

Learning GUI Completions with User-defined Constraints

LUKAS BRÜCKNER, Aalto University, Finland

LUIS A. LEIVA, University of Luxembourg, Luxembourg

ANTTI OULASVIRTA, Aalto University, Finland

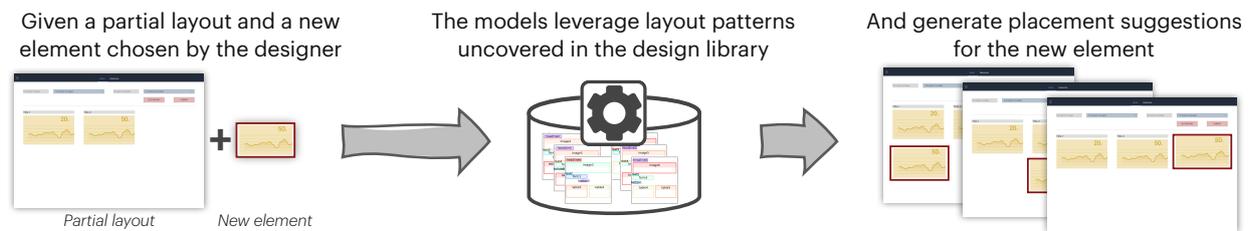


Fig. 1. Completion of a layout with a user-defined element according to layout patterns in reference designs.

A key objective in the design of graphical user interfaces (GUIs) is to ensure consistency across screens of the same product. However, designing a compliant layout is time-consuming and can distract designers from creative thinking. This paper studies layout recommendation methods that fulfill such consistency requirements using machine learning. Given a desired element type and size, the methods suggest element placements following real-world GUI design processes. Consistency requirements are given implicitly through previous layouts from which patterns are to be learned, comparable to existing screens of a software product. We adopt two recently proposed methods for this task, a Graph Neural Network (GNN) and a Transformer model, and compare them with a custom approach based on sequence alignment and nearest neighbor search (kNN). The methods were tested on handcrafted datasets with explicit layout patterns, as well as large-scale public datasets of diverse mobile design layouts. Our results show that our instance-based learning algorithm outperforms both neural network approaches. Ultimately, this work contributes to establishing smarter design tools for professional designers with explainable algorithms that increase their efficacy.

CCS Concepts: • **Human-centered computing** → **Systems and tools for interaction design**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: user interfaces, machine learning, design, layouts, layout completion

ACM Reference Format:

Lukas Brückner, Luis A. Leiva, and Antti Oulasvirta. 2022. Learning GUI Completions with User-defined Constraints. *ACM Trans. Interact. Intell. Syst.* 12, 1, Article 6 (March 2022), 41 pages. <https://doi.org/10.1145/3490034>

1 INTRODUCTION

This paper addresses a key objective in the design of graphical user interfaces (GUIs): *consistency* [30], understood as any reoccurring feature across designs. This includes visual attributes, such as

Authors' addresses: Lukas Brückner, paper@luksurious.me, Aalto University, Finland; Luis A. Leiva, name.surname@uni.lu, University of Luxembourg, Luxembourg; Antti Oulasvirta, antti.oulasvirta@aalto.fi, Aalto University, Finland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2160-6455/2022/3-ART6 \$15.00

<https://doi.org/10.1145/3490034>

50 typography, colors, shapes, etc., but also spatial patterns such as the layout base grid, and recurring
51 patterns of element groups, e.g., forms or navigation bars. Maintaining design consistency manually
52 can be difficult and time-consuming [25] and contributes to a substantial amount of time designers
53 spend on simple and repetitive tasks [31]. Automating such kind of work is expected to increase
54 designer's efficiency and allow them to focus on creative thinking instead.

55 To this end, organizations create *design systems* that establish consistency between different
56 screens of a product. Typical design systems maintain a large set of rules and guidelines, for both
57 layout and GUI elements, that can later be embedded into new designs. However, not all reoccurring
58 patterns of element placements can be anticipated beforehand. Instead, they emerge in the lifecycle
59 of a system as an increasing number of screens are designed. We refer to these reoccurring patterns
60 of element placements as *layout patterns*. Maintaining consistency of layout patterns between
61 multiple screens requires, thus, explicit knowledge of all other screens in the company's design
62 portfolio. This can become infeasible for large products or new designers in larger organizations.

63 Previous research into design assistance tools for design systems focused on explicit rules of
64 such design systems as well as aligning, packing, and optimizing layouts [3, 4, 32]. These goals
65 can be expressed as mathematical requirements and accurately optimized using Mixed Integer
66 Linear Programming [29]. However, to the best of our knowledge, consistency of implicit layout
67 patterns has not been studied yet. Providing design assistance for layout patterns should help
68 designers create GUIs that are more consistent and at the same time increase their efficiency. Since
69 layout patterns may not be defined upfront, modeling them with explicit methods is a tedious and
70 error-prone task. Instead, we focus on machine learning methods to understand these patterns
71 from examples.

74 1.1 Problem statement

75 This work tackles the question of where to place a new element of a specific type and size onto an
76 existing, partial layout such that the resulting layout is consistent with a set of reference designs.
77 This is depicted in Figure 1. A consistent placement of a new element is achieved if the layout
78 patterns exhibited in the set of reference designs are followed. We decompose layout patterns into
79 two attributes between elements: (1) positional dependencies and (2) alignment relations between
80 elements. While these attributes may not suffice to describe layout patterns rigorously, we argue
81 they indicate the most important features in a similar fashion a human designer would describe
82 them at a high level. subsection 3.2 explains these attributes in more detail.

83 We focus on *flat layouts*, i.e., we assume there are no container elements (such as card elements
84 that may contain images or text). Further, we frame the design process of a new GUI layout as
85 a sequential process of placing individual elements onto it, one after another. We do not restrict
86 that a group of elements can only have a single pattern placement but there can be multiple valid
87 compositions thereof. As such, to best assist human designers, the output is expected to provide
88 different variations where possible, and declare a measurement of goodness for these variations.
89 Finally, to allow the integration into the design process, the runtime of a competent method should
90 provide results in a short amount of time, so that it does not impact the designer's flow.

91 We explicitly do not aim for a scenario in which a method produces complete layouts on its
92 own, as that would require a much deeper understanding of the design task and the goals of a UI
93 to be applicable to real-world design processes. Instead, we aim to augment the designer in their
94 current workflow and to assist them in placement decisions based on the context of the existing
95 design such that they can focus on creative thinking. This approach was informed by the tools and
96 processes of professional designers that envisaged such a new assisting tool.

1.2 Contributions

Our work is motivated by reports of professional designers that uncovered consistency as a key requirement. By considering layout patterns explicitly, we concretize the general layout completion problem of previous works [10, 23] and make it applicable to real-world, product-focused design tasks. Focusing on layout patterns allows for a more principled evaluation as opposed to generic goodness qualities of completions. In contrast to previous work on layout completion [10, 19, 23], we argue that designers should ultimately be in charge of deciding on the final layout. Since a specific task requires particular elements, we allow designers to condition the layout completion process and ensure that user-specified constraints, i.e. element types and sizes, are followed.

We contribute by benchmarking different methods on standard as well as novel data sets that help to understand the potentials and limits of these approaches in realistic, commercial user interface design settings. Concretely, we evaluate two recently proposed methods for layout completion, a Graph Neural Network (GNN) [19] and a Transformer model [10, 23] that have shown promising results. These approaches have shown state-of-the-art results in related tasks, making them natural choices given their representative fit to layout problems.

Finally, inspired by the shortcoming of existing neural methods, we present a novel approach that leverages a sequence alignment algorithm to calculate features based on layout principles that are used in a nearest neighbor search (kNN). With this work, ultimately, we contribute to establishing smarter design tools for professional designers that ensure consistency in the design process and allow them to focus on less repetitive tasks.

2 RELATED WORK

The layout problem or derivations thereof have been approached with combinatorial optimization techniques, machine learning, and other techniques such as Bayesian methods. In the following, we provide a brief overview of these techniques that have been applied to either *layout generation* or *layout optimization* with a summary shown in Table 1.

2.1 Layout generation

Hart and Yi-Hsin [11] developed the first formal description of the layout problem as a rectangular packing problem for integer linear programming. This model aimed to fit as many objects as possible onto a window without overlap and clipping of the objects and showed that it can be solved efficiently. Damera-Venkata et al. [2] formulated a probabilistic document model to automatically generate multi-page document compositions given text and images. It requires a set of probabilistic templates which are evaluated via a Bayesian Network to find the best combination of templates and template parameters to achieve the best document layout. Later, Talton et al. [36] proposed a method that learned design patterns from a set of webpages with Bayesian Grammar Induction. The learned grammar could then be used to synthesize new web layouts.

O'Donovan et al. [28] developed an energy-based model to automatically generate graphic designs. The design principles considered include alignment, symmetric balance, white space, reading flow, overlap, and a saliency model learned via linear regression. Further, model parameters could be learned from a small number of example designs via non-linear inverse optimization, such that the style is transferred to the new design.

More recently, methods based on generative adversarial networks (GAN) that have shown success in natural image generation tasks have been applied to layout generation. With LAYOUTGAN, Li et al. [22] proposed a method that captures relations between all elements through stacked self-attention modules, leveraging geometric features and element probabilities. They found that a discriminator operating in the visual domain performs better than a discriminator using the generated geometric

Authors	Year	Task type	Layout	Method	Design features
Hart and Yi-Hsin [11]	1995	Generation	Generic	Optimization	overlap
Damera-Venkata et al. [2]	2011	Generation	Document	Bayesian	templates
Talton et al. [36]	2012	Generation	UI	Bayesian	templates
O'Donovan et al. [28]	2014	Generation	Poster	Energy	alignment, symmetry, flow, white space, overlap, saliency
Li et al. [22]	2019	Generation	Document, UI, Scene	Deep Learning	visual & spatial context
Zheng et al. [40]	2019	Generation	Document, Poster	Deep Learning	visual & semantic context
Tabata et al. [35]	2019	Generation	Document	Deep Learning	grid alignment, overlap, visual features
Jyothi et al. [15]	2019	Generation	Scene	Deep Learning	visual features
Johnson et al. [14]	2018	Generation	Scene	Deep Learning	semantic relations
Gajos et al. [6–8]	2004–2010	Adaptation	UI	Optimization	task time, usage pattern, device, motor capabilities
Kumar et al. [16]	2011	Transfer	UI	Machine Learning	visual features, font size, colors, structural similarities
Leiva [20]	2012	Adaptation	UI	Model-free	usage pattern
Xu et al. [39]	2014	Optimization	UI	Optimization	alignment
Todi et al. [37]	2016	Optimization	UI	Optimization	sensorimotor performance, perceptions
Dayama et al. [4]	2020	Optimization	UI	Optimization	grid alignment, rectangularity
Laine et al. [17]	2020	Adaptation	UI	Optimization	selection time, saliency, device constraints
Swearngin et al. [33]	2020	Optimization	UI	Optimization	size, balance, alignment, grouping, order, emphasis, grid
Lee et al. [19]	2019	Generation, Completion	UI	Deep Learning	semantic relations (size, position)
Li et al. [23]	2020	Completion	UI	Deep Learning	spatial context
Gupta et al. [10]	2020	Completion	UI	Deep Learning	spatial context
Dayama et al. [3]	2021	Transfer, Optimization	UI	Optimization	grid alignment, templates, structural similarities
Ours	2021	Completion	UI	Machine learning	spatial context, alignment, overlap

Table 1. Overview of related work, including task type, layout type, method, and design features used.

features directly. It was applied to both document layouts and mobile app layouts but it lacked control over the generation process.

Zheng et al. [40] proposed a content-aware GAN for generating magazine layouts that are conditioned on the desired content, including the topic, keywords, and image contents to be placed. These conditions are embedded into a feature vector that is then concatenated to the latent vector of

197 the generator. It further allows specifying desired locations of certain elements as soft-constraints.
198 To ensure results follow these additional constraints, a large number of results are generated and
199 filtered to contain the desired number of elements per type and rough locations.

200 Focusing on creative results for graphic designs, Tabata et al. [35] created a system that generates
201 layouts based on the user input of the text and images that need to be laid out. It first generates
202 a large number of random layouts with a minimum set of rules, such as grid alignment and non-
203 overlapping, and then scores candidates based on visual features processed by convolutional neural
204 network layers such that results are similar to real magazine layouts it was trained on.

205 Representations studied in the field of natural scene generation offer some resemblance but differ
206 greatly from the requirements of UI layouts. LAYOUTVAE employs variational autoencoders to
207 predict bounding boxes of target objects for a new scene which is then filled with images to achieve
208 the final scene [15]. They did not model any further constraints of the objects and did not consider
209 alignment, as it is not a requirement in natural images. Similarly, Johnson et al. [14] proposed a
210 method of encoding the input as a graph that is then used to generate bounding boxes for a scene
211 image.

212

213

2.2 Layout optimization and adaptation

214

215 Gajos et al. presented a model to treat user interface adaptation as a discrete constrained optimization
216 problem [6–8]. Their SUPPLE++/ ARNAULD system adapts a UI to different device constraints
217 and optimizes it for a user’s usage patterns and motor capabilities such that the navigation time
218 required to navigate the UI is minimized according to the usage history.

219 BRICOLAGE [16] is one the early works based on machine learning techniques where the content
220 of a web page is transferred into the style of another. It used 30 manual features to create mappings
221 between source and target elements, together with a sophisticated tree-matching algorithm whose
222 weights were learned from human examples via a simple perceptron network. It employed features
223 from the visual domain, like dimensions, font sizes, and colors, as well as structural similarities
224 based on sibling and ancestry nodes.

225 ACE [20] enabled adaptation of web interfaces to a user’s behavior without an explicit user
226 model. Instead, it leverages information induced by implicit interaction of the user to inform about
227 the relevance of different parts of the website, which are then slightly modified according to their
228 relative importance. As such, it uses a straight-forward mathematical formulation for scoring the
229 website elements without machine learning models or combinatorial optimization.

230 Xu et al. [39] proposed an optimization model for alignment improvements in UIs with a sparse
231 linear system solving technique that dynamically evaluates constraints based on the input to find
232 the optimal layout. They noted the issue of resolving ambiguity from the input and studied a
233 gesture-based design tool that allows to interactively update the input constraints to best match
234 the desired properties of the designer.

235 SKETCHPLORE [37] studied the integration of a layout optimization model into a design sketching
236 tool. It was designed to assist during the creation process of a new design and inferred the designer’s
237 task based on the input and offered local and global optimizations. The optimizer used predictive
238 models of sensorimotor performance and perceptions to find better designs.

239 GRIDS [4] is a wireframing tool with an integrated grid alignment optimizer based on mixed-
240 integer linear programming. It follows the assumption that many good layouts are following a grid
241 system to define placement areas and optimize results with respect to grid alignment, rectangularity
242 of the overall outline, and respecting preferential placement of elements. While it aims to minimize
243 the alignment edges of the elements of the layout, it does not follow a predefined grid system
244 typically found in professional designs.

245

246 Dayama et al. [3] combined layout adaptation based on templates with predefined grid systems
 247 using integer programming optimization to ensure designs adhere to the guidelines and rules of
 248 such a design system. As with most explicit methods, it requires to specify all rules beforehand and
 249 cannot adapt to conventions easily, as well as not providing a method to decide the placement of
 250 new elements.

251 Further work in this area include LAYOUT-AS-A-SERVICE [17] and SCOUT [33]. The former applies
 252 layout personalization and optimization to websites with different targets such as selection time,
 253 visual saliency, and device size. The latter presented a layout exploration tool based on high-level
 254 constraints that supports alternative design elements, grouping, placement preferences, and others.
 255 As such, it supports exploration for new design ideas but does not consider reference designs.

256

257

2.3 Layout completion and assistance

258 Lee et al. [19] proposed the *Neural Design Network* to generate and complete layouts with constraints
 259 by representing a layout as a graph and employing graph neural networks. They modeled edges
 260 as component relations of relative position and size as input constraints and constructed a three-
 261 layered system that processes incomplete layout graphs into coordinates. It serves as the basis
 262 for our graph network approach that we extend to support better the alignment requirements of
 263 design systems.

264 Li et al. [23] designed a system based on Transformers [38] to auto-complete a partial mobile app
 265 layout. Layouts are represented as sequences based on their tree structure and fed into different
 266 transformer structures to predict the types and positions of additional components. However, no
 267 control over the result of the generation was considered. This was addressed recently by Gupta et al.
 268 [10]. They also employed a transformer model to complete a layout based on an input sequence, but
 269 instead of encoding an element as a single embedding, they modeled every element as a sequence
 270 of attributes and tokens. This allows to condition the generation on partial attributes of the next
 271 element. As such, we adapt this proposal in our work and evaluate it according to our problem
 272 statement.

273 Our work builds upon the recent research by Gupta et al. [10] and Lee et al. [19]. We adjusted their
 274 proposed methods to our particular problem statement and addressed the encountered limitations
 275 in a novel approach as described in the next section.

276

277

3 ELEMENT PLACEMENT TASK

278 Our ultimate goal is to assist designers in the creation process of new designs for software products
 279 and enable them to create more consistent layouts with respect to layout patterns. As such, we
 280 expect that there exists a set of curated designs with a consistent layout following a design system
 281 that can serve as the basis for learning these layout patterns. To formalize this, we introduce a
 282 consistent layout notation and define the layout completion task studied in this paper.

283

284

3.1 Layout notation

285 We denote a layout L as an abstract, structural representation of a design canvas with size $w_L \times h_L$
 286 (width, height) in pixels. It is composed of a set of elements $\{e_i, \dots, e_n\}$. Every element is defined by
 287 $e = (c_e, s_e)$ where $c_e \in C$ is the component (i.e., functional user interface element with a specific
 288 function, e.g., ‘button’, ‘input field’, etc.) out of the valid component set C , and $s_e = (x_e^0, y_e^0, w_e, h_e)$
 289 is the ‘size box’ of the element. x_e^0, y_e^0, w_e, h_e describe the x and y coordinates, and the width
 290 and height of the element in pixels. Note that the ‘size box’ is not the ‘bounding box’ of an
 291 element $b_e = (x_e^0, y_e^0, x_e^1, y_e^1)$ since the latter describes the upper-left (x_e^0, y_e^0) and lower-right (x_e^1, y_e^1)
 292 coordinate points. Finally, we refer to the ‘center point’ of an element as x_e^c, y_e^c for the x and y
 293 coordinates, respectively.

294

In this work, we assume that all elements are rectangular, i.e., the actual area of an element equals to the area of its bounding box or size box. For many calculations it is useful to standardize the element size between $[0, 1]$, which is achieved by dividing the width and x-coordinates by the layout width w_L , and the height and y-coordinates by the layout height h_L . In this case, the corresponding (standardized) attributes are denoted with a tilde, \tilde{x} , \tilde{y} , \tilde{w} , \tilde{h} . It applies to all variants described above.

3.2 Task definition

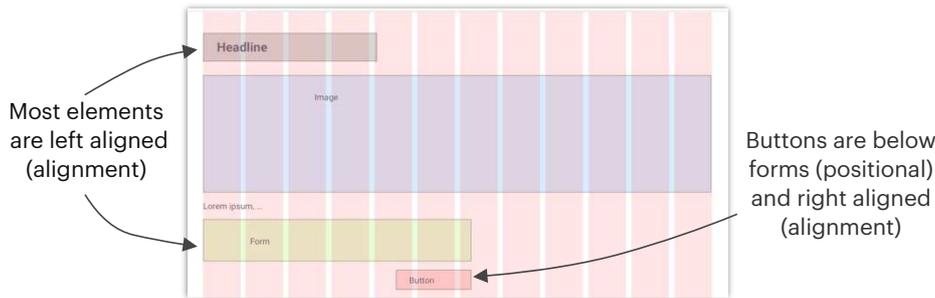


Fig. 2. A layout exhibiting patterns that should be followed when placing new elements in similar compositions.

Figure 2 shows an example of how we describe layout patterns using *positional* and *alignment* relations between groups of elements, assuming further layouts with these compositions in an existing design library. The layout in this example figure is following a common grid system (indicated by the reddish columns laid over the canvas) that is overall left-aligned, so most elements are attached on the left edge to the first layout column. This *general pattern* implicitly creates an alignment relation “left-aligned” between most elements (e.g., the image (blue box) is left-aligned to the headline (gray box)). There might further be an implicit positional relation between some elements (e.g., headline being “above” all other elements), and non-existent between others. On the contrary, the button element (red box) is shown to be right-aligned to the form element (green box), and positioned below which constitutes a *local pattern*. While this is just a single example for demonstration purposes, design patterns like these will be evident from investigating a set of reference layouts.

The layout attributes can be described in more detail as follows: *Positional* dependency refers to the interplay between elements in which the presence or location of one component influences the relative position of a second component. For example, a button might be placed directly below a form if the form is short while it is placed in a fixed position on the bottom of a screen if the form is long. More specifically, we consider the positional classes *above*, *below*, *left*, *right*. Figure 3 shows examples of this relation type. We argue that vertical positioning (*above*, *below*) is generally more descriptive than horizontal (*left*, *right*), so, if a component is fully below and to the side of another one, it will be considered as *below* only, and not *left*, or *right*.

On the other hand, *alignment* relations refer to whether elements share common alignment lines of edges or the center, both horizontally and vertically. For example, the button might be always aligned to the left side of a form while an image might be centered on a page. Alignment relations are shown in Figure 4. The set of valid relations between two elements depends, however, on their relative positioning. Elements that are *above/below* each other can be either *left*, *right*, or *vertically center-aligned*. Elements to the *left/right* side of each other can be either *top*, *bottom*, or *horizontally center-aligned*. Further, we include in this relation category also the relation to the overall canvas of the layout. If an element is within a reasonably small margin to either of the edges of the screen, it

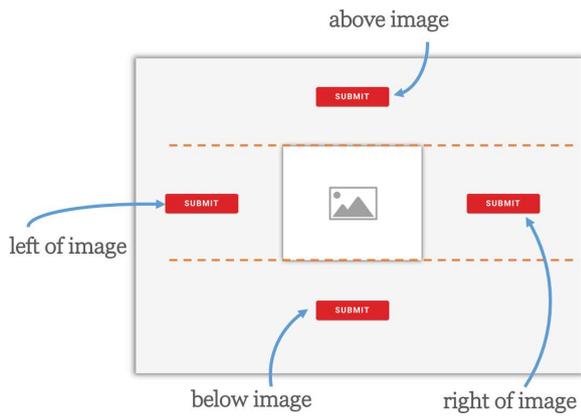


Fig. 3. Positional relations towards the center.

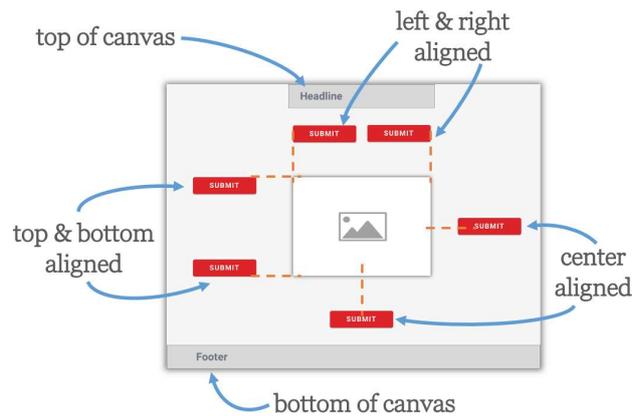


Fig. 4. Inter-element alignment and to the canvas.

will be described as being *at the top*, *bottom*, *right* or *left* of the canvas. Finally, as not every element might be aligned in some way to another component, a special *none* alignment type exists that does not restrict the relative placement of the two elements.

As we focus on flat layouts, every component is assumed to be “complete” by itself such that it contains every element to make it a meaningful and comprehensive user interface component. This means in turn, that we do not support containers of elements, nor do we expect that any elements overlap. Instead, we consider overlaps indicate a bad placement.

The task is then defined as follows: Given a set of reference layouts $\{L_0, \dots, L_n\}$, the user generates a partial input layout L (that might be empty) and picks the next element e_{new} to be placed, defined by its component type $c_{e_{\text{new}}}$ and the desired dimensions $(w_{e_{\text{new}}}, h_{e_{\text{new}}})$. A computational model is expected to return as pair of $\{x_i, y_i\}$ coordinates as recommendations for the placement of the new element e_{new} . The result is valid if the resulting placement is non-overlapping with existing elements and its bounding box is completely inside the canvas. Since only coordinates are returned, placing an element at extreme values of the canvas can produce a placement that extends beyond the valid canvas size. Further, results should exhibit layout patterns, i.e., relations to neighboring elements in both positional and alignment terms, that can be found in the set of reference layouts. Pattern matching is described further in subsection 5.3.

4 METHODS

Layout completion of GUIs is a problem that is less studied than the more generic GUI generation problem. Previous methods mostly focused on automatic completion without user constraints (see subsection 2.3). Therefore, we adapt two previously described methods which allow such conditioning and showed promising results in the more generic layout generation and completion tasks: the graph-based “Neural Design Network” (GNN) [19], and the “LayoutTransformer” [10].

As preliminary inspections of results produced by those approaches revealed shortcomings, we propose a novel sequence-based method that incorporates both layout representations. The sequence representation is used for determining the rough element insertion based on a sequence alignment algorithm and nearest neighbor search, and the graph representation powers the placement algorithm given a neighboring layout.

4.1 Layout representations

We employ two complementary layout representations in the evaluated methods: a *sequential* and a *graph-based* representation.

4.1.1 *Representing layouts as sequences.* Layouts are most naturally depicted on a 2D canvas. However, comparing layouts with different elements at different locations is challenging as it is not straight-forward which element from one layout to compare with from the other. A naïve approach would enumerate all combinations but this becomes intractable very quickly. One might try to compare those elements that are closest together but it can fail in cases where similar elements are shifted as additional rows are present in one layout.

It is desirable to simplify the representation such that the number of mapping combinations is greatly reduced. To this end, we employ a simple sequence representation by reading elements from top-left to bottom-right. For western left-to-right systems this representation has been shown to be resemble user scanning behavior [12], so despite this apparently stark simplification, this model is able to capture a natural understanding of layouts to a reasonable degree.

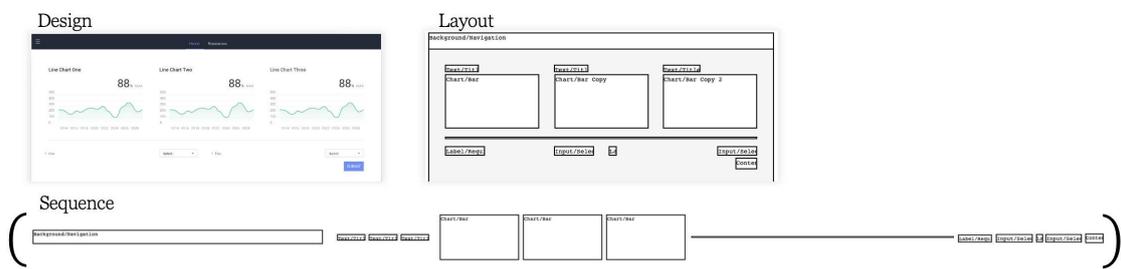


Fig. 5. Representing a layout as a sequence. The component view shows a simpler bounding box representation. The sequence follows the natural reading order from top-left to bottom-right.

Figure 5 shows an example of how a layout is represented as a sequence using this approach. While there might be ambiguities regarding the perceived element order when hierarchies are present (e.g., the groups of charts and text), we do not consider container elements currently and restrict ourselves to flat layouts. As such, elements that are perceived to span “multiple rows” are added the first time they are encountered in a row. More formally, a layout L with n elements $\{e_1, \dots, e_n\}$ is represented by its sequence S_L as follows:

$$S_L = (e_{p_1}, e_{p_2}, \dots, e_{p_n}), \quad (1)$$

where e_{p_i} is the i th element according to the placement order p_i . The placement order p_i is determined according to the reading order from top-left to bottom-right, such that for any two subsequent elements e_{p_i} and $e_{p_{i+1}}$ the following condition must follow:

$$y_{e_{p_i}}^0 \leq y_{e_{p_{i+1}}}^0 \wedge x_{e_{p_i}}^0 \leq x_{e_{p_{i+1}}}^0, \quad (2)$$

where x_e^0, y_e^0 represents the x and y coordinates of the top-left corner of an element e . This is the same approach as in LayoutTransformer [10].

4.1.2 *Representing layouts as graphs.* A more flexible representation of a layout can be achieved by utilizing a graph. Graphs allow capturing certain relations of its elements without operating in a large pixel space. In this representation, every element is modeled as a node in the graph, and edges between elements describe some relative features, such as a positional attribute (e.g., ‘left’, ‘above’, etc.). Considering that graph neural networks are using message passing between nodes, we use bidirectional edges to allow information flowing between all elements, even if it introduces some redundancy. Since we want to achieve consistent placement of elements according to their positional and alignment relations, two types of edges are added between elements, one for each relation type.

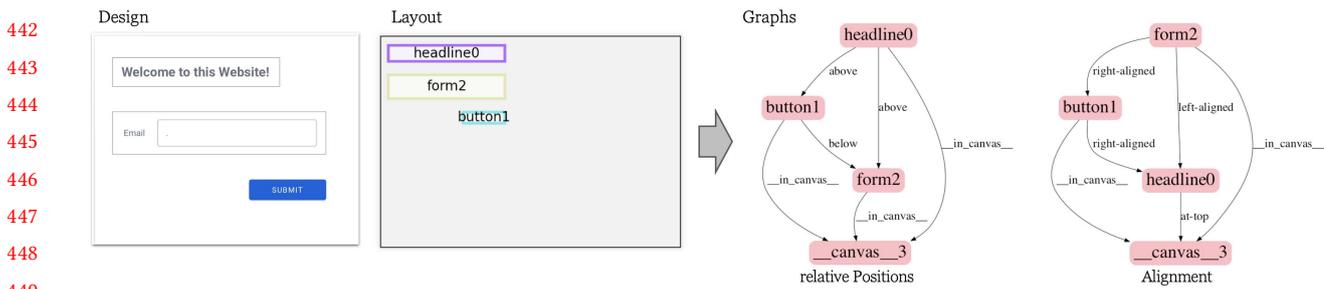


Fig. 6. Graph representation of a simple layout with 3 elements.

More formally, we represent layouts as bidirectional heterogeneous graphs $G = (V, A)$, where $V = \{v_i\}$ is the set of vertices that correspond to the elements of a layout $\{v_i\} = \{e_0, \dots, e_n\}$, and A is the set of labeled arrows (i.e., directed edges) between vertices (v_i, r, v_j) where v_i is the source of the edge, v_j its target, and r a relation label from the set R . To represent both relation types, we use separate graphs G^{pos} and G^{align} where the vertices are the same $V^{\text{pos}} = V^{\text{align}}$ but the arrows A differ in their assigned labels, and based on the valid set R^{pos} and R^{align} respectively. We create bidirectional graphs with redundant edges to allow information to flow in all directions.

For every pair of elements e_i, e_j , two directed edges (e_i, r, e_j) and (e_j, \bar{r}, e_i) are created where \bar{r} is the inverse relation of r (e.g., *button* –*below*– *text* and *text* –*above*– *button*). This is done for both graphs G^{pos} and G^{align} such that the edges between both are equal except for the relational types. Finally, the canvas is represented by a separate node. This canvas node has only incoming edges for specifying the relations of the elements to the canvas but has no outgoing edges. This allows specifying positions to the canvas, e.g., ‘at top’ or ‘at right’. An example layout with a corresponding graph is shown in Figure 6.

4.2 Graph neural network with constraints

We adapt the ‘Neural Design Network’ by Lee et al. [19] to our layout task. Reasons for evaluating this method include the strong representative power of the graph-based model, and the natural capabilities to allow constraining the network to user-defined inputs such that it can be integrated into a real-world design process. The network model is depicted in Figure 7. It comprises three modules that operate independently. The first module, *Relation Prediction*, completes a partial graph such that a new element has labeled edges to the existing elements. Then, the *Layout* module decides on the position of the new element on the canvas given the existing canvas and the completed input graph. Finally, the *Refinement* module optimizes the placement so that it better adheres to alignment rules.

The input layout is modeled as a directed graph $G = (V, A)$, according to the graph representation explained previously, where vertices $v \in V$ correspond to layout elements and the arrow labels $a \in A$ describe the relationship between two elements. The vertices always hold information about the component type c . In the layout and refinement steps, the existing ‘size box’ s is added to the vertex data as well. The canvas is included as a special node and its size box corresponds to the canvas size.

The two relation types (positional and alignment) are encoded in two separate sets of edges A^{pos} and A^{align} . This is different from the original implementation, which used relative *size* as a second edge type. Since in our work the element sizes are given by the user, we substitute the size property with alignment, as it follows more closely our problem definition. Then, the canvas element only has incoming relations that specify special positions of elements inside the canvas (e.g., ‘at top’), if applicable, or simply ‘in canvas’. All other elements have bidirectional relations.

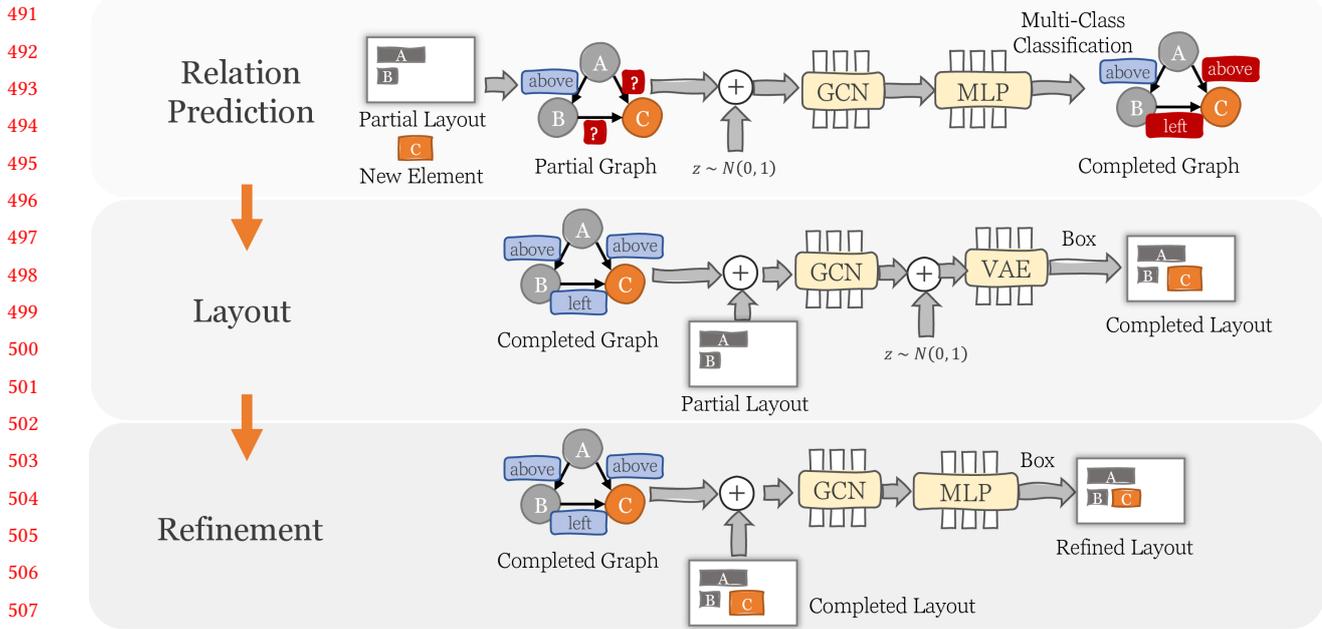


Fig. 7. The graph neural network implemented according to [19] is composed of three modules to predict a location for a new element.

Adding a new element e_{new} to the canvas is done by inserting a new node into the graph with type $c_{e_{\text{new}}}$ and the desired size $(w_{e_{\text{new}}}, h_{e_{\text{new}}})$. Edges between this new element and all other elements are labeled with a special 'unknown' category. Through the processing modules of the network, a final size box for the new element is predicted $\hat{s}_{e_{\text{new}}}$ that contains the suggested placement position.

4.2.1 Model details. Overall, the network architecture follows the description from [19] and is detailed in appendix A. However, we focus exclusively on the single element placement task. That means, only a single new element is added to the partial layout graph that has a defined type and width and height.

Relation module. The input to the relation module Rel is the partial graph G^p where edges to and from the new element e_{new} have the special label *unknown*, and a complete graph \hat{G} is generated that contains label predictions for the previously unknown edges: $\hat{G} = \text{Rel}(G^p)$. Depending on the set of training layouts, there might be cases for which the same partial input graph can have multiple valid output graphs. That is why the prediction is conditioned on a learned latent variable z_{rel} that allows representing different output variations.

Layout module. In the layout module, the elements' size boxes $\{s_{e_i}\}$ are added to the completed graph $\hat{G} = (V, \hat{A}^{\text{pos}}, \hat{A}^{\text{align}})$ to predict the size box of the new element $\hat{s}_{e_{\text{new}}}$. We adjust the training procedure according to our task definition, and convert the recursive approach originally described to perform a single box generation instead.

Refinement module. The final module takes the generated layout as input, aiming to fine-tune it. Its input is the completed graph $\hat{G} = (V, \hat{A})$, with the size boxes of the all elements $\{s_{e_i}\}$, and it produces an updated size box for the new element $\hat{s}'_{e_{\text{new}}}$. Originally, when multiple elements are placed in the layout module, this step allowed us to adjust the new boxes knowing the other placements. However, in our case the refinement module only operates on a single element, so this step becomes unnecessary. Thus, the output includes both refined and unrefined results to counter cases with a low rate of valid results and provide a higher variability in the final suggestions.

Training. Compared to the original description of the method that reconstructed a single training layout inside the network, we generate separate training items from a single training layout before training. We refer to subsection 5.2 for more details.

4.3 Transformer-based prediction

Transformers [38] have become a popular and powerful neural network architecture choice for many problems that can be represented as a sequence. Two recent papers also proposed a transformer-based model to address layout completion: Li et al. [23] and Gupta et al. [10]. Here, we use a similar approach to Gupta et al. as it is closer to our stated problem and allows us to specify layout constraints on prediction more easily.

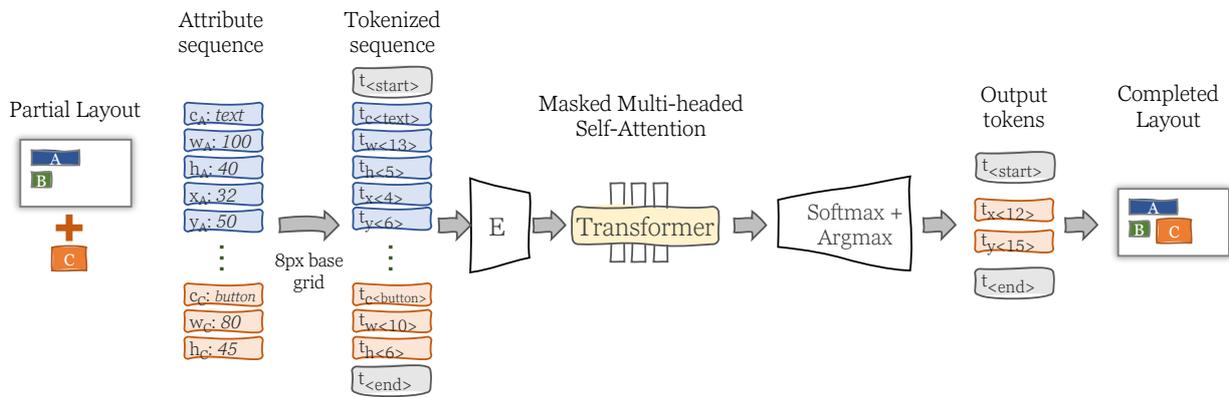


Fig. 8. The transformer network, modeled after [10].

4.3.1 Representation of layouts. In this model, layouts are decomposed into sequences of elements as described in subsection 4.1.1. The input is a partial layout L and a new element e_{new} with a given element type $c_{e_{\text{new}}}$ and size $w_{e_{\text{new}}}, h_{e_{\text{new}}}$. The result of the transformer network is a token sequence that can be decoded into the positions $\hat{x}_{e_{\text{new}}}^0, \hat{y}_{e_{\text{new}}}^0$.

Instead of combining the properties of an element to form a single embedding for the transformer network as in [23], the element properties are given as individual tokens that are embedded separately as in [10]. This makes it possible to condition the network output on the new element type and size. A single element e is represented as $(c_e, w_e, h_e, x_e^0, y_e^0)$. Compared to [10], we move the width and height of an element in front of the coordinates to allow defining the desired size of the new element. A layout L is then represented by the concatenation of all element attributes in a flat sequence S_L :

$$S_L = c_{e_0}, w_{e_0}, h_{e_0}, x_{e_0}^0, y_{e_0}^0, \dots, c_{e_n}, w_{e_n}, h_{e_n}, x_{e_n}^0, y_{e_n}^0. \quad (3)$$

To form the query sequence with a new element $(c_{e_{\text{new}}}, w_{e_{\text{new}}}, h_{e_{\text{new}}})$ is appended. We apply the same base grid transformation as in the LayoutTransformer paper [10], thus, reducing the number of possible tokens. The complete tokenization process is described in appendix B.1.

4.3.2 Model architecture. The model follows a standard transformer architecture [38] and the data flow is depicted in Figure 8. It is composed of stacked encoders and decoders that consist of multi-head attention layers and feed-forward layers. They take in embedded token sequences with explicit positional information. In the decoder, the input must be partially masked to prevent reverse information flow of the expected output of the sequences. The final step of the decoder is passing the result through a linear transformation followed by a *softmax* activation function to

589 generate a probability distribution of the next token prediction. We employ similar parameters in
 590 our implementation as proposed by Gupta et al. [10], see appendix B.3.

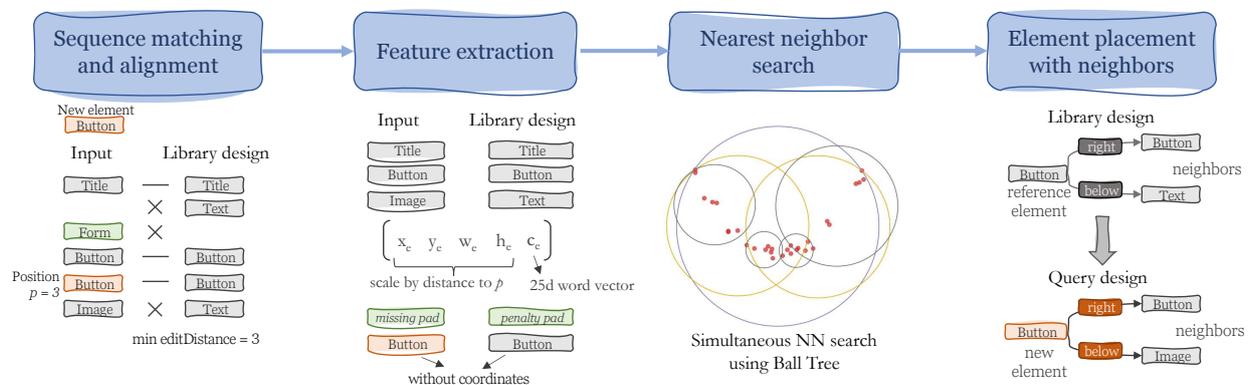
591 We specialize our transformer network on the single element prediction task of our problem
 592 statement. For that, we generate the training data to only contain the single next new element to
 593 be added to the design. Training details are described in appendix B.2.

594 **4.3.3 Prediction and decoding.** Since a single element is decomposed into 5 distinct attributes, every
 595 position in the token sequence is associated with a particular attribute. Thus, we need to control the
 596 decoding of an output sequence such that valid sequences are produced. During inference, we add a
 597 partially defined element to the end of an input sequences, i.e., the attributes $(c_{e_{\text{new}}}, w'_{e_{\text{new}}}, h'_{e_{\text{new}}})$, and
 598 request the prediction of the missing coordinate attributes $\hat{x}_{e_{\text{new}}}^{0'}, \hat{y}_{e_{\text{new}}}^{0'}$. Valid tokens must then be
 599 part of the corresponding token ranges $\{t_{x'}\}, \{t_{y'}\}$. To achieve this, we restrict the decoding to the
 600 corresponding token range and disregard probabilities for other token ranges (i.e., only considering
 601 the probability distribution over tokens representing x- and y-coordinates in the standard task).
 602

603 Finally, we use a beam search algorithm to generate diverse variations for the same input,
 604 with a temperature parameter T that modifies the output probabilities according to a Boltzmann
 605 distribution which allows controlling the ‘creativity’ of the model. Therein, a low temperature
 606 amplifies high probabilities, while a high temperature levels all probabilities. In our trials, we used
 607 a temperature of $T = 0.2$, which results in a slightly conservative prediction of the model.

608 4.4 Sequence alignment with nearest neighbors

609 Achieving consistency with existing designs naturally requires attending to the patterns in those
 610 designs. This can be directly exploited by a nearest neighbor search algorithm that suggests
 611 placements based on specific instances of previous designs without trying to generalize.
 612



625 Fig. 9. Process overview of the sequence-aligned nearest neighbor search with neighborhood placement of
 626 the new element.

628 Our process is visualized in Figure 9. We model the placement problem as finding the best
 629 insertion position in the target layout sequence such that the distance to the layout sequences
 630 in the design library is minimized. The overall process is as follows: (1) For every library layout,
 631 find the insertion point in the target sequence such that the sequence alignment is maximized.
 632 (2) Create features for all aligned layout sequences and run the nearest neighbor algorithm to find
 633 the library layouts with the lowest distance. (3) Place the new element on the target design based on
 634 the neighborhood relations in the library layout. The time and space complexity for the complete
 635 algorithm is $\Theta(N|L_{\text{max}}|^2)$ and $\Omega(N|L_{\text{max}}|^3)$, where $N = |\{L^i\}|$ is the number of library designs and
 636 $|L_{\text{max}}|$ is the maximum number of elements any layout (library or query) contains.
 637

638 **4.4.1 Insertion point and sequence alignment.** Before we can compare a completed layout from the
 639 library with an input layout, the new element has to be inserted into the target layout sequence. To
 640 determine sensible candidates efficiently, we identify the likely best insertion points by minimizing
 641 the edit distance between the target layout and the new element inserted at position p and each
 642 library layout based on the element type sequences S^{type} (i.e., ‘button’, ‘text’, etc.). This is described
 643 in more detail in appendix C.1.

644 For the resulting candidates, we calculate the sequence alignment path according to Hirschberg’s
 645 algorithm [13]. The alignment path allows to construct a pair of target \tilde{S}_{t_p} and library \tilde{S}_{l_p} element
 646 subsequences for every library item. In these subsequences, the type matches are maximized. The
 647 alignment path then provides a mapping between the library and the target layout that is used to
 648 construct the feature representations for the nearest neighbor search.

649 **4.4.2 Feature representation.** Every library sequence \tilde{S}_{l_i} and target sequence \tilde{S}_t is converted to a
 650 flat feature representation that can be used as input to a nearest neighbor algorithm. Therein, every
 651 element is represented by its *top-left* coordinate, its *width* and *height*, as well as a 25-dimensional
 652 word vector of its *type*. We use the top-left coordinate of an element and its size instead of a
 653 bounding box to better handle shifted subsequences in which sizes might be very similar but actual
 654 positions are shifted by a fixed value.

655 More formally, every element e in the layout is represented by its feature $f_e = (\tilde{x}_e^0, \tilde{y}_e^0, \tilde{w}_e, \tilde{h}_e, v_{c_e})$,
 656 where $\tilde{x}_e^0, \tilde{y}_e^0$ are the standardized coordinates of the top-left corner, \tilde{w}_e, \tilde{h}_e the standardized width
 657 and height of the element, and $v_{c_e} \in \mathbb{R}^{25}$ the word vector of the component name. The word
 658 vector encodes the component type and allows calculating a distance between components such
 659 that for similar concepts, e.g., ‘label’ and ‘text’, the distance is smaller, while unrelated concepts
 660 such as ‘button’ and ‘table’ are separated by a longer distance. We use *fastText*’s pretrained word
 661 representations that are trained on Common Crawl and Wikipedia text [9]. The basic form of the
 662 layout sequence \tilde{S} is then a concatenation of the individual element features:

$$663 \quad \tilde{S} = f_{e_{p_0}}, f_{e_{p_1}}, \dots, f_{e_{p_n}}, \quad (4)$$

664 where $f_{e_{p_i}}$ is the feature of the element in the i th position in the aligned sequence. All features are
 665 “unwrapped” to ensure that a 1-dimensional feature representation is created.

666 The new element of the target layout does not have assigned coordinates yet. Hence, only its
 667 dimensions and area are used to encode it: $f_{e_t} = (\tilde{w}_{e_t}, \tilde{h}_{e_t}, \alpha \sqrt{\tilde{w}_{e_t} \tilde{h}_{e_t}}, v_{c_{e_t}})$. We want to place a high
 668 emphasis on finding a placement for the new element where the reference element in the library
 669 layout has a similar size. Thus, in addition to the width and height of the new element, also a scaled
 670 value of the area is added to its feature representation. To keep it in the same magnitude as the
 671 individual coordinates, the square root of the area is used. This applies both to the new element
 672 in the target layout e_t as well as the matched element e_{p^*} in the library layout according to the
 673 sequence alignment result. With the scaling factor α , one can increase or decrease the size matching
 674 requirements for the new element.

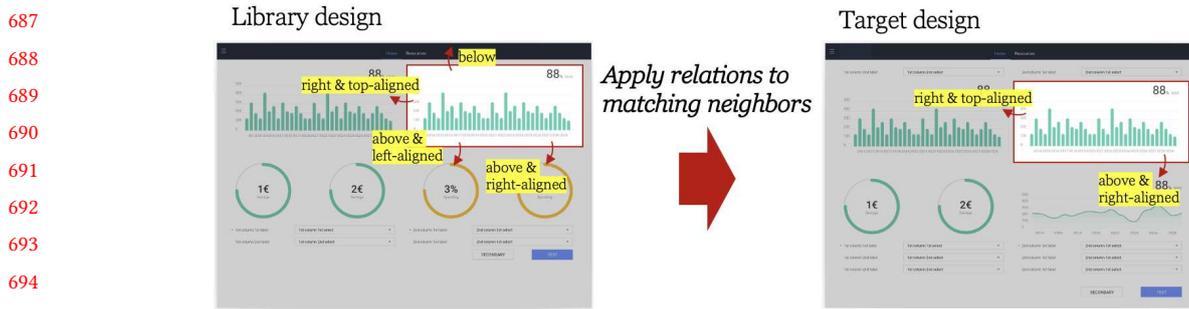
675 In addition, we argue that elements that are closer to the insertion point in the sequence are
 676 more relevant than elements that are farther. So, to accommodate this, we scale each feature vector
 677 by its distance to the insertion point of the new element. See appendix C.2 for more details.

678 Not all elements of a layout might be directly matched against other layouts in the design
 679 library. For example, if the library contains more elements than the target layout (including the new
 680 element), some elements in the library layout will not be encoded in the feature representation. On
 681 the other hand, the more elements that can be matched, the better. Since the sequence alignment
 682 algorithm finds the best matching component type subsequence, it could suggest a very small
 683

684

685

686



696 Fig. 10. Placement approach using relations to neighbor elements.

697

698

699 overlap that could lead to a small distance if mismatches are ignored. To counter this effect, every
 700 element from the target layout that is not matched in the library is added as a penalty feature f_e^p
 701 to the target layout (but not the other way around). See appendix C.3 for our specific formulation,
 702 and C.4 for an example of a feature representation.

703

704

705 **4.4.3 Nearest neighbor search.** As the number of elements in the layouts may vary and efficient
 706 search structures require fixed input lengths, short sequences are padded with dummy values.
 707 Further, to be able to use efficient search structures such as Ball Tree or KDTree, the features
 708 must be comparable by a single true metric that exhibits the properties of identity, symmetry, and
 709 triangle inequality. While it is more common to use cosine similarity for word similarity, it has
 710 the disadvantage of not exhibiting the triangle inequality. Thus, we use euclidean distance for all
 711 parts of our feature vectors. Finally, the desired number of neighbors is increased in the neighbor
 712 search by a factor $\eta = 3$ to account for the effect that a resulting match might not produce a feasible
 layout.

713

714

715 **4.4.4 Producing final layouts.** Once a neighbor for the query layout is identified, the new element
 716 can be placed on it. To avoid collision with existing elements when naively copying the position,
 717 we consider the *neighborhood* of the mapped element in the neighbor layout and try to apply
 718 the relative positioning rules to the query layout. Specifically, we consider the neighborhood
 719 of the mapped target element in the neighbor layout as the set of elements that have no other
 720 element between the target and itself in any of the four directions (above, below, left, right)
 721 of the element. Of this neighborhood, we consider the graph representation as detailed in section
 722 4.1.2, i.e., the relations in regards to position and alignment, to the mapped target element. These
 723 relations are then applied to the query layout, using the mapping between the elements produced
 724 by the sequence alignment algorithm. Neighbor elements that are not mapped are ignored. This
 725 is shown in Figure 10. If the resulting placement creates an invalid layout because the element is
 726 overlapping with another element or is placed outside the canvas, the relations are successively
 727 reduced according to the distance to the target element such that farther neighbors are ignored for
 728 the subsequent layout trial. If no relaxation creates a good layout, the candidate is discarded. The
 729 resulting layouts are finally presented to the user as placement suggestions, sorted by the distance
 computed in the nearest neighbor search (the smaller the distance, the better the suggestion).

730

731

730 4.5 Design tool integration

731

732

732 For these methods to be used properly in a design process it must be integrated into a fully-fledged
 733 user interface design program. For this purpose, we created a plugin for the Sketch application,¹ a

733

734

734 ¹<https://www.sketch.com>

735

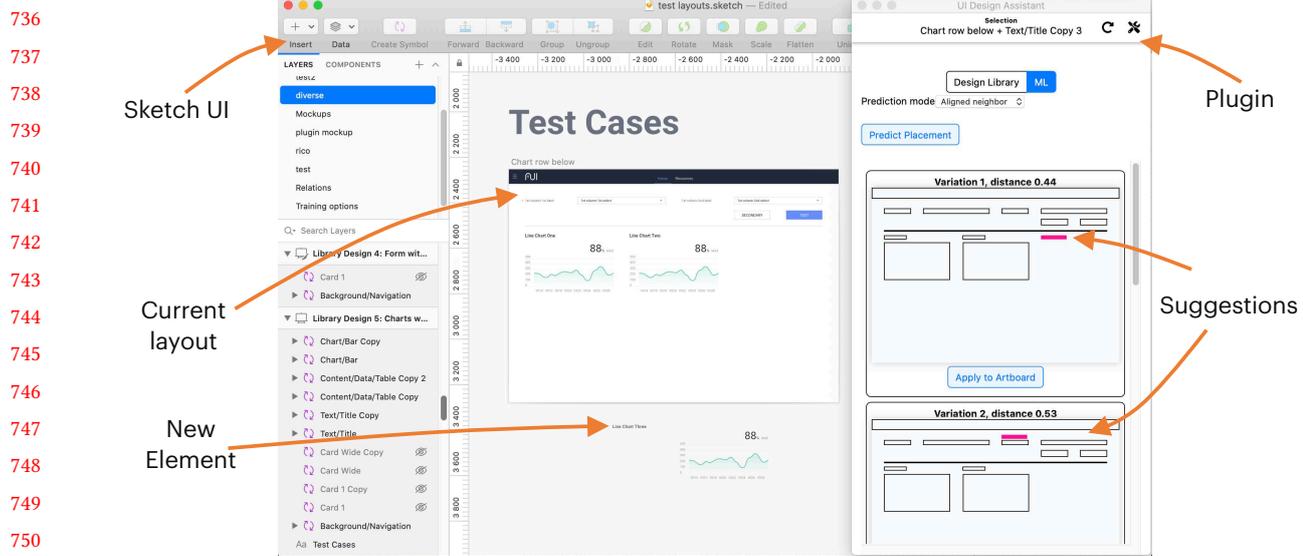


Fig. 11. The user interface of *Sketch* with our plugin loaded on the right side. It shows the placement suggestion for a new text element onto the artboard (canvas) in the middle of the screen.

popular design tool used by the designers that motivated our work. Figure 11 shows the plugin overlaid over the regular *Sketch* window. *Sketch* represents designs in a nested structure according to its layer-based structure. Since our work focuses on higher-order components instead of low-level elements (i.e., rectangle), we require that the layout is composed of previously defined components, known as ‘symbols’ in *Sketch*. These symbols represent the components that are available via the design system to the UI designer. To identify components, we assume a naming convention that allows mapping components and their variations to a set of known components. For example, a ‘button’ component might have different styles with names such as ‘button/primary’ and ‘button/secondary’, but we consider them both to be of the same component type (‘button’).

The workflow of our plugin is described as follows. First, the user selects the canvas with a partial layout or any element inside of it. Next, a new element must be added to an area outside of the canvas and selected. Then, the user can generate placement suggestions in the plugin by choosing the prediction method and requesting placement candidates by pressing the “Predict placement” button. A list of candidates is computed in the backend application that communicates via an HTTP API with the plugin. The plugin generates stylized previews of the suggestions with the new element highlighted, which can be applied to the design canvas via the “Apply” button.

We should mention that this interaction method may not provide the best user workflow. More sophisticated techniques such as snapping or ghosting should be considered to display the suggested candidates. However such techniques cannot be implemented with the existing *Sketch* plugins API² as the current plugins architecture is limited in this regard.

5 EVALUATION

Evaluating the models requires appropriate data sets that can serve as reference layouts for finding layout patterns. Previous work mostly leveraged the Rico data set [5]. However, it was shown to be of mixed quality [18, 21]. Further, it contains layouts from various apps, thus, no specific layout patterns can be expected. While we still evaluate the Rico data set to match previous works, we note that its expressiveness is limited. To address the quality issue, we test the curated subset of

²<https://developer.sketch.com/plugins/>

785 Rico called Enrico [21] containing well-formed layouts, to assess if this has an effect on the learning
 786 despite it containing a mixture of applications without clear patterns. Lastly, we address the lack
 787 of consistent layout patterns in existing data sets by creating two custom data sets that contain
 788 layout patterns and follow a design system.

789 After an exhaustive quantitative evaluation, we investigated results across the different methods
 790 and data sets manually to better understand the capabilities of each approach. We evaluate the
 791 methods by measuring the rate of valid results, an alignment score, and a pattern matching accuracy
 792 where applicable. Alignment has been used before to evaluate UI layout generation [19]. We argue
 793 that the rate of valid results is an important measure to estimate the applicability of a method in
 794 practical applications and should be reported to prevent cherry-picking results. Finally, to test for
 795 layout patterns, we leverage the pattern attributes and check for a threshold overlap to determine
 796 whether a pattern was matched or not.

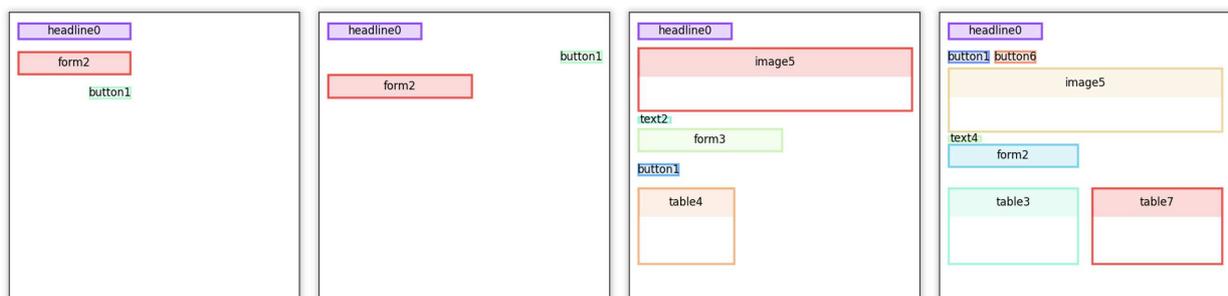
797

798 5.1 Data sets

799 We created two layout data sets with pre-defined patterns using probabilistic templates: *Varying*
 800 *buttons* and *Artificial web layouts*. These allow to test the methods against an applicable benchmark
 801 with layout patterns, similar to existing designs in an organization with professional designers,
 802 where clear layout patterns are expected to be present. While the first data set is small and contains
 803 little variation, the latter consists of dense layouts with a wider variety of patterns and noise levels
 804 resulting in complex layouts representative of real-world designs. On the other hand, both *Rico* and
 805 *Enrico* comprise real-world layouts of mobile apps with layout complexity ranging from low to high.
 806 These data sets provide results on common benchmarks and allow to investigate if the training
 807 data size has an effect on the neural network methods. However, they do not allow to evaluate
 808 layout patterns and we can only test general goodness qualities. The different layout complexity of
 809 the studied data set is reported in the column ‘Elements’ in Table 2.

810

811



820

821

821 Fig. 12. Layouts in the varying buttons data set.

822

823 *Varying buttons*. This data set consists of small web-like layouts with 3–7 high-level components
 824 (*headline*, *form*, *text*, *button*, *image*, *table*). The layouts follow a simple 2 column grid layout with the
 825 majority of the elements being positioned in the first column and being left-aligned. While most
 826 components are placed in the same (relative) position, the *button* component is placed at 5 different
 827 positions: (1) below the headline and left-aligned, (2) below the headline and at the right edge of
 828 the canvas, (3) below the form and left-aligned, (4) below the form and right-aligned, and (5) below
 829 the form and at the right edge of the canvas. This is the main pattern that is encoded in all of the
 830 layouts. Repeated elements form a secondary pattern. Therein, the second element is placed in the
 831 same row next to the first element of the same type. In total, the set comprises 36 layouts of which
 832 25 form the design library, and 11 layouts the test cases. Samples of this set are shown in Figure 12.

833

When completing such layouts, the placement of the next element is mostly unambiguous except for buttons. This represents a simple yet non-trivial case, as there is not a single correct result in reference to the data set when placing a button. Further, it allows to validate the requirement that multiple variations are expected when appropriate. A challenge for relation prediction lies in the imbalance of classes. There exist predominantly vertical positions (*above/below*), as well as *left-aligned* relations.

Artificial web layouts. The second custom data set contains a larger number of more complex layouts. The layouts use a typical grid system with a horizontally centered, top-to-bottom column layout. They comprise 2–3 sections with multiple inner columns and a static *navigation bar* and *footer*. Samples of such layouts are shown in Figure 13.

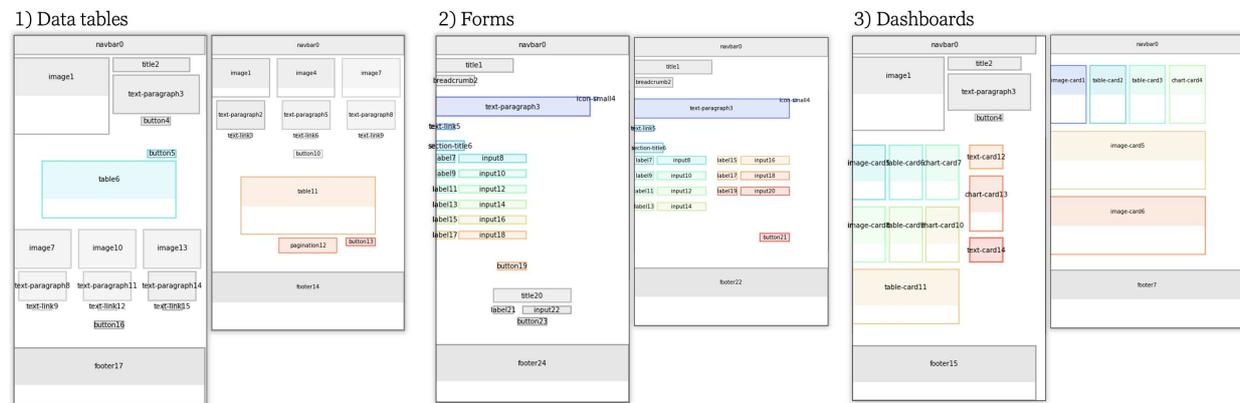


Fig. 13. The different patterns in the artificial web layouts data set.

We encoded three distinct layout patterns in this data set: (1) data tables, (2) forms, and (3) dashboards, and each contains multiple variations. Most layouts contain additional context components to form more realistic web pages. It consists of 359 layouts of which 52 belong to the test set. The number of elements ranges from 4 to 38 per layout with 18 different component types. Hence, repeated components are very common. A detailed description of the patterns can be found in appendix D.1. Completing such layouts is a more challenging task: Firstly, the layouts are more packed. Secondly, the context components create a certain noise level. And thirdly, the layouts contain a high number more components.

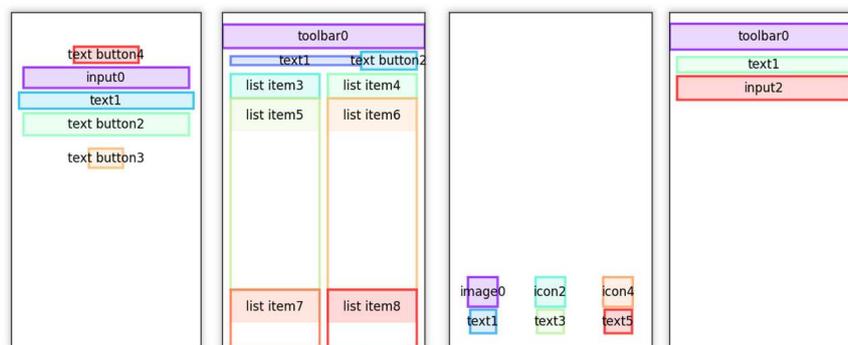


Fig. 14. Example layouts from the Rico (NDN) data set. All layouts have less than 10 elements, hence, the complexity is lower than the Enrico set.

Rico. (NDN) Rico [5] is a large set of mobile layouts extracted from the Android Play Store. We employ the same selection method as in the “Neural Design Network” [19] and retain only layouts with less than 10 elements that are composed of the most commonly used component types. In addition, we ignore layouts that have only 1 element and those with overlap. This results in 21,557 layouts for the final data set³. Figure 14 shows example layouts of this data set. One has to bear in mind that these in-the-wild layouts do not contain common patterns. As such, it is not suitable to evaluate pattern detection capabilities. Instead, general layouting capabilities are tested on it.

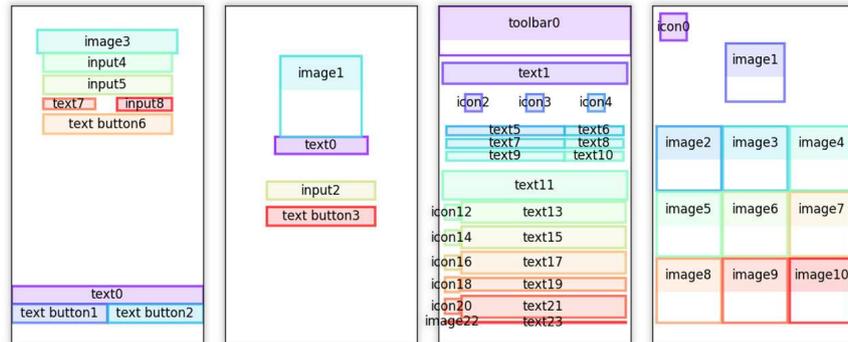


Fig. 15. Example layouts that are considered high-quality based on Enrico and are compatible with our assumptions (no overlap, valid representations as graphs and sequences).

Enrico. Finally, we evaluate the methods on a subset of “good” Rico layouts provided by Enrico [21]. Taking a set of layouts of better quality could reduce the noise in the data and might increase the likelihood of commonalities and overall better layout results. We filter the layouts further to eliminate overlap, resulting in 766 items with element counts between 3 and 38. Examples of these are shown in Figure 15. For completion, the same considerations as in the Rico superset apply.

5.2 Training and test sets

To generate the training data, the set of reference designs are taken and decomposed sequentially according to the sequence representation (i.e., scanned in rows from top-left to bottom-right). For example, a vertical layout with three elements ‘headline’, ‘form’, and ‘button’ below each other would generate the sublayouts (‘headline’), (‘headline’, ‘form’) and (‘headline’, ‘form’, ‘button’). In every sublayout created as such, we generate all possible inputs by removing every element once and marking it as the new element.

This produces an exhaustive number of actual inputs for the methods to be used during training and for testing generalization capabilities. These numbers are shown in Table 2. For the handcrafted data sets ‘Varying buttons’ and ‘Artificial web’, the test set is carefully created to contain the encoded patterns without overlap in the training data. For ‘Enrico’ and ‘Rico (NDN)’, we randomly sample with a fixed seed 10% of the data set as the test set.

5.3 Evaluation metrics

Predictions should follow general layout principles and be consistent with layout patterns in the data set. To measure these, we employ the following metrics.

³Out of 66,261; 24,262 contain more than 9 elements, 8,084 contain only 1 element, 12,121 of the remaining are with overlap, and 237 contain one of the least used components

Data set	Layouts			Queries	
	Train	Test	Elements	Train	Test
Varying buttons	25	11	3–9	355	169
Artificial web	307	52	4–38	41,284	4,998
Enrico	689	77	2–38	34,943	2,652
Rico (NDN)	19,401	2,156	2–9	260,046	25,317

Table 2. Data set statistics. Queries are constructed by decomposing layouts and placing each element in sequentially growing compositions.

Exact and close match. In case a clear placement expectation exists, which is the case for training data and our custom data sets, we compare suggestions to this expectation and measure two levels of matches: exact and close matches as depicted in Figure 22 (appendix). An *exact match* is achieved if the predicted placement matches the expectation in terms of relative positions and alignments to neighboring elements, and the Intersection over Union (IoU) is above a threshold η_{exact} . We define the set of elements that are neighboring elements of e as K_e . An element e_j is considered a neighbor of e_i if you can draw a straight line from any edge of e_i to e_j without crossing any other element e_k . Additionally, the special *canvas* element is always contained in the neighbor set. Since this does not take into account the exact position, we test the IoU between the prediction \hat{b}_e and the expectation b_e^* (see appendix E). We allow for a small offset of a few pixels (keeping alignment and rel. position correct), and require an IoU of $\eta_{\text{exact}} > 0.7$ to be considered an exact match. In practice, this allows, e.g., a small offset of 10 pixels for small elements like buttons. If this check fails, we test for a *close match*, requiring that the placement has a non-insignificant overlap with the expected placement. We consider this if the IoU is above the threshold of $\eta_{\text{close}} > 0.15$.

Valid results, overlap and outside of canvas. Following our premise of the problem and data, we do not expect any overlap (see appendix E). Note that we do not consider overlap to occur if two elements' borders touch each other, as this can be a valid arrangement in graphical layouts. Predictions that are overlapping are considered invalid and are counted as N_{overlap} . Further, if an element's bounding box extends beyond the canvas size, it is considered invalid as well, as counted in N_{outside} . All other cases are considered valid results. The rate of valid results is then the share of the results that did not match the overlap and outside-of-canvas tests of the total number of results.

Alignment. We measure the alignment of the generated element placement to the rest of the layout. For this, we follow a similar approach as described in [19], and calculate the alignment index α as the mean of the horizontal and vertical alignment as the minimum of the distance of any alignment line of the new element e_{new} to any other element e_i in standardized form:

$$\alpha_{e_{\text{new}}} = \frac{1}{2} \min_{e_i} \{ \|\tilde{x}_{e_{\text{new}}}^0 - \tilde{x}_{e_i}^0\|_1, \|\tilde{x}_{e_{\text{new}}}^1 - \tilde{x}_{e_i}^1\|_1, \|\tilde{x}_{e_{\text{new}}}^c - \tilde{x}_{e_i}^c\|_1 \} \\ + \min_{e_i} \{ \|\tilde{y}_{e_{\text{new}}}^0 - \tilde{y}_{e_i}^0\|_1, \|\tilde{y}_{e_{\text{new}}}^1 - \tilde{y}_{e_i}^1\|_1, \|\tilde{y}_{e_{\text{new}}}^c - \tilde{y}_{e_i}^c\|_1 \}. \quad (5)$$

Figure 23 (appendix) shows an example that depicts the minimum differences of both the horizontal and vertical alignment lines to the other elements.

Test queries: Retrieval and Generalization. We divide test queries into two buckets, depending on their (dis-)similarity to the training data. We refer to queries that are contained in the training data as *retrieval queries*. These types of queries can be objectively compared to a ground truth in all data sets. However, in the generic Rico-based sets, there are no systematic layout patterns encoded,

981 and the ground truth may be ambiguous. These queries test the models abilities against a base
982 scenario where no generalization is needed and a simpler search algo rithm would solve these
983 perfectly (excluding ambiguity). They are measured by the *exact* and *close match*. Additionally, we
984 also measure general layout qualities with the *valid rate*, *overlap* and *outside counts*.

985 Queries that are not exact copies of the training data are referred to as *generalization queries*.
986 This is the ultimate target scenario and allows to evaluate whether patterns have been learned or
987 can be constructed correctly. Since the Rico-based sets have no defined patterns, we can only test
988 the general layout qualities *valid rate*, *overlap* and *outside counts*, and we cannot ascertain if the
989 resulting placement is derived from a pattern. However, a good general placement is a necessity
990 for also pattern-based placements and give an indication to how strong the model is overall. In
991 the handcrafted data sets, we have encoded specific layout patterns, such that we can construct
992 new combinations with these patterns. As such, for these data sets, we measure the *exact* and *close*
993 *match* scores as before, along with the general layout measures. This then, allows to evaluate the
994 true pattern-learning ability of a model.

995

996

996 5.4 Quantitative results

997

998

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

1028

1029

Since a single input can resolve to multiple valid results (e.g., there are multiple valid positions for placing a new *button*), we query 3 suggestions for placement for each prediction task. Increasing the number of suggestions could increase the matching scores, but might at the same time affect negatively the alignment scores if more diverse results are generated to account for the higher number. We found 3 suggestions to leave enough room for predicting useful results with ambiguous input without largely distorting results. When testing for a placement match for an input query, we count a match if any of the 3 suggestions results in a match. If an exact match is encountered, it will be counted only as such, and no count for a close match is given for any other suggestion.

To counter the effect of possible invalid results, we run up to 100 trials per input and take the first 3 valid placements. We should note that the larger data sets contain a vast number of queries (as noted in Table 2), and testing every possibility is computationally expensive. Hence, we limit the number of test queries to 1,000 in each of the retrieval and generalization cases, while preserving the distribution of element types, and considering all areas of placements on the canvas equally. Tables 3, 4, 6, and 5 show the quantitative results of our evaluation. The alignment index show the median.

Overall, the kNN method achieves very high combined retrieval match scores on all data sets of more than 95%, and the best combined generalization match scores in the custom data sets of more than 86% with the majority of matches being exact. Only in the first data set, around 20% of generalization matches are only close (absolute 14%), otherwise less than 1% of matches are not exact. The rate of valid results decreases as the layouts become more complex, from around 70% in the simplest ‘varying buttons’ data, to about 25% in the Enrico data set that has the highest number of elements per layout, with the rate being similar for retrieval and generalization queries. Invalid results are mainly due to overlap (between 28% and 58%), placements outside the canvas vary between 5% and 25%. The alignment index (smaller is better) are the lowest between the methods for all data sets with a median of below 0.1 in all cases.

The Transformer model generates valid results in 30–50% of cases. The rate of overlapping elements ranges between 26% and 45%, while placements outside the canvas occur in 20–30% of cases. It achieves high retrieval matching scores of 70–80% of cases except for the Enrico data set, where it is below 30%. The generalization accuracy is similarly high at around 80%. However, the rate of exact matches accounts for only 25–75% of matches. The alignment index are three times the highest between the methods and surpass 0.1 in two cases. For the handcrafted data sets they

Varying buttons	kNN	Transformer	GNN
<i>Retrieval</i>			
Valid	70.1%	46.0%	50.1%
Overlap	28.4%	31.5%	44.2%
Outside	1.5%	22.5%	5.7%
Alignment	.052	.086	.065
Exact match	99.7%	61.2%	0.3%
Close match	0.0%	21.6%	62.9%
Failure	0.3%	17.2%	36.8%
<i>Generalization</i>			
Valid	68.5%	46.2%	46.2%
Overlap	29.2%	35.1%	51.1%
Outside	2.3%	18.7%	2.7%
Alignment	.046	.086	.055
Exact match	72.4%	36.5%	1.1%
Close match	14.4%	43.1%	48.6%
Failure	13.2%	20.4%	50.3%

Table 3. Evaluation results ‘varying buttons’.

Artificial web	kNN	Transformer	GNN
<i>Retrieval</i>			
Valid	37.6%	31.1%	24.8%
Overlap	57.6%	42.0%	64.8%
Outside	4.8%	26.8%	10.4%
Alignment	.000	.032	.035
Exact match	99.9%	45.4%	0.3%
Close match	0.0%	33.7%	20.2%
Failure	0.1%	20.9%	79.5%
<i>Generalization</i>			
Valid	36.0%	27.5%	26.5%
Overlap	58.6%	44.2%	64.4%
Outside	5.4%	28.3%	9.1%
Alignment	.000	.037	.039
Exact match	97.7%	31.9%	0.0%
Close match	0.8%	45.2%	20.6%
Failure	1.5%	22.9%	79.4%

Table 4. Evaluation results for ‘artificial web’.

Rico NDN	kNN	Transformer	GNN
<i>Retrieval</i>			
Valid	34.6%	53.2%	27.9%
Overlap	42.6%	26.8%	66.9%
Outside	22.8%	20.0%	5.2%
Alignment	.087	.096	.123
Exact match	97.8%	19.2%	1.3%
Close match	1.0%	49.8%	13.6%
Failure	1.2%	31.0%	85.1%
<i>Generalization</i>			
Valid	36.0%	53.5%	32.4%
Overlap	39.4%	26.0%	64.0%
Outside	24.6%	20.5%	3.6%
Alignment	.092	.112	.128

Table 5. Evaluation results for ‘Rico (NDN)’

Enrico	kNN	Transformer	GNN
<i>Retrieval</i>			
Valid	25.8%	35.6%	19.5%
Overlap	53.3%	44.7%	76.3%
Outside	20.9%	19.7%	4.2%
Alignment	.056	.096	.067
Exact match	98.4%	8.3%	0.0%
Close match	1.0%	29.5%	9.0%
Failure	0.6%	62.2%	91.0%
<i>Generalization</i>			
Valid	23.9%	34.1%	25.4%
Overlap	55.4%	38.0%	68.3%
Outside	20.7%	27.9%	6.3%
Alignment	.084	.112	.104

Table 6. Evaluation results for ‘Enrico’.

are similar between the retrieval and generalization queries, while for the mobile layouts Enrico and Rico, there is an visible increase between the conditions.

The GNN approach produces only 20–50% valid results on the different data sets and often produces overlapping placements. While for the smallest and simplest data set ‘varying buttons’ it closely matches patterns with decent rates of 63% in the retrieval case and 50% in the generalization case, in the more complex and larger data sets, it drops to less than 10–20% and around 90% unmatched patterns. There are hardly any exact matches, only around 1%. The alignment index varies widely and is very close to the kNN score on the simple data sets but increases to the highest score for the ‘Rico NDN’ data set where the median is 14–40% above the alignment index of the other methods.

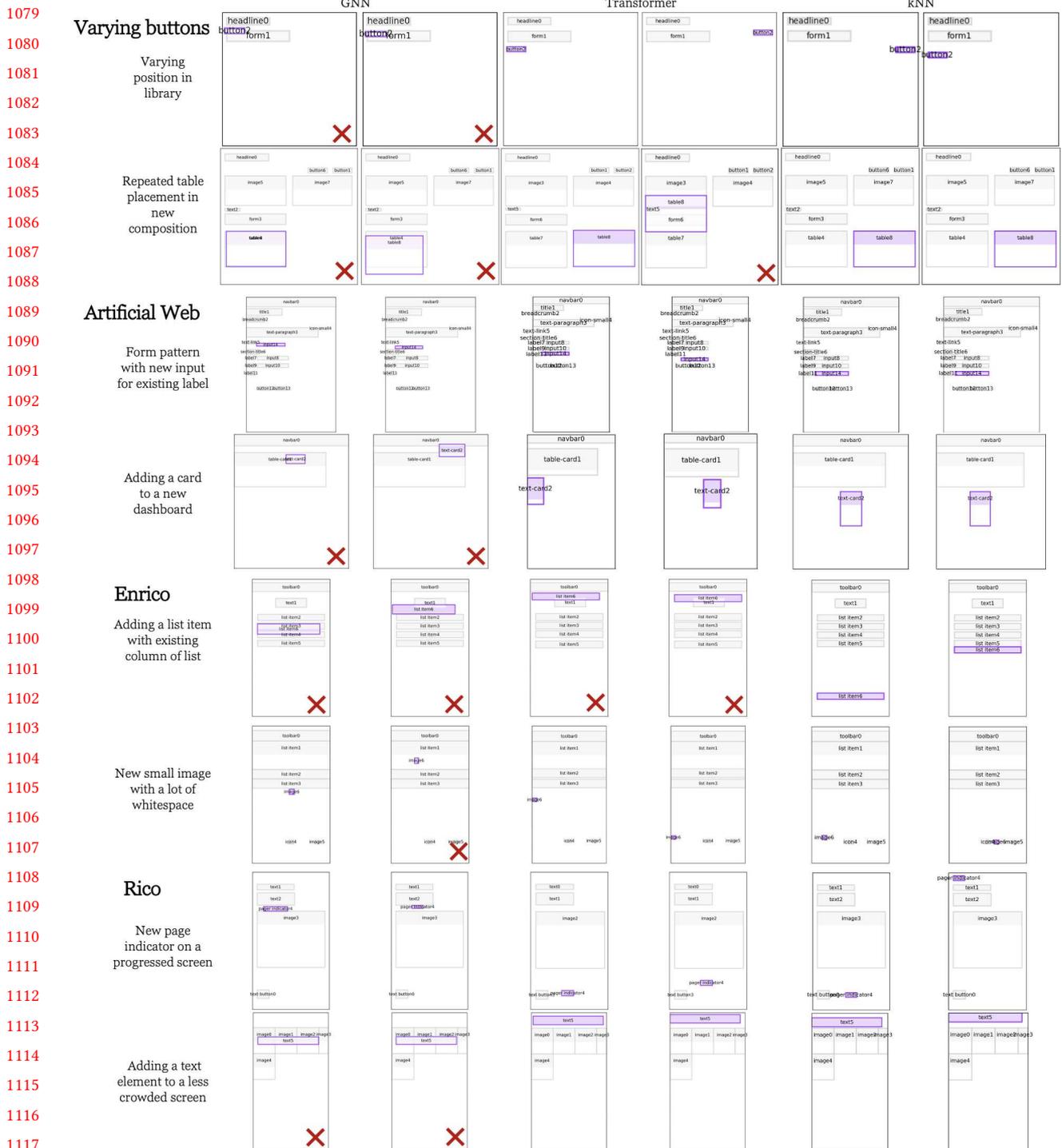


Fig. 16. Qualitative results for all data sets with one retrieval and one generalization query with two results per method. Invalid results are marked with a red cross.

5.5 Qualitative inspection

To better understand the capabilities of each method, we show results for all data sets and contrast them with expectations. For each data set, one retrieval query and one generalization query is used and two results per methods are shown in Figure 16.

1128 *Varying buttons.* This data set contains similar placements for most elements except for buttons
1129 that can have multiple valid locations. If two elements of the same types are present, they should be
1130 located next to each other. For the retrieval query, with a varying position in the training set of the
1131 target element, the results are quite different. The GNN method tries to squeeze the button between
1132 headline and form which also exists in a similar composition (although then there is more space
1133 between those elements) and produces overlapping placements. The transformer model suggests
1134 a left-aligned placement and a placement on the right side, which is, however, not following the
1135 pattern of being below the form. The kNN model suggests two placements according to neighbors,
1136 at the right side of the canvas and below the form, which matches different training elements. For
1137 the generalization query, there is only really a single placement possible and valid. As all methods
1138 converge to the same result, only a single variation is shown. The GNN method puts the new table
1139 at the position of the existing table, resulting in an invalid overlapping placement. The transformer
1140 model and the kNN method return the same position at the expected position.

1141 *Artificial web layouts.* This data set encodes three layout patterns (Table + Button, Form arrange-
1142 ment, and Dashboard layout) which are present in many different contexts. For the retrieval query,
1143 where a small form is already in the layout, one row is with a label but without an input and a
1144 new input should be added. We expect it to be placed next to the label so it aligns well with the
1145 other form rows. The GNN model places the input above the form in both variations, and although
1146 rather aligned, it is not a sensible result in this context. The Transformer suggests the expected
1147 placement next to the label and another one in a new row. The kNN method only suggests the
1148 expected position. For the generalization query, where a new card is to be added to a rather empty
1149 dashboard layout, we expect that it is placed in the second row, and ideally at the start of the
1150 row. The GNN method fails to predict a valid result and shrinks the element unexpectedly. The
1151 transformer model returns two valid positions that both are in line with our expectation, where
1152 one is also placed at the start of the row. The kNN model suggests placements in the new row but
1153 does not return a placement at the start of the row.

1154 *Enrico.* The mobile layouts based on Enrico's data set features mobile layouts with element
1155 counts ranging from 2 to 38. As there are no known layout patterns encoded, we can only evaluate
1156 if the generated placements are generally producing well-formed layouts and are valid. In the
1157 retrieval query, a set of list items is present in the layout and another one should be added. The GNN
1158 again fails to present valid non-overlapping results. The Transformer method suggests placements
1159 towards the top of the screen that results in overlaps with existing elements. The kNN approach
1160 returns two placements below the existing list items, and even though the gaps with the previous
1161 list item is not the same as with the other list items, we consider it well-formed suggestions. In the
1162 generalization query, a small image element should be added. The top of the screen is occupied
1163 by existing elements and at the bottom are two other small images and icons. The GNN method
1164 predicts one valid result where the image is centered below the list items and one invalid result
1165 where the image is overlapping with the existing elements. The transformer model shows two
1166 results where the image element is at the edge of the screen, and both can be considered good
1167 suggestions. The kNN model also predicts two valid placements, one where it is well-spaced with
1168 the other small images at the bottom, and one where it is placed between the two small images.
1169 Both can be considered good suggestions.

1170 *Rico (NDN).* In this data set, the models have a much larger set of training items available where
1171 all layouts have less than 10 elements per page. In the retrieval query, four elements that span the
1172 whole screen already exist in the layout and a pager indicator element is to be added. The GNN
1173 approach places the pager indicator between the text and image elements without alignment. The
1174 Transformer model predicts results where the new element is towards the bottom of the screen
1175 and horizontally centered, but with two different y-coordinates. The first result is aligned with
1176

1177 the bottom text button, while the second result is only center aligned with the screen. The kNN
1178 method places the pager indicator also either centered at the bottom of the screen, or at the top,
1179 and left-aligned. Both results can be considered good suggestions, thus. In the generalization query,
1180 five image elements are placed at the top of the screen and a new text element is to be added. The
1181 GNN method returns again two invalid positions that overlap with other elements. The transformer
1182 model predicts placements at the top of the screen, above the images, where one result is centered,
1183 and the other left-aligned. The kNN model shows similar results and places the text element above
1184 the images but keeps it left-aligned in both cases. Both the Transformer and the kNN results can be
1185 considered good suggestions.

1186

1187 6 DISCUSSION AND LIMITATIONS

1188 *Layout model and representations.* We focused on a simple layout model requiring layouts to be
1189 flat (i.e., no hierarchies) and their elements to be non-overlapping. While this does not fully reflect
1190 complex UI layouts, it is expressive enough to be a first model for evaluation, and simplify the task
1191 by eliminating one additional dimension. This simplified model should be solved first before adding
1192 more complexity. To add support for more complex layouts in the GNN, one might add nesting
1193 relations (i.e., ‘inside of’, ‘contains’) to the graph vocabulary or create a tree graph and process the
1194 outer and inner layers separately. For the sequential models (Transformer and kNN), Li et al. [23]
1195 evaluated different tree representations that could be applicable here as well.

1196 Both graph and transformer layout representations contain assumptions that may not always be
1197 valid. In the graph representation, for example, the specific relation categories can greatly affect the
1198 resulting layouts and the expressiveness of the graph representation, and we did not test greatly
1199 varying categorization schemes. Defining relations to the canvas is especially difficult as it can be
1200 unclear whether proximity to an edge is a design decision or an implicit result of a packed layout.

1201 Similarly, the sequential decomposition follows a simple reading order without taking into
1202 consideration nested columns or segments of the layout (e.g., as indicated with whitespace). This is
1203 a difficult problem with existing research on segmenting layouts from visual representations [1].
1204 Encoded hierarchies can create better sequences, taking into account columns and segments, similar
1205 to the tree representation by Li et al. [23].

1206

1207 *Evaluation.* Requiring a robust and consistent set of layouts is a major obstacle for further
1208 research in this particular area, which may also impact a possible user study. Our main limitation
1209 with regards to such an evaluation is the lack of annotated user feedback that could be used as
1210 reference (or ground-truth) data.

1211 Compared to previous research, we are the first to introduce systematic layout pattern evaluation.
1212 The previous reliance on composite data sets prevents more rigorous studying of layout completion
1213 capabilities, beyond generic goodness qualities. We argue that this is an important characteristic
1214 when evaluating methods for UI design. Certainly, there are general layout principles that indicate
1215 if a layout can be considered well-formed, such as alignment, however, layout assistance tools
1216 should consider the local context (i.e., other elements) as well as the global context (i.e., other
1217 designs) and thus, become more task-aware to better model real-world design scenarios.

1218 We focused on a data-driven evaluation and validated the methods by analyzing layout quality
1219 (both quantitatively and qualitatively) and comparing the results to expected layout patterns where
1220 applicable. However, a user-centered evaluation with professional UI designers is the next logical
1221 step to further validate our approach. Such an evaluation is rather challenging, as it requires
1222 designers to be familiar with a set of reference layouts and the patterns therein, besides a strong
1223 knowledge of user interface design in general. Critically, designer’s acceptance towards this kind of
1224 assistance is dependent (1) on the quality of the suggestions and (2) how well they are integrated

1225

1226 with the designer’s software and workflow. As discussed previously, our work addresses the former
1227 case but has not considered the latter case, for which a substantial engineering effort is needed to
1228 properly instrument existing design tools. In general, the positive applicability of similar design
1229 suggestions has been shown several times before [4, 18, 37], therefore we believe that similar
1230 findings would be observed in our case as well.

1231 *Practical implementation.* Connecting with the previous discussion, our interaction method as a
1232 Sketch plugin may not be ideal but we believe it is a good compromise solution. A custom research
1233 prototype, developed from scratch, would allow us to implement better interaction methods to
1234 display the suggested placement candidates, but then UI designers would need to learn how to use
1235 the prototype instead of their own software. Still, a better plugin interface for Sketch would be
1236 possible and could be improved in a subsequent version.

1238 *Graph neural network.* We found that the graph neural network following the description of
1239 the “Neural Design Network” [19] achieves the lowest scores among the tested methods across
1240 all data sets. Especially problematic is the high number of invalid results due to overlapping with
1241 other elements. Further, it has the lowest pattern matching scores, indicating that it is not learning
1242 the design patterns as expected. Its ability to produce diverse results was limited, and the model
1243 seemed to have converged to a small region. While high model creativity is not required, there
1244 are often multiple valid placements that would be appreciated if returned. Thus, even though the
1245 graph representation is a natural and useful format for layouts, our implementation is unable to
1246 produce high-quality results most of the time.

1247 Certainly, there are many parameters and details in this method and it cannot be excluded that
1248 differences to previous descriptions are responsible for these low scores. Further, our restrictions
1249 on the problem, requiring non-overlapping results, and performing single-element predictions
1250 might require more substantial changes to the previously proposed model. We tested different
1251 variations to improve the results, but we did not achieve significantly better scores. For example,
1252 we tested using the relation module to produce constraints for a combinatorial optimization system,
1253 however, the predicted edges were often not consistent with each other such that no feasible result
1254 was possible. These conflicting relations might also be a reason for the layout module to produce
1255 overlapping placements. We found that when a consistent set of relations is given directly to the
1256 layout module, the results improved significantly in some cases.

1257 *Transformer model.* The transformer model following [10] produced acceptable results overall.
1258 It was able to produce close results for most patterns in the custom data sets and generated well-
1259 aligned results in many cases. Modeling element coordinates as categories instead of a continuous
1260 variable allows the model to learn alignment lines of the layouts. That has the disadvantage,
1261 however, that vertically shifted patterns (e.g., due to additional elements in the layout) might not
1262 result in equally shifted predictions. While the results are reasonable even for our small data set,
1263 the results are not consistent and robust enough to be fully usable in practical applications. The
1264 qualitative results show that for new compositions, many predictions are invalid, especially if the
1265 layout is already crowded and only a particular empty space would be available. Since there is no
1266 explicit understanding of the visual nature of the layout, these layout ‘holes’ are not recognized.

1268 *Sequence-aligned nearest neighbor search.* The custom nearest neighbor method was able to
1269 predict layout patterns with the best scores, as it explicitly leverages the reference designs that
1270 contain these patterns. However, as it does not learn generalizations and takes a single neighbor
1271 as the reference, it is not able to use information from multiple designs to inform the prediction.
1272 Certainly, one possible extension is to take into account multiple neighbors for a single input layout.
1273 However, since element placement is based on the neighborhood relations of the returned neighbor,
1274

multiple neighbors might return conflicting relation categories. Already with a single neighbor, our approach can lead to failures when the neighborhood is sparsely populated, or the relevant reference element for placement is not in the direct vicinity. A more sophisticated placement module could further improve this approach.

While the instance-based learning approach does not require a large set of data points to be useful, the set must be comprehensive and representative of the different types and layout compositions that designers might create, otherwise the generalization capabilities would be limited. On the other hand, its runtime complexity is unfavorable, rendering this approach unsuitable for very large data sets: Running on a recent MacBook Pro⁴, predictions for Enrico with around 700 elements took usually less than 1 second, while querying the larger Rico data set with over 19.000 elements took between 10-40 seconds, depending on the target layout size. The biggest bottleneck in this regard is the enumeration of reference layouts to align each with the input and produce adjusted features. Intelligent filtering of promising candidates could alleviate this issue and improve its runtime. Further, it has a limited ability to produce very diverse results and the number of overlapping results increases with the complexity of the layouts. Since our sequence alignment algorithm does not consider the final placement (as this is too costly to perform on all candidates in the design library), there is a certain chance of generating neighbors that cannot be applied to the current input.

Nevertheless, a major advantage of this method is that it is applicable to many design data sets found in commercial settings, which are not particularly large. The second benefit of this method is its high interpretability. Since a result is based on a particular neighbor, the outcome can be explained to the user; e.g. “this element is recommended to be placed here because the system found an existing design layout with the same element in a similar position”. Additionally, unexpected or wrong behaviors can be easily investigated. This is in contrast to the neural network-based approaches which are inherently difficult to interpret as they learn an abstract mathematical model of the underlying data set. While a lot of progress has been made to make neural networks more understandable, interpreting the results often requires a solid understanding of machine learning and neural networks.

1303

1304 7 CONCLUSION AND FUTURE WORK

We evaluated three methods for layout completion with element constraints. We focused on a practical application for commercial designers by giving them control over the generation process and having them decide the type of the next element to be added. Following a real-world scenario for a single design system, we target layout patterns exhibited in a set of reference designs that are expected to be adhered to when generating new element placements. We modeled these layout patterns as positional and alignment relations of element pairs.

Our graph neural network mostly failed to produce high-quality results and did not learn fine-grained patterns. The layout transformer learned many encoded patterns but its generalization is limited due to its inability to transfer patterns to different areas. Our custom nearest neighbor search finds insertion points via a sequence alignment algorithm and encodes layout principles in the feature generation to emphasize local similarity over global matches. The results showed that layout patterns are returned with high accuracy, and it achieves satisfying alignment scores overall. Its disadvantages are its limited scalability and limitations of the sequential decomposition and corresponding alignment. Nevertheless, its interpretability and high overall scores make it most suitable for practical use of professional designers.

1320

1321

1322 ⁴MacBook Pro 16", 2019, 2,6 GHz 6-Core Intel Core i7, 16 GB RAM

1323

1324 Future extensions of this research should address our restriction on flat layouts, and expand the
 1325 methods to support more complex and hierarchical designs as described in section 6. Further, while
 1326 all methods can benefit from improvements, extending our kNN approach appears most promising
 1327 by tuning the sequence alignment algorithm and improving the placement strategy. While we
 1328 aimed for a rigorous, data-driven evaluation, we did not conduct a user study. Future work should
 1329 conduct a user evaluation to further validate our experimental results. Our analysis showed that,
 1330 even today, classic machine learning with principled methods for feature generation can be most
 1331 suitable for practical applications in commercial environments where consistency of results and
 1332 interpretability are important factors. With this work, we have contributed to the understanding of
 1333 the practical applicability of different machine learning methods for design assistance tools, which
 1334 ultimately will help companies to improve consistency in their GUI designs.

1336 ACKNOWLEDGMENTS

1337 We thank all our colleagues and reviewers for their helpful comments. This work was funded by
 1338 the Department of Communications and Networking (Comnet), the Finnish Center for Artificial
 1339 Intelligence (FCAI), Academy of Finland projects ‘Human Automata’ and ‘BAD’, HumaneAI Net
 1340 (H2020 ICT 48 Network of Centers of Excellence), and Huawei Technologies. ;

1342 REFERENCES

- 1344 [1] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. Extracting Content Structure for Web Pages Based on
 1345 Visual Representation. In *Web Technologies and Applications*, Xiaofang Zhou, Maria E. Orlowska, and Yanchun Zhang
 1346 (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–417.
- 1347 [2] Niranjan Damera-Venkata, José Bento, and Eamonn O’Brien-Strain. 2011. Probabilistic Document Model for Automated
 1348 Document Composition. In *Proceedings of the 11th ACM Symposium on Document Engineering (Mountain View, California, USA) (DocEng ’11)*. Association for Computing Machinery, New York, NY, USA, 3–12.
- 1349 [3] Niraj Ramesh Dayama, Simo Santala, Lukas Brückner, Kashyap Todi, Jingzhou Du, and Antti Oulasvirta. 2021. Interactive
 1350 Layout Transfer. In *26th International Conference on Intelligent User Interfaces (College Station, TX, USA) (IUI ’21)*.
 1351 Association for Computing Machinery, New York, NY, USA, 70–80. <https://doi.org/10.1145/3397481.3450652>
- 1352 [4] Niraj Ramesh Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. 2020. GRIDS: Interactive Layout Design
 1353 with Integer Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (Honolulu, HI, USA) (CHI ’20)*. Association for Computing Machinery, New York, NY, USA, 1–13. <https://doi.org/10.1145/3313831.3376553>
- 1354 [5] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschan, Daniel Afegan, Yang Li, Jeffrey Nichols, and Ranjitha
 1355 Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th
 1356 Annual ACM Symposium on User Interface Software and Technology (Québec City, QC, Canada) (UIST ’17)*. Association
 1357 for Computing Machinery, New York, NY, USA, 845–854. <https://doi.org/10.1145/3126594.3126651>
- 1358 [6] Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: Automatically Generating User Interfaces. In *Proceedings of the 9th
 1359 International Conference on Intelligent User Interfaces (Funchal, Madeira, Portugal) (IUI ’04)*. Association for Computing
 1360 Machinery, New York, NY, USA, 93–100. <https://doi.org/10.1145/964442.964461>
- 1361 [7] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically generating personalized user interfaces
 1362 with Supple. *Artificial Intelligence* 174, 12 (2010), 910 – 950.
- 1363 [8] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically generating user interfaces adapted
 1364 to users’ motor and vision capabilities. In *UIST ’07: Proceedings of the 20th annual ACM symposium on User interface
 1365 software and technology*. ACM Press, New York, NY, USA, 231–240.
- 1366 [9] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for
 1367 157 Languages. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*.
 European Language Resources Association (ELRA), Miyazaki, Japan. <https://www.aclweb.org/anthology/L18-1550>
- 1368 [10] Kamal Gupta, Alessandro Achille, Justin Lazarow, Larry Davis, Vijay Mahadevan, and Abhinav Shrivastava. 2020.
 1369 Layout Generation and Completion with Self-attention. *CoRR* abs/2006.14615 (2020). arXiv:2006.14615 <https://arxiv.org/abs/2006.14615>
- 1370 [11] Stephen M Hart and Liu Yi-Hsin. 1995. The application of integer linear programming to the implementation of a
 1371 graphical user interface: a new rectangular packing problem. *Applied mathematical modelling* 19, 4 (1995), 244–254.

- 1373 [12] Bruce Hilliard, Jocelyn Armarego, and Tanya McGill. 2016. Optimising visual layout for training and learning
1374 technologies. In *Proceedings of the 27th Australasian Conference on Information Systems, ACIS 2016*. University of
1375 Wollongong, Faculty of Business.
- 1376 [13] D. S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18,
1377 6 (June 1975), 341–343.
- 1378 [14] Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image Generation from Scene Graphs. *CoRR* abs/1804.01622 (2018).
1379 arXiv:1804.01622 <http://arxiv.org/abs/1804.01622>
- 1380 [15] Akash Abdu Jyothi, Thibaut Durand, Jiawei He, Leonid Sigal, and Greg Mori. 2019. LayoutVAE: Stochastic Scene
1381 Layout Generation from a Label Set. *CoRR* abs/1907.10719 (2019).
- 1382 [16] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-Based Retargeting
1383 for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC,
1384 Canada) (*CHI '11*). Association for Computing Machinery, New York, NY, USA, 2197–2206. [https://doi.org/10.1145/](https://doi.org/10.1145/1978942.1979262)
1385 1978942.1979262
- 1386 [17] Markku Laine, Ai Nakajima, Niraj Dayama, and Antti Oulasvirta. 2020. Layout as a Service (LaaS): A Service Platform
1387 for Self-Optimizing Web Layouts. In *Web Engineering*, Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso (Eds.).
1388 Springer International Publishing, Cham, 19–26.
- 1389 [18] Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. 2020.
1390 GUIComp: A GUI Design Assistant with Real-Time, Multi-Faceted Feedback. In *Proceedings of the 2020 CHI Conference*
1391 *on Human Factors in Computing Systems* (Honolulu, HI, USA) (*CHI '20*). Association for Computing Machinery, New
1392 York, NY, USA, 1–13. <https://doi.org/10.1145/3313831.3376327>
- 1393 [19] Hsin-Ying Lee, Weilong Yang, Lu Jiang, Madison Le, Irfan Essa, Haifeng Gong, and Ming-Hsuan Yang. 2019. Neural
1394 Design Network: Graphic Layout Generation with Constraints. *CoRR* abs/1912.09421 (2019). arXiv:1912.09421
1395 <http://arxiv.org/abs/1912.09421>
- 1396 [20] Luis Leiva. 2012. Interaction-Based User Interface Redesign. In *Proceedings of the 2012 ACM International Conference*
1397 *on Intelligent User Interfaces* (Lisbon, Portugal) (*IUI '12*). Association for Computing Machinery, New York, NY, USA,
1398 311–312. <https://doi.org/10.1145/2166966.2167028>
- 1399 [21] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico: A Dataset for Topic Modeling of Mobile UI Designs. In
1400 *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services, Oldenburg, Germany,*
1401 *October 5–8, 2020 (MobileHCI '20)*. ACM, New York, NY, USA, 7.
- 1402 [22] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2019. LayoutGAN: Generating Graphic
1403 Layouts with Wireframe Discriminators. In *7th International Conference on Learning Representations, ICLR 2019, New*
1404 *Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. <https://openreview.net/forum?id=HJxB5sRcFQ>
- 1405 [23] Yang Li, Julien Amelot, Xin Zhou, Samy Bengio, and Si Si. 2020. Auto Completion of User Interface Layout Design
1406 Using Transformer-Based Tree Decoders. *ArXiv* (2020), 1–11.
- 1407 [24] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on*
1408 *Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. [https://openreview.net/](https://openreview.net/forum?id=Bkg6RiCqY7)
1409 [forum?id=Bkg6RiCqY7](https://openreview.net/forum?id=Bkg6RiCqY7)
- 1410 [25] Jonas Löwgren and Ulrika Laurén. 1993. Supporting the use of guidelines and style guides in professional user interface
1411 design. *Interacting with Computers* 5, 4 (1993), 385 – 396. [https://doi.org/10.1016/0953-5438\(93\)90003-C](https://doi.org/10.1016/0953-5438(93)90003-C)
- 1412 [26] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. 2019. When does label smoothing help?. In *Advances in Neural*
1413 *Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.),
1414 Vol. 32. Curran Associates, Inc. [https://proceedings.neurips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-](https://proceedings.neurips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-Paper.pdf)
1415 [Paper.pdf](https://proceedings.neurips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-Paper.pdf)
- 1416 [27] Gene Myers. 1999. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *J.*
1417 *ACM* 46, 3 (May 1999), 395–415.
- 1418 [28] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2014. Learning layouts for single-page graphic designs.
1419 *IEEE Transactions on Visualization and Computer Graphics* 20, 8 (2014), 1200–1213.
- 1420 [29] Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer. 2020. Combi-
1421 natorial Optimization of Graphical User Interface Designs. *Proc. IEEE* 108, 3 (2020), 434–464.
- [30] A. Ant Ozok and Gavriel Salvendy. 2000. Measuring consistency of web page design and its effects on performance
and satisfaction. *Ergonomics* 43, 4 (2000), 443–460.
- [31] Andreas Pfeiffer. 2018. *Creativity and technology in the age of AI*. Technical Report. Pfeiffer Consulting. Retrieved
18.09.2020 from <http://www.pfeifferreport.com/essays/creativity-and-technology-in-the-age-of-ai/>
- [32] Simo Santala. 2020. *Optimising User Interface Layouts for Design System Compliance*. Master's thesis. Aalto University.
- [33] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. 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 (Honolulu, HI, USA) (*CHI '20*). Association for Computing Machinery, New York,

- 1422 NY, USA, 1–13. <https://doi.org/10.1145/3313831.3376593>
- 1423 [34] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception
1424 Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE
1425 Computer Society, Los Alamitos, CA, USA, 2818–2826. <https://doi.org/10.1109/CVPR.2016.308>
- 1426 [35] Sou Tabata, Hiroki Yoshihara, Haruka Maeda, and Kei Yokoyama. 2019. Automatic Layout Generation for Graphical
1427 Design Magazines. In *ACM SIGGRAPH 2019 Posters* (Los Angeles, California) (*SIGGRAPH '19*). Association for Computing
1428 Machinery, New York, NY, USA, Article 9, 2 pages.
- 1429 [36] Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. 2012. Learning Design
1430 Patterns with Bayesian Grammar Induction. In *Proceedings of the 25th Annual ACM Symposium on User Interface
1431 Software and Technology* (Cambridge, Massachusetts, USA) (*UIST '12*). Association for Computing Machinery, New
1432 York, NY, USA, 63–74. <https://doi.org/10.1145/2380116.2380127>
- 1433 [37] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. 2016. Sketchplore: Sketch and explore layout designs with an optimiser.
1434 *Conference on Human Factors in Computing Systems - Proceedings 07-12-May- (2016)*, 3780–3783.
- 1435 [38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia
1436 Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V.
1437 Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.
1438 <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>
- 1439 [39] Pengfei Xu, Hongbo Fu, Takeo Igarashi, and Chiew-Lan Tai. 2014. Global Beautification of Layouts with Interactive
1440 Ambiguity Resolution. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*
1441 (Honolulu, Hawaii, USA) (*UIST '14*). Association for Computing Machinery, New York, NY, USA, 243–252. <https://doi.org/10.1145/2642918.2647398>
- 1442 [40] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson W. H. Lau. 2019. Content-Aware Generative Modeling of Graphic
1443 Design Layouts. *ACM Trans. Graph.* 38, 4, Article 133 (July 2019), 15 pages. <https://doi.org/10.1145/3306346.3322971>

1471 A GRAPH NEURAL NETWORK DETAILS

1472 A.1 Relation prediction

1473 The input to the relation module Rel is the partial graph G^P where edges to and from the new
 1474 element e_{new} have the special label *unknown*, and a complete graph \hat{G} is generated that contains
 1475 label predictions for the previously unknown edges: $\hat{G} = \text{Rel}(G^P)$. In this module, the bounding
 1476 boxes of the elements are not used as part of the features of the nodes. Depending on the set of
 1477 training layouts, there might be cases for which the same partial input graph can have multiple
 1478 different valid output graphs. That is why we condition the graph completion task on a learned
 1479 latent variable z_{rel} that allows differentiating between multiple variations of the same input. The
 1480 embedded vertices and arrows F_V, F_A of the input graph of dimension D_{embed} are joined by a latent
 1481 vector z_{rel} of dimension $D_{z_{\text{rel}}}$. During inference, z_{rel} is sampled from a standard normal distribution
 1482 $z_{\text{rel}} \sim \mathcal{N}(0, 1)$, while during training, it is encoded from the ground truth graph $z_{\text{rel}} = \text{Enc}(G^*)$ to
 1483 allow reconstruction of different source graphs from the same input. These joined feature vectors
 1484 are the inputs to the actual graph convolution network GCN_{pred} . It produces updated feature vectors
 1485 for both nodes and edges. Finally, the convoluted feature vectors of the edges are fed into a multilayer
 1486 perceptron network MLP_{pred} that performs a multi-class classification task on each input vector,
 1487 and returns a probability vector on the relation categories for each edge:
 1488

$$1489 F'_V, F'_A = \text{GCN}_{\text{pred}}((F_V, z_{\text{rel}}), (F_A, z_{\text{rel}})), \quad (6)$$

$$1490 \hat{p}_{A_r} = \sigma(\text{MLP}_{\text{pred}}(F'_A)), \quad (7)$$

1492 where $\hat{p}_{r_i} \in \hat{p}_{A_r}$ are the probability vectors for the label of every edge as returned by a softmax
 1493 activation function σ . The output is then an $|R|$ -dimensional vector that assigns a specific score
 1494 to each possible label category for every input edge. The predicted labels $\{\hat{r}_i\}$ are computed by
 1495 the arg max on these vectors. By replacing the unknown edge labels from the partial graph with
 1496 the predicted labels from the network, a complete graph \hat{G} is generated that can then be used in a
 1497 subsequent layouting module. Note that the relation prediction are learned independently for the
 1498 two relation categories. Hence, we create two models $\text{Rel}^{\text{pos}}, \text{Rel}^{\text{align}}$, one for each relation type.

1499 *Latent vector encoding.* The latent vector of dimension $D_{z_{\text{rel}}}$ is encoded from the ground truth
 1500 graph G^* . Embedded graph features are passed into a graph convolution network $F'_{V^*}, F'_{A^*} =$
 1501 $\text{GCN}_{\text{enc}}(F_{V^*}, F_{A^*})$, which are then fed into a two-headed dense neural network MLP_{enc} to generate
 1502 μ, σ , from which the latent vector is generated via the reparameterization trick $z_{\text{enc}} = \mu + \sigma\epsilon$, where
 1503 ϵ is a random noise variable sampled from $\mathcal{N}(0, 1)$.
 1504

1505 *Loss.* The network is trained with a reconstruction loss \mathcal{L}_{rec} on the category prediction, and
 1506 an entropy loss $\mathcal{L}_{\text{kl}_1}$ between the generated latent vectors z_{rel} and the prior distribution $\mathcal{N}(0, 1)$:
 1507 $\mathcal{L}_{\text{rec}_1} = \text{CE}(\{\hat{r}_i\}, \{r^*\}, w)$, where $\text{CE} = -\sum_n^{|R|} r_n^* \log \hat{r}_n$ is the cross-entropy function, $\{\hat{r}_i\}$ the set of
 1508 predicted relation categories, $\{r^*\}$ the set of true relation categories, and w is a weight parameter
 1509 to account for imbalanced data sets.

1510 $\mathcal{L}_{\text{kl}_1} = \text{KL}(z_{\text{rel}}, \mathcal{N}(0, 1))$, where KL is the Kullback-Leiber divergence function, z_{rel} the encoded
 1511 latent variable, and $\mathcal{N}(0, 1)$ the standard normal distribution. Both losses are summed with weights
 1512 $\lambda_{\text{rec}_1}, \lambda_{\text{kl}_1}$ to form the complete loss for the relation module:
 1513

$$1514 \mathcal{L}_{\text{rel}} = \lambda_{\text{rec}_1} \mathcal{L}_{\text{rec}_1} + \lambda_{\text{kl}_1} \mathcal{L}_{\text{kl}_1}. \quad (8)$$

1515 A.2 Layout generation

1517 In the layout generation module, the completed graph $\hat{G} = (V, \hat{A}^{\text{pos}}, \hat{A}^{\text{align}})$ along with the elements
 1518 size boxes $\{s_{e_i}\}$ are used to predict the size box of the new element $\hat{s}_{e_{\text{new}}}$, completing the layout.
 1519

We do not need to work with the two relation types separately in the layout module, so the edges are merged and the vocabulary is combined. In the original description of the method, an iterative process was used to generate the boxes for new elements, to support adding multiple new elements. To be consistent with the stated problem, we only consider a single new element and employ a single prediction step. To do so, first, a learned graph representation is generated by passing the embedded features through a graph convolution network:

$$F'_V, F'_A = \text{GCN}_{\text{lay1}}(F_V, F_A). \quad (9)$$

To generate a placement for a new element, the existing standardized size boxes $\{\tilde{s}_{e_i}\}$ are concatenated onto the feature vectors of the nodes, and onto the new element e_{new} , the half-empty size box is concatenated $\tilde{s}_{e_{\text{new}}} = (\emptyset, \emptyset, \tilde{w}_{e_{\text{new}}}, \tilde{h}_{e_{\text{new}}})$: $F''_V = (F'_V, \{\tilde{s}_{e_i}\})$. To adjust the dimensionality of the edge vectors F_A to match the node vectors, a 4-dimensional vector of zeros is concatenated onto them $F''_A = (F'_A, (0, 0, 0, 0))$. These features are passed through another graph convolution network GCN_{lay2} , and a variational auto encoder-decoder network of fully-connected layers $h_s^{\text{enc}}, h_s^{\text{dec}}$ generates the box prediction from the new node feature of the new element e_{new} :

$$F'''_V, F'''_A = \text{GCN}_{\text{lay2}}(F''_V, F''_A), \quad (10)$$

$$z_{\text{lay}} = h_s^{\text{enc}}(s_{e_{\text{new}}}^*), \quad (11)$$

$$\hat{s}_{e_{\text{new}}} = h_s^{\text{dec}}(F'''_{e_{\text{new}}}, z_{\text{lay}}), \quad (12)$$

where $s_{e_{\text{new}}}^*$ is the ground truth box of the new element, which is used during training to generate the latent code from. During inference, the latent code z_{lay} is sampled from a prior distribution.

Loss. The network is trained with a reconstruction loss $\mathcal{L}_{\text{rec}_2}$ on the size box differences, and an entropy loss $\mathcal{L}_{\text{kl}_2}$ between the generated latent vectors z_{lay} and the prior distribution $\mathcal{N}(0, 1)$. Additionally, to force keeping the input sizes of the elements, a size reconstruction loss $\mathcal{L}_{\text{size}}$ is added: $\mathcal{L}_{\text{rec}_2} = \|\hat{s}_i - s_i^*\|_1$, where $\|\cdot\|_1$ is the L1-loss between the predicted box and the ground truth box s^* , $\mathcal{L}_{\text{size}} = \|(\hat{w}, \hat{h}) - (w^*, h^*)\|_1$, where w, h represent the widths and heights of new elements, and $\mathcal{L}_{\text{kl}_2} = \text{KL}(z_{\text{lay}}, \mathcal{N}(0, 1))$, where KL is the Kullback-Leiber divergence function, z_{lay} the encoded latent variable, and $\mathcal{N}(0, 1)$ the standard normal distribution. As before, the losses are summed with weights to form the complete layout loss:

$$\mathcal{L}_{\text{lay}} = \lambda_{\text{rec}_2} \mathcal{L}_{\text{rec}_2} + \lambda_{\text{size}} \mathcal{L}_{\text{size}} + \lambda_{\text{kl}_2} \mathcal{L}_{\text{kl}_2}. \quad (13)$$

A.3 Refinement module

The final refinement module operates on the generated complete layout with the goal of fine-tuning the previous result and producing a more aesthetically pleasing layout. Its input is the completed graph $\hat{G} = (V, \hat{A})$, along with the size boxes of the existing layout $\{s_{e_i}\}$ and the size box of the new element $\hat{s}_{e_{\text{new}}}$, and it produces an updated size box $\hat{s}'_{e_{\text{new}}}$. The completed layout from the previous model is concatenated onto the embedded features of the nodes, and the result is convoluted in a graph convolution network $\text{GCN}_{\text{refine}}$. The updated feature vector of the new element is fed into a multi-layer perceptron network $\text{MLP}_{\text{refine}}$ to generate the refined size box $\hat{s}'_{e_{\text{new}}}$:

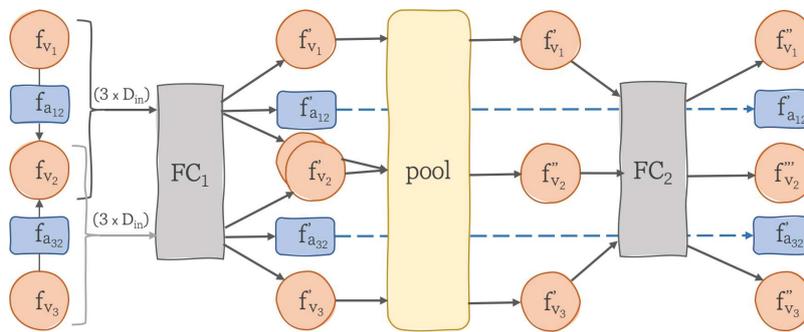
$$F'_V, F'_A = \text{GCN}_{\text{refine}}((F_V, \{s_{e_i}\}), (F_{\hat{A}}, (0, 0, 0, 0))), \quad (14)$$

$$\hat{s}'_{e_{\text{new}}} = \text{MLP}_{\text{refine}}(F'_{e_{\text{new}}}). \quad (15)$$

1569 *Training and loss.* To teach the network to refine layouts, the training layouts are perturbed
 1570 by a random strength $\delta = U(-0.05, 0.05)$ which is added to the coordinates of new elements
 1571 $s'_{e_{\text{new}}} = (\tilde{x}_{e_{\text{new}}} + \delta, \tilde{y}_{e_{\text{new}}} + \delta, \tilde{w}_{e_{\text{new}}}, \tilde{h}_{e_{\text{new}}})$. The boxes of the other elements are kept as is such that
 1572 there are well-aligned elements to align to. The network is trained with a reconstruction loss $\mathcal{L}_{\text{rec}_3}$
 1573 on the size box differences $\mathcal{L}_{\text{rec}_3} = \|\hat{s}'_i - s_i^*\|_1$, where $\|\cdot\|_1$ is the L1-loss between the predicted box
 1574 and the ground truth box s^* .

1576 A.4 Graph convolution network

1577 To perform the actual graph convolutions, we use the same approach as described in the original
 1578 ‘Neural Design Network’ [19] which employs the architecture described in ‘sg2im’ by Johnson *et*
 1579 *al.* [14].



1591 Fig. 17. The graph convolutional layer. f_v represents the features of a vertex, f_a the features of an arrow (edge). FC describe fully-connected layers.

1594 This graph convolution layer GC is depicted in Figure 17. The input are feature vectors of the
 1595 vertices f_{v_i} and of the arrows f_{a_i} with dimension D_{in} . First, they are passed as triples in the form of
 1596 $(f_{v_i}, f_{a_i}, f_{v_j})$ where v_i is a source of the triple a_i and v_j the target through a fully-connected neural
 1597 network FC_1 , resulting in $(f'_{v_i}, f'_{a_i}, f'_{v_j})$. Vertices that appear multiples times in any of the triples are
 1598 then pooled in the next step which is by default implemented as an *average* pooling, creating $\{f''_{v_i}\}$.
 1599 The pooled vertex features are finally passed through a second fully-connected neural network FC_2
 1600 to generate the output of the graph convolution for the vertices $\{f''_{v_i}\}$. The features of the arrows
 1601 $\{f'_{a_i}\}$ are output directly from the first network FC_1 . Every fully-connected layer is followed by an
 1602 optional dropout layer, and the output is processed by an activation function σ . Parameters of this
 1603 layer are the input Dimension D_{in} , the hidden dimension D_{hidden} and the output dimension D_{out} .
 1604 We will use the shorthand $(D_{in}, D_{hidden}, D_{out})$ to describe the dimensions going forward. Graph
 1605 convolution networks as denoted GCN above are then stacked layers of the described GC layer.

1607 A.5 GNN parameters

1608 We use the same parameters as described in the ‘Neural Design Network’ supplementary material
 1609 [19] with minor differences. For GC , we present the dimensions of each layer with $(D_{in}, D_{hidden}, D_{out})$,
 1610 and for fully connected layers FC as (D_{in}, D_{out}) , where D_{in}, D_{out} denote the input and output dimen-
 1611 sions correspondingly.

1613 *Relation module.* In the relation module, the embedding dimension is $D_{\text{embed}} = 128$ and the
 1614 dimension of the latent variable is $D_{z_{\text{rel}}} = 32$. GCN_{pred} consists of three graph convolutional layers
 1615 as detailed in Table 7. MLP_{pred} consists of two fully-connected layers as shown in Table 8. The
 1616 embedding dimension of the ground-truth encoding is $D_{\text{embed}}^{\text{enc}} = 64$, and GCN_{enc} is a three-layered
 1617

1618 graph convolutional network as described in Table 9. MLP_{enc} is a two-headed network shown in
 1619 Table 10.

	GCN_{pred}	σ	d
1620			
1621			
1622	1 GC(128+32, 512, 128)	ReLU	0.1
1623	2 GC(128, 512, 128)	ReLU	0.1
1624	3 GC(128, 128, 128)	ReLU	0.1

1625 Table 7. GCN_{pred} layers.

	MLP_{pred}	σ	d
1620			
1621			
1622	1 FC(128, 512)	ReLU	0.1
1623	2 FC(512, $ R $)	softmax	0

1624 Table 8. MLP_{pred} layers.

1626
 1627
 1628 We use the loss weights $\lambda_{\text{rec}_1} = 1$ and $\lambda_{\text{kl}_1} = 0.005$.

1630 *Layout module.* The dimension of the embedding is the same as above with $D_{\text{embed}} = 128$.
 1631 GCN_{I1} , GCN_{I2} consist of three layers according to the definitions in Table 11 and Table 12.

1632 The variational auto-encoder networks are defined as follows: h^{enc} is a network of fully-connected
 1633 layers, followed by a two-headed output of 32 again as shown in Table 13. h^{dec} is a simple fully-
 1634 connected network with three layers as detailed in Table 14. The dimension of the latent code in
 1635 this module is correspondingly $D_{z_{\text{lay}}} = 32$.

1636 The employed loss weights of this module are $\lambda_{\text{rec}_2} = 1$, $\lambda_{\text{rec}_3} = 10$ (to prioritize size reconstruc-
 1637 tion), and $\lambda_{\text{kl}_2} = 0.01$.

1638 *Refinement module.* The embedding dimension in the refinement module are given as $D_{\text{embed}_V}^{\text{refine}} =$
 1639 60 and $D_{\text{embed}_A}^{\text{refine}} = 64$. $\text{GCN}_{\text{refine}}$ is a graph convolution network with three layers according to
 1640 Table 15. Finally, $\text{MLP}_{\text{refine}}$ is a network of two fully-connected layers as shown in Table 16.

	GCN_{enc}	σ	d
1643			
1644			
1645	1 GC(64, 512, 128)	ReLU	0.1
1646	2 GC(128, 512, 128)	ReLU	0.1
1647	3 GC(128, 128, 128)	ReLU	0.1

1648 Table 9. GCN_{enc} layers.

	MLP_{enc}	σ	d
1643			
1644			
1645	1_{μ} FC(128, 32)	-	0
1646	1_{σ} FC(128, 32)	-	0

1647 Table 10. MLP_{enc} layers.

	GCN_{I1}	σ	d
1650			
1651			
1652	1 GC(128, 512, 124)	ReLU	0.1
1653	2 GC(124, 512, 124)	ReLU	0.1
1654	3 GC(124, 512, 124)	ReLU	0.1

1655 Table 11. GCN_{I1} layers.

	GCN_{I2}	σ	d
1650			
1651			
1652	1 GC(124+4, 512, 128)	ReLU	0.1
1653	2 GC(128, 512, 128)	ReLU	0.1
1654	3 GC(128, 512, 128)	ReLU	0.1

1655 Table 12. GCN_{I2} layers.

	h^{enc}	σ	d
1658			
1659			
1660	1 FC(4+128, 128)	ReLU	0.1
1661	2 FC(128, 128)	ReLU	0.1
1662	3_{μ} FC(128, 32)	-	0
1663	3_{σ} FC(128, 32)	-	0

1664 Table 13. h^{enc} layers.

	h^{dec}	σ	d
1658			
1659			
1660	1 FC(32+128, 128)	ReLU	0.1
1661	2 FC(128, 64)	ReLU	0.1
1662	3 FC(64, 4)	-	0

1663 Table 14. h^{dec} layers.

	GCN _{refine}	σ	d
1	GC(64, 512, 128)	ReLU	0.1
2	GC(128, 512, 128)	ReLU	0.1
3	GC(128, 128, 128)	ReLU	0.1

Table 15. GCN_{refine} layers.

	MLP _{refine}	σ	d
1	FC(128, 512)	ReLU	0.1
2	FC(512, 4)	-	0

Table 16. MLP_{refine} layers.

Training parameters. In all fully-connected layers, we add a batch normalization layer, and a dropout layer with a rate of 0.1. We use the optimizer Adam with a learning rate of 10^{-4} , $\beta = (.9, .999)$ and an l2 regularization of 10^{-4} .

We also tried using only the relation module to produce constraints for a combinatorial optimization system. However, the predicted edges were very often not completely consistent such that considering all, no feasible result was possible.

B LAYOUTTRANSFORMER DETAILS

B.1 Input tokenization

Since tokens need to be embedded in the network, we limit the amount of coordinate and size tokens by employing a base grid size g onto which all positions and sizes are “snapped” to. This is a common approach in user interface design and automatically prevents misalignments by few pixels. Consequently, all w, h, x^0, y^0 in the layout sequence are divided by g and rounded: $w' = \lfloor \frac{w}{g} \rfloor, h' = \lfloor \frac{h}{g} \rfloor, x^{0'} = \lfloor \frac{x^0}{g} \rfloor, y^{0'} = \lfloor \frac{y^0}{g} \rfloor$, where $\lfloor \cdot \rfloor$ denotes a rounding function to the nearest integer.

This updated layout sequence is then converted into a token sequence S_L^{in} for the transformer network. For that, a token dictionary is created with specifically allocated ranges for the different types of element attributes, as well as special transformer tokens for *start*, *end*, and *padding*. The token vocabulary Vocab is then the following set:

$$\text{Vocab} = (t_{\langle \text{pad} \rangle}, t_{\langle \text{start} \rangle}, t_{\langle \text{end} \rangle}, \{t_{c_i}\}, \{t_{x'_i}\}, \{t_{y'_i}\}, \{t_{w'_i}\}, \{t_{h'_i}\}), \quad (16)$$

where $\{t_{c_i}\}$ is the token set of the different component types (i.e., every component type is assigned a token), $\{t_{x'_i}\}$ is the set of tokens for all possible x-coordinates as given by the base grid, and $\{t_{y'_i}\}, \{t_{w'_i}\}, \{t_{h'_i}\}$ are defined correspondingly for the other grid-adjusted element attributes. This requires a maximum canvas size $W_{\text{max}}, H_{\text{max}}$ to be defined that determines the maximum number of tokens for the positions and sizes. With this token vocabulary, each element attribute is mapped to the corresponding token, i.e., the token representation of an element e is given by $t_e = t_{c_e}, t_{w'_e}, t_{h'_e}, t_{x^{0'}_e}, t_{y^{0'}_e}$. The tokens are encapsulated by a special start and stop token to produce the final input sequence:

$$S_L^{\text{in}} = t_{\langle \text{start} \rangle}, t_{c_{e_0}}, t_{w'_{e_0}}, t_{h'_{e_0}}, t_{x^{0'}_{e_0}}, t_{y^{0'}_{e_0}}, \dots, t_{c_{e_{\text{new}}}}, t_{w'_{e_{\text{new}}}}, t_{h'_{e_{\text{new}}}}, t_{\langle \text{end} \rangle}. \quad (17)$$

B.2 Training

We train the network with the cross-entropy loss $CE = -\sum_n^T y_n \log \hat{y}_n$ between the generated token probability and the ground-truth token with Label Smoothing of strength l [34] where $T = |\text{Vocab}|$ is the token vocabulary size, and y_n, \hat{y}_n denote the ground truth probability and predicted probability of token n respectively. Label smoothing puts a high probability on the ground truth category and distributes a small probability uniformly on the other categories. Using such soft targets has been shown to improve the learning and generalization capabilities of neural network models [26].

1716 B.3 Network parameters

1717 The embedding dimensions is $D_{\text{embed}} = 512$, the number of layers per encoder/decoder is $n_{\text{layers}} = 6$,
 1718 with $n_{\text{heads}} = 8$ attention heads, and $D_{\text{ff}} = 1024$ the dimension of units in the feed-forward layers,
 1719 with the *ReLU* activation function in both the attention and feed-forward layers.

1720 As in [10], we use the optimizer *AdamW* with parameters lr, β that employs learning rate
 1721 scheduling such that first a warmup schedule on the learning rate is employed, followed by a
 1722 decay schedule of the learning rate [24]. In addition, a dropout layer with a rate of d is added after
 1723 every layer to counter overfitting. We use a label smoothing strength of $l = 0.1$, a learning rate of
 1724 $lr = 10^{-4}$ with $\beta = (0.9, 0.999)$, and a dropout rate of $d = 0.1$.

1726 C SEQUENCE-ALIGNED NEIGHBOR SEARCH DETAILS

1727 C.1 Insertion point details

1728 The candidate positions of the target element p_t^* are defined by:

$$1730 p_t^* = \arg \min_{p \in [0, n_t]} \text{editDistance}(S_{t_p}^{\text{type}}, S_{l_i}^{\text{type}}), \quad (18)$$

1732 where $p \in [0, n_t]$ are the possible positions in the target type sequence $S_{t_p}^{\text{type}}$, $S_{l_i}^{\text{type}}$ represents the
 1733 target sequence with the new element inserted at position p , $S_{l_i}^{\text{type}}$ is the library type sequence and
 1734 *editDistance* is a function that calculates the edit distance between two sequences. For the edit
 1735 distance algorithm, we utilize an implementation of the Myers's bit vector algorithm [27].

1737 C.2 Feature scaling

1738 We scale each feature vector by its distance to the insertion point of the new element $t(p_e, p_t)f_e$
 1739 where $t(., .)$ is a function returning the scaling factor between the position of the current element
 1740 p_e and the position of the target element p_t . For $t(., .)$ we use the following formula:

$$1742 t(p_e, p_t) = \max(\lambda_{\min}, 1 - \min(1, \frac{|p_t - p_e|^2}{\gamma^2})), \quad (19)$$

1744 where λ_{\min} is the minimum scaling factor for far away elements, and γ is the distance at which
 1745 point the minimum scaling takes effect.

1747 C.3 Penalty feature

1748 The penalty feature is composed of the distance of the element to an approximate location of the
 1749 new element and its size: $f_e^p = (\tilde{\delta}\tilde{x}_e, \tilde{\delta}\tilde{y}_e, \tilde{w}_e, \tilde{h}_e)$. The distance to the new element is approximated
 1750 since the final location is not known yet. For this, the center position of the elements surrounding
 1751 the insertion point is taken as an approximation for the new element, and the distance to the center
 1752 point of the missing element is calculated:

$$1754 \tilde{\delta}\tilde{x}_e = \left| \frac{(\tilde{x}_{e_{p^*-1}}^0 + \frac{\tilde{w}_{e_{p^*-1}}}{2}) + (\tilde{x}_{e_{p^*+1}}^0 + \frac{\tilde{w}_{e_{p^*+1}}}{2})}{2} - (\tilde{x}_e^0 + \frac{\tilde{w}_e}{2}) \right|, \quad (20)$$

1757 where e_{p^*-1}, e_{p^*+1} are the elements before and after the insertion point in the sequence. The same
 1758 formula is applied to the y-coordinate with corresponding attributes. Finally, the approximate
 1759 distance is inverted so that close elements that are not mapped produce a higher penalty than those
 1760 that are far away. The following formula is applied:

$$1761 \tilde{\delta}\tilde{x}_e = \max(0, 0.5 - \tilde{\delta}\tilde{x}_e), \quad (21)$$

$$1763 \tilde{\delta}\tilde{y}_e = 0.5 \max(0, 0.5 - \tilde{\delta}\tilde{y}_e). \quad (22)$$

1765 The vertical distance is scaled down because we argue that missing elements in the same row are
 1766 more relevant than those in other rows even if the horizontal distance is less in the second case.
 1767 Lastly, the penalty feature is scaled with the same function $t(p_e, p_t)$ but with different parameters
 1768 λ_{\min} and γ .

1769

1770 C.4 Feature representation

1771 The dimension of the feature vector is a function of the number of maximally matched elements n^*
 1772 (incl. the new element) in the sequence: $|\tilde{S}| = 4(n^* - 1) + 3 + 25n^*$. An example is given in Listing 1.

1773

1774 Listing 1. Example feature representation of a 6+1 element layout search.

```
1775 # box features of matched elements in the layout
1776 (0.02, 0.03, 0.21, 0.03,
1777 0.03, 0.10, 0.79, 0.18,
1778 0.03, 0.35, 0.11, 0.02,
1779 0.03, 0.40, 0.45, 0.08,
1780 0.03, 0.51, 0.37, 0.22,
1781 # box padding to match longest feature sequence
1782 100.0, 100.0, 100.0, 100.0,
1783 # box feature of target element
1784 0.14, 0.04, 3.59,
1785 # word vector of the component names
1786 0.02, -0.01, 0.00, ..., 0.09, -0.02, 0.05, -0.04, -0.03,
1787 0.03, -0.11, -0.02, ..., 0.08, 0.01, 0.05, 0.03, -0.00,
1788 -0.01, -0.13, 0.04, ..., 0.13, -0.01, 0.16, 0.02, 0.04,
1789 0.17, -0.19, 0.00, ..., -0.02, 0.07, 0.04, 0.01, 0.03,
1790 0.07, -0.11, 0.03, ..., 0.02, 0.02, 0.17, -0.10, -0.01,
1791 # word vector padding
1792 10.00, 10.00, 10.00, ..., 10.00, 10.00, 10.00, 10.00, 10.00,
1793 # word vector of target element
1794 -0.03, -0.17, 0.15, ..., 0.13, 0.00, 0.05, -0.03, 0.02)
```

1794

1795

1796 D DATA SET DETAILS

1797

1798 D.1 Artificial web patterns

1799

1798 (1) A *data table* pattern combines a *table* component with a *button* (e.g., to execute an action with a
 1799 selection of items from the data table). The button is always right-aligned with the table but can be
 1800 on either side. An optional *pagination* component is vertically centered with the table element if
 1801 present. In that case, the button and pagination components must be on the same row.

1802

1802 (2) For *forms*, we follow a common web pattern where the label is placed to the left of an input
 1803 element. The complete form is either arranged in a single column (70 % of times) or in two columns
 1804 (30 %). There is a set of fixed elements preceding the form (title, breadcrumb, text paragraph, icon,
 1805 link, section title), some of which are exclusive to forms. As before, the submit button has multiple,
 1806 valid positions with defined frequencies to simulate major and minor patterns. When only a single
 1807 button exists below the form, it is either aligned to the left of the last input field (20%) or right-
 1808 aligned (80%). This applies to both column variations. When two buttons are placed below the
 1809 form, they are next to each other in the same row, and the right button is right-aligned with the
 1810 last input field.

1811

1811 (3) Lastly, different *dashboard* layouts are present in the data set. It is composed of at least two
 1812 rows of different widgets. There are three different layout arrangements: a two-column layout,

1813

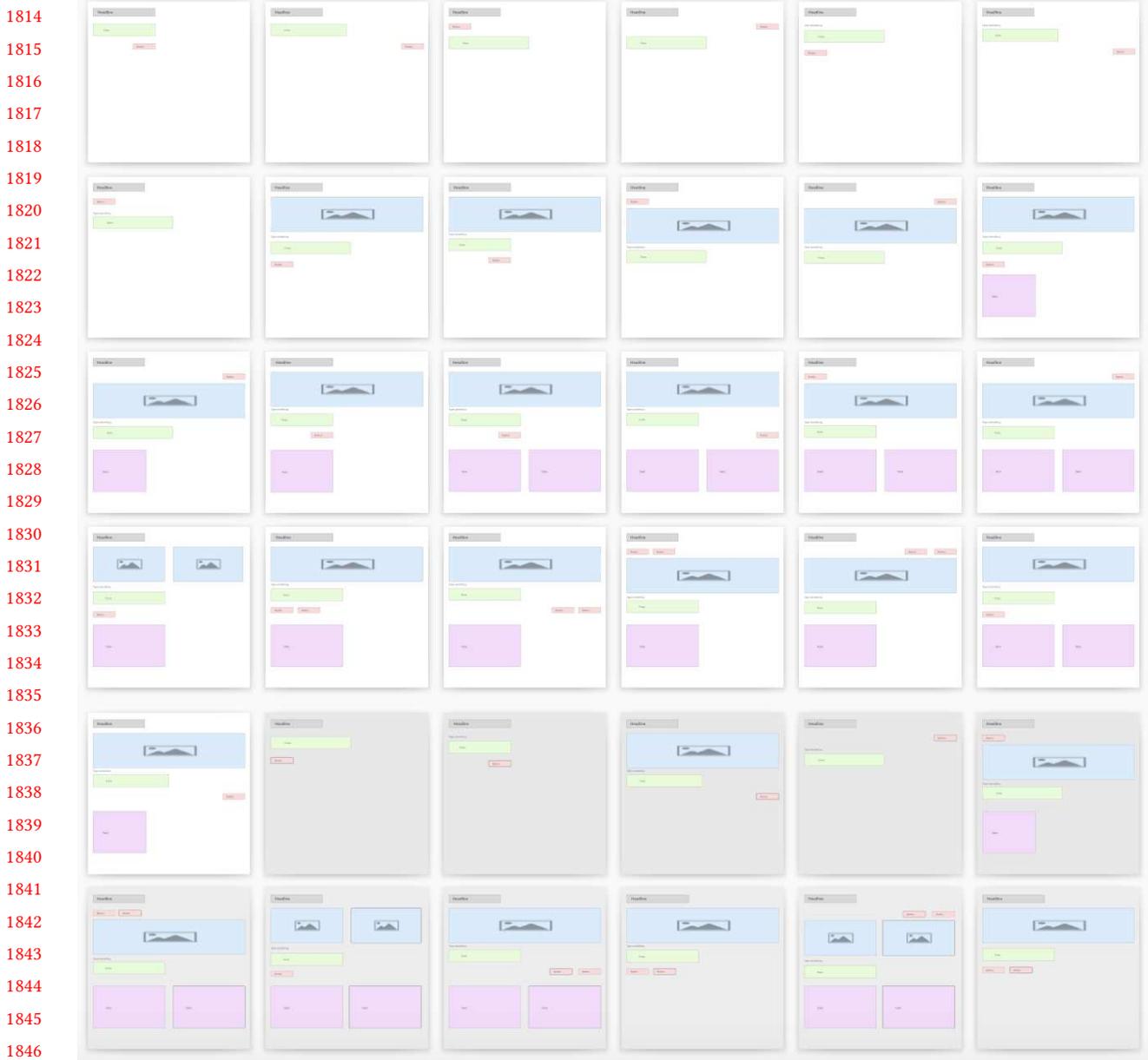


Fig. 18. The set of layouts in the varying buttons data set. Grey boxes correspond to headlines, green boxes to forms, red boxes to buttons, blue boxes to images, pink boxes to tables and black text represents a text paragraph. Grey layouts indicate test items that are not used during training.

a four-column layout, and a three-column layout with a separate sidebar column. Each widget can take the full width of the columns or a single column. There are four different widget types that are used randomly in every layout (text, chart, table, image). The different layouts are present uniformly distributed in the data set. For these dashboards, the first section acting as a filler is optional.

E METRIC DETAILS

Intersection over Union. We calculate the Intersection over Union (IoU) according to:

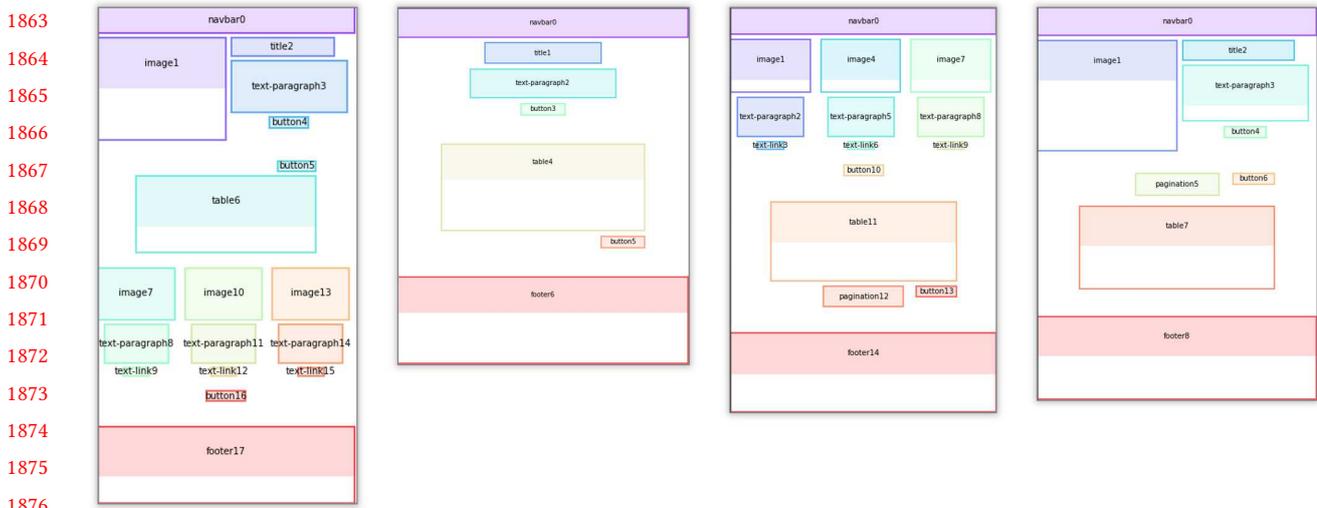


Fig. 19. The table pattern. The button accompanying the table is always right-aligned with the table and can be either above or below it. If the pagination is present, it is next to it.

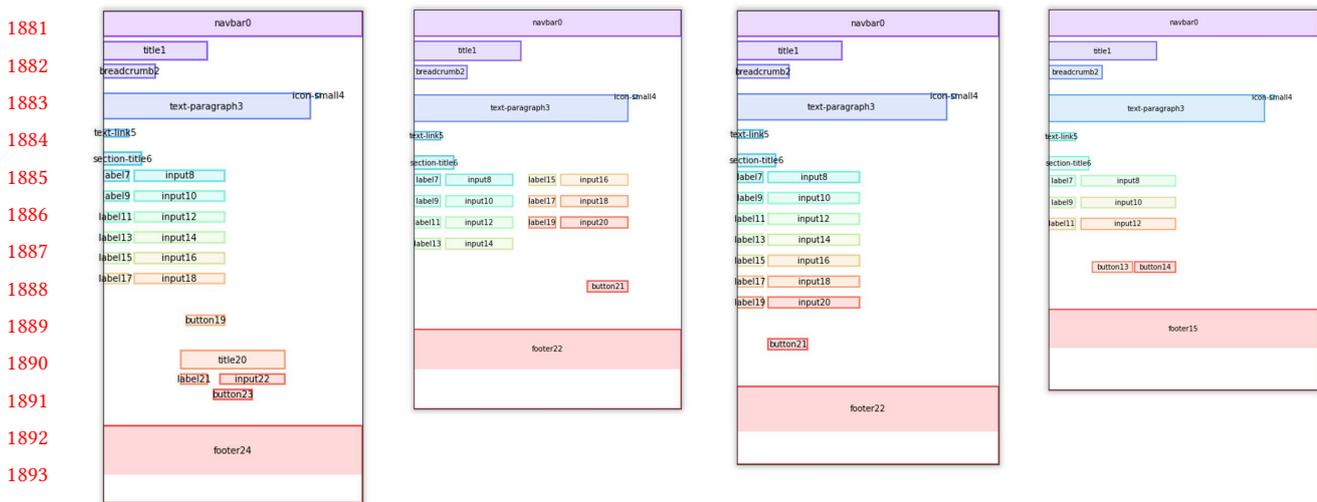


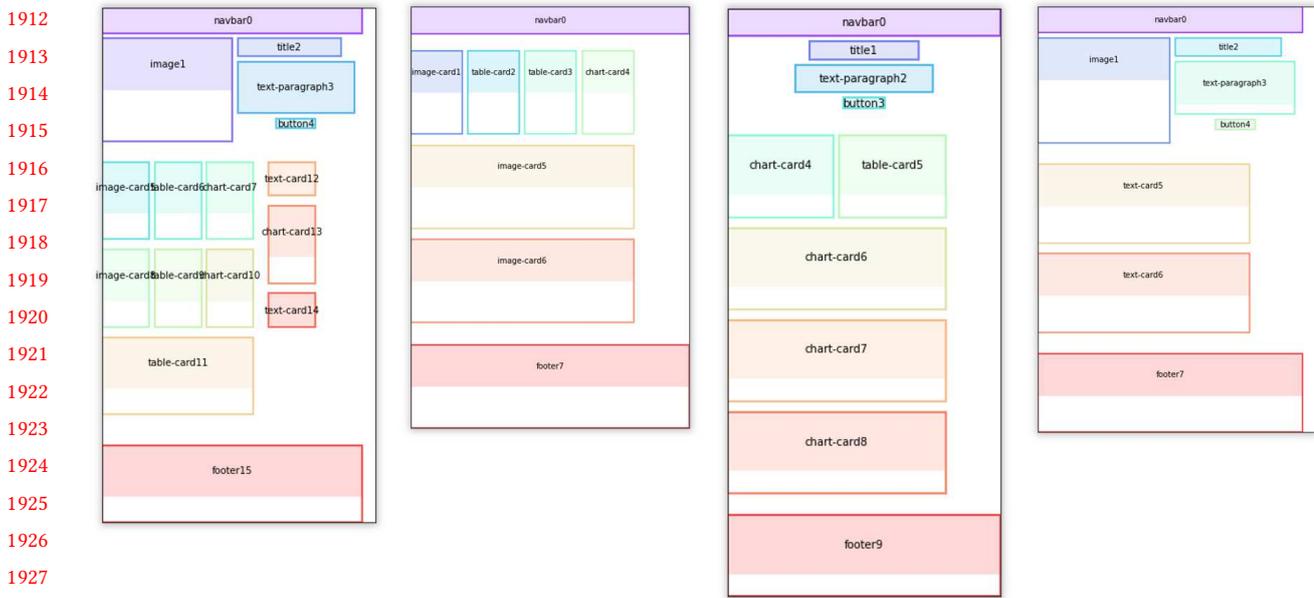
Fig. 20. The form pattern. There can be forms with one or two columns with the label to the left of the input. A single button can be either placed left or right-aligned with the right-most input field. With two buttons, they are placed next to each other and are right-aligned with the last input element.

$$IoU(\hat{b}_e, b_e^*) = \frac{|\hat{b}_e \cap b_e^*|}{|\hat{b}_e \cup b_e^*|}. \quad (23)$$

Overlap. We use the following formula to determine if there is overlap between two elements e_1, e_2 :

$$\min(x_{e_1}^1, x_{e_2}^1) > \max(x_{e_1}^0, x_{e_2}^0) \wedge \min(y_{e_1}^1, y_{e_2}^1) > \max(y_{e_1}^0, y_{e_2}^0). \quad (24)$$

Outside the canvas. The following test determines if an element e is outside the canvas of the layout L :



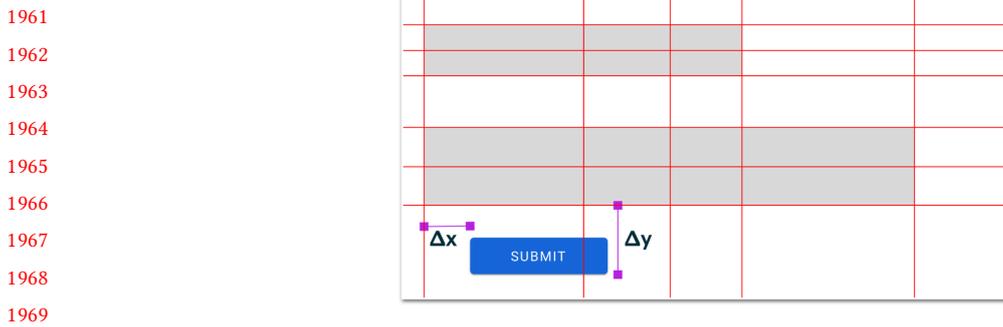
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960

Fig. 21. The dashboard pattern arranges different type of cards in either a 2-column layout, 4-column layout, or a 3-column layout with an additional sidebar. Cards can either span a single column or all columns.



Fig. 22. Examples of exact and close matches. In the exact match, the alignment to the neighboring element is according to the expectation in green, even if it has a minor offset vertically. The close match has a major overlap but misses the alignment.

$$\min(x_e^0, y_e^0) < 0 \vee x_e^1 > w_L \vee y_e^1 > h_L. \quad (25)$$



1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009

Fig. 23. Alignment takes the average of the minimum differences between any x^0, x^c, x^1 of the new element and the existing elements (shown with Δx) and between any y^0, y^c, y^1 of the new element and the existing elements (Δy).