# Learning GUI Completions with User-defined Constraints

LUKAS BRÜCKNER, Aalto University, Finland
LUIS A. LEIVA, University of Luxembourg, Luxembourg
ANTTI OULASVIRTA, Aalto University, Finland

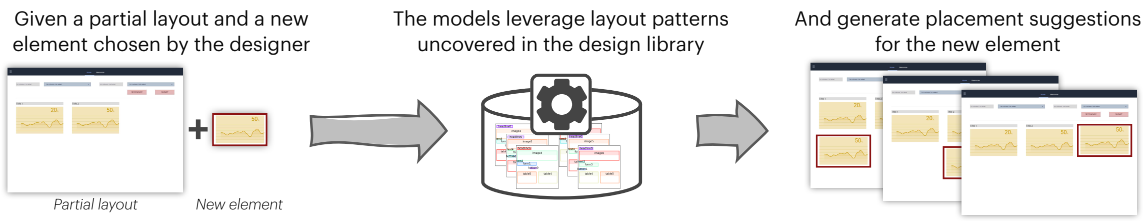
Given a partial layout and a new element chosen by the designer
The models leverage layout patterns uncovered in the design library
And generate placement suggestions for the new element
Partial layout
New element

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. 2021. Learning GUI Completions with User-defined Constraints. *ACM Trans. Interact. Intell. Syst.* 1, 1, Article 1 (January 2021), 41 pages. https://doi.org/10.1145/3490034

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.

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

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

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

### 1.1 Problem statement

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

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

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

| 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.

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

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

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

### 2.2 Layout optimization and adaptation

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

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

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

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

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

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

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

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

### 2.3 Layout completion and assistance

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

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

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

## 3 ELEMENT PLACEMENT TASK

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

### 3.1 Layout notation

We denote a layout $L$ as an abstract, structural representation of a design canvas with size $w_L \times h_L$ (width, height) in pixels. It is composed of a set of elements $\{e_i, ..., e_n\}$. Every element is defined by $e = (c_e, s_e)$ where $c_e \in C$ is the component (i.e., functional user interface element with a specific function, e.g., 'button', 'input field', etc.) out of the valid component set $C$, and $s_e = (x_e^0, y_e^0, w_e, h_e)$ 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 and height of the element in pixels. Note that the 'size box' is not the 'bounding box' of an 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 $(x_e^1, y_e^1)$ coordinate points. Finally, we refer to the 'center point' of an element as $x_e^c, y_e^c$ for the $x$ and $y$ coordinates, respectively.

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

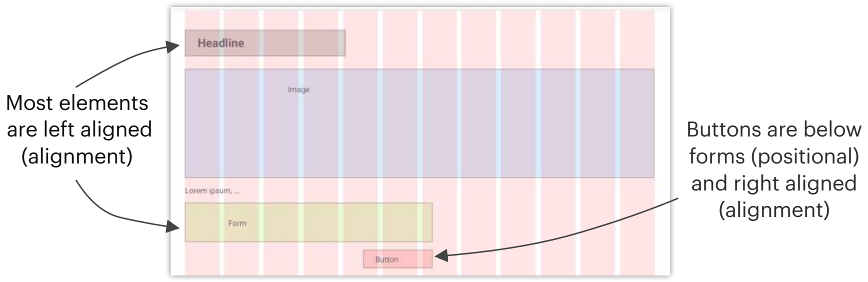
Most elements are left aligned (alignment)
Headline
Image
Lorem ipsum, ...
Form
Button
Buttons are below forms (positional) and right aligned (alignment)

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 will be described as being *at the top,*

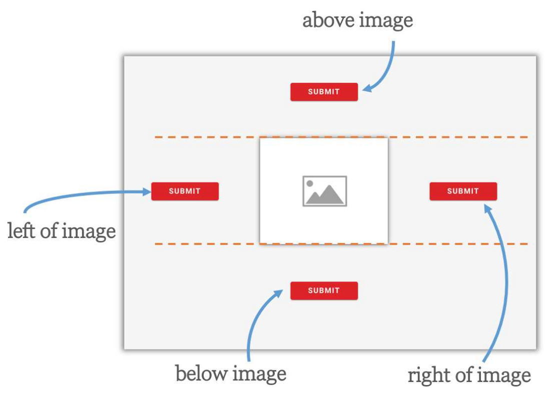

Fig. 3. Positional relations towards the center.

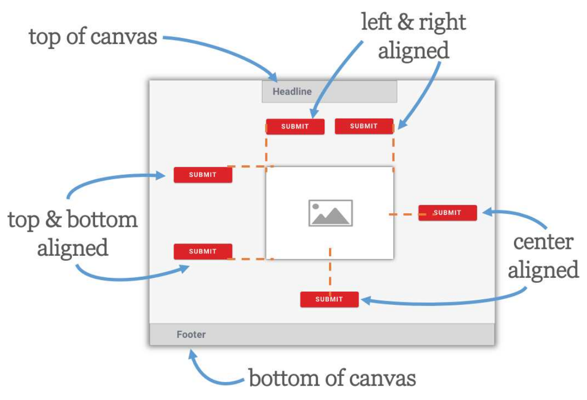

Fig. 4. Inter-element alignment and to the canvas.

*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 $\{L_0, ..., L_n\}$, the user generates a partial input layout $L$ (that might be empty) and picks the next element $e_{\text{new}}$ 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 $e_{\text{new}}$. 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.

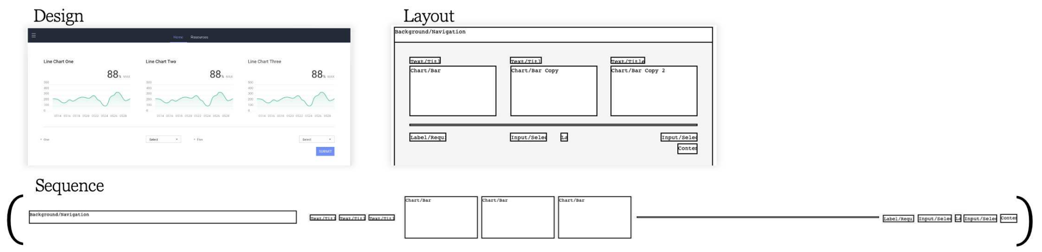

Fig. 5. Representing a layout as a sequence. The component view shows a simpler bounding box representation. The sequence follows the natural reading order from top-left to bottom-right.

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 $\{e_i, ..., e_n\}$ is represented by its sequence $S_L$ as follows:

$$S_L = (e_{p_1}, e_{p_2}, ..., e_{p_n}), \tag{1}$$

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:

$$y^0_{e_{p_i}} \le y^0_{e_{p_{i+1}}} \land x^0_{e_{p_i}} \le x^0_{e_{p_{i+1}}}, \tag{2}$$

where $x^0_e, y^0_e$ 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

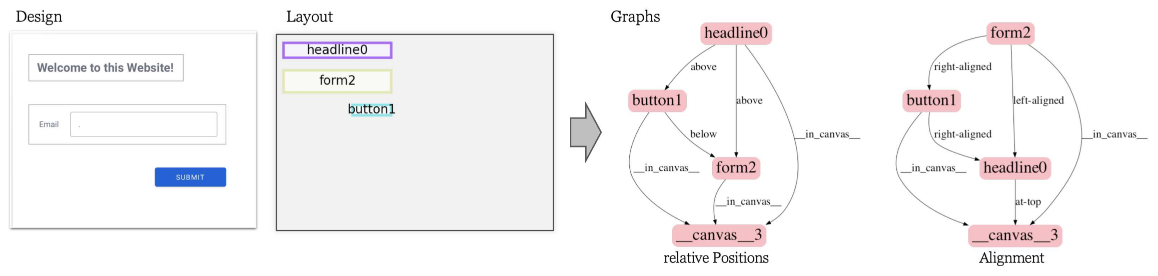

Fig. 6. Graph representation of a simple layout with 3 elements.

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.

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 $\{v_i\} = \{e_0, ..., e_n\}$, 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 $G^{\text{pos}}$ and $G^{\text{align}}$ 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 $R^{\text{pos}}$ and $R^{\text{align}}$ 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 $(e_j, \bar{r}, e_j)$ 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 $G^{\text{pos}}$ and $G^{\text{align}}$ 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 $A^{\text{pos}}$ and $A^{\text{align}}$. This is different from the original implementation, which used relative *size* as a second edge type. Since in our work the

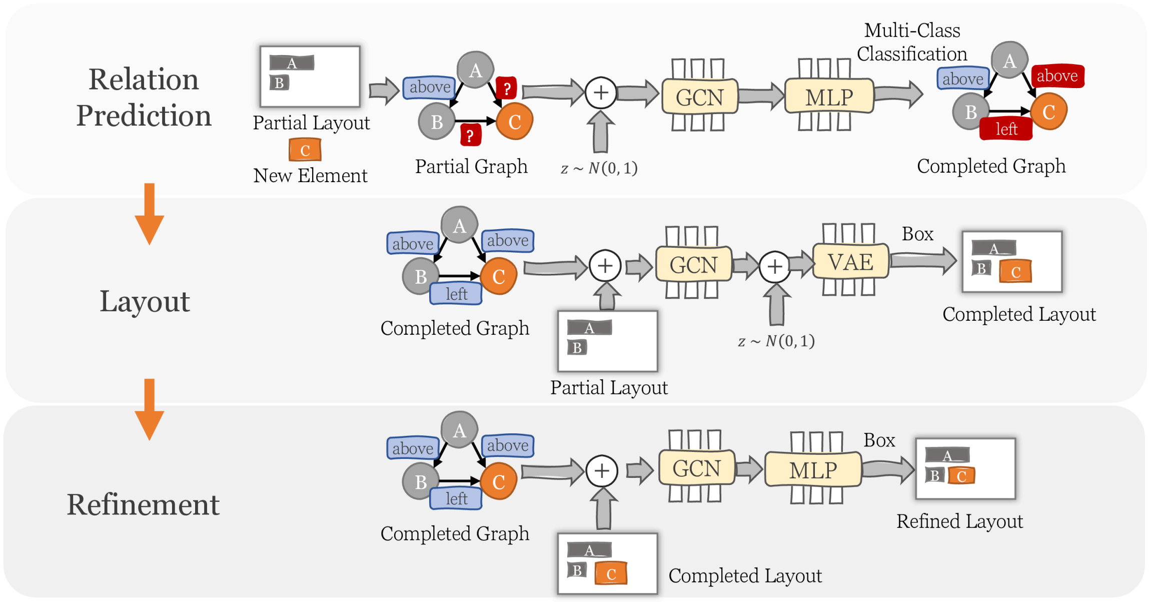

Fig. 7. The graph neural network implemented according to [19] is composed of three modules to predict a location for a new element.

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.

Adding a new element $e_{\text{new}}$ 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 $G^{\text{p}}$ where edges to and from the new element $e_{\text{new}}$ have the special label *unknown*, and a complete graph $\hat{G}$ is generated that contains label predictions for the previously unknown edges: $\hat{G} = \text{Rel}(G^{\text{p}})$. 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 $z_{\text{rel}}$ 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.

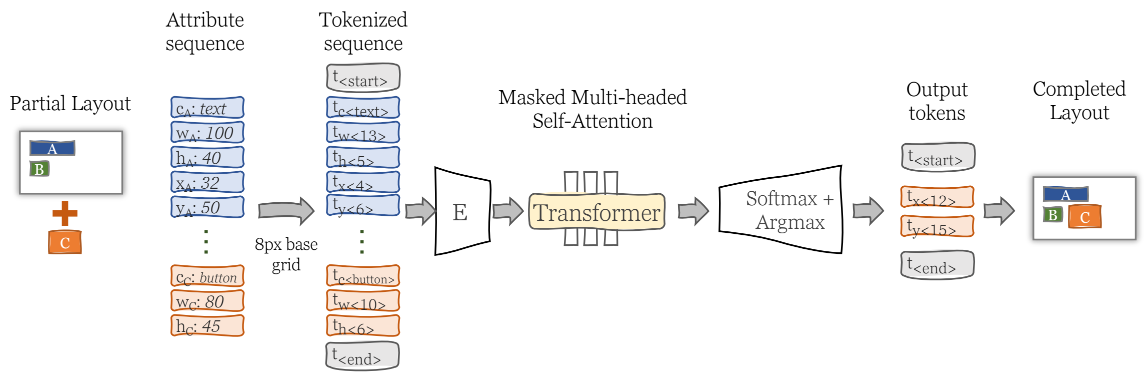

Fig. 8. The transformer network, modeled after [10].

*4.3.1 Representation of layouts.* In this model, layouts are decomposed into sequences of elements as described in subsubsection 4.1.1. The input is a partial layout $L$ and a new element $e_{\text{new}}$ 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 $\hat{x}^0_{e_{\text{new}}}, \hat{y}^0_{e_{\text{new}}}$.

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 $(c_e, w_e, h_e, x^0_e, y^0_e)$. 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 $S_L$:

$$S_L = c_{e_0}, w_{e_0}, h_{e_0}, x^0_{e_0}, y^0_{e_0}, \ldots, c_{e_n}, w_{e_n}, h_{e_n}, x^0_{e_n}, y^0_{e_n}. \tag{3}$$

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

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

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

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

### 4.4 Sequence alignment with nearest neighbors

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

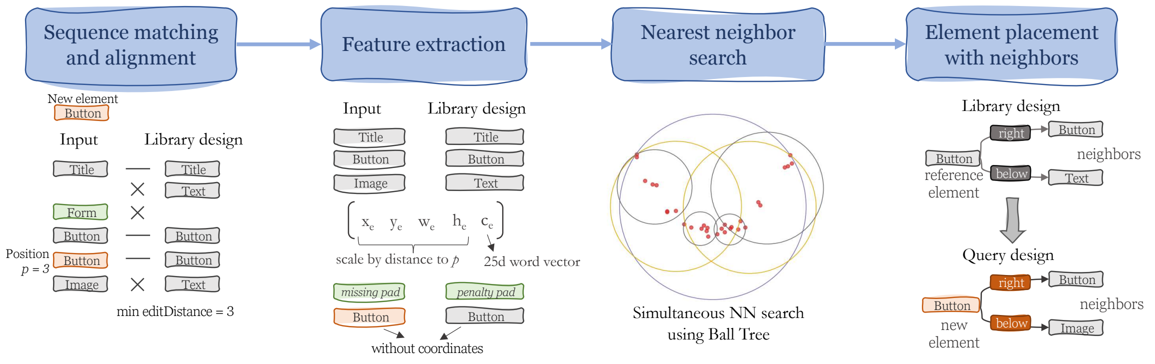

Fig. 9. Process overview of the sequence-aligned nearest neighbor search with neighborhood placement of the new element.

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

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

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

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

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

$$\tilde{S} = f_{e_{p_0}}, f_{e_{p_1}}, ..., f_{e_{pn}}, \tag{4}$$

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

The new element of the target layout does not have assigned coordinates yet. Hence, only its 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 emphasis on finding a placement for the new element where the reference element in the library layout has a similar size. Thus, in addition to the width and height of the new element, also a scaled value of the area is added to its feature representation. To keep it in the same magnitude as the individual coordinates, the square root of the area is used. This applies both to the new element in the target layout $e_t$ as well as the matched element $e_{p^*}$ in the library layout according to the sequence alignment result. With the scaling factor $\alpha$, one can increase or decrease the size matching requirements for the new element.

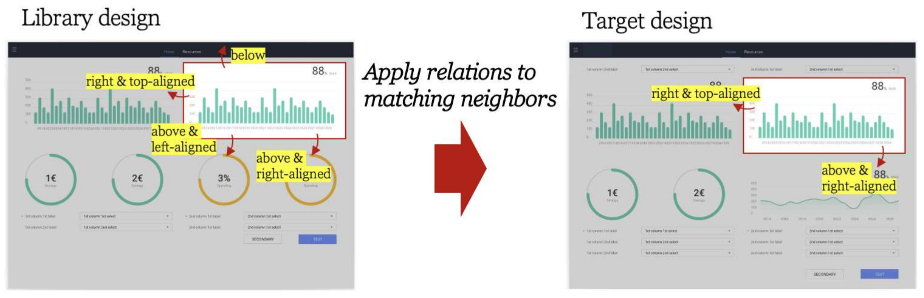

Fig. 10. Placement approach using relations to neighbor elements.

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

Not all elements of a layout might be directly matched against other layouts in the design library. For example, if the library contains more elements than the target layout (including the new element), some elements in the library layout will not be encoded in the feature representation. On the other hand, the more elements that can be matched, the better. Since the sequence alignment algorithm finds the best matching component type subsequence, it could suggest a very small overlap that could lead to a small distance if mismatches are ignored. To counter this effect, every element from the target layout that is not matched in the library is added as a penalty feature $f_e^p$ to the target layout (but not the other way around). See appendix C.3 for our specific formulation, and C.4 for an example of a feature representation.

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

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

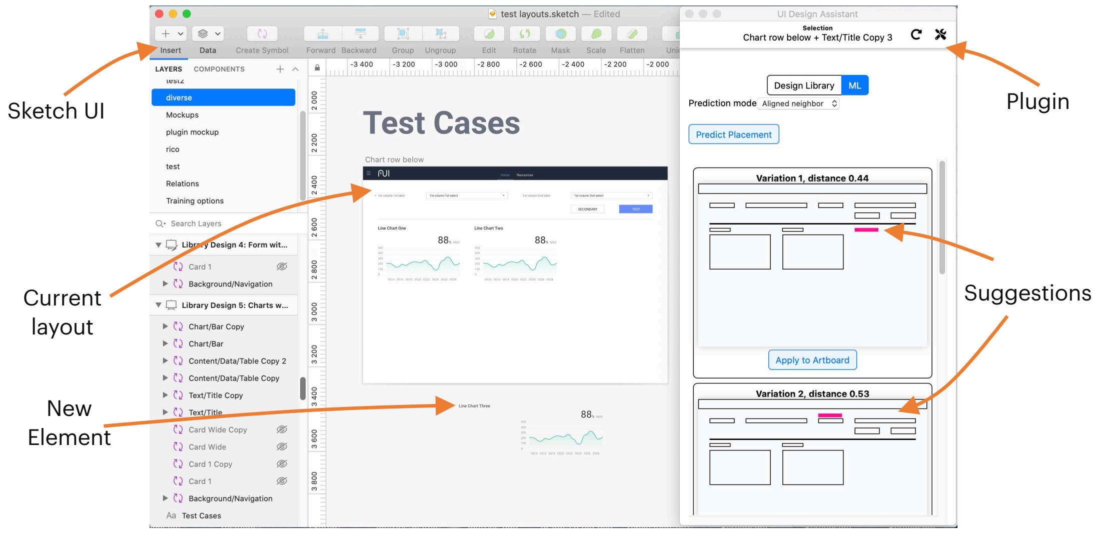

Fig. 11. The user interface of *Sketch* with our plugin loaded on the right side. It shows the placement suggestion for a new text element onto the artboard (canvas) in the middle of the screen.

placement suggestions, sorted by the distance computed in the nearest neighbor search (the smaller the distance, the better he suggestion).

### 4.5 Design tool integration

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

[1]https://www.sketch.com

techniques cannot be implemented with the existing Sketch plugins API[2] 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 Rico called Enrico [21] containing well-formed layouts, to assess if this has an effect on the learning despite it containing a mixture of applications without clear patterns. Lastly, we address the lack of consistent layout patterns in existing data sets by creating two custom data sets that contain layout patterns and follow a design system.

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

### 5.1 Data sets

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

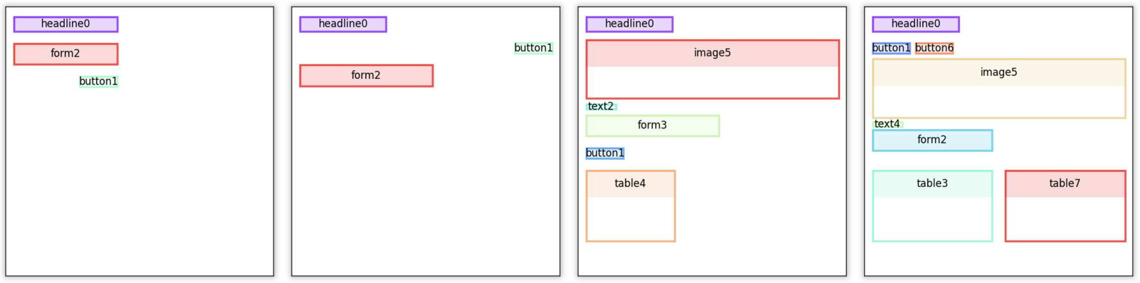

Fig. 12. Layouts in the varying buttons data set.

[2]https://developer.sketch.com/plugins/

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

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.

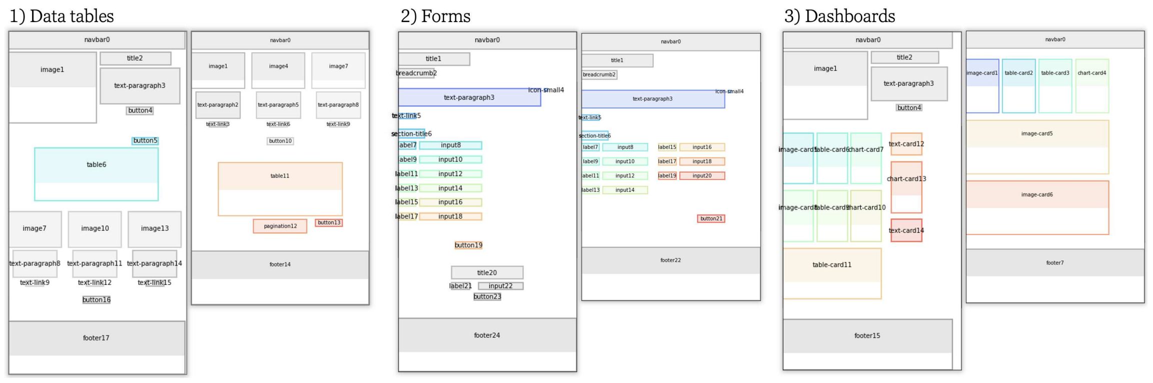

Fig. 13. The different patterns in the artificial web layouts data set.

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.

*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

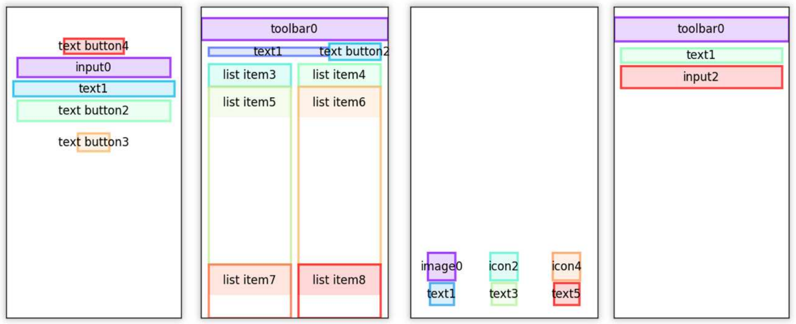

Fig. 14. Example layouts from the *Rico (NDN)* data set. All layouts have less than 10 elements, hence, the complexity is lower than the *Enrico* set.

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 set[3]. 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.

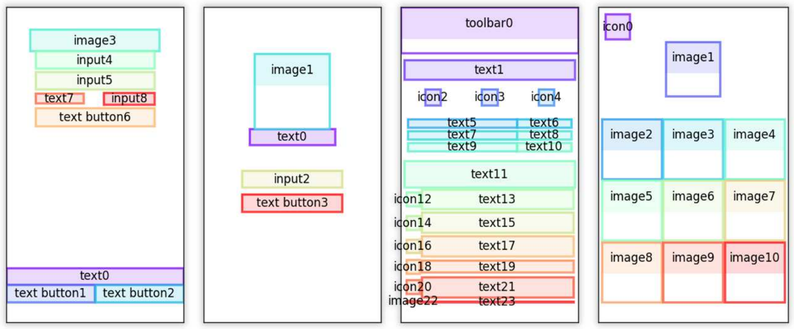

Fig. 15. Example layouts that are considered high-quality based on Enrico and are compatible with our assumptions (no overlap, valid representations as graphs and sequences).

*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

[3]Out 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

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.

| | *Layouts* | | | *Queries* | |
|---|---|---|---|---|---|
| **Data set** | **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.

### 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.

*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 $\eta_{\text{exact}}$. 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 $N_{\text{overlap}}$. Further, if an element's bounding box extends beyond the canvas size, it is considered invalid as well, as counted in $N_{\text{outside}}$. 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 $\alpha$ 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:

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

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

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

## 5.4 Quantitative results

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

| **Varying buttons** | kNN | Transformer | GNN |
|---|---|---|---|
| *Retrieval* | | | |
| 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% |
| *Generalization* | | | |
| 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 |
|---|---|---|---|
| *Retrieval* | | | |
| 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% |
| *Generalization* | | | |
| 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 |
|---|---|---|---|
| *Retrieval* | | | |
| 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% |
| *Generalization* | | | |
| 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 |
|---|---|---|---|
| *Retrieval* | | | |
| 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% |
| *Generalization* | | | |
| 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'.

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 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.

### 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.

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

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

*Enrico.* The mobile layouts based on Enrico's data set features mobile layouts with element counts ranging from 2 to 38. As there are no known layout patterns encoded, we can only evaluate if the generated placements are generally producing well-formed layouts and are valid. In the retrieval query, a set of list items is present in the layout and another one should be added. The GNN again fails to present valid non-overlapping results. The Transformer method suggests placements towards the top of the screen that results in overlaps with existing elements. The kNN approach returns two placements below the existing list items, and even though the gaps with the previous list item is not the

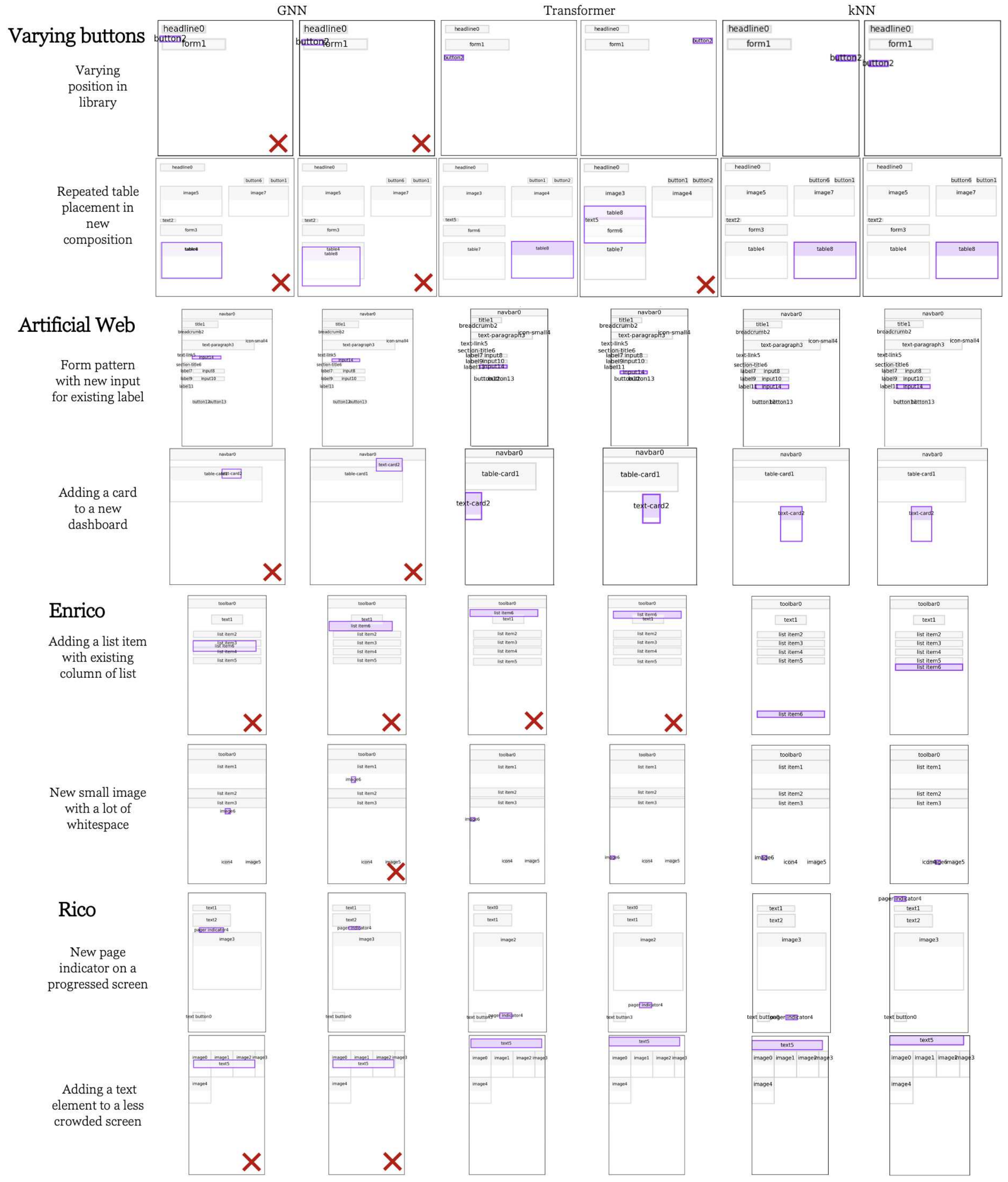
GNN
Transformer
kNN
Varying buttons
Varying position in library
Repeated table placement in new composition
Artificial Web
Form pattern with new input for existing label
Adding a card to a new dashboard
Enrico
Adding a list item with existing column of list
New small image with a lot of whitespace
Rico
New page indicator on a progressed screen
Adding a text element to a less crowded screen

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.

same as with the other list items, we consider it well-formed suggestions. In the generalization query, a small image element should be added. The top of the screen is occupied by existing elements and at the bottom are two other small images and icons. The GNN method predicts one valid result where the image is centered below the list items and one invalid result where the image is overlapping with the existing elements. The transformer model shows two results where the image element is at the edge of the screen, and both can be considered good suggestions. The kNN model also predicts two valid placements, one where it is well-spaced with the other small images at the bottom, and one where it is placed between the two small images. Both can be considered good suggestions.

*Rico (NDN).* In this data set, the models have a much larger set of training items available where all layouts have less than 10 elements per page. In the retrieval query, four elements that span the whole screen already exist in the layout and a pager indicator element is to be added. The GNN approach places the pager indicator between the text and image elements without alignment. The Transformer model predicts results where the new element is towards the bottom of the screen and horizontally centered, but with two different y-coordinates. The first result is aligned with the bottom text button, while the second result is only center aligned with the screen. The kNN method places the pager indicator also either centered at the bottom of the screen, or at the top, and left-aligned. Both results can be considered good suggestions, thus. In the generalization query, five image elements are placed at the top of the screen and a new text element is to be added. The GNN method returns again two invalid positions that overlap with other elements. The transformer model predicts placements at the top of the screen, above the images, where one result is centered, and the other left-aligned. The kNN model shows similar results and places the text element above the images but keeps it left-aligned in both cases. Both the Transformer and the kNN results can be considered good suggestions.

## 6 DISCUSSION AND LIMITATIONS

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

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

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

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

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

We focused on a data-driven evaluation and validated the methods by analyzing layout quality (both quantitatively and qualitatively) and comparing the results to expected layout patterns where applicable. However, a user-centered evaluation with professional UI designers is the next logical step to further validate our approach. Such an evaluation is rather challenging, as it requires designers to be familiar with a set of reference layouts and the patterns therein, besides a strong knowledge of user interface design in general. Critically, designer's acceptance towards this kind of assistance is dependent (1) on the quality of the suggestions and (2) how well they are integrated with the designer's software and workflow. As discussed previously, our work addresses the former case but has not considered the latter case, for which a substantial engineering effort is needed to properly instrument existing design tools. In general, the positive applicability of similar design suggestions has been shown several times before [4, 18, 37], therefore we believe that similar findings would be observed in our case as well.

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

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

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

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

*Sequence-aligned nearest neighbor search.* The custom nearest neighbor method was able to predict layout patterns with the best scores, as it explicitly leverages the reference designs that contain these patterns. However, as it does not learn generalizations and takes a single neighbor as the reference, it is not able to use information from multiple designs to inform the prediction. Certainly, one possible extension is to take into account multiple neighbors for a single input layout. However, since element placement is based on the neighborhood relations of the returned neighbor, 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 Pro[4], 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.

## 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

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

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.

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

## ACKNOWLEDGMENTS

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

## REFERENCES

[1] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. Extracting Content Structure for Web Pages Based on Visual Representation. In *Web Technologies and Applications*, Xiaofang Zhou, Maria E. Orlowska, and Yanchun Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–417.

[2] Niranjan Damera-Venkata, José Bento, and Eamonn O'Brien-Strain. 2011. Probabilistic Document Model for Automated 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.

[3] Niraj Ramesh Dayama, Simo Santala, Lukas Brückner, Kashyap Todi, Jingzhou Du, and Antti Oulasvirta. 2021. Interactive Layout Transfer. In *26th International Conference on Intelligent User Interfaces* (College Station, TX, USA) *(IUI '21)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/3397481.3450652

[4] Niraj Ramesh Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. 2020. GRIDS: Interactive Layout Design 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

[5] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854. https://doi.org/10.1145/3126594.3126651

[6] Krzysztof Gajos and Daniel S. Weld. 2004. SUPPLE: Automatically Generating User Interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces* (Funchal, Madeira, Portugal) *(IUI '04)*. Association for Computing Machinery, New York, NY, USA, 93–100. https://doi.org/10.1145/964442.964461

[7] Krzysztof Z. Gajos, Daniel S. Weld, and Jacob O. Wobbrock. 2010. Automatically generating personalized user interfaces with Supple. *Artificial Intelligence* 174, 12 (2010), 910 – 950.

[8] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*. ACM Press, New York, NY, USA, 231–240.

[9] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning Word Vectors for 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

[10] Kamal Gupta, Alessandro Achille, Justin Lazarow, Larry Davis, Vijay Mahadevan, and Abhinav Shrivastava. 2020. Layout Generation and Completion with Self-attention. *CoRR* abs/2006.14615 (2020). arXiv:2006.14615 https://arxiv.org/abs/2006.14615

[11] Stephen M Hart and Liu Yi-Hsin. 1995. The application of integer linear programming to the implementation of a graphical user interface: a new rectangular packing problem. *Applied mathematical modelling* 19, 4 (1995), 244–254.

[12] Bruce Hilliard, Jocelyn Armarego, and Tanya McGill. 2016. Optimising visual layout for training and learning technologies. In *Proceedings of the 27th Australasian Conference on Information Systems, ACIS 2016*. University of Wollongong, Faculty of Business.

[13] D. S. Hirschberg. 1975. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Commun. ACM* 18, 6 (June 1975), 341–343.

[14] Justin Johnson, Agrim Gupta, and Li Fei-Fei. 2018. Image Generation from Scene Graphs. *CoRR* abs/1804.01622 (2018). arXiv:1804.01622 http://arxiv.org/abs/1804.01622

[15] Akash Abdu Jyothi, Thibaut Durand, Jiawei He, Leonid Sigal, and Greg Mori. 2019. LayoutVAE: Stochastic Scene Layout Generation from a Label Set. *CoRR* abs/1907.10719 (2019).

[16] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, and Scott R. Klemmer. 2011. Bricolage: Example-Based Retargeting for Web Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vancouver, BC, Canada) *(CHI '11)*. Association for Computing Machinery, New York, NY, USA, 2197–2206. https://doi.org/10.1145/1978942.1979262

[17] Markku Laine, Ai Nakajima, Niraj Dayama, and Antti Oulasvirta. 2020. Layout as a Service (LaaS): A Service Platform for Self-Optimizing Web Layouts. In *Web Engineering*, Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso (Eds.). Springer International Publishing, Cham, 19–26.

[18] Chunggi Lee, Sanghoon Kim, Dongyun Han, Hongjun Yang, Young-Woo Park, Bum Chul Kwon, and Sungahn Ko. 2020. GUIComp: A GUI Design Assistant with Real-Time, Multi-Faceted Feedback. 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/10.1145/3313831.3376327

[19] Hsin-Ying Lee, Weilong Yang, Lu Jiang, Madison Le, Irfan Essa, Haifeng Gong, and Ming-Hsuan Yang. 2019. Neural Design Network: Graphic Layout Generation with Constraints. *CoRR* abs/1912.09421 (2019). arXiv:1912.09421 http://arxiv.org/abs/1912.09421

[20] Luis Leiva. 2012. Interaction-Based User Interface Redesign. In *Proceedings of the 2012 ACM International Conference on Intelligent User Interfaces* (Lisbon, Portugal) *(IUI '12)*. Association for Computing Machinery, New York, NY, USA, 311–312. https://doi.org/10.1145/2166966.2167028

[21] Luis A Leiva, Asutosh Hota, and Antti Oulasvirta. 2020. Enrico: A Dataset for Topic Modeling of Mobile UI Designs. In *22nd International Conference on Human-Computer Interaction with Mobile Devices and Services, Oldenburg, Germany, October 5–8, 2020 (MobileHCI '20)*. ACM, New York, NY, USA, 7.

[22] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2019. LayoutGAN: Generating Graphic Layouts with Wireframe Discriminators. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=HJxB5sRcFQ

[23] Yang Li, Julien Amelot, Xin Zhou, Samy Bengio, and Si Si. 2020. Auto Completion of User Interface Layout Design Using Transformer-Based Tree Decoders. *ArXiv* (2020), 1–11.

[24] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[25] Jonas Löwgren and Ulrika Laurén. 1993. Supporting the use of guidelines and style guides in professional user interface design. *Interacting with Computers* 5, 4 (1993), 385 – 396. https://doi.org/10.1016/0953-5438(93)90003-C

[26] Rafael Müller, Simon Kornblith, and Geoffrey E Hinton. 2019. When does label smoothing help?. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/f1748d6b0fd9d439f71450117eba2725-Paper.pdf

[27] Gene Myers. 1999. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *J. ACM* 46, 3 (May 1999), 395–415.

[28] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2014. Learning layouts for single-page graphic designs. *IEEE Transactions on Visualization and Computer Graphics* 20, 8 (2014), 1200–1213.

[29] Antti Oulasvirta, Niraj Ramesh Dayama, Morteza Shiripour, Maximilian John, and Andreas Karrenbauer. 2020. Combinatorial 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, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376593

[34] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2818–2826. https://doi.org/10.1109/CVPR.2016.308

[35] Sou Tabata, Hiroki Yoshihara, Haruka Maeda, and Kei Yokoyama. 2019. Automatic Layout Generation for Graphical Design Magazines. In *ACM SIGGRAPH 2019 Posters* (Los Angeles, California) *(SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 2 pages.

[36] Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. 2012. Learning Design Patterns with Bayesian Grammar Induction. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST '12)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/2380116.2380127

[37] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. 2016. Sketchplore: Sketch and explore layout designs with an optimiser. *Conference on Human Factors in Computing Systems - Proceedings* 07-12-May- (2016), 3780–3783.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[39] Pengfei Xu, Hongbo Fu, Takeo Igarashi, and Chiew-Lan Tai. 2014. Global Beautification of Layouts with Interactive Ambiguity Resolution. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (Honolulu, Hawaii, USA) *(UIST '14)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/2642918.2647398

[40] Xinru Zheng, Xiaotian Qiao, Ying Cao, and Rynson W. H. Lau. 2019. Content-Aware Generative Modeling of Graphic Design Layouts. *ACM Trans. Graph.* 38, 4, Article 133 (July 2019), 15 pages. https://doi.org/10.1145/3306346.3322971

## A GRAPH NEURAL NETWORK DETAILS

### A.1 Relation prediction

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

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

$$\hat{p}_{A_r} = \sigma(\mathrm{MLP}_{\mathrm{pred}}(F'_A)), \tag{7}$$

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 activation function $\sigma$. The output is then an $|R|$-dimensional vector that assigns a specific score to each possible label category for every input edge. The predicted labels $\{\hat{r}_i\}$ are computed by the arg max on these vectors. By replacing the unknown edge labels from the partial graph with the predicted labels from the network, a complete graph $\hat{G}$ is generated that can then be used in a subsequent layouting module. Note that the relation prediction are learned independently for the two relation categories. Hence, we create two models $\mathrm{Rel}^{\mathrm{pos}}, \mathrm{Rel}^{\mathrm{align}}$, one for each relation type.

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

*Loss.* The network is trained with a reconstruction loss $\mathcal{L}_{\mathrm{rec}}$ on the category prediction, and an entropy loss $\mathcal{L}_{\mathrm{kl}_1}$ between the generated latent vectors $z_{\mathrm{rel}}$ and the prior distribution $\mathcal{N}(0,1)$: $\mathcal{L}_{\mathrm{rec}_1} = \mathrm{CE}(\{\hat{r_i}\}, \{r^*\}, w)$, where $\mathrm{CE} = -\sum_n^{|R|} r_n^* \log \hat{r_n}$ is the cross-entropy function, $\{\hat{r_i}\}$ the set of predicted relation categories, $\{r^*\}$ the set of true relation categories, and $w$ is a weight parameter to account for imbalanced data sets.
$\mathcal{L}_{\mathrm{kl}_1} = \mathrm{KL}(z_{\mathrm{rel}}, \mathcal{N}(0,1))$, where KL is the Kullback-Leiber divergence function, $z_{\mathrm{rel}}$ the encoded latent variable, and $\mathcal{N}(0,1)$ the standard normal distribution. Both losses are summed with weights $\lambda_{\mathrm{rec}_1}, \lambda_{\mathrm{kl}_1}$ to form the complete loss for the relation module:

$$\mathcal{L}_{\mathrm{rel}} = \lambda_{\mathrm{rec}_1}\mathcal{L}_{\mathrm{rec}_1} + \lambda_{\mathrm{kl}_1}\mathcal{L}_{\mathrm{kl}_1}. \tag{8}$$

### A.2 Layout generation

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

$$F'_V, F'_{\hat{A}} = \mathrm{GCN}_{\mathrm{lay1}}(F_V, F_{\hat{A}}). \tag{9}$$

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 $e_{\mathrm{new}}$, the half-empty size box is concatenated $\tilde{s}_{e_{\mathrm{new}}} = (\emptyset, \emptyset, \tilde{w}_{e_{\mathrm{new}}}, \tilde{h}_{e_{\mathrm{new}}})$: $F''_V = (F'_V, \{\tilde{s}_{e_i}\})$. 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 $F''_{\hat{A}} = (F'_{\hat{A}}, (0, 0, 0, 0))$. These features are passed through another graph convolution network $\mathrm{GCN}_{\mathrm{lay2}}$, and a variational auto encoder-decoder network of fully-connected layers $\mathrm{h}_s^{\mathrm{enc}}, \mathrm{h}_s^{\mathrm{dec}}$ generates the box prediction from the new node feature of the new element $e_{\mathrm{new}}$:

$$F'''_V, F'''_{\hat{A}} = \mathrm{GCN}_{\mathrm{lay2}}(F''_V, F''_{\hat{A}}), \tag{10}$$

$$z_{\mathrm{lay}} = \mathrm{h}_s^{\mathrm{enc}}(s^*_{e_{\mathrm{new}}}), \tag{11}$$

$$\hat{s}_{e_{\mathrm{new}}} = \mathrm{h}_s^{\mathrm{dec}}(F'''_{e_{\mathrm{new}}}, z_{\mathrm{lay}}), \tag{12}$$

where $s^*_{e_{\mathrm{new}}}$ 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 $z_{\mathrm{lay}}$ is sampled from a prior distribution.

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

$$\mathcal{L}_{\mathrm{lay}} = \lambda_{\mathrm{rec}_2}\mathcal{L}_{\mathrm{rec}_2} + \lambda_{\mathrm{size}}\mathcal{L}_{\mathrm{size}} + \lambda_{\mathrm{kl}_2}\mathcal{L}_{\mathrm{kl}_2}. \tag{13}$$

## 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 $\hat{s}_{e_{\mathrm{new}}}$, and it produces an updated size box $\hat{s}'_{e_{\mathrm{new}}}$. 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 $\mathrm{GCN}_{\mathrm{refine}}$. The updated feature vector of the new element is fed into a multi-layer perceptron network $\mathrm{MLP}_{\mathrm{refine}}$ to generate the refined size box $\hat{s}'_{e_{\mathrm{new}}}$:

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

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

*Training and loss.* To teach the network to refine layouts, the training layouts are perturbed by a random strength $\delta = U(-0.05, 0.05)$ which is added to the coordinates of new elements $s'_{e_{\mathrm{new}}} = (\tilde{x}_{e_{\mathrm{new}}} + \delta, \tilde{y}_{e_{\mathrm{new}}} + \delta, \tilde{w}_{e_{\mathrm{new}}}, \tilde{h}_{e_{\mathrm{new}}})$. The boxes of the other elements are kept as is such that there are well-aligned elements to align to. The network is trained with a reconstruction loss $\mathcal{L}_{\text{rec}_3}$ 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 and the ground truth box $s^*$.

### A.4 Graph convolution network

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

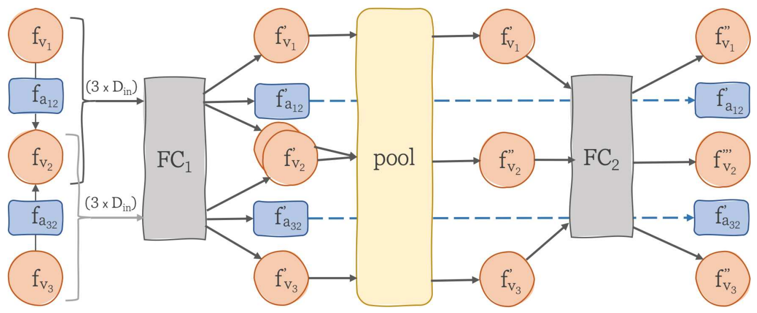

Fig. 17. The graph convolutional layer. $f_v$ represents the features of a vertex, $f_a$ the features of an arrow (edge). *FC* describe fully-connected layers.

This graph convolution layer GC is depicted in Figure 17. The input are feature vectors of the vertices $f_{v_i}$ and of the arrows $f_{a_i}$ with dimension $D_{\text{in}}$. First, they are passed as triples in the form of $(f_{v_i}, f_{a_i}, f_{v_j})$ where $v_i$ is a source of the triple $a_i$ and $v_j$ the target through a fully-connected neural 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 then pooled in the next step which is by default implemented as an *average* pooling, creating $\{f''_{v_i}\}$. The pooled vertex features are finally passed through a second fully-connected neural network FC$_2$ to generate the output of the graph convolution for the vertices $\{f'''_{v_i}\}$. The features of the arrows $\{f'_{a_i}\}$ are output directly from the first network FC$_1$. Every fully-connected layer is followed by an optional dropout layer, and the output is processed by an activation function $\sigma$. Parameters of this layer are the input Dimension $D_{\text{in}}$, the hidden dimension $D_{\text{hidden}}$ and the output dimension $D_{\text{out}}$. We will use the shorthand $(D_{\text{in}}, D_{\text{hidden}}, D_{\text{out}})$ to describe the dimensions going forward. Graph convolution networks as denoted GCN above are then stacked layers of the described GC layer.

### A.5 GNN parameters

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

*Relation module.* In the relation module, the embedding dimension is $D_{\text{embed}} = 128$ and the dimension of the latent variable is $D_{z_{\text{rel}}} = 32$. GCN$_{\text{pred}}$ consists of three graph convolutional layers as detailed in Table 7. MLP$_{\text{pred}}$ consists of two fully-connected layers as shown in Table 8. The embedding dimension of the ground-truth encoding is $D^{\text{enc}}_{\text{embed}} = 64$, and GCN$_{\text{enc}}$ is a three-layered graph convolutional network as described in Table 9. MLP$_{\text{enc}}$ is a two-headed network shown in Table 10.

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

| | $\mathrm{GCN_{pred}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | GC(128+32, 512, 128) | ReLU | 0.1 |
| 2 | GC(128, 512, 128) | ReLU | 0.1 |
| 3 | GC(128, 128, 128) | ReLU | 0.1 |

Table 7. $\mathrm{GCN_{pred}}$ layers.

| | $\mathrm{MLP_{pred}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | FC(128, 512) | ReLU | 0.1 |
| 2 | FC(512, $\lvert R \rvert$) | softmax | 0 |

Table 8. $\mathrm{MLP_{pred}}$ layers.

| | $\mathrm{GCN_{enc}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 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 9. $\mathrm{GCN_{enc}}$ layers.

| | $\mathrm{MLP_{enc}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| $1_\mu$ | FC(128, 32) | - | 0 |
| $1_\sigma$ | FC(128, 32) | - | 0 |

Table 10. $\mathrm{MLP_{enc}}$ layers.

| | $\mathrm{GCN_{l1}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | GC(128, 512, 124) | ReLU | 0.1 |
| 2 | GC(124, 512, 124) | ReLU | 0.1 |
| 3 | GC(124, 512, 124) | ReLU | 0.1 |

Table 11. $\mathrm{GCN_{l1}}$ layers.

| | $\mathrm{GCN_{l2}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | GC(124+4, 512, 128) | ReLU | 0.1 |
| 2 | GC(128, 512, 128) | ReLU | 0.1 |
| 3 | GC(128, 512, 128) | ReLU | 0.1 |

Table 12. $\mathrm{GCN_{l2}}$ layers.

| | $\mathrm{h^{enc}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | FC(4+128, 128) | ReLU | 0.1 |
| 2 | FC(128, 128) | ReLU | 0.1 |
| $3_\mu$ | FC(128, 32) | - | 0 |
| $3_\sigma$ | FC(128, 32) | - | 0 |

Table 13. $\mathrm{h^{enc}}$ layers.

| | $\mathrm{h^{dec}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | FC(32+128, 128) | ReLU | 0.1 |
| 2 | FC(128, 64) | ReLU | 0.1 |
| 3 | FC(64, 4) | - | 0 |

Table 14. $\mathrm{h^{dec}}$ layers.

*Layout module.* The dimension of the embedding is the same as above with $D_{\mathrm{embed}} = 128$. $\mathrm{GCN_{l1}}, \mathrm{GCN_{l2}}$ consist of three layers according to the definitions in Table 11 and Table 12.

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

The employed loss weights of this module are $\lambda_{\mathrm{rec}_2} = 1$, $\lambda_{\mathrm{rec}_3} = 10$ (to prioritize size reconstruction), and $\lambda_{\mathrm{kl}_2} = 0.01$.

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

*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.

| | $\text{GCN}_{\text{refine}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 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. $\text{GCN}_{\text{refine}}$ layers.

| | $\text{MLP}_{\text{refine}}$ | $\sigma$ | $d$ |
|---|---|---|---|
| 1 | FC(128, 512) | ReLU | 0.1 |
| 2 | FC(512, 4) | - | 0 |

Table 16. $\text{MLP}_{\text{refine}}$ layers.

## 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: $w' = \lfloor \frac{w}{g} \rceil, h' = \lfloor \frac{h}{g} \rceil, x^{0'} = \lfloor \frac{x^0}{g} \rceil, y^{0'} = \lfloor \frac{y^0}{g} \rceil$, where $\lfloor \cdot \rceil$ denotes a rounding function to the nearest integer.

This updated layout sequence is then converted into a token sequence $S_L^{\text{in}}$ 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:

$$\text{Vocab} = (t_{\text{<pad>}}, t_{\text{<start>}}, t_{\text{<end>}}, \{t_{c_i}\}, \{t_{x'_i}\}, \{t_{y'_i}\}, \{t_{w'_i}\}, \{t_{h'_i}\}), \tag{16}$$

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 $t_e = t_{c_e}, t_{w'_e}, t_{h'_e}, t_{x_e^{0'}}, t_{y_e^{0'}}$. The tokens are encapsulated by a special start and stop token to produce the final input sequence:

$$S_L^{\text{in}} = t_{\text{<start>}}, t_{c_{e_0}}, t_{w'_{e_0}}, t_{h'_{e_0}}, t_{x_{e_0}^{0'}}, t_{y_{e_0}^{0'}}, ..., t_{c_{e_{\text{new}}}}, t_{w'_{e_{\text{new}}}}, t_{h'_{e_{\text{new}}}}, t_{\text{<end>}}. \tag{17}$$

### 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].

### B.3 Network parameters

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

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

## C SEQUENCE-ALIGNED NEIGHBOR SEARCH DETAILS

### C.1 Insertion point details

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

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

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

### C.2 Feature scaling

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

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

where $\lambda_{\min}$ is the minimum scaling factor for far away elements, and $\gamma$ is the distance at which point the minimum scaling takes effect.

### C.3 Penalty feature

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

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

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

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

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

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

### C.4 Feature representation

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

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

```
# box features of matched elements in the layout
(0.02,  0.03,  0.21,  0.03,
 0.03,  0.10,  0.79,  0.18,
 0.03,  0.35,  0.11,  0.02,
 0.03,  0.40,  0.45,  0.08,
 0.03,  0.51,  0.37,  0.22,
# box padding to match longest feature sequence
100.0, 100.0, 100.0, 100.0,
# box feature of target element
 0.14,  0.04,  3.59,

# word vector of the component names
 0.02, -0.01,  0.00, ...,  0.09, -0.02,  0.05, -0.04, -0.03,
 0.03, -0.11, -0.02, ...,  0.08,  0.01,  0.05,  0.03, -0.00,
-0.01, -0.13,  0.04, ...,  0.13, -0.01,  0.16,  0.02,  0.04,
 0.17, -0.19,  0.00, ..., -0.02,  0.07,  0.04,  0.01,  0.03,
 0.07, -0.11,  0.03, ...,  0.02,  0.02,  0.17, -0.10, -0.01,
# word vector padding
10.00, 10.00, 10.00, ..., 10.00, 10.00, 10.00, 10.00, 10.00,
# word vector of target element
-0.03, -0.17,  0.15, ...,  0.13,  0.00,  0.05, -0.03,  0.02)
```

## D DATA SET DETAILS

### D.1 Artificial web patterns

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

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

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

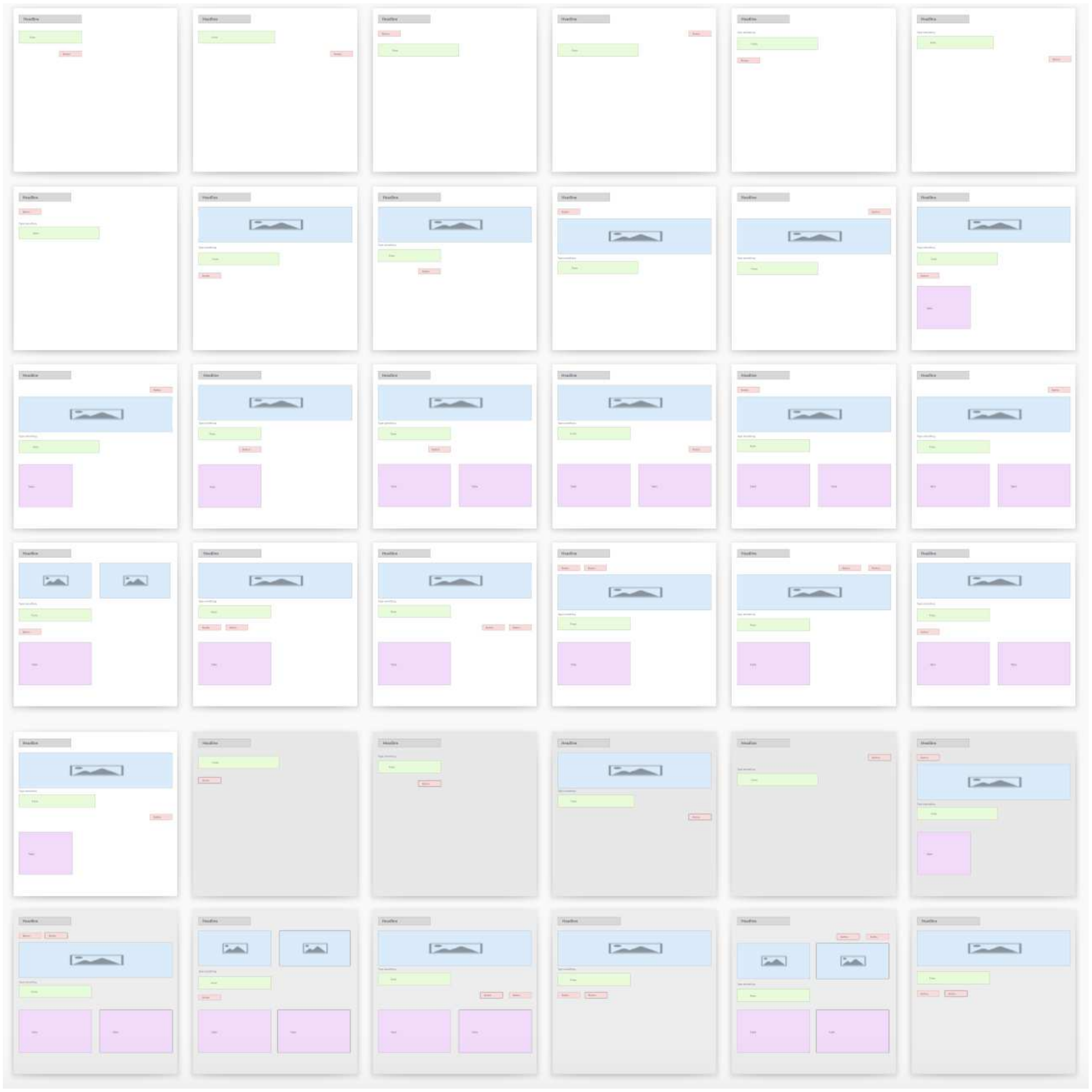

Fig. 18. The set of layouts in the varying buttons data set. Grey boxes correspond to headlines, green boxes to forms, red boxes to buttons, blue boxes to images, pink boxes to tables and black text represents a text paragraph. Grey layouts indicate test items that are not used during training.

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:

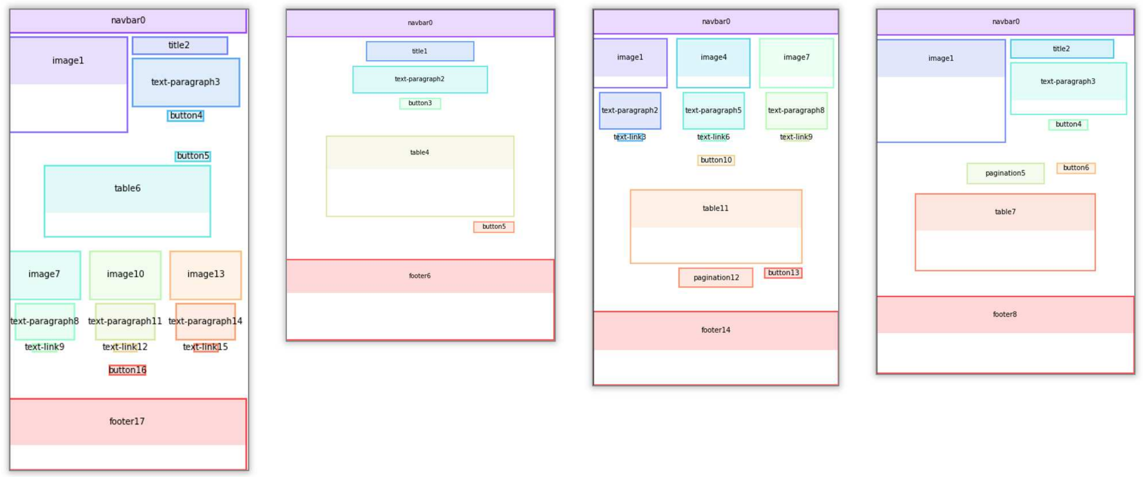

Fig. 19. The table pattern. The button accompanying the table is always right-aligned with the table and can be either above or below it. If the pagination is present, it is next to it.

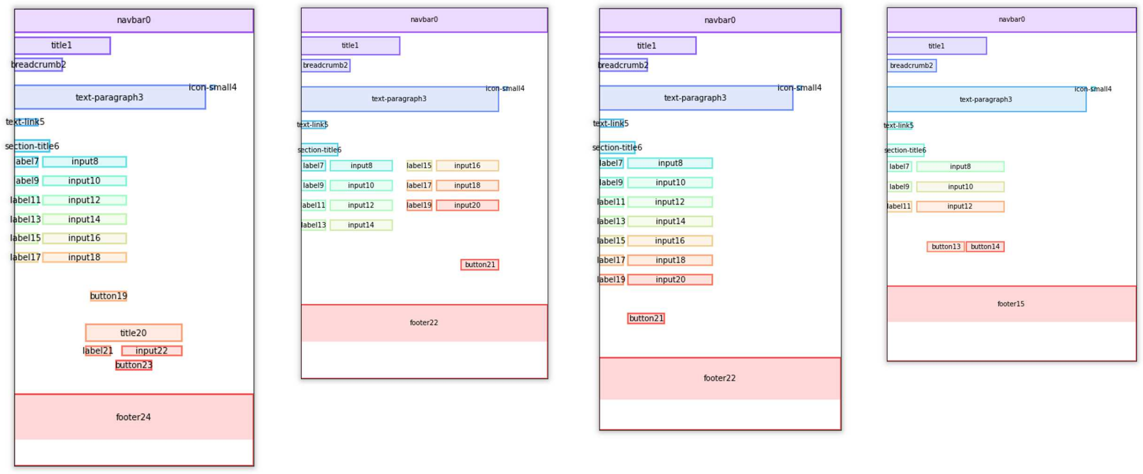

Fig. 20. The form pattern. There can be forms with one or two columns with the label to the left of the input. A single button can be either placed left or right-aligned with the right-most input field. With two buttons, they are placed next to each other and are right-aligned with the last input element.

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

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

$$\min(x^1_{e_1}, x^1_{e_2}) > \max(x^0_{e_1}, x^0_{e_2}) \wedge \min(y^1_{e_1}, y^1_{e_2}) > \max(y^0_{e_1}, y^0_{e_2}). \tag{24}$$

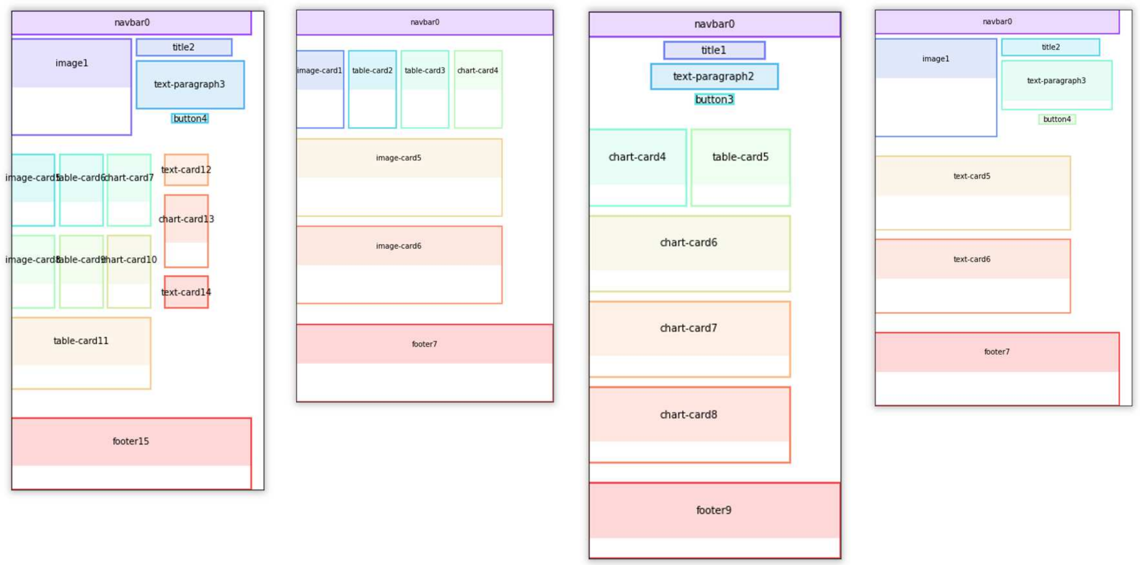

Fig. 21. The dashboard pattern arranges different type of cards in either a 2-column layout, 4-column layout, or a 3-column layout with an additional sidebar. Cards can either span a single column or all columns.

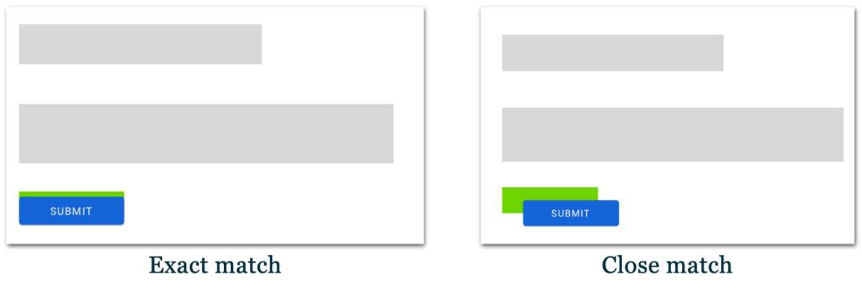

Fig. 22. Examples of exact and close matches. In the exact match, the alignment to the neighboring element is according to the expectation in green, even if it has a minor offset vertically. The close match has a major overlap but misses the alignment.

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

$$\min(x_e^0, y_e^0) < 0 \lor x_e^1 > w_L \lor y_e^1 > h_L. \tag{25}$$

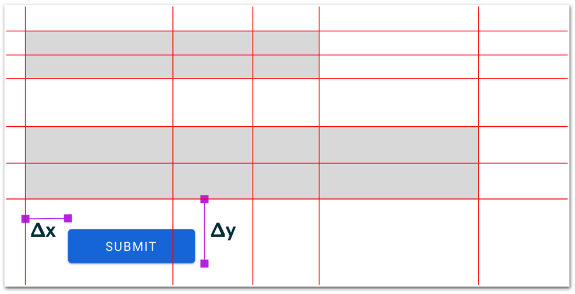

Fig. 23. Alignment takes the average of the minimum differences between any $x^0, x^c, x^1$ of the new element and the existing elements (shown with $\Delta x$) and between any $y^0, y^c, y^1$ of the new element and the existing elements ($\Delta y$).