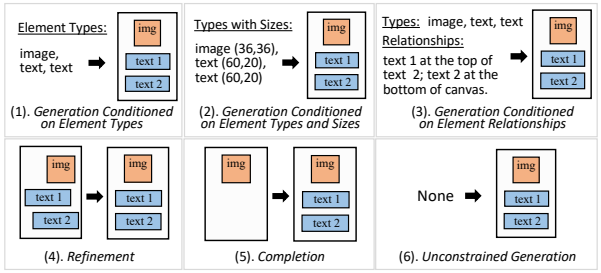


Figure 2. Typical tasks for conditional layout generation.

We conduct extensive experiments on two public datasets [15, 30] and six layout generation tasks with different user constraints, to evaluate *LayoutFormer++* and compare it with state-of-the-art approaches. Experimental results show that LayoutFormer++ can successfully tackle all six layout generation tasks that are handled separately by previous work, demonstrating that it is able to provide sufficient flexibility. Furthermore, LayoutFormer++ significantly outperforms previous approaches in terms of both better generation quality and less constraint violation, indicating that it achieves good controllability.

2. Related Work

Graphic Layout Generation. Graphic layout generation aims at generating aesthetic layouts to tackle users' needs. As users' needs are diverse in real-world applications, various tasks of layout generation are explored (see Figure 2), such as generation condition on element types [8, 10, 12], generation condition on types and sizes [10], generation condition on element relationships [8, 12], refinement [23] and completion [3]. Some other approaches focus on unconstrained generation [2, 7, 13, 27].

However, none of existing approaches can achieve sufficient flexibility to handle diverse layout generation tasks, due to the limitations in constraint modeling. Some works represent the constraints in the same format of the target layout, thus can not handle other complex constraints [3, 10, 23]. For example, LayoutTransformer [3] can only handle completion task but cannot deal with relationships between elements. Other works design the modeling formats specific to the constraint, which are hard to adapt to other tasks. For example, [12] leverages graph format to represent the relationships among the elements, which is inconvenient for the size constraints. In this work, we propose LayoutFormer++ with constraint serialization scheme, modeling different constraints into same format of the sequence, eliminating the isolation between layout generation tasks.

Existing approaches also consider different methods for constraint satisfaction. Some approaches directly reset the predicted attributes by the values specified in the constraints [3, 10]. Other approaches define the heuristic rules and cost functions to control the generation [8, 12]. For example, [8] designs cost functions for each constraint and takes the constrained generation as an optimization problem. However, these methods often decrease the generation quality significantly. For [3, 10], directly resetting the sequence will make the generated layout deviate from the distribution learned by model. For [8, 12], it is hard to define the rules and cost functions to handle the trade-off between constraint satisfaction and generation quality. Different from existing approaches, we introduce decoding space restriction strategy to prune the infeasible values during decoding process, which does not rely on sophisticated rules or cost functions, and also fits the learned distribution.

Decoding Methods. Decoding method plays an important role in autoregressive generation. The most prominent decoding methods include Greedy Search, Beam Search, Top-K sampling and Top-P sampling [5], etc. To enable the control over the output sequence, constrained decoding is first proposed in controllable text generation. [1,4,6,16,17,21] propose constrained decoding methods for hard lexical constraints where a set of tokens are required to occur in target output. [22, 28] present decoding methods for soft constraints which assign a value between 0 and 1 to indicate how constraints are satisfied. [11, 24, 25] introduce decoding methods for structural constraints which impose a rigorous structures on the output sequence. Inspired by these methods, we propose a new constrained decoding method tailored for conditional layout generation. We carefully design two pruning modules and a backtracking mechanism to improve the satisfaction of constraints while maintaining high layout quality.

3. Approach

3.1. Overview

A layout is made up of a set of elements, where each element can be described by its type c, left and top coordinate x and y, as well as width w and height h. The continuous attributes x, y, w and h are quantized, which is proven to be helpful for graphic layouts [2, 3, 23]. Thus, an element can be represented by five tokens. Following the state-of-the-art approaches, we represent a layout by concatenating all the elements' tokens, i.e., $L = \{\langle \text{sos} \rangle c_1 x_1 y_1 w_1 h_1 \dots c_N x_N y_N w_N h_N \langle \text{eos} \rangle\}$. Here, N denotes the total number of elements, $\langle \text{sos} \rangle$ and $\langle \text{eos} \rangle$ are special tokens indicating the start and end of a sequence.

In this work, we propose a new approach named LayoutFormer++ for conditional layout generation. The overall model architecture is illustrated in Figure 3a.

To achieve sufficient flexibility, different constraints are formatted uniformly into sequence through constraint serialization. The details of the constraint serialization scheme will be introduced in Section 3.2.

We leverage a Transformer encoder-decoder architecture for LayoutFormer++ based on the experience of previous works. The bidirectional encoder takes the constraint sequence as input and outputs their contextualized representations. The autoregressive decoder iteratively predicts the probabilities of future tokens. The model is trained to minimize the negative log-likelihood of layout tokens, i.e.,

$$\text{minimize} \sum_{t=1}^{N} -\log P_{\boldsymbol{\theta}}(L_t | L_{<t}, S), \tag{1}$$

where S is the input constraint sequence, L is the output layout sequence, $\boldsymbol{\theta}$ refers to the model parameters and N denotes the length of the layout sequence.

Finally, we introduce the decoding space restriction strategy with a backtracking mechanism into inference process to achieve good controllability. The details of the decoding space restriction strategy are given in Section 3.3.

3.2. Serializing Constraints

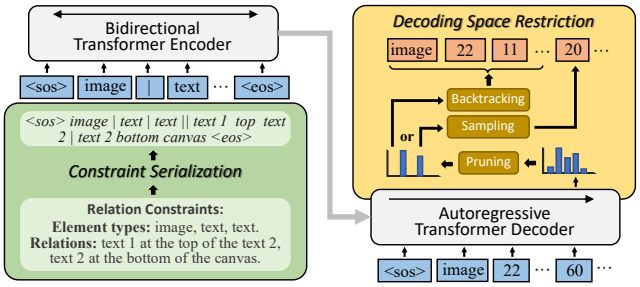
3.2.1 Principles

Instead of directly presenting concrete sequences for the constraints considered in previous work, we first introduce the principles for serializing constraints. These principles not only summarize how the existing constraints are serialized but also provide a general guidance for formulating other constraints. There are two critical questions in serializing constraints. First, how to represent each constraint in a sequence format. Second, how to combine different constraints into a complete sequence.

Constraint Representation. To present a constraint as a sequence, the basic idea is to first define a set of vocabularies and then concatenate the tokens from these vocabularies. Specifically, the constraints often fall into two categories.

First, some constraints are only related to a single element. For example, put an image on the page or the element should be 10 pixel wide. The serialization for such constraint is quite straightforward: we just define a vocabulary and then represent the constraint by a single token from the vocabulary. For example, for the constraints about element type, the vocabulary could be $\{\texttt{image}, \texttt{toolbar}, \dots, \texttt{button}\}$, and the constraint 'an image' can be represented as `image`.

Second, other constraints are related to multiple elements. For example, put an image on top of a button. To distinguish the elements referred in the constraint, we first build a vocabulary representing unique IDs for the elements. Then, we define a vocabulary for the relationships between elements. At last, we concatenate the tokens of element IDs and relationships.



(a) Model Architecture

Input Sequence	
Gen-T: <sos> image | text | text <eos>	*Gen-TS*: <sos> image 36 36 | text 60 20 | text 60 20 <eos>
Gen-R: <sos> image | text | text || text 1 top text 2 | text 2 bottom canvas <eos>	*Refinement*: <sos> image 20 13 35 34 | text 11 59 61 21 | text 9 87 63 19 <eos>
Completion: <sos> image 20 13 35 34 <eos>	*UGen*: <sos> <eos>
Output Sequence	
<sos> image 22 11 36 36 | text 10 58 60 20 | text 10 89 60 20 <eos>	

(b) Input and Output Sequences

Figure 3. a The overall model architecture of LayoutFormer++, illustrated by taking the task Gen-R as an example. b The examples of input and output sequences for different conditional layout generation tasks through constraint serialization.

For example, the vocabulary for element IDs could be $\{\texttt{image}_1, \ldots, \texttt{image}_{K_1}, \ldots, \texttt{button}_1, \ldots, \texttt{button}_{K_t}\}$, and the vocabulary for relationships could be $\{\texttt{top}, \ldots, \texttt{small}\}$. Then, the constraint 'an image on top of a button' is represented as $\texttt{image}_1\ \texttt{top}\ \texttt{button}_1$.

Constraint Combination. To combine different constraints, we concatenate their sequences. To ease the learning of neural network, we use a fixed order instead of a random order when concatenating different constraints. First, for the constraints only related to a single element, we first concatenate those for the same element and then concatenate those from different elements. Second, for the constraints related to multiple elements, we put them after all the constraints for the single elements.

3.2.2 Examples for Typical Tasks

In this work, we consider six typical layout generation tasks which has been explored in previous work. We demonstrate how we formulate the constraint sequences for these tasks according to Section 3.2.1 as followed.

Generation conditioned on types (Gen-T) is to generate layouts from the element types specified by user constraints. We formulate the input constraint sequence by concatenating the single-element constraints of element type, i.e., $S_{\text{Gen-T}} = \{\langle\text{sos}\rangle c_1|c_2|\ldots|c_N\langle\text{eos}\rangle\}$.[1]

Generation conditioned on types and sizes (Gen-TS) is to generate layouts when user constraints specify the element types and sizes. We formulate the input sequence by concatenating the type, the width and the height constraints as $S_{\text{Gen-TS}} = \{\langle\text{sos}\rangle c_1 w_1 h_1|c_2 w_2 h_2|\ldots|c_N w_N h_N\langle\text{eos}\rangle\}$.

Generation conditioned on relationships (Gen-R) generates layouts conditioned on element relationships. For example, a user could expect to put a large image at the top of a small text box. We formulate a relationship between two elements as $\{c_{k_{2m-1}} k_{2m-1} r_{k_{2m-1},k_{2m}} c_{k_{2m}} k_{2m}\}$, where $c_{k_{2m-1}}$ and $c_{k_{2m}}$ are the element types, k_{2m-1} and k_{2m} are the indexes for the elements, and $r_{k_{2m-1},k_{2m}}$ is an extra token introduced to denote one kind of relationships. We concatenate the sequences of type constraints and relationship constraints as $S_{\text{Gen-R}} = \{\langle\text{sos}\rangle c_1|c_2|\ldots|c_N||c_{k_1} k_1 r_{k_1,k_2} c_{k_2} k_2|\ldots|c_{k_{2M-1}} k_{2M-1} r_{k_{2M-1},k_{2M}} c_{k_{2M}} k_{2M}\langle\text{eos}\rangle\}$, where M is the number of relationships.

Refinement applies local changes to the elements that need improvements while maintaining the original layout design. We formulate the input as $S_{\text{Refinement}} = \{\langle\text{sos}\rangle c_1 x_1 y_1 w_1 h_1|\ldots|c_N x_N y_N w_N h_N\langle\text{eos}\rangle\}$.

Completion aims to complete layout from a set of specified elements. We formulate the input as $S_{\text{Completion}} = \{\langle\text{sos}\rangle c_1 x_1 y_1 w_1 h_1|\ldots|c_P x_P y_P w_P h_P\langle\text{eos}\rangle\}$, where P is the number of known elements.

Unconstrained generation (UGen) aims to generate layouts without any user requirements. We formulate the input as an empty sequence with necessary special tokens, i.e., $S_{\text{UGen}} = \{\langle\text{sos}\rangle\langle\text{eos}\rangle\}$.

3.3. Decoding Space Restriction

During the inference, we introduce the decoding space restriction strategy to the decoding process. Algorithm 1 presents the pseudo-code for the decoding space restriction.

In each decoding step t, the decoder predicts the probabilities P of the possible values for current attribute. Then, P will be pruned by two modules. The *ConstraintPruning* prunes the infeasible values which may violate the user given constraints S. The *ProbabilityPruning* prunes the values which the probabilities are lower than the threshold θ.

If all the probabilities in P' are pruned, it means there is no feasible value for current attribute. We propose a backtracking mechanism to solve this problem. When P' is empty and the back time $B[t]$ is less than the max back time $maxBack$, the backtracking mechanism will roll back the decoding process to a previous step t' and restart the prediction. Otherwise, the value o will be sampled from P' and the generation O will be updated. The decoding process will move to the next step by increasing t by 1.

While the end-of-sequence token EOS is predicted, or

[1] According to the practical experience, we add the special tokens $\langle\text{sos}\rangle$ and $\langle\text{eos}\rangle$ at the start and the end of the sequence, and use a separator token | to distinguish the single constraints.

the length of O reaches $maxLen$, the generation finishes. In the following, we will give more details about the two pruning modules and the backtracking mechanism.

Constraint Pruning Module. The constraint pruning module prunes the value in P which may violate the related constraints. Take the relationship constraint as example, suppose at step t the decoder predicts possibilities P for the value of the attribute w_i of the i-th element, there is a relation constraint $s = \{w_i \leq w_j\}$, which specifies the relative size relationships between element i and j. If w_j has not been predicted at step t, w_i will not be influenced by w_j in current step and the P will not be pruned. Otherwise, if w_j has been predicted at previous step, the feasible values of w_i will be restricted to $(0, w_j]$ according to s, and the infeasible values' probabilities in P will be set as 0.

Probability Pruning Module. The probability pruning module checks each value's probability in P'. The probabilities that are lower than the predetermined threshold θ will be pruned by setting as 0. The threshold θ is tuned to achieve the best performance by task.

Backtracking Mechanism. When the probabilities in P' are all set as 0, the backtracking mechanism works to roll back the decoding process to a previous step and restart the prediction. The function $Backtracking$ will check why the P' is pruned as empty, and decide which step t' to backtrack to. One situation is that the current token is restricted by previous token through the constraint. For example, the relation constraint $s = \{w_i \leq w_j\}$ restricts the feasible values of w_i by w_j. In this case, we choose the step of w_j as the backtracking step. Otherwise, since all the previous predictions collectively lead to current circumstance, we randomly pick one previous step to back to.

Algorithm 1: Decoding Space Restriction

Input: Encoder hidden state M; User constraints S.
Output: Layout sequence O.

Initialize step index t, the back time for each step B and the predicted sequence O.
while $(O[-1] \neq \text{EOS})$ ***and*** $(t{<}maxLen)$ **do**
  $P \leftarrow \text{Decoder}(O, M)[t]$;
  $P' \leftarrow \text{ConstraintPruning}(P, S)$;
  $P' \leftarrow \text{ProbabilityPruning}(P', \theta)$;
  if (P' is $\varnothing$) ***and*** $(B[t]{<}maxBack)$ **then**
    $t' \leftarrow \text{Backtracking}(P, S, t)$;
    $B[t] \leftarrow B[t] + 1$;
    $O \leftarrow O[:t]$;
    $t \leftarrow t'$;
  else
    $o \leftarrow \text{Sampling}(P')$;
    $O \leftarrow O \cup o$;
    $t \leftarrow t + 1$;
  end
end

4. Experiments

4.1. Setups

Datasets. We compare LayoutFormer++ to existing approaches on two public datasets. *RICO* [15] is a dataset of mobile app UI that contains 66K+ UI layouts with 25 element types. *PubLayNet* [30] contains 360K+ document layouts with 5 element types. On both datasets, there are a few layouts with quite a lot of elements, which easily leads to an out-of-memory problem. Existing studies use multifarious rules to filter out these layouts. In our experiments, we simply filter out the layouts with more than 20 elements. For RICO, we use 90%, 5% and 5% of data for training, validation and testing. For PubLayNet, we use 95% and 5% of official training split for training and validation, and the official validation split for testing.

Baselines. We try our best to reproduce all the existing approaches. Even so, we regret to not include some approaches for the following reasons. First, due to the missing implementation details and hyperparameter settings, we fail to reproduce some approaches [13, 14, 18, 20]. Second, a few approaches consider data-specific attributes and are difficult to be applied to arbitrary datasets about graphic layouts [27, 29]. Ultimately, we compare LayoutFormer++ against 1) *NDN-none* [12], *LayoutGAN++* [8] and *BLT* [10] on Gen-T, 2) *BLT* [10] on Gen-TS, 3) *NDN* [12] and *CLG-LO* [8] on Gen-R, 4) *RUITE* [23] on refinement, 5) *LayoutTransformer* [3] on completion, and 6) *VTN* [2], *Coarse2Fine* [7] and *LayoutTransformer* [3] on UGen.

Evaluation Metrics. We adopt the metrics proposed by existing works for comprehensive evaluation.

Maximum Intersection over Union (mIoU) measures the similarity between the generated layouts and the real layouts, which is based on the averaged IoU of bounding boxes. We use the same implementation as [8].

Alignment (Align.) measures whether the elements are well-aligned. We modify the metric from [14] by normalizing it by the number of elements.

Overlap measures the abnormal overlap area between elements. We improve the metric from [14] by ignoring the elements that serve as a background or padding, e.g., card, background and modal on RICO.

Frechet Inception Distance (FID) describes the distribution difference between real and generated layouts. Following [8], we train a neural network to convert the layouts into representative features and then calculate the distribution difference based on learned features.

Constraint Violation Rate (Vio.%) measures the rate of the violated constraints. We follow the implementation of [8] for Gen-R. For other tasks, we calculate Vio.% by dividing the number of the violated constraints by the total

Tasks	Methods	RICO				PubLayNet			
		mIoU ↑	FID ↓	Align. ↓	Overlap ↓	mIoU ↑	FID ↓	Align. ↓	Overlap ↓
Gen-T	NDN-none	0.35	13.76	0.56	0.55	0.31	35.67	0.35	0.17
	LayoutGAN++	0.298	5.954	0.261	0.620	0.297	14.875	0.124	0.148
	BLT	0.216	25.633	0.150	0.983	0.140	38.684	0.036	0.196
	LayoutFormer++	**0.432**	**1.096**	0.230	**0.530**	**0.348**	**8.411**	**0.020**	**0.008**
Gen-TS	BLT	0.604	0.951	0.181	0.660	0.428	7.914	0.021	0.419
	LayoutFormer++	**0.620**	**0.757**	0.202	**0.542**	**0.471**	**0.720**	0.024	**0.037**
Gen-R	NDN	0.36	-	0.56	-	0.31	-	0.36	-
	CLG-LO	0.286	8.898	0.311	0.615	0.277	19.738	0.123	0.200
	LayoutFormer++	**0.424**	**5.972**	0.332	**0.537**	**0.353**	**4.954**	**0.025**	**0.076**
Refinement	RUITE	0.811	0.107	0.133	0.483	0.781	0.061	0.029	0.020.
	LayoutFormer++	**0.816**	**0.032**	**0.123**	0.489	**0.785**	0.086	**0.024**	**0.006**
Completion	LayoutTransformer	0.363	6.679	0.194	0.478	0.077	14.769	0.019	0.0013
	LayoutFormer++	**0.732**	**4.574**	**0.077**	0.487	**0.471**	**10.251**	0.020	0.0022
UGen	LayoutTransformer	0.439	22.884	0.052	0.471	0.062	36.304	0.031	0.0009
	VTN	0.686	76.064	0.461	0.694	0.210	103.373	0.205	0.211
	Coarse2Fine	0.360	46.483	0.128	0.676	0.361	50.854	0.221	0.142
	LayoutFormer++	**0.742**	**19.688**	**0.047**	0.547	**0.417**	46.522	**0.029**	**0.0009**

Table 1. Quantitative comparisons with existing approaches on six layout generation tasks.

number of constraints for all the evaluated layouts.

Among the above metrics, mIoU, Align., Overlap and FID evaluate the generation quality, and Vio.% evaluate the constraint satisfaction. A larger value for mIoU indicates better performance, while smaller values for other metrics indicate better performance.

Implementation Details. We implement LayoutFormer++ by PyTorch [19]. The model is trained using the Adam optimizer [9] with NVIDIA V100 GPUs. For Transformer blocks, we use 8 layers, 8 heads for multi-attention, 512 embedding dimensions and 2048 feed-forward dimensions. Other hyper-parameters, e.g., the batch size, the learning rate, and the threshold θ in decoding space restriction, are tuned to achieve the best performance on the validation set[2]. For Gen-R, following CLG-LO [8], we randomly sample 10% element relationships as the input. For refinement, following RUITE [23], we synthesize the input by adding random noise to the position and size of each element, which is sampled from a normal distribution with the mean 0 and the standard variance 0.01.

4.2. Evaluations on Sufficient Flexibility

To demonstrate the sufficient flexibility of LayoutFormer++, we compare LayoutFormer++ with existing approaches on all six layout generation tasks.

Table 1 shows the quantitative comparisons to existing works[3]. It should be noted that the existing approaches can only handle up to two tasks, while LayoutFormer++ can handle all the six tasks. We highlight the results of LayoutFormer++ by bold when it achieves the best performance among all the existing approaches in Table 1. According to the results, LayoutFormer++ achieves significantly better performance than the baselines on most metrics. This demonstrates that LayoutFormer++ not only can flexibly handle diverse user constraints, but also has advantages in the generation ability over existing approaches.

Figure 4 and 5 show qualitative comparisons on RICO and PubLayNet. Compared to baselines, LayoutFormer++ generates layouts with better spacing, less misalignment, and fewer unreasonable overlaps.

4.3. Evaluations on Good Controllability

In this section, we compare LayoutFormer++ with LayoutGAN++ [13], BLT [10] and CLG-LO [8], which pay attention to the constraint satisfaction. These approaches handle Gen-T, Gen-TS and Gen-R respectively.

We bold the results of LayoutFormer++ in Table 2 when it outperforms the baselines. For Gen-T and Gen-TS, since all approaches have no constraint violation, we focus on the performance of quality. As the results, LayoutFormer++ outperforms the baselines on both two datasets. For Gen-R, LayoutFormer++ achieves better performance on both quality metrics and Vio.%, except a worse Vio.% than CLG-LO on RICO. However, CLG-LO does not make a good trade-off: the quality of layouts generated by CLG-LO is significantly worse than that of LayoutFormer++. This demonstrates that LayoutFormer++ achieves the best controllability among existing approaches.

[2]Details of the hyper-parameters can be find in supplemental material.

[3]As we fail to reproduce NDN by ourselves, we directly use the results from [8] in Table 1. It is evaluated on simple datasets that only consider layouts with less than 10 elements.

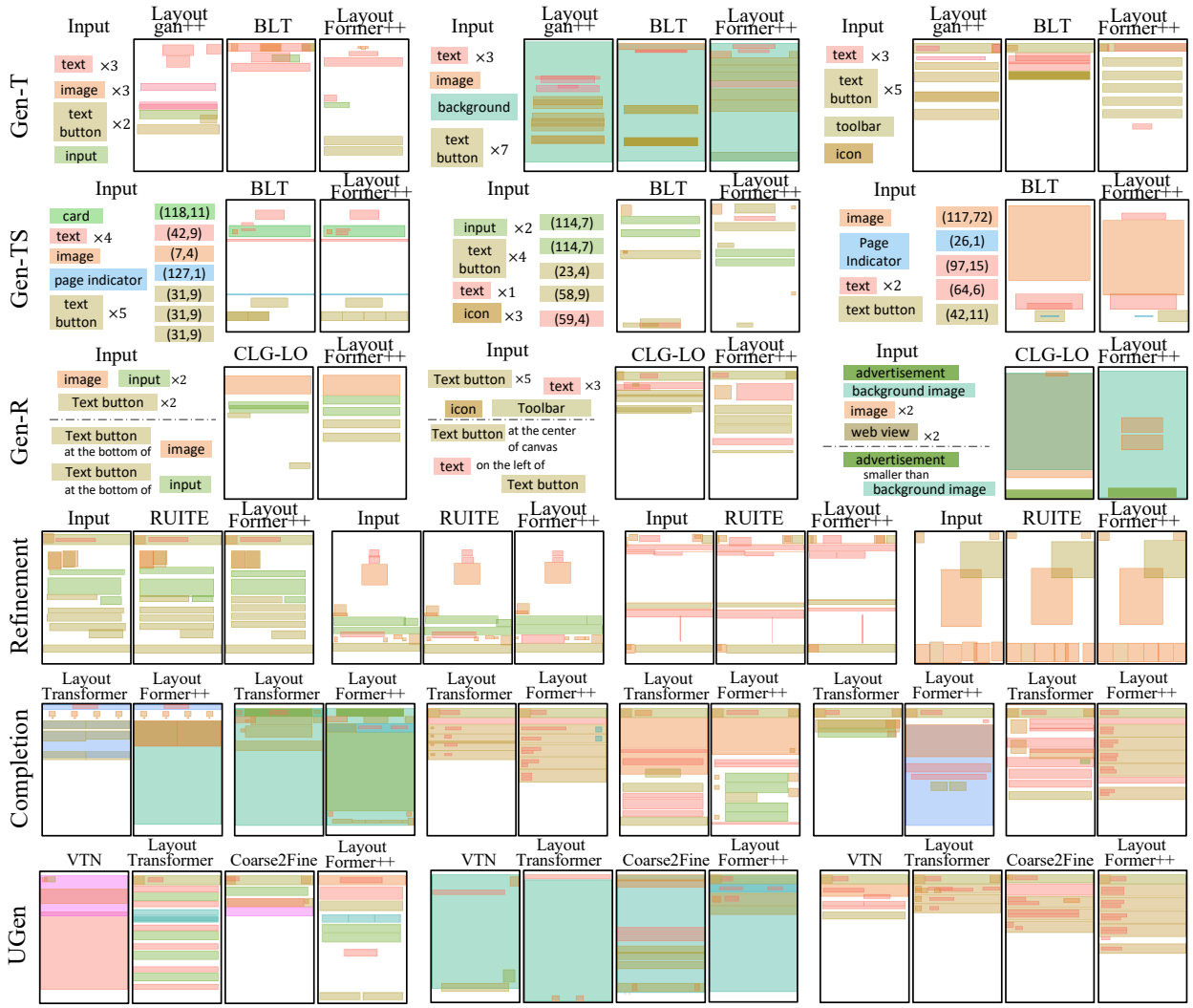


Figure 4. Qualitative results on RICO.

Tasks	Method		RICO mIoU ↑	RICO FID ↓	RICO Align. ↓	RICO Overlap ↓	RICO Vio. % ↓	PubLayNet mIoU ↑	PubLayNet FID ↓	PubLayNet Align. ↓	PubLayNet Overlap ↓	PubLayNet Vio. % ↓
Gen-T	LayoutGAN++		0.298	5.954	0.261	0.620	0.	0.297	14.875	0.124	0.148	0.
	Layout Former++	Full	**0.432**	**1.096**	**0.230**	**0.530**	**0.**	**0.348**	**8.411**	**0.020**	**0.008**	**0.**
		- Back	0.431	1.320	0.272	0.550	0.	0.345	9.367	0.020	0.009	0.
		- Back&Prune	0.439	1.392	0.206	0.545	5.5	0.345	9.373	0.020	0.009	0.05
Gen-TS	BLT		0.604	0.951	0.181	0.660	0.	0.428	7.914	0.021	0.419	0.
	Layout Former++	Full	**0.620**	**0.757**	0.202	**0.542**	**0.**	**0.471**	**0.720**	0.024	**0.037**	**0.**
		- Back	0.613	0.782	0.206	0.543	0.	0.464	0.903	0.026	0.044	0.
		- Back&Prune	0.613	0.801	0.206	0.545	≈0.	0.464	0.903	0.026	0.044	≈0.
Gen-R	CLG-LO		0.286	8.898	0.311	0.615	3.66	0.277	19.738	0.123	0.200	6.66
	Layout Former++	Full	**0.424**	**5.972**	0.332	**0.537**	11.84	**0.353**	**4.954**	**0.025**	**0.076**	**3.9**
		- Back	0.419	8.604	0.284	0.544	12.75	0.352	5.152	0.023	0.075	5.70
		- Back&Prune	0.458	5.126	0.221	0.546	33.04	0.358	4.620	0.022	0.030	16.09

Table 2. Comparisons with baselines and the model variants to evaluate the controllability.

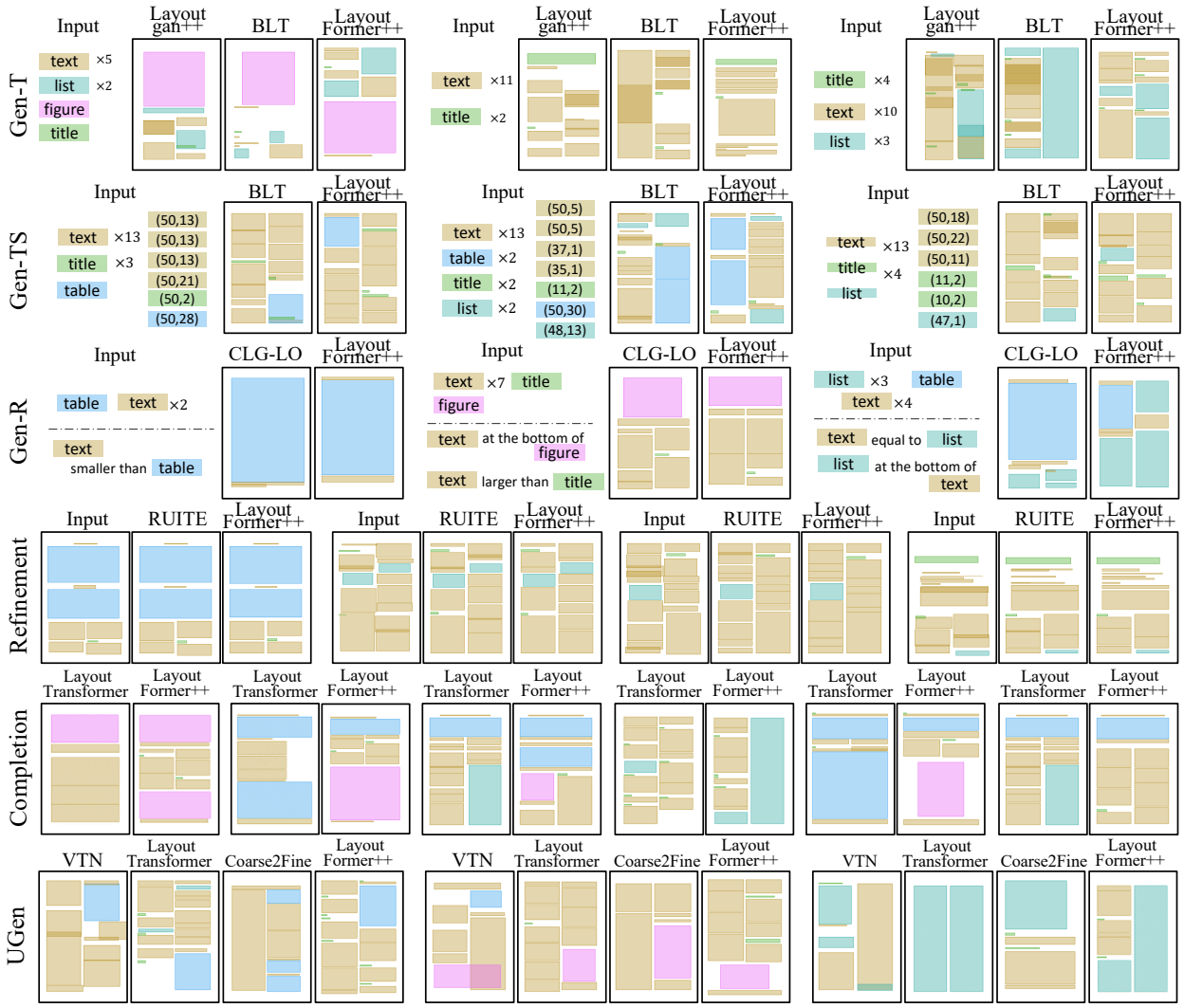


Figure 5. Qualitative results on PubLayNet.

Besides, we compare LayoutFormer++ framework with two variants. The first denoted as *-Back* indicates LayoutFormer++ without backtracking mechanism. The second denoted as *-Back&Prune.* is LayoutFormer++ without both pruning modules and the backtracking mechanism. For each task in Table 2, the last three lines show the comparison between the variants. Comparing *-Full* with *-Back*, we find that after disabling the backtracking mechanism, the generation quality decreases. And on Gen-R, the Vio.% also gets worse. Comparing *-Full* with *-Back&Prune*, we find that the Vio.% significantly decreases. It shows that the full model consistently decreases the violation while having little impact on quality, which demonstrates that both the pruning modules and the backtracking mechanism play important roles in helping LayoutFormer++ achieve good controllability.

5. Conclusion

In this work, we propose *LayoutFormer++* for conditional layout generation. To achieve the sufficient flexibility, we propose constraint serialization to represent different user constraints by the same sequence format. To achieve the good controllability, we propose decoding space restriction to prune the predicted distribution by disabling the options that violate the constraints or lead to bad quality. Experiments show that LayoutFormer++ can flexibly handle different layout generation tasks with better generation quality and less constraint violation compared to the existing task-specific approaches. In the future, we plan to investigate more practical user requirements for layout design. Besides, as LayoutFormer++ enables a unified way to handle different layout generation tasks, it is possible to develop a powerful pretrained model for layout generation in the future.

References

[1] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Guided open vocabulary image captioning with constrained beam search. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 936–945, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics. 3

[2] Diego Martin Arroyo, Janis Postels, and Federico Tombari. Variational transformer networks for layout generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13642–13652, June 2021. 2, 3, 5

[3] Kamal Gupta, Justin Lazarow, Alessandro Achille, Larry S. Davis, Vijay Mahadevan, and Abhinav Shrivastava. Layouttransformer: Layout generation and completion with self-attention. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1004–1014, October 2021. 2, 3, 5

[4] Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1535–1546, Vancouver, Canada, July 2017. Association for Computational Linguistics. 3

[5] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. 3

[6] J. Edward Hu, Huda Khayrallah, Ryan Culkin, Patrick Xia, Tongfei Chen, Matt Post, and Benjamin Van Durme. Improved lexically constrained decoding for translation and monolingual rewriting. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 839–850, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. 3

[7] Zhaoyun Jiang, Shizhao Sun, Jihua Zhu, Jian-Guang Lou, and Dongmei Zhang. Coarse-to-fine generative modeling for graphic layouts. In *AAAI'22*, February 2022. 2, 5

[8] Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. Constrained graphic layout generation via latent optimization. In *Proceedings of the 29th ACM International Conference on Multimedia*, MM '21, page 88–96, New York, NY, USA, 2021. Association for Computing Machinery. 2, 3, 5, 6

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015. 6

[10] Xiang Kong, Lu Jiang, Huiwen Chang, Han Zhang, Yuan Hao, Haifeng Gong, and Irfan Essa. Blt: Bidirectional layout transformer for controllable layout generation, 2021. 2, 3, 5, 6

[11] Jayant Krishnamurthy, Pradeep Dasigi, and Matt Gardner. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics. 3

[12] Hsin-Ying Lee, Lu Jiang, Irfan Essa, Phuong B. Le, Haifeng Gong, Ming-Hsuan Yang, and Weilong Yang. Neural design network: Graphic layout generation with constraints. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 491–506, Cham, 2020. Springer International Publishing. 2, 3, 5

[13] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. Layoutgan: Generating graphic layouts with wireframe discriminators, 2019. 2, 5, 6

[14] Jianan Li, Jimei Yang, Jianming Zhang, Chang Liu, Christina Wang, and Tingfa Xu. Attribute-conditioned layout gan for automatic graphic design. *IEEE Transactions on Visualization and Computer Graphics*, 27(10):4039–4048, aug 2021. 5

[15] Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. Learning design semantics for mobile apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 569–579, New York, NY, USA, 2018. ACM. 2, 5

[16] Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States, July 2022. Association for Computational Linguistics. 3

[17] Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. NeuroLogic decoding: (un)supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4288–4299, Online, June 2021. Association for Computational Linguistics. 3

[18] David D. Nguyen, Surya Nepal, and Salil S. Kanhere. Diverse multimedia layout generation with multi choice learning. In *Proceedings of the 29th ACM International Conference on Multimedia*, MM '21, page 218–226, New York, NY, USA, 2021. Association for Computing Machinery. 5

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 6

[20] Akshay Gadi Patil, Omri Ben-Eliezer, Or Perel, and Hadar Averbuch-Elor. Read: Recursive autoencoders for document

layout generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020. 5

[21] Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1314–1324, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. 3

[22] Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. COLD decoding: Energy-based constrained text generation with langevin dynamics. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. 3

[23] Soliha Rahman, Vinoth Pandian Sermuga Pandian, and Matthias Jarke. Ruite: Refining ui layout aesthetics using transformer encoder. In *26th International Conference on Intelligent User Interfaces - Companion*, IUI '21 Companion, page 81–83, New York, NY, USA, 2021. Association for Computing Machinery. 2, 3, 5, 6

[24] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. 3

[25] Richard Shin and Benjamin Van Durme. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5417–5425, Seattle, United States, July 2022. Association for Computational Linguistics. 3

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. 2

[27] Kota Yamaguchi. Canvasvae: Learning to generate vector graphic documents. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5481–5489, October 2021. 2, 5

[28] Maosen Zhang, Nan Jiang, Lei Li, and Yexiang Xue. Language generation via combinatorial constraint satisfaction: A tree search enhanced Monte-Carlo approach. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1286–1298, Online, Nov. 2020. Association for Computational Linguistics. 3

[29] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson W. H. Lau. Content-aware generative modeling of graphic design layouts. *ACM Trans. Graph.*, 38(4), jul 2019. 5

[30] Xu Zhong, Jianbin Tang, and Antonio Jimeno Yepes. Publaynet: largest dataset ever for document layout analysis. *arXiv preprint arXiv:1908.07836*, 2019. 2, 5